TURBO VISION™

PROGRAMMING GUIDE

- TUTORIAL
- USING TURBO VISION
- REFERENCE

BORLAND

Turbo Vision™

Version 2.0

Programming Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN THE USA.

R1

15 14 13

CONTENTS

Part 1 Learning Turbo Vision

Chapter 11 Window and dialog box objects 195

Chapter 12 Control objects 211

Chapter 13 Data validation objects 237

Chapter 17 Streams 291

Chapter 18 Resources 309

Part 3 Turbo Vision Reference

Chapter 19 Turbo Vision reference 317

TABLES

F I G U R E S

LISTINGS

INTRODUCTION

This volume contains complete documentation for Turbo Vision, the object-oriented application framework. It describes not only *what* Turbo Vision can do and *how*, but also *why*. If you take the time to understand the underlying principles of Turbo Vision, you will find it a rewarding, time-saving, and productive tool: You can build sophisticated, consistent interactive applications in less time than you thought possible.

What's new in Turbo Vision?

Turbo Vision 2.0 adds new objects to the hierarchy and adds some new capabilities to the existing objects. Changes to existing objects are backward-compatible, so existing Turbo Vision code should compile without changes, and existing streams and resources should load without error.

Turbo Vision 2.0 has the following new features:

- Support for data validation (see Chapter 13)
- More built-in application functions, including
  - DOS shell
  - Window tiling and cascading
  - Safety checks on windows and dialog boxes
  - Standard menu functions
- Multi-state check boxes
- A new outline viewer object
- Stream versioning
- Larger clusters of check boxes and radio buttons

In addition, this manual includes the following new material:

- Expanded tutorial
- More example programs
- Chapters explaining how to use windows, applications, controls, editors, and data validators
- Expanded explanations of views and events
- More complete inheritance information in the reference section

What is Turbo Vision?

Turbo Vision is an object-oriented application framework for windowing programs. We created Turbo Vision to save you from endlessly recreating the basic platform on which you build your application programs.

Turbo Vision is a complete object-oriented application framework, including:

- Multiple, resizeable, overlapping windows
- Pull-down menus
- Mouse support
- Dialog boxes
- Data validation
- Built-in color installation
- Buttons, scroll bars, input boxes, check boxes and radio buttons
- Standard handling of keystrokes and mouse clicks

You might have used libraries of procedures and functions or objects, and at first glance Turbo Vision sounds a lot like a library. After all, you can buy libraries that give you menus, windows, mouse bindings, and more. But beneath that superficial resemblance is an important difference: Turbo Vision is not just a library; it's an *application framework.*

With Turbo Vision, you never have to modify the source code. You "change" Turbo Vision by extending it. The *TApplication* application skeleton remains unchanged inside APP.TPU. You add to it by deriving new object types and change what you need to by overriding the inherited methods with new methods that you write for your new objects.

Also, *Turbo Vision is a hierarchy,* not just a disjointed box full of tools. If you use any of it at all, you should use *all* of it. There is a single architectural vision behind every component of Turbo Vision, and they all work together in many subtle, interlocking ways. You shouldn't try to just "pull out" mouse support and use it—the "pulling out" would be more work than writing your own mouse bindings from scratch.

Turbo Vision is also *event-driven*, enabling you to write flexible programs that give your users control over what part of the program they want to access, rather than having the program dictate to them. The event-driven model is the same one used by modern graphical environments like Microsoft Windows.

We created Turbo Vision to save you an enormous amount of unnecessary, repetitive work, and to provide you with a proven application framework you can trust and build on. To get the most benefit from it, let Turbo Vision work for you.

Turbo Vision provides the basis for the integrated development environment, which we produced in a fraction of the time it would have taken to write it from scratch. Turbo Vision lets you use this same foundation for your own applications.

Why Turbo Vision?

After creating a number of programs with windows, dialog boxes, menus, and mouse support at Borland, we decided to package all that functionality into a reusable set of tools. Object-oriented programming gave us the vehicle, and Turbo Vision is the result.

Because Turbo Vision takes a standardized, rational approach to screen design, your applications acquire a familiar look and feel. That look and feel is identical to that of the Turbo languages themselves, and is based on years of experience and usability testing. Having a common and well-understood look to an application is a distinct advantage to your users and to yourself. No matter how arcane your application is in terms of what it *does*, the way to *use* it will always be familiar ground, and the learning curve will be easier to ascend.

Turbo Vision is also fast. Using Pascal and assembly language, we've optimized Turbo Vision to make it smooth and flicker-free, so it doesn't slow down your applications.

What you need to know

You need to be comfortable with object-oriented programming to use Turbo Vision. Turbo Vision makes extensive use of object-oriented techniques, including inheritance and polymorphism. These topics are covered in the chapter "Object-oriented programming," in the *User's Guide.*

In addition to object-oriented techniques, you also need to be familiar with the use of pointers and dynamic variables, because nearly all of Turbo Vision's object instances are dynamically allocated on the heap. If you're not familiar with pointers, or if you want to review the use of pointers, see the chapter "Using Pointers" in the *User's Guide.*

How to use this book

The Turbo Vision Programming Guide is expanded to make it more complete and easier to use. If you're already familiar with Turbo Vision, you'll probably want to skim Chapters 7, 13, and 19 to see what's new. If you're new to Turbo Vision, you should read through all of Part 1, "Learning Turbo Vision." The tutorial walks you through building a complete Turbo Vision application, explaining the principles of Turbo Vision and event-driven programming along the way.

What's in this book?

This manual has four parts:

- Part 1 introduces you to the principles behind Turbo Vision and provides a tutorial that walks you through writing a complete Turbo Vision application.
- Part 2 gives greater detail on all the essential elements of Turbo Vision, including explanations of the objects in the Turbo Vision hierarchy and suggestions for how to write better applications. Part 2 also covers collections, streams, and resources. These are important data management tools provided with Turbo Vision.
- Part 3 is a complete reference lookup for all the objects and other elements included in the Turbo Vision units.

PART 1

Learning Turbo Vision

CHAPTER

1

Stepping into Turbo Vision

In the next several chapters, you'll build a complete Turbo Vision application, starting from the very simplest instance of the bare framework and working up to a fairly complex data-entry system with input validation and context-sensitive prompts.

The walk-through consists of twelve steps:

- Step 1: Creating an application
- Step 2: Customizing menus and status lines
- Step 3: Responding to commands
- Step 4: Adding a window
- Step 5: Adding a clipboard window
- Step 6: Saving and loading the desktop
- Step 7: Using resources
- Step 8: Creating a data-entry window
- Step 9: Setting control values
- Step 10: Validating data entry
- Step 11: Adding collections of data
- Step 12: Creating a custom view

The source code for the application in this tutorial is provided at various stages on your distribution disks. The files are named TUTOR01.PAS, TUTOR02.PAS, and so on, corresponding to the numbered steps in the tutorial.

At the end of the tutorial, you'll find some suggestions on how you might add onto the finished program to give it even more capability.

What's in a Turbo Vision application?

Before you start building your first Turbo Vision application, let's take a look at "what's in the box"—what tools Turbo Vision gives you to build your applications.

Views, events and engines

A Turbo Vision application is a cooperating society of *views*, *events*, and *engines*. Let's look at each of those.

Views

Views are covered in detail in Chapter 8.

A *view* is any program element that is visible on the screen—and all such elements are objects. In a Turbo Vision context, if you can see it, it's a view. Fields, field captions, window borders, scroll bars, menu bars, and push buttons are all views. Views can be combined to form more complex elements like windows and dialog boxes. These collective views are called *groups*, and they operate together as though they were a single view. Conceptually, groups may be considered views.

Views are always rectangular. This includes rectangles that contain a single character, or lines which are only one character high or one character wide.

Events

Events are explained in detail in Chapter 9.

An *event* is some sort of occurrence to which your application must respond. Events come from the keyboard, from the mouse, or from other parts of Turbo Vision. For example, a keystroke is an event, as is a click of a mouse button. Events are queued by Turbo Vision's application skeleton as they occur, then they are processed in order by an event handler. The *TApplication* object, which is the body of your application, contains an event handler. Through a mechanism that will be explained later on, events that are not serviced by *TApplication* are passed along to other views owned by the program until either a view is found to handle the event, or an "abandoned event" error occurs.

For example, an *F1* keystroke invokes the help system. Unless each view has its own entry to the help system (as might happen in a context-sensitive help system), the *F1* keystroke is handled by the main program's event handler. Ordinary alphanumeric keys or the line-editing keys, by contrast, need to be handled by the view that currently has the *focus;* that is, the view that is currently interacting with the user.

Engines

Engines, sometimes called "mute objects," are any other objects in the program that are not views. They are "mute" because they don't speak to the screen themselves. They perform calculations, communicate with peripherals, and generally do the work of the application. When an engine needs to display some output to the screen, it must do so through the cooperation of a view.

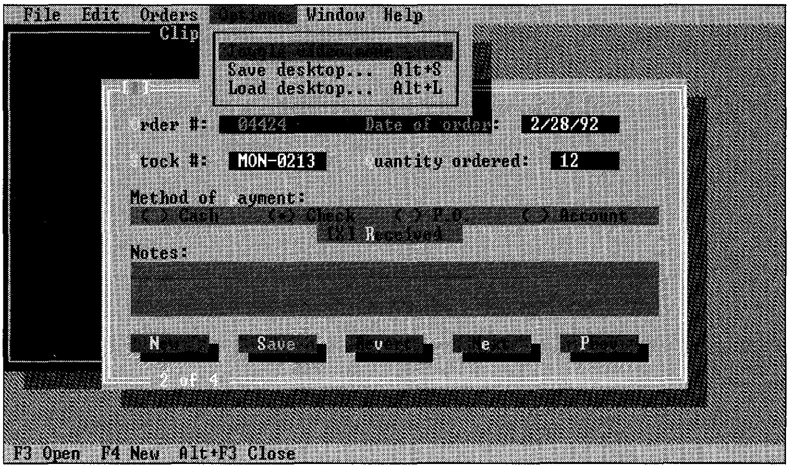
☞ This concept is very important to keeping order in a Turbo Vision application: *Only views may access the display.*

Nothing will stop your engines from writing to the display with Pascal's *Write* or *Writeln* statements. However, if you write to the display "on your own," the text you write will disrupt the text that Turbo Vision writes, and the text that Turbo Vision writes (by moving or sizing windows, for example) will obliterate this "renegade" text.

All these items are described in Chapter 8, "Views."

Figure 1.1 shows a collection of common objects that might appear as part of a Turbo Vision application. The *desktop* is the shaded background against which the rest of the application appears. Like everything else in Turbo Vision, the desktop is an object. So are the *menu bar* at the top of the display and the *status line* at the bottom. Words in the menu bar represent menus, which are "pulled down" by clicking the words with the mouse or by pressing *hot keys.*

Figure 1.1
Turbo Vision objects onscreen



rder #: 04424 Date of order: 2/28/92

tock #: MON-0213 uantity ordered: 12

Method of ayment:

Notes:

The text that appears in the status line is up to you, but typically it displays messages about the current state of your application, shows available hot keys, or prompts for commands that are currently available to the user.

When a menu is pulled down, a *highlight bar* slides up and down the menu's list of selections in response to movements of the mouse or cursor keys. When you press *Enter* or click the left mouse button, the item highlighted at the time of the button press is selected. Selecting a menu item transmits a command to some part of the application.

Your application typically communicates with the user through one or more *windows* or *dialog boxes*, which appear and disappear on the desktop in response to commands from the mouse or the keyboard. Turbo Vision provides a great assortment of window machinery for entering and displaying information. Window interiors can be made *scrollable*, which enbles windows to act as portals into larger data displays such as document files. Scrolling the window across the data is done by moving a *scroll bar* along the bottom of the window, the right side of the window, or both. The scroll bar indicates the window's position relative to the entirety of the data being displayed.

Step 1: Creating an application

The usual way to get started with a new language or library is to write the simplest program possible, such as a very short program that displays the text "Hello, World!" on the screen. In this step, you'll

- Create the absolute minimum Turbo Vision program
- Extend the basic application

Constructing the simplest program

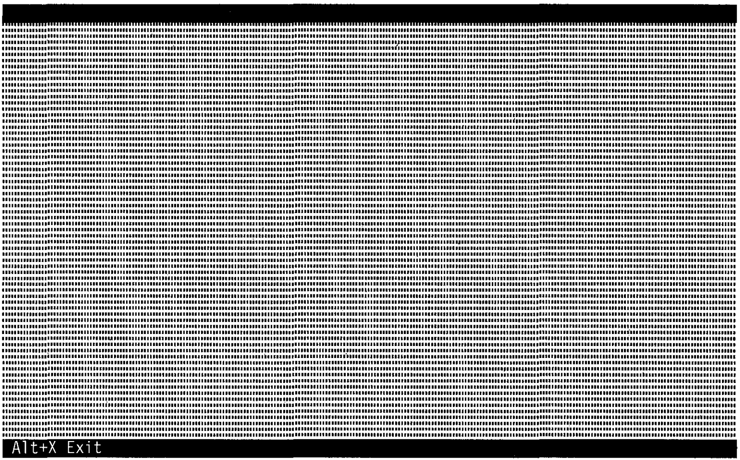
The application object provides the framework on which you'll build a real application. The simplest Turbo Vision program, then, is just an instance of the base application object, *TApplication*. Listing 1.1 shows the very simplest Turbo Vision application.

Listing 1.1
The simplest Turbo Vision program

```
program Minimal;

uses App;

var MyApp: TApplication;

begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.
```

In the object-oriented world of Turbo Vision, even your application is an object. As you'll see later, that object is also a view and a group. The definition of the basic application object is in a unit called *App*. Although the program in Listing 1.1 only uses that one unit directly, *App* itself makes use of several other Turbo Vision units. As you add to this program, you'll use parts of all of them.

If you run the program, you'll see a screen that looks like the one in Figure 1.2. Note that the application has a blank menu bar at the top of the screen, a status line at the bottom that indicates the availability of the *Alt+X* hot key to exit the program, and a shaded desktop in between.



Alt+X Exit

Figure 1.2
Default TApplication screen

Program *Minimal* shows the default behavior of the object type *TApplication*. In fact, *TApplication* can do a lot more than just respond to *Alt+X* or a click the status line. What you see is just the bare frame of a real application. As you start hanging more items on the application framework, you'll find that the default application already has functions built in to handle most of them.

Extending the application object

In the remaining steps of this tutorial, you'll add new abilities to the application object. If you're not accustomed to using libraries of objects, you might be tempted to open APP.PAS and make your changes directly to *TApplication*'s source code. You should resist that temptation for several reasons.

- The purpose of an application framework is to provide a standard, reliable foundation for all your applications. If you modify that basis for each of your programs, you defeat one of the greatest benefits of using the framework.
- Modifying proven source code is a good way to introduce bugs. Turbo Vision objects interact with each other in numerous interlocking ways, so making a change in one of the standard objects could have unforeseen consequences in apparently unrelated places.
- One of the great benefits of object-oriented programming is extensibility. Instead of rewriting your code, you can derive a new object type from an existing one, and you only have to write code for the parts that will differ. That way you keep your

solid, reliable base for all your applications, and all your customizations are in one convenient place.

The first step you should take is to derive a new application object to which you'll add your changes, as shown in Listing 1.2.

Listing 1.2
TUTOR01.PAS, an extensible application

```
program Tutor01;

uses App;                        { APP.TPU holds application objects }

type
  TTutorApp = object(TApplication)    { define your application type }
  end;                          { leaving room for future extensions }

var TutorApp: TTutorApp;      { declare an instance of your new type }

begin
  TutorApp.Init;                    { set up the application object }
  TutorApp.Run;                             { interact with the user }
  TutorApp.Done;                 { dispose of the application object }
end.
```

Normally you wouldn't declare a new object type with no new fields or methods, but *TTutorApp* will have new field and method declarations added in future steps. *Tutor01* behaves exactly like *Minimal*, since at this point *TTutorApp* is exactly like its ancestor object type, *TApplication*.

Creating a command unit

One aspect of making sure a Turbo Vision application is flexible and extensible is making sure that commands are available at any point in the program. Turbo Vision commands are integer-type constants. The easiest way to handle this is to create a separate unit that contains only constant definitions. Listing 1.3 shows part of a unit containing all the command constant definitions for *Tutorial*.

Don't worry too much about these command constants right now. The important thing is to make them available. You'll be using them extensively in the next several steps, and you'll see the advantages of having them in a single location.

Listing 1.3
TUTCONST.PAS, a unit defining command constants

```
unit TutConst;          { global constants for Turbo Vision Tutorial }

interface

const
  cmOrderNew = 251;
  cmOrderWin = 252;
  cmOrderSave = 253;
  cmOrderCancel = 254;
  cmOrderNext = 255;
  cmOrderPrev = 250;
  cmClipShow = 260;
  cmAbout = 270;
  cmFindOrderWindow = 2000;

const
  cmOptionsVideo = 1502;
  cmOptionsSave = 1503;
  cmOptionsLoad = 1504;

implementation
end.
```

Keeping constants in a separate unit has several advantages. The constant unit serves as a single, central location for all constants, which helps you avoid duplicating constant definitions. It also speeds up compilation of the program somewhat, as the unit will rarely have to be recompiled.

Step 2: Customizing menus and status lines

```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

Turbo Vision application objects divide the screen into three main parts: the desktop, the menu bar, and the status line. In this step, you'll learn a bit about each, then you'll learn how to

- Customize the status line
- Customize the menu bar

You'll rarely have occasion to modify the desktop object, but you'll learn how to use its capabilities in Step 4.

Initializing the application

When you initialize an application object, the *Init* constructor calls three virtual methods called *InitDesktop*, *InitMenuBar*, and *InitStatusLine* to set up objects to handle the desktop, the menu bar, and the status line. This means you can change any of those three objects without having to change the application's constructor. You just override the method that sets up the particular object. You'll rarely want to change the desktop object, since its operation is quite straightforward. But you'll nearly always customize the status line and menu bar of your applications.

In Step 3 you'll extend Init itself, but you'll build on the existing method, rather than having to reproduce all its operations.

Customizing the status line

The application object's virtual method *InitStatusLine* initializes a status line object and assigns it to the global variable *StatusLine.* To create a custom status line, you need to override *InitStatusLine* to construct a new status line object and assign it to *StatusLine.*

Constructing a status line object takes three steps:

- Setting the boundaries of the status line
- Defining ranges of help contexts
- Defining status keys

Be sure to add the declaration of *InitStatusLine* to the type declaration of *TTutorApp.*

Setting the boundaries

In general, the status line is the bottom line of the application screen. You should rarely have occasion to put it elsewhere. The default application assumes that its bottom line is a status line, so if you move the status line somewhere else, you'll need to make sure some other object, such as the desktop, takes over that last screen line.

Because the boundaries of the status line depend on the boundaries of the application itself, and because the application's boundaries change depending on video modes, *InitStatusLine* should query the application object for its boundaries and set the status line bounds accordingly.

Since all Turbo Vision views are rectangular, they store their boundaries in a rectangle object of type *TRect*. *TRect* has two fields, *A* and *B*, which represent the upper left and bottom right corners of the view. *A* and *B* in turn are point objects, which have two fields *X* and *Y*, which represent the column and row coordinates of the point.

Views have a method called *GetExtent* that returns the bounding rectangle of the view in its single **var** parameter. *InitStatusLine* will call the application's *GetExtent*, then modify the returned rectangle:

```
type
  TTutorApp = object(TApplication)
    procedure InitStatusLine; virtual;      { declare the new method }
  end;
```

```
procedure TTutorApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);                        { get the application's boundaries }
  R.A.Y := R.B.Y - 1;                       { set top to one above bottom }
   ⋮
```

Defining ranges of help contexts

For full information on help contexts, see Chapter 10, "Application objects."

Every view has an object field of type *Word* that holds its *help context*. Help contexts serve two main purposes. They provide a number that a context-sensitive help system can use to determine what help screen to display, and they determine which status line shows at the bottom of the screen.

A status line object contains a linked list of records, called *status definitions*. A status definition holds a range of *help contexts* and a list of status line items or *status keys* to display when the application's help context falls within the specified range. You create status definitions by calling the function *NewStatusDef*.

The default status line object assigned to *StatusLine* defines a single status definition with a range covering all possible help contexts, so the same status line shows, no matter what the current help context:

```
New(StatusLine, Init(R,                   { use the boundaries passed in R }
  NewStatusDef(0, $FFFF,              { cover help context range 0..$FFFF }
    StdStatusKeys(nil),                   { include standard status keys }
  nil)));                                        { no further definitions }
```

For this application, you need two status definitions: one for most help contexts, and a special one for help contexts $F000 and higher. In Step 7, you'll create a customized data entry window that sets the help context to values higher than $F000. Creating a second status definition is just a matter of supplying another nested call to *NewStatusDef*:

```
procedure TTutorApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);
  R.A.Y. := R.B.Y - 1;
  New(StatusLine, Init(R,
    NewStatusDef(0, $EFFF,                    { note the different range }
      StdStatusKeys(nil),
    NewStatusDef($F000, $FFFF,                 { new range $F000..$FFFF }
      StdStatusKeys(nil),                     { same status keys for now }
    nil))));                                 { note one more parenthesis }
end;
```

Now you have two status definitions covering two ranges of help contexts, but they both display the same set of items. In the next section, you'll define custom status line items for each range.

Defining status keys

Each status definition has its own linked list of status keys, which are the items you actually see on the status line (although you can define keys with no text, as you'll see). Each status key consists of four items:

- A text string that shows on the status line
- A keyboard scan code for a hot key
- A command
- A pointer to the next status key

Briefly, the text for a status key defines what shows on the status line. Any text enclosed by tildes ('~') shows up highlighted. An empty text string means the item doesn't show up on the screen at all, but it still binds the hot key to the command.

The hot key can be any "special" key, such as a function key, an *Alt+* key combination, or a *Ctrl +* key combination. Turbo Vision's *Drivers* unit defines mnemonic constants corresponding to all common key combinations.

You'll use commands extensively in Step 3.

All you need to know about Turbo Vision commands at this point is that they are integer constants. Turbo Vision defines some standard commands, and you can define your own. Specifying a command in a status key declaration *binds* the command to the hot key and the status line item. Clicking the status key or pressing the hot key generates the command.

Listing 1.4 shows the *StdStatusKeys* function, which returns the default status keys.

Listing 1.4
Creating the standard status keys

```
function StdStatusKeys(Next: PStatusItem): PStatusItem;
begin
  StdStatusKeys :=
    NewStatusKey('', kbAltX, cmQuit,          { bind Alt+X to cmQuit }
    NewStatusKey('', kbF10, cmMenu,     { these keys are invisible... }
    NewStatusKey('', kbAltF3, cmClose,   { ...but still bind hot keys }
    NewStatusKey('', kbF5, cmZoom,
    NewStatusKey('', kbCtrlF5, cmResize,
    NewStatusKey('', kbF6, cmNext,
    Next))))));                         { append any keys passed in Next }
end;
```

The items displayed on the status line for each status definition form a linked list of status keys, created by nested calls to the function *NewStatusKey*. In complicated status line declarations, all these nested function calls can get confusing. One way to simplify this code is to define functions that return status definitions or lists of status keys, especially if you have items in common between multiple status definitions. For example, calling *StdStatusKeys* keeps you from having to declare *Alt+X, F10,* and the other standard keys in *every* status definition.

Listing 1.5 shows the revised program, including the declararion of several new status keys.

Listing 1.5
TUTOR02A.PAS, the application with a custom status line

```
program Tutor02a;

uses App, Objects, Menus, Drivers, Views, TutConst;

type
  TTutorApp = object(TApplication)
    procedure InitStatusLine; virtual;      { declare the new method }
  end;

procedure TTutorApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  New(StatusLine, Init(R,
    NewStatusDef(0, $EFFF,             { this is the "normal" range }
      NewStatusKey('~F3~ Open', kbF3, cmOpen,              { bind F3 }
      NewStatusKey('~F4~ New', kbF4, cmNew,                 { and F4 }
      NewStatusKey('~Alt+F3~ Close', kbAltF3, cmClose,  { and Alt+F3 }
      StdStatusKeys(nil)))),           { and add the standard keys }
    NewStatusDef($F000, $FFFF,              { define another range  }
      NewStatusKey('~F6~ Next', kbF6, cmOrderNext,
      NewStatusKey('~Shift+F6~ Prev', kbShiftF6, cmOrderPrev,
      StdStatusKeys(nil))), nil))));
    nil)                           { no more defs for this status line }
  ));                               { closing parens for New and Init }
end;

var TutorApp: TTutorApp;

begin
  TutorApp.Init;
  TutorApp.Run;
  TutorApp.Done;
end.
```

If you run the program now, you'll see it looks just the same, except for the additional status keys. The *Alt+F3* item is not highlighted, however, and clicking it has no effect because the *cmClose* command you bound to *Alt+F3* is disabled by default.

Turbo Vision automatically disables items that generate disabled commands. Once you open a window, Turbo Vision will enable *cmClose* and the *Alt+F3* status line item.

Customizing the menu bar

Just as the application's *InitStatusLine* method constructs the status line, *InitMenuBar* constructs the application's menu bar and assigns it to the global variable *MenuBar*. To customize your application's menus, you override *InitMenuBar*.

Like the status line, the menu bar is made up of a linked list of items. The items on the menu bar can be either menu commands or links to a drop-down menu (called a *submenu*). The default application has no items of any sort on its menu bar, so to create a meaningful menu bar, you need to build one from scratch.

Creating a menu bar takes three steps:

- Setting the boundaries of the menu bar
- Defining menu items
- Defining submenus

After doing those three steps, you'll see how you can cut some corners by using functions to return menu items and submenus.

Setting the boundaries

Like the status bar, the menu bar needs to set its boundaries based on the application's boundaries. But instead of the bottom line of the screen, the menu bar occupies the top line on the screen. Again, the easiest way to do this is to call the application view's *GetExtent* method:

```
procedure TTutorApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  New(MenuBar, Init(R, ...));
   ⋮
```

Defining menu items

In its simplest form, the application's menu bar looks and acts almost like the status line: a horizontal list of items the user can click to generate commands. Unlike the status line, however, the menu bar is not context sensitive. The menu bar stays the same unless the application explicitly alters it or replaces it.

Each item in a menu has of six parts:

- A text label describing the menu command
- Another text label describing any hot keys
- A keyboard scan code for the hot key
- A command
- A help context
- A pointer to the next item

When a menu item appears directly on the menu bar, however, the hot key description doesn't show up, but the hot key itself still works. The help context is useful in case you want to provide context-sensitive descriptions of menu items. Lines between menu items don't show up on the menu bar, although they do show in vertical menus.

Listing 1.6 shows an *InitMenuBar* method that declares a menu bar with items for opening and saving files.

Listing 1.6
Constructing a simple menu bar in TUTOR02B.PAS

Turbo Vision constants, including the hcXXXX and kbXXXX codes, are listed in Chapter 19, "Turbo Vision reference."

```
procedure TTutorApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(

      NewItem('~N~ew', '', kbNoKey, cmNew, hcNew,
      NewItem('~O~pen...', 'F3', kbF3, cmOpen, hcOpen,
      NewItem('~S~ave', 'F2', kbF2, cmSave, hcSave,
      NewItem('Save ~a~s...', '', kbNoKey, cmSaveAs, hcSaveAs,
      NewLine(
      NewItem('E~x~it', 'Alt+X', kbAltX, cmQuit, hcExit,
      nil))))))))));
end;
```

As with the status line you created, the only item that results in any action is the *Alt+X* item that terminates the application. The others generate commands you haven't yet defined responses to.

Menu items on the menu bar are quite limited, so they're rarely used this way. You generally group menu items into *submenus*, the vertical menu boxes that come from other menu items.

Defining submenus

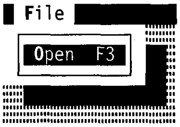
Items that create submenus have no hot keys, so they don't have as many parameters as other items. The parameters they take are

- A text label for the submenu
- A help context

- A pointer to the first item in the submenu list
- A pointer to the next item or next submenu

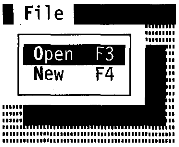
To create a simple menu bar with a single submenu called 'File' that has one item on it called 'Open', you override *InitMenuBar* like this:

Listing 1.7
Constructing a simple menu bar



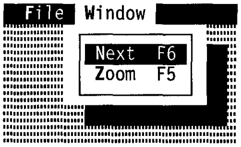
```
procedure TMyApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(    { create bar with menu }
    NewSubMenu('~F~ile', hcNoContext, NewMenu(        { define menu }
      NewItem('~O~pen', 'F3', kbF3, cmOpen, hcOpen,          { item }
      nil)),                                         { no more items }
    nil)                                          { no more submenus }
  )));                                              { end of the bar }
end;
```

To add another item to the 'File' menu, replace the **nil** that's the last parameter passed to *NewItem* with another call to *NewItem*:



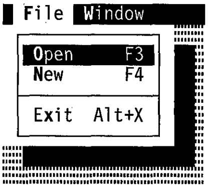
```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmOpen, hcOpen,
    NewItem('~N~ew', 'F4', kbF4, cmNew, hcNew,
    nil))),
  nil)
)));
```

To add another submenu to the menu bar, replace the **nil** that's the last parameter passed to *NewSubMenu* with another call to *NewSubMenu*:



```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmOpen, hcOpen,
    NewItem('~N~ew', 'F4', kbF4, cmNew, hcNew,
    nil))),                  { closing parens for menu selections }
  NewSubMenu('~W~indow', hcNoContext, NewMenu(
    NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
    NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
    nil)),                   { closing parens for menu selections }
  nil)))                               { closing parens for menus }
)));
```

To add a line between menu items, call the function *NewLine* between *NewItem* calls:



```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmOpen, hcOpen,
    NewItem('~N~ew', 'F4', kbF4, cmNew, hcNew,
    NewLine(
    NewItem('E~x~it', 'Alt+X', kbAltX, cmQuit, hcExit,
    nil))))),
  NewSubMenu('~W~indow', hcNoContext, NewMenu(
    NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
    NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
    nil))),
  nil))
)));
```

Using functions to return menus

Menu declarations can become complicated, especially if you have menus nested within menus. One way to tame this complexity is to use functions to return linked lists of menu items. Turbo Vision provides several such functions in the *App* unit. For example, the *StdWindowMenuItems* function returns a pointer to a list of standard window menu items:

App also defines standard File and Edit menus.

```
function StdWindowMenuItems(Next: PMenuItem): PMenuItem;
begin
  StdWindowMenuItems :=
    NewItem('~T~ile', '', kbNoKey, cmTile, hcTile,
    NewItem('C~a~scade', '', kbNoKey, cmCascade, hcCascade,
    NewItem('Cl~o~se all', '', kbNoKey, cmCloseAll, hcCloseAll,
    NewLine(
    NewItem('~S~ize/Move','Ctrl+F5', kbCtrlF5, cmResize, hcResize,
    NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcZoom,
    NewItem('~N~ext', 'F6', kbF6, cmNext, hcNext,
    NewItem('~P~revious', 'Shift+F6', kbShiftF6, cmPrev, hcPrev,
    NewItem('~C~lose', 'Alt+F3', kbAltF3, cmClose, hcClose,
    Next)))))))));
end;
```

Although *Tutorial* has a fairly complex menu bar, its declaration is much less complex because it relies on the standard menu functions, *StdFileMenuItems*, *StdEditMenuItems*, and *StdWindowMenuItems*:

Listing 1.8 TUTOR02C.PAS defines a complex menu.

```
procedure TTutorApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      StdFileMenuItems(nil)),
    NewSubMenu('~E~dit', hcNoContext, NewMenu(
      StdEditMenuItems(
      NewLine(
      NewItem('~S~how clipboard', '', kbNoKey, cmClipShow,
hcNoContext,
      nil)))),
    NewSubMenu('~O~rders', hcNoContext, NewMenu(
      NewItem('~N~ew', 'F9', kbF9, cmOrderNew, hcNoContext,
      NewItem('~S~ave', '', kbNoKey, cmOrderSave, hcNoContext,
      NewLine(
      NewItem('Next', 'PgDn', kbPgDn, cmOrderNext, hcNoContext,
      NewItem('Prev', 'PgUp', kbPgUp, cmOrderPrev, hcNoContext,
      nil)))))),
    NewSubMenu('O~p~tions', hcNoContext, NewMenu(
      NewItem('~T~oggle video', '', kbNoKey, cmOptionsVideo,
hcNoContext,
      NewItem('~S~ave desktop', '', kbNoKey, cmOptionsSave,
hcNoContext,
      NewItem('~L~oad desktop', '', kbNoKey, cmOptionsLoad,
hcNoContext,
      nil)))),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('Orders', '', kbNoKey, cmOrderWin, hcNoContext,
      NewItem('Stock items', '', kbNoKey, cmStockWin, hcNoContext,
      NewItem('Suppliers', '', kbNoKey, cmSupplierWin, hcNoContext,
      NewLine(
      StdWindowMenuItems(nil)))))),
    NewSubMenu('~H~elp', hcNoContext, NewMenu(
      NewItem('~A~bout...', '', kbNoKey, cmAbout, hcNoContext,
      nil)), nil)))))))));
end;
```

What you've accomplished

At this point it might not seem like you've done much. Although you've defined a number of commands and set up ways to generate the commands through menu items and status keys, most of the commands are either disabled or just don't do anything yet. If you're disappointed — don't be! You've accomplished a lot.

Separating events from responses

Event-driven programming is explained in Chapter 9, "Event-driven programming."

In traditional, non-event-driven programming, if you wanted to respond to the commands you've defined, you'd have to go back into the code you'd just written and indicate what procedure should be called when the user chooses each menu item, then do the same for each status key. But you don't have to do that in Turbo Vision. Each of those menu items and status keys generates a command. You just have to write a few routines that respond to those commands—without touching the menu or status line code.

The Turbo Vision application framework takes you a step beyond traditional modular programming. Not only do you code in functional, reusable blocks, but those blocks are more independent and interchangeable.

Programming flexibly

Tutorial now has three ways to generate the command *cmNewWin*: a status key, a menu item, and a hot key. In the next step, you'll see how easy it is to tell your application to open a window when that command shows up. But the most important thing is that the application doesn't care *how* the command was generated, and neither will the window.

Later on, if you decide you want to change the binding of the command—move the menu selection or remap the hot keys, for example—you don't have to worry or even *think* about how it affects the response code. That's the biggest benefit of event-driven programming. It separates your user interface design from your program workings and lets different parts of your program function independently.

CHAPTER

2

Responding to commands

Now that your program generates commands, you need to add the ability to respond to those commands. In this, chapter you'll learn about how commands work, then add code to the *Tutorial* program to

- Change the video mode
- Display an "About" box
- Enable and disable commands

What are commands?

Throughout the last step, you set up ways for *Tutorial* to generate commands, but we really touched only lightly on what commands really are. In this step, you'll learn a lot more about commands and then write some code to respond to some of the commands you generated in Step 2.

Understanding commands

So far, you've learned that commands are integer constants, usually represented by identifiers beginning with *cm* (short for "command"). You've seen several of the standard Turbo Vision commands, such as *cmQuit* and *cmNext*, and defined your own commands for *Tutorial*, such as *cmOpenWindow*.You've also gotten an indication that these commands are tied to certain actions, such as pressing a hot key or choosing a menu item. In this section, you'll see just what we mean by "generating" a command, how commands relate to events, and what it means to "respond" to a command.

What are events?

Already in this manual, we've referred several times to "event-driven programming" and some of its benefits. You've probably understood that it involves writing your program so that it responds to outside occurrences such as mouse clicks or keystrokes. Any such occurrence that your program needs to take note of is called an *event*.

Traditionally, we've all written programs that perform some action, wait for input from a user, then act on that input. Central to this model of programming is the input loop, usually followed by a branching statement. A simplified version might look like this:

This is a simulated control loop for a typical Pascal program.

```
repeat
  GetCommand;
until Command <> 0;
case Command of
  1: DoSomething;
  2: DoSomethingElse;
  else EtCetera;
end;
```

The procedures *DoSomething* and *DoSomethingElse* either perform some action and come back to the loop or perhaps contain their own input/action loops. The main drawback to this kind of programming is that your code has to be tightly coupled—different parts of the program have to be aware of what other parts are available or not available at any given time.

Event-driven programming is often called "modeless programming" because the user can access any part of the program, not just the current "mode."

In event-driven programming, instead of having numerous input loops, the whole program has *one*, called the *event loop*, which interacts with all interfaces to the outside world and channels information about what went on to the appropriate part of the program. This structure allows great flexibility. For example, if you pull down a menu, then decide that you want to click on the status line, you don't have to first close the menu and get out of "menu mode" before moving on to something else. The event loop recognizes that your click on the status line is meant for the status line, and tells the status line object to respond to a mouse click. Closing the open menu happens automatically when you move the input focus to the status line.

Responding to events

For the moment we won't dwell on how the event loop decides where to send events. The event-routing mechanism is described in detail in Chapter 9, "Event-driven programming." The important thing to know is that the event loop packages information about the event into a variant record of type *TEvent* and sends it to the object that needs to know about the event.

All visible Turbo Vision objects have virtual methods called *event handlers*. These methods are always called *HandleEvent*, and always take a single **var** parameter of type *TEvent*. Thus, when the application's event loop detects an event, it figures out which object should handle the event, creates an *event record*, and passes that record to the object's *HandleEvent* method. The object then examines the event record and decides what to do with the event, if anything.

For example, if the event was a mouse click, the event record contains information as to where on the screen the click took place, which button was clicked, and whether it was a double click. Pressing a key on the keyboard sends an event that includes the scan code or character code of the key pressed.

In this step, you'll learn how to handle command events.

Handling command events

Every event record has a *Word*-type field called *What* that the event loop fills with a constant indicating the type of event described in the record. One of those constants is *evCommand*, indicating a command event. If the event is a command event, the record also contains a field called *Command*, which holds the command constant bound to the menu item, status key, or hot key that generated the command event.

For example, if you click the *Alt+F3* status line item (or press *Alt+F3*), the event loop generates a command event, setting the event record's *What* field to *evCommand*, and the *Command* field to *cmClose*. It then routes the event record to the active window. Window objects know that when they receive a *cmClose* command, they are to close, specifically by calling a method called *Close*:

This is a greatly simplified portion of TWindow.HandleEvent.

```
if Event.What = evCommand then
  case Event.Command of
    cmClose:
      begin
        Close;
        ClearEvent(Event);
      end;
  :
```

Notice that after responding to the event, the object *clears* the event by calling the method *ClearEvent*. This indicates to other objects that the event has been handled and that no further processing is necessary.

Step 3: Responding to commands

```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

Now that you've seen the process in theory, it's time to actually respond to some command events. In this step you'll respond to commands to

- Change the video mode
- Display an About box

In addition, you'll learn to enable and disable commands.

Changing the video mode

If you pull down *Tutorial*'s Options menu, you'll notice that the first item is called Toggle Video Mode. The *InitMenuBar* method binds that menu item to the command *cmOptionsVideo*. To define a response to that command, you need to give *TTutorApp* a *HandleEvent* method that knows how to respond to *cmOptionsVideo*. Listing 2.1 shows the event handler.

Listing 2.1 Redefining the application's event handler, from TUTOR03A.PAS

```
procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);      { call the inherited method first }
  if Event.What = evCommand then   { if unhandled, check for commands }
  case Event.Command of                   { check for known commands }
    cmOptionsVideo:
      begin
        SetScreenMode(ScreenMode xor smFont8x8);    { toggle mode bit }
        ClearEvent(Event);                  { mark the event as handled }
      end;
  end;
end;
```

Manipulating bit flags is explained in Chapter 7.

If you run the program now, choosing the Toggle item on the Options menu toggles the program's video mode between standard 25-line mode and 43- or 50-line EGA/VGA mode by toggling a bit called *smFont8x8* in the *ScreenMode* variable. On monochrome or CGA systems, this command will have no effect.

Calling inherited methods

Notice that the first thing the new *HandleEvent* does is call the *HandleEvent* inherited from *TApplication*. As a rule, when you redefine a virtual method in Turbo Vision, you want your new method to call its inherited method at some point.

Calling the inherited method essentially tells the new method to act like its ancestor type. In Listing 2.1, *TTutorApp* calls its inherited *HandleEvent* method and then defines some more behavior. That's like saying, "*TTutorApp* should handle events like *TApplication* and also handle some others."

You can also define *HandleEvent* methods that remove some of their inherited behavior by checking for certain events *before* calling the inherited method, then clearing the event so the inherited method doesn't get a chance to handle it. That's like saying, "This object should handle events like its ancestor, *except* for these certain events."

In general, if you want to eliminate some inherited behavior, you either trap that behavior before calling the inherited method, or don't call the inherited method at all. If you want to add to the inherited behavior, you call the inherited method first, and then define the desired additional actions.

Displaying an About box

Programs often have a menu option that brings up a box that displays information about the program. This box is usually called the "About box." Turbo Vision provides a utility called a message box that you can use to show messages to users. In the next section, you'll use a message box to create a simple About box. Later, you'll create a somewhat fancier About box on your own.

Tutorial's Help menu has an "About..." item. The "..." after the name of the item indicates that the item brings up a dialog box. That menu item is bound to the command *cmAbout*, so to display your about box, you need to tell *TTutorApp*'s *HandleEvent* method to respond to *cmAbout*. This time, instead of actually displaying the About box from within *HandleEvent*, you'll call another method, called *DoAboutBox*, which actually displays the About box. Listing 2.2 shows the necessary code changes.

Listing 2.2
Adding an About box, from TUTOR03B.PAS

```
procedure TTutorApp.DoAboutBox;
begin                           { #3 centers a line; #13 is a line break }
  MessageBox(#3'Turbo Vision Tutorial Application'#13 +
    #3'Copyright 1992'#13#3'Borland International',
    nil, mfInformation or mfOKButton);       { specify title & button }
end;

procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);     { call the inherited method first }
  if Event.What = evCommand then { if unhandled, check for commands }
  case Event.Command of                   { check for known commands }
    cmOptionsVideo:
      begin
        SetScreenMode(ScreenMode xor smFont8x8);   { toggle mode bit }
        ClearEvent(Event);               { mark the event as handled }
      end;
    cmAbout:
      begin
        DoAboutBox;                           { call about box method }
        ClearEvent(Event);               { mark the event as handled }
      end;
  end;
end;
```

Now when you run the program, you can bring up the About box from the menu, and close it by clicking OK.

Using message boxes

The *MessageBox* function gives you an easy way to inform or warn the user of a limited amount of information, and also enables you to get limited feedback based on the button the user presses.

MessageBox is in the MsgBox unit.

MessageBox takes three parameters. The first is the message string to display. The message box automatically wraps the text if it exceeds one line, but you can force a line break (as *DoAboutBox* does) by putting a carriage return character (#13) in the string. If a line begins with #3, the message box centers that line instead of left-aligning it.

The second parameter is a pointer to an array or record of data items to substitute into the message string, if any. The message string can contain formatting characters that get replaced by the data items in the second parameter. For simple text messages, this second parameter is **nil**.

Combining message box flags

The last parameter to *MessageBox* is a set of flag bits that indicate the title to put on the message box and the buttons to place under the text. The easiest way to set these bits is to use the predefined message flag constants, which have identifiers starting with *mf*.

Use the **or** operator to combine one of *mfInformation, mfWarning, mfConfirmation,* or *mfError* with one of *mfOKButton, mfOKCancel,* or *mfYesNoCancel.*

Try substituting different combinations of *mfXXXX* constants into the About box in *Tutorial* to see their different effects.

Reading message box return values

When the user clicks one of the buttons in a message box, the box closes, and *MessageBox* returns the value of the command bound to the clicked button. That value will always be *cmOK, cmCancel, cmYes,* or *cmNo,* so you can use message boxes to ask simple questions of the user and get simple yes-no or OK-not OK answers.

Extended syntax {$X+} is the default setting.

In the case of an About box, you don't care *how* the user closes the box—that's not important information, so you can ignore the value returned by *MessageBox*. Using Turbo Pascal's extended syntax, you can call a function as if it were a procedure, essentially throwing away the return value.

Enabling and disabling commands

Now that you've defined responses to some of the menu commands, it's a good time to learn more about Turbo Vision commands in general. You've already seen that Turbo Vision automatically disables some commands, such as disabling *cmClose* where there's nothing to close. You'll also notice that on the Window menu, the items for Next, Previous, Resize and Zoom are disabled because there's nothing for them to act on. In the next step, you'll actually add windows to the desktop, and you'll see those commands become enabled.

Of course Turbo Vision can't automatically handle commands that you define, and there might be times when you want to, for example, disable a standard command that would otherwise be available. In this section you'll learn how to enable and disable single commands and groups of commands.

Which commands can I disable?

You've already seen that commands are *Word*-type constants, but you can only disable commands in the range 0..255 because command disabling operates on *sets* of commands, and Turbo Pascal sets contain only elements in that range. When you define commands, then, consider whether you'll ever need to disable them before you assign a value. Since you can only disable a limited number, you need to assign values accordingly.

Keep in mind also that Turbo Vision reserves some commands for its standard commands, including the range 0..99 of commands you can disable and 256..999 of commands that you can't. So you can define commands 100..255 that you can disable and 1,000..65,535 that you can't.

Disabling commands

Turbo Vision provides a set type for holding sets of commands, called *TCommandSet*. Every visual Turbo Vision object has a *DisableCommands* method that takes a *TCommandSet* as its one parameter and disables the commands in that set.

When you disable a command, it is disabled throughout the application, because you don't want some other part of the program generating a command you don't expect to have to handle. All menu items, status keys, and buttons that generate a disabled command are themselves disabled. You can click them, but they have no effect, so they show up dimmed.

For example, none of the first commands on the Window menu do anything yet, so you might want to have the program disable them initially, only enabling them when there's actually something for them to do. A good place to do this is in the application object's constructor:

```
constructor TTutorApp.Init;
begin
  inherited Init;                    { do standard application setup }
  DisableCommands([cmOrderWin, cmStockWin, cmSupplierWin]);
end;
```

Enabling commands

Just as each visible Turbo Vision object can disable commands, it has a corresponding *EnableCommands* method. In Step 11, you'll use *EnableCommands* to reenable the Orders menu commands.

CHAPTER

3

Adding windows

So far you've customized your application's menu bar and status line and seen how to respond to their commands. In this chapter, you'll start adding windows to the desktop and managing them.

In this chapter, you'll do the following steps:

- Add a simple window
- Tile and cascade windows
- Add file editor windows
- Use a standard file open dialog box
- Add a clipboard window

Step 4: Adding a window

```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

One of the great benefits of Turbo Vision is that it makes it easy to create and manage multiple, overlapping, resizeable windows. The key to managing windows is the desktop, which knows how to keep track of all the windows you give it and which can handle such operations as cascading, tiling, and cycling through the available windows.

The desktop is one example of a *group* in Turbo Vision; that is, a visible object that holds and manages other visible items. You've already used one group—the application itself, which handles the menu bar, the status line, and the desktop. As you proceed, you'll find that windows and dialog boxes are also groups.

Like the menu bar and the status line, the desktop object is constructed in a virtual method of the application object called *InitDesktop* and assigned to a global variable, *Desktop*. By default, *Desktop* covers all of the application screen that isn't covered by the menu bar and status line.

Adding a window to the desktop in an application takes three steps:

- Assigning the boundaries for the window
- Constructing the window object
- Inserting the window into the desktop

Adding a simple window

As a first step, you can add a plain window to the desktop in response to the New item on the File menu. That item generates a *cmNew* command, so you need to define a response to that command in the application's *HandleEvent* method. In this case, just call a method called *NewWindow*, which you'll modify a few times before you're through:

```
procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNew:
        begin
          NewWindow;
          ClearEvent(Event);
        end;
  ⋮
end;

procedure TTutorApp.NewWindow;
var
  R: TRect;
  TheWindow: PWindow;
begin
  R.Assign(0, 0, 60, 20);          { assign boundaries for the window }
  The Window := New(PWindow,
    Init(R, 'A window', wnNoNumber));                { construct window }
  Desktop^.Insert(TheWindow);            { insert window into desktop }
end;
```

The changes to *HandleEvent* should seem familiar by now. *NewWindow*, though, contains some new things.

Assigning the window boundaries

You've seen *TRect*-type variables before. However, for the menu bar and status line, you set their sizes based on the size of the application (using the *GetExtent* method). In *NewWindow*, you assign the new window an absolute set of coordinates using *Assign*.

Constructing the window object

The next statement constructs a dynamic instance of the generic window object type, *TWindow*. Constructing a window requires three parameters: the boundaries of the window, a string containing the title for the window, and a number for the window. In this case, your window has the title 'A window' and no number, because you've passed the constant *wnNoNumber*.

If you assign a number to a window, the user can activate the window on the desktop by holding down the *Alt* key and typing the window's number.

Inserting the window

Insert is a method common to all Turbo Vision groups, and it's the way a group gets control of the objects within it. When you insert *TheWindow* into the desktop, you're telling the desktop that it is supposed to manage *TheWindow*.

If you run the program now and choose New from the File menu, an empty blue window with the title 'A window' appears on the desktop. If you choose New again, another, identical window appears in the same place, because *NewWindow* assigns exact coordinates for the window. Using your mouse, you can select different windows.

☞ The menu items under Window and the hot keys bound in the status line now operate on the windows. Note that the menu and status line items haven't changed. They don't know anything about your windows. They just issue commands which the windows and desktop already know how to respond to.

Inserting more safely

The application object has several methods you can use to both simplify some common operations and make those operations "safer." By safer, we mean that it's less likely to cause a problem, such as running out of memory. Your application object inherits a method called *InsertWindow* that takes care of the *Desktop^.Insert()* part of inserting a window. In addition, *InsertWindow* makes sure the window you're inserting was constructed successfully and that you haven't run out of memory.

Using *InsertWindow*, the *NewWindow* method looks like this:

Listing 3.1
Inserting a window safely, in TUTOR04A.PAS

```
procedure TTutorApp.NewWindow;
var
  R: TRect;
  TheWindow: PWindow;
begin
  R.Assign(0, 0, 60, 20);
  New(TheWindow, Init(R, 'A window', wnNoNumber));
  InsertWindow(TheWindow);              { insert window into desktop }
end;
```

It's a good idea to use *InsertWindow* to insert windows into the desktop unless you have a good reason to circumvent the safety precautions it takes.

Tiling and cascading

One thing the desktop knows how to do is tile and cascade windows. The application just needs to tell the desktop when to do that. The default event handler in *TApplication* responds to the standard Window menu commands *cmTile* and *cmCascade*, calling the *TApplication* methods *Tile* and *Cascade*, respectively.

Such inherited standard behavior is one important reason to remember to call inherited *HandleEvent* methods.

Adding an editor window

Now that you've seen how windows behave in general, you might want to include a more useful window, such as a file editor window. The *Editors* unit in Turbo Vision defines just such a window, so you can change *NewWindow* to insert an editor window instead of a generic window.

Adding an editor window requires only two additional steps and one changed step:

- Defining a file edit buffer
- Setting up editor dialog boxes
- Constructing a file editor window

Defining the file editor buffer

If you want your application to use any file editors (including the clipboard), you need to initialize the *MaxHeapSize* variable from the *Memory* unit, and you have to do it *before* constructing the application object. *MaxHeapSize* sets aside a part of memory above the regular heap to be used for file-editor buffers.

MaxHeapSize and file editor buffers are explained fully in Chapter 15.

MaxHeapSize sets the number of 16-byte paragraphs the application can use for its regular heap, leaving the rest of free memory for file-editor buffers. The changes in *Tutorial* shown in Listing 3.2 include setting *MaxHeapSize* to 8192, meaning it sets aside 128K for the application heap, which is much more than enough for this simple program.

Setting up editor dialog boxes

The *Editors* unit has a procedural variable called *EditorDialog* that handles all the dialog boxes for all editor objects in your program. By default, *EditorDialog* doesn't really do anything, so before you use editor objects, you should assign a function to *EditorDialog* that shows useful dialog boxes and returns the proper values.

Turbo Vision provides such a function that you can use, called *StdEditorDialog*. If you want fancier dialog boxes, you can define your own, but *StdEditorDialog* is a good starting point. To use the standard editor dialog boxes, just put the statement

```
EditorDialog := StdEditorDialog;
```

in the application's constructor. Listing 3.2 shows such an addition to *TTutorApp.Init*.

Constructing the editor window

The constructor for an edit window takes exactly the same parameters as the generic window you already constructed. The main things you have to change are the type of the window to construct (*PEditWindow* instead of *PWindow*) and the title passed for the window.

Constructing a file editor window with an empty title string produces a window with the title "Untitled," which indicates that whatever you type into the editor has not yet been assigned to a specific file. Since you're creating this editor in response to the File | New command, it's appropriate to create an untitled editor window, as shown in Listing 3.2.

Listing 3.2
Inserting a file editor window, completing TUTOR04B.PAS

```
constructor TTutorApp.Init;
begin
  MaxHeapSize := 8192;     { set up file edit buffer area above heap }
  EditorDialog := StdEditorDialog;     { use standard editor dialogs }
  inherited Init;
  DisableCommands([cmOrderWin, cmStockWin, cmSupplierWin]);
end;

procedure TTutorApp.NewWindow;
var
  R: TRect;
  TheWindow: PEditWindow;              { note the change of type here }
begin
  R.Assign(0, 0, 60, 20);
  New(TheWindow, Init(R, '', wnNoNumber));    { construct edit window }
  InsertWindow(TheWindow);
end;
```

Using standard dialog boxes

Having a file editor that creates new files is useful, but you need to be able to edit existing files, too. To do that, you need to tell the file editor which file you want to edit. Although you could use a simple prompt that reads a file name from the user, a much better approach is to show the user what files are available and allow navigation to different directories. Turbo Vision's standard dialogs unit, *StdDlgs*, provides a dialog box object that does just that.

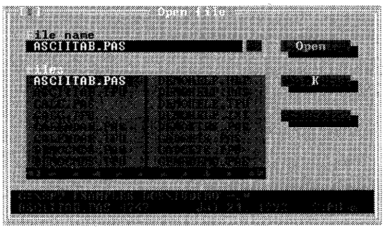


Figure 3.1
The File Open dialog box from the StdDlgs unit

Once you've gotten the file dialog box object, you need to *execute* it. Executing is a lot like inserting, as you did with the editor windows, but it not only inserts the dialog box into the desktop, it also makes the dialog box *modal*. Modal means that the dialog box is the only active part of the application—you can click other parts of the application, such as the menu bar, but they don't react. Once you make a window or dialog box modal, you can't interact

The status line is always available, no matter what window or dialog box is modal.

with any part of the application outside that window or dialog box until you close it or execute another dialog box.

To edit existing files, you need to do the following:

- Construct a file dialog box
- Execute the file dialog box to prompt the user for a file name
- Construct an editor window for that file

Constructing a file dialog box

The constructor for a file dialog box object takes five parameters: three strings, a word containing option flags, and the number of a history list. The strings passed are the initial file-name mask, such as '*.*', the title of the dialog box, and the label for the input line where the user will type the file name, in that order.

The options flags work much like those you used for the message box earlier. They indicate what buttons appear in the box in addition to the Cancel button that's always included. Depending on whether you're using the dialog box to choose a file to open or a file to save into, you use different combinations of constants starting with *fd*.

For now, just pass any nonzero number as the number of the history list. You'll see how easy it is to keep track of the file names you've opened.

Executing the dialog box

ExecuteDialog works much like *InsertWindow*. It checks to make sure you've passed it a valid dialog box object and that it has enough memory to complete the action. It then inserts the dialog box into the desktop and makes it modal.

Controls and their initialization are explained in Chapter 12, "Control objects."

The second parameter passed to *ExecuteDialog* points to a data record the dialog box can use for initialization when it becomes modal. Every dialog box requires different data. For example, a file dialog box takes a string to contain the file name. Passing **nil** in this parameter indicates that you don't want the dialog box to initialize its controls and that you don't want to read the control values when it's done.

ExecuteDialog is a function that, like *MessageBox*, returns the value of the command that closed the box. So if the user presses the Cancel button, *ExecuteDialog* returns *cmCancel*; choosing OK causes *ExecuteDialog* to return *cmOK*; and so on. Your program can therefore tell if the user accepted the dialog box or canceled it, reading the data from the controls only if the dialog box wasn't canceled. Notice that *OpenWindow* opens an edit window only if the value returned by *ExecuteDialog* is not equal to *cmCancel*.

Constructing the file editor window

Constructing the file editor window looks very familiar. It's just what you did in *NewWindow*, except you pass the name of the file instead of an empty string when you construct the editor window.

In response to the *cmOpen* command from the Open item on the File menu, your application should call a new method called *OpenWindow*, as shown in Listing 3.3.

Listing 3.3
Opening a file to edit, making TUTOR04C.PAS

```
procedure TTutorApp.OpenWindow;
var
  R: TRect;                                 { boundaries for edit window }
  FileDialog: PFileDialog;                   { file selection dialog box }
  TheFile: FNameStr;                           { string for the file name }
const
  FDOptions: Word = fdOKButton or fdOpenButton;          { dialog options }
begin
  TheFile := '*.*';                        { initial mask for file names }
  New(FileDialog, Init(TheFile, 'Open file', '~F~ile name',
    FDOptions, 1));
  if ExecuteDialog(FileDialog, @TheFile) <> cmCancel then
  begin
    R.Assign(0, 0, 75, 20);
    InsertWindow(New(PEditWindow, Init(R, TheFile, wnNoNumber)));
  end;
end;
```

Step 5: Adding a clipboard window

Step 1: Basic App
Step 2: Menu/Status
Step 3: Commands
Step 4: Windows
Step 5: Clipboard
Step 6: Streams
Step 7: Resources
Step 8: Data entry
Step 9: Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view

Editors and editor windows are much more useful if you can cut and paste text to and from a clipboard, meaning you can exchange text between windows, rearrange text, and so on. Turbo Vision's editors fully support a clipboard feature.

The clipboard is just an editor object that's always present in the application. If you only want the clipboard to work in the background, you don't even need to give the clipboard a window. In *Tutorial*, however, you'll create a clipboard window so you can display the clipboard and its contents.

Adding a clipboard window takes two steps:

- Constructing an editor window
- Making the editor the clipboard

Constructing an editor window

Constructing a window for the clipboard is just like constructing a new file editor window. You assign the boundaries for the window, construct an unnamed window with those boundaries, and insert the window into the desktop. In the case of the clipboard window, however, you don't want to *see* the window unless you specifically ask for it. Before inserting the window, therefore, call its *Hide* method. *Hide* makes an object invisible until you call its *Show* method.

You should hide the window before inserting it. Otherwise, the window will flash on the screen and disappear, which is annoying to users. Since the validity check is usually handled by *InsertWindow*, you need a separate check. *TApplication*'s *ValidView* function is the same validity test used by *InsertWindow* and *ExecuteDialog*. *ValidView* returns **nil** if the view is invalid, or a pointer to the view if it's valid. Listing 3.4 shows the use of *ValidView*. Once you've validated the view, you can hide and insert it.

Assigning the clipboard editor

The *Editors* unit defines a variable called *Clipboard* which other editor objects use for cutting and pasting operations. If *Clipboard* is **nil**, those operations have no effect. Since you're constructing a clipboard window, you need to set *Clipboard* to point to the editor in your clipboard window, as shown in Listing 3.4.

The only other thing you have to worry about with the clipboard is that you disable its undo capability. Turbo Vision editor objects can usually undo the most recent editing changes, but the clipboard editor can't support that. All that's required to disable undo is to set the editor's *CanUndo* field to *False*.

Listing 3.4 Creating a clipboard window

```
constructor TTutorApp.Init;
var R: TRect;
begin
  MaxHeapSize := 8192;
  inherited Init;
  Desktop^.GetExtent(R);                 { get boundaries for window }
  ClipboardWindow := New(PEditWindow, Init(R, '', wnNoNumber));
```

ClipboardWindow is a new field in the TTutorApp object.

```
    if ValidView(ClipboardWindow) <> nil then    { make sure it worked }
    begin
      ClipboardWindow^.Hide;                        { hide clipboard window }
      InsertWindow(ClipboardWindow);  { insert hidden clipboard window }
      Clipboard := ClipboardWindow^.Editor;    { make editor clipboard }
      Clipboard^.CanUndo := False;           { can't undo in clipboard }
    end;
  end;
```

Showing the clipboard window

Now that you have a clipboard, you can cut and paste text between windows at will. But what if you want to see or edit what's in the clipboard? The clipboard window is hidden, so you need a way to show it.

The standard Edit menu includes an item labeled 'Show clipboard' that generates the *cmClipShow* command. In response to that command, have your application show the clipboard window, as shown in Listing 3.5.

Listing 3.5
Showing the clipboard window

```
procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmClipShow:
        begin
          ClipBoardWindow^.Show;
          ClearEvent(Event);
        end;
      ⋮
```

If you have other windows open, you'll notice there's a problem with this approach. The other windows are in front of the clipboard window. After all, it was the first window inserted into the desktop; all the others were opened on top of the hidden clipboard window. This layering of visible objects is known as *Z-order*, and it's what allows groups to know which objects show up in front of others, which window to activate when you use the Window | Next command, and so on.

Z-order is explained fully in Chapter 8, "Views."

The solution is to make sure the clipboard window is in front of all the other editors before you show it. All visible Turbo Vision objects inherit a method called *Select*, which you can call to make the given object the *selected* subview, or frontmost, in its group. If you change the response to *cmClipShow* to include *Select*, it looks like Listing 3.6.

Listing 3.6
Showing the clipboard window in front, which completes TUTOR05.PAS

```
procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmClipShow:
        with ClipBoardWindow^ do
        begin
          Select;
          Show;
          ClearEvent(Event);
        end;
         :
```

CHAPTER

4

Using streams and resources

Now that you've made *Tutorial* do actual work, the next logical step is to be able to save that work. Using Turbo Vision's *streams* to store objects to your disk, you'll be able to save the state of your desktop and restore it at a later time. You'll also see how you can use an extension of streams, called *resources*, to simplify your use of menus and status lines.

The two steps in this chapter will walk you through

- Saving objects to the disk
- Restoring objects from the disk
- Defining objects in resources

Step 6: Saving and restoring the desktop

```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

In order to save your desktop in a file, you need a mechanism for storing numerous object types. After all, you need to store the desktop object, various window objects, editor objects, and so on, and then be able to read them back in.

Turbo Vision uses streams to store objects, either to a disk file or to EMS memory. Rather than treating the file like a normal Pascal file, streams represent a stream of bytes, written or read sequentially. Writing an object to a stream involves telling the stream what kind of object it's getting, then sending the information that describes the object. When reading the object back in from the stream, you first get back what kind of object it is, so you know how to interpret subsequent bytes.

Using streams is a lot easier than it sounds, as you'll see in this step. Saving and loading the desktop takes three steps:

- Registering objects with streams
- Saving the desktop
- Loading the desktop

Registering with streams

In order to use an object type with streams, you have to *register* the type with Turbo Vision's streams. Registering is a way of telling the stream what kind of objects they'll have to deal with, how to identify them, and how to read or write the object's data. Registration is best handled in the application's constructor, ensuring that stream access occurs only after the objects are registered.

Turbo Vision's units all have procedures that register their objects for stream usage. For example, to register the objects in the *Editors* unit, you call *RegisterEditors*. The desktop object itself is in the *App* unit, and the windows and their components are in the *Views* unit, so you'll have to call *RegisterApp* and *RegisterViews*, too. Listing 4.1 shows the revised application constructor:

Listing 4.1 Registering standard objects with streams

```
constructor TTutorApp.Init;
var R: TRect;
begin
  MaxHeapSize := 8192;
  EditorDialog := StdEditorDialog;
  StreamError := @TutorStreamError;
  RegisterObjects;
  RegisterViews;
  RegisterEditors;
  RegisterApp;
  inherited Init;
  Desktop^.GetExtent(R);
  ClipBoardWindow := New(PEditWindow, Init(R, '', 0));
  if ValidView(ClipboardWindow) <> nil then
  begin
    ClipboardWindow^.SetState(sfVisible, False);
    InsertWindow(ClipboardWindow);
    Clipboard := ClipboardWindow^.Editor;
    Clipboard^.CanUndo := False;
  end;
end;
```

That's all there is to it. *Tutorial* can now use object types from the *Objects, Views, App,* and *Editors* units with streams.

Trapping stream errors

In addition to registering objects, Listing 4.1 adds a safety feature to the application. The *StreamError* variable points to a procedure that's called by any Turbo Vision stream when it encounters an error. By default, *StreamError* is **nil**, so it's never called. *Tutor06a* assigns it to point to a procedure called *TutorStreamError*, which reports errors and halts, as shown in Listing 4.2.

Listing 4.2
Defining a simple stream error procedure

```
procedure TutorStreamError(var S: TStream); far;
var ErrorMessage: String;
begin
  case S.Status of
    stError: ErrorMessage := 'Stream access error';
    stInitError: ErrorMessage := 'Cannot initialize stream';
    stReadError: ErrorMessage := 'Read beyond end of stream';
    stWriteError: ErrorMessage := 'Cannot expand stream';
    stGetError: ErrorMessage := 'Unregistered type read from stream';
    stPutError: ErrorMessage :=
      'Unregistered type written to stream';
  end;
  ClearScreen;                                  { clear the display }
  PrintStr('Error: ' + ErrorMessage);          { show error message }
  Halt(Abs(S.Status));                        { halt with errorlevel }
end;
```

TutorStreamError is not a very elegant error handler, but it reports any errors encountered with streams.

Saving the desktop

As you saw in Step 4, the desktop object is a group, meaning it's a view that manages other views. Part of managing those views is making sure they are written to a stream when the group object is written. That ability is built into *TGroup*, so when you write any group to a stream, make sure it calls the stream writing method it inherits.

To save the desktop and all the windows it contains, you need to do three things:

- Open the stream
- Store the desktop object
- Close the stream

In this case, since you're writing to a disk file, you could use the *TDosStream* type, but you'll get somewhat better performance using *TBufStream*, a buffered version of *TDosStream*. The DOS stream associates a Turbo Vision stream with a DOS file, and the buffered DOS stream lets you specify a buffer size for reading and writing from the file.

Writing the objects to a stream

The Store Desktop item on the Options menu in *Tutorial* generates the command *cmOptionsSave*, so you should extend the application's event handler to respond to *cmOptionsSave* by calling a new method called *SaveDesktop*:

Listing 4.3
Saving the desktop

```
procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmOptionsSave:
        begin
          SaveDesktop;
          ClearEvent(Event);
        end;
         ⋮
end;

procedure TTutorApp.SaveDesktop;
var DesktopFile: TBufStream;
begin
  DesktopFile.Init('DESKTOP.TUT', stCreate, 1024);     { open stream }
  DesktopFile.Put(Desktop);                          { store desktop }
  DesktopFile.Done;                               { close the stream }
end;
```

SaveDesktop is extremely simple. It initializes the buffered stream, associating it with the file DESKTOP.TUT and giving it a 1K buffer. Using the *stCreate* constant tells the stream to create a new file, even if one already exists, much like Pascal's *Rewrite* procedure.

Writing the entire desktop, including any windows on the desktop, is accomplished by calling a single method, the stream's *Put* method. Calling *Put* writes information about the object to the stream, which then calls the *Store* method of the object passed as the parameter.

Any Turbo Vision object that will be used with streams needs to have a *Store* method (and, as you'll see in the next part of this step, a corresponding *Load* constructor) that writes its information to the stream. Since *TDesktop* descends from *TGroup*, it writes all the subviews inserted into it, including its background and windows.

Don't call *Store* directly. You tell the stream to put the object, and the stream calls *Store* at the appropriate time.

Calling the stream's *Done* method flushes the stream's buffer and closes the associated file.

Preserving the clipboard

Right now you're probably thinking, "It can't be that simple." And you're both right and wrong. It *is* that simple, and the method in Listing 4.3 will save the desktop in such a way that you can retrieve it again. But you're also right that there are other things to consider, specifically the clipboard.

As you'll recall, when you created the clipboard window, you set the global variable *Clipboard* to point to the clipboard window's editor. Unfortunately, if you write the clipboard window to the disk and then read it back, that pointer will no longer be valid, and there will be no way to set it to point to the restored window's editor without a great deal of effort.

The simple solution is to exclude the clipboard from loading and saving operations. After all, there's rarely anything in the clipboard you want to save across sessions. Listing 4.4 shows a safer way to save the desktop.

Listing 4.4
Saving the desktop without the clipboard, making TUTOR06A.PAS

```
procedure TTutorApp.SaveDesktop;
var DesktopFile: TBufStream;
begin
  Desktop^.Delete(ClipboardWindow); { remove clipbaord from desktop }
  DesktopFile.Init('DESKTOP.TUT', stCreate, 1024);    { open stream }
  DesktopFile.Put(Desktop);                         { store desktop }
  DesktopFile.Done;                              { close the stream }
  InsertWindow(ClipboardWindow);         { restore clipboard window }
end;
```

Excluding the clipboard from the desktop save essentially makes the clipboard part of the application, rather than part of the desktop, although the desktop does get to manage it. In the next section, you'll have to make arrangements to handle this as well.

Restoring the desktop

Restoring the desktop is essentially the inverse of storing it, but you need to take some precautions. Rather than just reading a desktop object and calling it the desktop, you should check to make sure it's a working desktop object. In this step, you'll

- Load the object from a stream
- Ensure that it's a valid object
- Replace the existing desktop

Because loading a new desktop affects the way your program works, you need to be more cautious than you were when you simply wrote the desktop object to the stream. Ensuring the validity of the loaded object and careful replacement of the existing desktop are important steps.

Loading the desktop object

The steps for loading the object correspond exactly to those for saving it:

- Open the stream
- Read the object
- Close the stream

Listing 4.5 shows the code to retrieve the desktop object.

Listing 4.5
Reading the desktop from a stream

```
DesktopFile.Init('DESKTOP.TUT', stOpenRead, 1024);      { open stream }
TempDesktop := PDesktop(DesktopFile.Get);            { get the desktop }
DesktopFile.Done;                                   { close the stream }
```

There are two interesting points to note about this fragment of code. First, it assigns the loaded desktop to a temporary variable, rather than to *Desktop* itself, since you don't want to lose the old *Desktop* should this newly-read object prove invalid, and since you need to dispose of the old *Desktop* anyway to free the memory it used. Second, you'll notice the use of the *Get* method. *Get* is the counterpart of *Put*, which was the method you used to write the object to the stream.

Get reads the information that *Put* wrote, so it knows what kind of object it's loading, and calls the object's *Load* constructor to read the object from the stream. *Load* is a constructor, just like *Init*, but instead of constructing the object based on the parameters passed to it, *Load* constructs the object and.reads its values from the stream passed as its parameter.

Notice that *Get* returns a pointer of type *PObject*, you you have to typecast the result into the proper type for your object. Since you're reading a desktop object, you typecast the pointer returned by *Get* into a *PDesktop*.

Validating the object

Once you've obtained a desktop object from the stream, you need to ensure that it's a valid object before using it to replace the existing desktop. To do this, you call one of the application's methods, *ValidView*.

ValidView returns a pointer to the object if the object is valid, or **nil** if it's invalid, after performing two important checks on a view object:

- First, *ValidView* checks to make sure the application didn't run out of memory when constructing the object. The application reserves some memory at the end of the heap for a *safety pool*. If a memory allocation (such as constructing a dynamic object) crosses into that safety pool, *ValidView* returns **nil**.
- Second, *ValidView* calls the view's *Valid* method, which checks to make sure the object was correctly constructed. If *Valid* returns *False*, *ValidView* returns **nil**.

After loading the object from the stream, it's best to only use the object after passing it by *ValidView*:

```
{ load the object }
if ValidView(TempDesktop) <> nil then
  { replace Desktop with TempDesktop }
```

Replacing the desktop

Once you've decided that you have a valid desktop object, you're ready to replace the existing desktop. Replacing the desktop takes five steps:

- Deleting the desktop from the application
- Disposing of the old desktop object
- Setting the *Desktop* variable
- Inserting the new desktop
- Positioning the desktop

Deleting the old desktop and disposing of the object are important. Remember that the application is a group, too, and it's holding a pointer to the desktop object as one of the views it's supposed to manage. If you dispose of that object without deleting it from the application object, the application object will still try to dispose of the old view when it shuts down, causing a runtime error. Once you've deleted the old desktop, you can safely dispose of it:

```
Delete(Desktop);    { delete the desktop from the application object }
Dispose(Desktop, Done);                     { dispose of the old desktop }
```

With the old desktop safely out of the way, you can now insert the new desktop:

```
Desktop := TempDesktop;
Insert(Desktop);
```

The last item isn't obvious, unless you've changed the video mode between the time you saved the desktop and the time you loaded it. Because the saved desktop object and any windows it contains have their sizes and positions set based on the size of the application, you need to make sure those get adjusted to the current size. Restoring a desktop meant for a 25-line screen on a 43- or 50-line application leaves a lot to be desired!

Luckily, it's easy to set the boundaries of the desktop to suit the current application view, and the desktop takes care of resizing its windows:

```
GetExtent(R);                  { get the boundaries of the application }
R.Grow(0, -1);                    { allow for menu bar and status line }
Desktop^.Locate(R);                { set the boundaries of the desktop }
```

The last consideration, of course, is the clipboard window. Since you excluded it when you saved the desktop, you need to do so when loading the new desktop, too. Be sure to delete the clipboard window from the desktop before getting rid of the old desktop, and reinsert it when the new desktop is in place.

Listing 4.6 shows the complete *LoadDesktop* method.

Listing 4.6
TUTOR06B.PAS loads the new desktop

```
procedure TTutorApp.LoadDesktop;
var
  DesktopFile: TBufStream;
  TempDesktop: PDesktop;
  R: TRect;
begin
  DesktopFile.Init('DESKTOP.TUT', stOpenRead, 1024);
  TempDesktop := PDesktop(DesktopFile.Get);
  DesktopFile.Done;
  if ValidView(TempDesktop) <> nil then
  begin
    Desktop^.Delete(ClipboardWindow);
    Delete(Desktop);
    Dispose(Desktop, Done);
    Desktop := TempDesktop;
    Insert(Desktop);
    GetExtent(R);
    R.Grow(0, -1);
    Desktop^.Locate(R);
    InsertWindow(ClipboardWindow);
  end;
end;
```

Step 7: Using resources

```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

Resources are a handy way to define some of the visual elements of your program and have the added benefit that you don't have to include the initialization code in your application.

In this step, you'll do the following with resources:

- Create a resource file
- Load a menu bar from a resource file
- Load a status line from a resource file
- Load an About box from a resource file

As you saw in Step 2, the code to construct status line and menu bar objects can get rather convoluted, involving numerous nested function calls. One way to insulate your program from that kind of complexity is to construct those objects from resources.

Creating a resource file

Before you can load objects into your program from a resource file, you need to have a resource file to load from. One of the beauties of resources is that your program doesn't care where the resources come from, it just reads the objects and uses them. Similarly, the program doesn't know or care what's in the resource. When you load a menu bar, for example, the program just gets a pointer to a menu bar object. Nothing in your code has to know how many items are on the menu, what order they come in, or what commands you've bound to them.

At some point, your program and your resources have to coordinate. After all, there's no point in loading a resource that only generates commands that your program doesn't respond to. Resources give you the flexibility to define the menu structure (or the dialog box layout or whatever) outside your program, meaning you can modify those aspects of the user interface without changing your program code.

What is a resource file?

Resource files are closely tied to streams. In fact, resource files use streams to store and retrieve objects. The main difference from the program's view is that the resource file allows you to name the stored objects and retrieve them in any order. When you initialize a resource file, you pass it the stream that holds its objects. The resource file itself maintains an index that keeps track of the names and locations of all the resources.

You can write any Turbo Vision object into a resource file, just as you can with a stream. Because the resource file has an underlying stream, you need to make sure you register any object types you'll be reading or writing as resources.

Writing resources to a file

Storing an object as a resource is almost exactly like storing it on a stream, but you also need to give it a name. For example, suppose you have a menu bar called *MyMenu* that you want to store in a resource file under the name 'MAINMENU'. The code looks like this:

Listing 4.7
Storing a menu object as a resource

```
var
  ResFile: TResourceFile;
  MyMenu: PMenuBar;

begin
  MyMenu := ...                               { Initialize the menu bar }
  ResFile.Init(New(PBufStream, Init('FILE.EXT', stCreate, 1024)));
  ResFile.Put(MyMenu, 'MAINMENU');                 { store the resource }
  ResFile.Done;            { dispose of resource file and its stream }
end;
```

The example in Listing 4.7 uses a buffered stream to hold the resources, but you can use any kind of stream. Often, if you have a large resource file, you'll copy the entire resource file to an EMS stream or memory stream for faster access.

The file TUTRES.PAS on your distribution disks contains the program that creates the resource file that holds the menu bar, status line, and About box resources used in the next three steps.

Loading a menu bar resource

Loading any object from a resource file takes three steps:

- Opening the resource file
- Loading the object
- Closing the resource file

If you load numerous objects from the same resource file, you usually open the file only once, read all the objects, and then close the file. If your application reads objects from the resource file at various times during its operation, you might want to open the file during program initialization and close it during shutdown.

Opening the resource file

Tutorial needs to access resources at various times during its operation. The menu bar and status line are set up during the initialization of the application, but a user might want to call up the About box at any time, so you need to have the resource file available at all times. The best solution for this is to open the resource file in the application object's constructor and close it in the application's destructor.

The order of the statements in the constructor is very important, because some steps depend on others having already happened. The following steps *must* be performed in order:

1. Stream registration
2. Resource file initialization
3. Application initialization

You need to register objects before opening the resource file, because you want to be sure you can load objects at any time when the resource file is open. The resource file needs to be open before you call the inherited application constructor, because it will call the virtual methods *InitMenuBar* and *InitStatusLine*, which you're about to modify to read from the resource file.

Listing 4.8 shows the *TTutorApp*'s constructor modified to initialize the resource file TUTORIAL.TVR.

Listing 4.8
Opening a resource file for an application

```
var ResFile: TResourceFile;
   ⋮
constructor TTutorApp.Init;
begin
  MaxHeapSize := 8192;
  RegisterMenus;                     { register objects with streams }
  RegisterViews;
  RegisterEditors;
  RegisterDialogs;
  RegisterApp;
  ResFile.Init(New(PBufStream, Init('TUTORIAL.TVR',    { init stream }
    stOpenRead, 1024)));               { open for reading, 1K buffer }
  inherited Init;                { initialize the application object }
   ⋮
end;
```

Loading the menu bar resource

Initializing the application's menu bar from a resource works just like creating it: You override the virtual method *InitMenuBar*. But instead of calling all the nested functions to create submenus and menu items, you load a menu object from the resource file. Listing 4.9 shows *InitMenuBar* modified to load a menu resource called 'MAINMENU' from the resource file.

Listing 4.9
Initializing a menu from a resource

```
procedure TTutorApp.InitMenuBar;
begin
  MenuBar := PMenuBar(ResFile.Get('MAINMENU'));
end;
```

That's all there is to it. The menu bar object loaded from the resource file functions exactly like one you create. This code makes the assumption that the menu bar resource was created with the proper boundaries. Since all video modes currently supported by Turbo Vision have the same screen width, this should not cause problems. When you load a status line resource, you need to make adjustments for different positions on the screen.

As with streams, the resource file's *Get* method returns a pointer of type *PObject*. You need to typecast that pointer into the appropriate type for the object you load.

Closing the resource file

Closing and disposing of the resource file is just a matter of calling the resource file object's destructor. Since you initialized the resource file in the application constructor, you should dispose of the resource file in the application destructor:

```
destructor TTutorApp.Done;
begin
  ResFile.Done;                  { flush and close the resource file }
  inherited Done;                { dispose of the application object }
end;
```

Loading a status line resource

Loading a status line object from a resource file works just the same way as loading a menu bar. Since you added the code to open and close the resource file in the last section, you don't have to repeat that, so there are only two steps to concern yourself with:

- Loading the status line object
- Adjusting the status line position

Loading the status line object

Loading a status line object from a resource file is just like loading a menu bar, but you need to specify the name of a status line resource:

```
procedure TTutorApp.InitStatusLine;
begin
  StatusLine := PStatusLine(ResFile.Get('STATUS'));
end;
```

In the menu bar example, it was safe to assume that the menu bar resource was designed to cover the top line of the screen, since that's virtually always where menus are, and all top menu lines have the same boundaries. But since different video modes put the bottom line at different positions, it's not safe to assume that a status line automatically has valid and useful boundaries. In the next section, you'll adjust the boundaries of the loaded status line to put it on the last line of the application screen.

Adjusting the status line position

When you created a status line object in Step 2, you assured that the status line was always on the last line of the application by reading the boundaries of the application and setting the status line boundaries relative to that. The only difference with the status line loaded from a resource is that you adjust its position after you load it, rather than setting the position when you create the object. The method, however, is the same and will give you an idea of how you can reposition existing views.

Listing 4.10 shows two alternative ways to position the status line on the last line of the screen.

Listing 4.10
Two ways to move the status line

```
procedure TTutorApp.InitStatusLine;
var R: TRect;
begin
  StatusLine := PStatusLine(ResFile.Get('STATUS'));
  GetExtent(R);
  StatusLine^.MoveTo(0, R.B.Y -1);
end;

procedure TTutorApp.InitStatusLine;
var R: TRect;
begin
  StatusLine := PStatusLine(ResFile.Get('STATUS'));
  GetExtent(R);
  R.A.Y := R.B.Y -1;
  StatusLine^.Locate(R);
end;
```

Neither of the approaches illustrated in Listing 4.10 is particularly "better" than the other. In fact, *MoveTo* sets up a rectangle based on the passed coordinates and the size of the view and then calls *Locate*, so the methods are nearly identical.

Loading an About box resource

One common use of resources is the definition of complex dialog boxes. The About box you defined in Step 3 by calling *MessageBox* is rather limited. You can't define the title of the box, and you can pass it only a single string of text. In this section, you'll create a more interesting About box and store it as a resource.

Using a dialog box resource takes three steps:

- Defining the dialog box resource
- Loading the dialog box resource
- Executing the dialog box

Defining a dialog box resource

Like any other Turbo Vision resource, a dialog box resource is just a named object stored on a stream, so to create a dialog box resource, you first need to create a dialog box object. Since you'll be spending all of Steps 7 and 8 creating dialog boxes, we won't go into all the details right now. For now, all you need to know is that a dialog box object is a group, with other views called *controls* inserted into it. Controls are specialized views that a user interacts with, such as buttons, list boxes, and check boxes.

To create a dialog box object (or a control object to insert into a dialog box), you need to do three things:

- Set the boundaries of the dialog box (or control)
- Construct the object
- Insert the control or store the dialog resource

Listing 4.11 shows the code to create your new About box and its controls.

Listing 4.11
Creating a dialog box resource

```
R.Assign(0, 0, 40, 11);                  { set dialog box boundaries }
AboutBox := New(PDialog, Init(R, 'About Tutorial'));  { construct it }
with AboutBox^ do
begin
  Options := Options or ofCentered;        { make sure it's centered }
  R.Assign(4, 2, 36, 4);                { set static text boundaries }
  Insert(New(PStaticText, Init(R,    { construct static text control }
    #3'Turbo Vision'#13#3'Tutorial program')));    { with this text }
  R.Assign(4, 5, 36, 7);         { set second static text boundaries }
  Insert(New(PStaticText, Init(R,    { construct static text control }
    #3'Copyright 1992'#13#3'Borland International')));
  R.Assign(15, 8, 25, 10);                { set OK button boundaries }
  Insert(New(PButton, Init(R, 'O~k~', cmOk, bfDefault)));
end;
ResFile.Put(AboutBox, 'ABOUTBOX');   { store dialog in resource file }
```

As you saw when you stored the desktop on a stream, a group object stores all the views it manages, so storing the dialog box automatically stores all the controls you inserted into it. When you load the About box from the resource, it automatically loads all its controls, too.

When you call the resource file's *Put* method, you specify the object you want to store and a name for that resource. That's the same name you later use to load the resource with *Get*. In Listing 4.11, the dialog box *AboutBox* is stored with the name 'ABOUTBOX'.

Loading the dialog box resource

Be sure to call RegisterDialogs before loading a dialog box from a stream or resource file.

Loading the dialog box resource is just like loading any other resource, so it should look familiar to you. All you have to do is call the *Get* method of the resource file object, passing it the name of your dialog box resource. Because *Get* returns a generic *PObject* pointer, you'll probably want to typecast the pointer into a *PDialog*:

```
MyDialog := PDialog(ResFile.Get('MYDIALOG'));
```

In many cases, you won't even need to assign the pointer to a variable, as you'll see in the next section.

Executing the dialog box

To execute your dialog box, you use the same *ExecuteDialog* method of the application object that you used in Step 4 to execute the standard file open dialog box:

Listing 4.12
Executing a custom About box, completing TUTOR07.PAS

```
procedure TTutorApp.DoAboutBox;
begin
  ExecuteDialog(PDialog(ResFile.Get('ABOUTBOX')), nil);
end;
```

Since *Get* allocates memory and returns a pointer to it, and since *ExecuteDialog* disposes of the dialog box after executing it, you don't need to assign the About box's pointer to anything, which makes *DoAboutBox* a very simple method.

CHAPTER

5

Creating a data-entry screen

Up to this point, all the objects you've used have been standard Turbo Vision objects, with the exception of the application object, which you've extended considerably. That gives you an idea of the power of Turbo Vision, but at some point you'll definitely want to create some objects of your own. In this chapter, you'll

- Create a data-entry window
- Send messages between views
- Use control objects
- Validate entered data

Over the next several steps, you'll implement a simple inventory system for a small business. The program isn't meant to be truly useful, but it illustrates a lot of useful principles you will want to use in your Turbo Vision applications.

Step 8: Creating a data-entry window

Data entry usually takes place in a dialog box. In this example, the dialog box you'll create will not be modal like the ones you've used so far. Rather than executing it (which makes it modal), you'll insert it, as you do with windows. A Turbo Vision dialog box is just a specialized kind of window—the *TDialog* type is a descendant of *TWindow*.

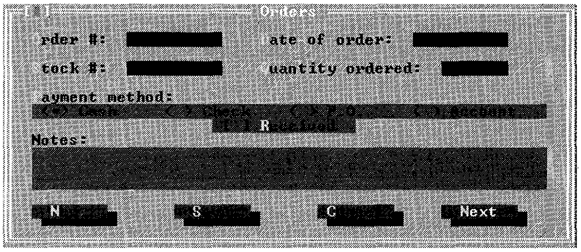
Creating your data-entry window will happen in three parts:

- Creating a new window type
- Preventing duplicate windows
- Adding controls to the window

Creating a new window type

Because you're going to make a number of customizations to your data-entry window, you'll need to define a new object type for that window, called *TOrderWindow*. Because the application needs to keep track of the order window, you'll give the application object a pointer to the order window object.

Figure 5.1
The finished order-entry window



You'll also add a response to the menu command *cmOrderWin*, which is bound to the Examine item on the Orders menu. When you choose Orders | Examine, you want the order-entry window to pop up, so you'll teach the application to handle that command. Listing 5.1 shows these changes.

Listing 5.1
Opening a customized window, from TUTOR08A.PAS

```
type
  POrderWindow = ^TOrderWindow;
  TOrderWindow = object(TDialog)      { order "window" is a dialog box }
    constructor Init;
  end;
  TTutorApp = object(TApplication)
    ClipboardWindow: PEditWindow;
    OrderWindow: POrderWindow;     { give app a pointer to order win }
      ⋮
    procedure OpenOrderWindow;
  end;

constructor TOrderWindow.Init;
var R: TRect;
begin
  R.Assign(0, 0, 60, 17);                    { assign the boundaries }
  inherited Init(R, 'Orders');             { construct the dialog box }
  Options := Options or ofCentered;         { make sure it's centered }
  HelpCtx := $F000;                          { set a new help context }
end;

procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then
      cmOrderWin: OpenOrderWindow;           { open the order window }
      ⋮
end;

procedure TTutorApp.OpenOrderWindow;
begin
  OrderWindow := New(POrderWindow, Init);    { create a new instance }
  InsertWindow(OrderWindow);             { insert it into the desktop }
end;
```

In the remainder of this step and the next one, you'll add more abilities to *TOrderWindow*.

Option flags are explained in Chapter 8, "Views."

If you run the program now, you'll notice several changes. First, if you choose Orders | Examine, a dialog box appears in the middle of the desktop, with the title 'Orders'. Setting the *ofCentered* bit in the window's *Options* field makes sure the window centers itself on the desktop.

You'll also notice that the status line changes when the dialog box appears. That's because *TOrderWindow* changes the current help context (with its *HelpCtx* field). Since you defined a separate status definition for the help context range $F000..$FFFF in Step 2, bringing up a view that sets the help context in that range automatically displays the proper status line. If you close the order window, the status line reverts because the help context changes back.

Limiting open windows

What happens if you choose Orders | Examine while there's already an order window open? *OpenOrderWindow* assigns a new order window to *OrderWindow* and inserts it into the desktop. Now you have two order windows, which is no problem for the desktop to manage, but the application object only knows about the most recent one. This could cause problems when you start to maintain your inventory, so you need to make sure you don't open a new order window if there's already one open. Instead, bring the open window to the front.

One way to approach this problem is to check *OrderWindow* and only assign a new window if it's non-**nil**. This adds some extra responsibilities for you, however, as you have to ensure that *OrderWindow* is always **nil** if there's no valid order window. A simpler solution is to let the window and the application themselves handle the situation.

Sending messages

A more reliable way to find out if there's an order window open is to let the order window itself tell you. Turbo Vision gives you the ability to send *messages* to views. Messages are special events, much like commands, which carry information to a specific view object and allow the receiving view to send information back. In this case, you'll use a *broadcast* message, which is a message that the recipient sends on to each of its subviews. By defining a special message that only order windows know how to handle, you'll be able to determine that there is an order window if (and only if) the message gets answered.

Sending messages is easy. You call a function called *Message*, passing it a pointer to the recipient, some information about the message, and a pointer to any data you might want to accompany the message. In return, *Message* returns **nil** if no view responded to the message, or a pointer to the view that handled the message event.

Listing 5.2 shows how to send a broadcast message to the desktop object, which it will then send to all the windows it's managing. If one of them responds, you can be sure it's the order window, and instead of creating a new order window, you can just bring the existing one to the front by calling *Select*.

Listing 5.2
Sending a broadcast message

```
procedure TTutorApp.OpenOrderWindow;
begin
  if Message(Desktop, evBroadcast, cmFindOrderWindow, nil) = nil then
  begin
    OrderWindow := New(POrderWindow, Init);
    Application^.InsertWindow(OrderWindow);
  end
  else
    if PView(OrderWindow) <> Desktop^.TopView then  { if not already }
      OrderWindow^.Select;              { put order window in front }
end;
```

Responding to messages

Since messages are just events, responding to messages is just like responding to other events. In this case, you know that you want the order window to respond to broadcast messages containing the command *cmFindOrderWindow*, so you give the order window object a *HandleEvent* method that knows how to respond to that:

Listing 5.3
Responding to a broadcast message, which completes TUTOR08B.PAS

```
procedure TOrderDialog.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);    { handle all normal dialog events }
  if (Event.What = evBroadcast) and        { look for a broadcast... }
    (Event.Command = cmFindOrderWindow) then  { ...with this command }
    ClearEvent(Event);                              { and clear it }
end;
```

All that's required to respond to a message is to clear the event. In addition to marking the event as handled, *ClearEvent* sets the event record's *InfoPtr* field to the address of the view that called *ClearEvent*, and *Message* returns the value from *InfoPtr*. So if you need to know *which* view responded to a message, you can check the value returned from *Message*.

In this case, you know what window responded (if any object did), because the whole point of this step was to keep you from creating more than one order window. Simply put, if *Message* returns **nil**, it means no view handled the broadcast, so there is no order window on the desktop. A non-**nil** return value indicates that there is an order window, so it should be put in front of all the windows on the desktop.

Adding controls to the window

In order to use the data-entry window you've created, you need to give it data-entry fields. These fields are made up of various kinds of Turbo Vision *controls*. Controls are the specialized views that enable users to enter or manipulate data in a dialog box, such as buttons, check boxes, and input lines.

Adding a control to a window takes three steps:

- Adding a field to the dialog box object (Optional)
- Setting the boundaries of the control
- Inserting the control

Adding object fields

Before you actually create the control object, you need to consider whether you'll need to access it directly later. In a modal dialog box, there's usually no need or opportunity to do that, so you'll rarely assign object fields to controls in them. In a modeless dialog box such as the order window, you might have occasion to set or read a particular control while the dialog box is open, so you might want to assign fields for them.

In Step 9, you'll look at ways to set and read the values of all the controls together. There are probably not a lot of occasions when you'll need to access individual controls, so you probably won't often create fields in your dialog box objects for specific controls.

Setting boundaries and inserting

Constructing control objects is similar to constructing the other views you've seen already. You assign a rectangle with the boundaries of the control, call the object's constructor, and insert the resulting object into the dialog box object. In some cases, this can be accomplished in a single statement, but in other cases, you'll want to keep a temporary pointer to the object so you can link another control (usually a label object) to the control.

Listing 5.4 shows the code to add an input field with an associated label to the dialog box. Notice that the label control takes only a single statement, while the input line object takes an extra step since you have to keep a pointer to it that you can pass to the label's constructor.

Listing 5.4
Adding a labeled control to a dialog box

```
constructor TOrderWindow.Init;
var
  R: TRect;
  Field: PInputLine;             { temporary variable for input fields }
```

```
begin
  R.Assign(0, 0, 60, 17);
  inherited Init(R, 'Orders');
  Options := Options or ofCentered;
  HelpCtx := $F000;

  R.Assign(13, 2, 23, 3);          { set boundaries for input field }
  Field := New(PInputLine, Init(R, 8));                { construct it }
  Insert(Field);                             { insert into dialog box }
  R.Assign(2, 2, 12, 3);                    { set boundaries for label }
  Insert(New(PLabel, Init(R,
    '~O~rder #:', Field)));   { construct & insert, linking to field }
end;
```

Listing 5.5 shows the full initialization for the data-entry window.

Listing 5.5
Constructing the data entry window, from TUTOR08C.PAS

```
constructor TOrderWindow.Init;
var
  R: TRect;
  Field: PInputLine;
  Cluster: PCluster;
  Memo: PMemo;
begin
  R.Assign(0, 0, 60, 17);
  inherited Init(R, 'Orders');
  Options := Options or ofCentered;
  HelpCtx := $F000;

  R.Assign(13, 2, 23, 3);
  Field := New(PInputLine, Init(R, 8));
  Insert(Field);
  R.Assign(2, 2, 12, 3);
  Insert(New(PLabel, Init(R, '~O~rder #:', Field)));

  R.Assign(43, 2, 53, 3);
  Field := New(PInputLine, Init(R, 8));
  Insert(Field);
  R.Assign(26, 2, 41, 3);
  Insert(New(PLabel, Init(R, '~D~ate of order:', Field)));

  R.Assign(13, 4, 23, 5);
  Field := New(PInputLine, Init(R, 8));
  Insert(Field);
  R.Assign(2, 4, 12, 5);
  Insert(New(PLabel, Init(R, '~S~tock #:', Field)));

  R.Assign(46, 4, 53, 5);
  Field := New(PInputLine, Init(R, 5));
  Insert(Field);
  R.Assign(26, 4, 44, 5);
  Insert(New(PLabel, Init(R, '~Q~uantity ordered:', Field)));
  R.Assign(3, 7, 57, 8);
  Cluster := New(PRadioButtons, Init(R,
    NewSItem('Cash   ',
    NewSItem('Check  ',
    NewSItem('P.O.   ',
    NewSItem('Account', nil))))));
  Insert(Cluster);
  R.Assign(2, 6, 21, 7);
  Insert(New(PLabel, Init(R, '~P~ayment method:', Cluster)));

  R.Assign(22, 8, 37, 9);
  Cluster := New(PCheckBoxes, Init(R, NewSItem('~R~eceived', nil)));
  Insert(Cluster);

  R.Assign(3, 10, 57, 13);
  Memo := New(PMemo, Init(R, nil, nil, nil, 255));  { add memo field }
  Insert(Memo);
  R.Assign(2, 9, 9, 10);
  Insert(New(PLabel, Init(R, 'Notes:', Memo)));

  R.Assign(2, 14, 12, 16);
  Insert(New(PButton, Init(R, '~N~ew', cmOrderNew, bfNormal)));
  R.Assign(13, 14, 23, 16);
  Insert(New(PButton, Init(R, '~S~ave', cmOrderSave, bfDefault)));
  R.Assign(24, 14, 34, 16);
  Insert(New(PButton, Init(R, 'Re~v~ert', cmOrderCancel, bfNormal)));
  R.Assign(35, 14, 45, 16);
  Insert(New(PButton, Init(R, 'N~e~xt', cmOrderNext, bfNormal)));
  R.Assign(46, 14, 56, 16);
  Insert(New(PButton, Init(R, '~P~rev', cmOrderPrev, bfNormal)));
  SelectNext(False);
end;
```

Note that the order in which you add controls is very important, because it determines the *tab order* for the dialog box. Tab order indicates where the input focus goes when the user presses *Tab.* Tab order is really the same as Z-order, which you learned about in Step 4, but since controls don't generally overlap, you don't notice that one is "in front of" another.

If you run the appliction now, you'll find that you have a fully functional data entry window. You can type data into the input lines, manipulate the radio buttons, and so on. In the next step, you'll learn how to set and read the values of the controls, and then add some responses to pressing the buttons along the bottom of the order window.

Step 9: Setting control values

```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

Now that you have a data-entry window, you need to be able to set initial values for the controls and read the data when you're done. You've created the user interface, so now you need to create the program interface. This step covers the three things you have to do to let your application communicate with your dialog boxes:

- Setting up a data record
- Setting controls from the data record
- Reading controls into the data record

Setting up a data record

To set or read the values of the controls in a dialog box, you need to create a buffer (usually a data record) to hold the data for each control. The order of the fields in the data records corresponds to the tab order of the controls; that is, the controls read their data from the record in the same order you inserted the controls into the dialog box.

To create the data record you have to do two things:

- Determine the data needs of each control
- Create a record structure

Determining data needs

Each type of control requires a specific kind or amount of data to initialize itself. For example, an input line reads a string of a particular length, a set of radio buttons reads a *Word*-type value, and a button reads nothing at all. The easiest way to organize the data is to write down each control in the order you insert it into the dialog box, and then write down next to it the data it takes, as shown in Table 5.1.

Table 5.1
Dialog box controls and their data needs

Field	Control	Data needed
Order #	input line	**string**[8]
	label	none
Date	input line	**string**[8]
	label	none
Stock #	input line	**string**[8]
	label	none
Quantity	input line	**string**[5]
	label	none
Payment method	radio buttons	*Word*
	label	none
Received	check boxes	*Word*

Table 5.1: Dialog box controls and their data needs (continued)

Notes	memo	*Word* and **array** of *Char*
	label	none
New	button	none
Save	button	none
Cancel	button	none
Next	button	none

Creating the record structure

Once you have all the information for the controls, you can define a record with the appropriate types of fields in the proper order. Listing 5.6 shows a record type for the controls in *TOrderWindow*.

Listing 5.6
A data record for the order window controls

```
POrder = ^TOrder;
TOrder = record
  OrderNum: string[8];
  OrderDate: string[8];
  StockNum: string[8];
  Quantity: string[5];
  Payment, Received, MemoLen: Word;
  MemoText: array[0..255] of Char;
end;
```

You'll use the same record structure for both setting and reading the control values.

Setting controls

To set the values of the controls in a dialog box, you set the desired values in a data record, then call the dialog box object's *SetData* method, passing it the data record. *SetData* is a method that dialog boxes inherit from *TGroup*. A group's *SetData* calls the *SetData* methods of each of the subviews it manages, following Z-order, which is why the fields of the record need to follow the insertion order of the controls.

Add a global variable to *Tutorial* called *OrderInfo*, of type *TOrder*, to hold data for the current order. Once you initialize *OrderInfo*, you can set the controls in the dialog box by calling *SetData* before executing the dialog box, as shown in Listing 5.7.

Listing 5.7
Using SetData to set control values

```
var OrderInfo: TOrder;
constructor TTutorApp.Init;
begin
   ⋮
  with OrderInfo do                           { set initial OrderInfo fields }
  begin
    OrderNum := '42';
    StockNum := 'AAA-9999';
    OrderDate := '01/15/61';
    Quantity := '1';
    Payment := 2;
    Received := 0;
    MemoLen := 0;
  end;
end;

procedure TTutorApp.OpenOrderWindow;
var R: TRect;
begin
  if Message(Desktop, evBroadcast, cmFindOrderWindow, nil) = nil then
  begin
    OrderWindow := New(POrderWindow, Init);
    Application^.InsertWindow(OrderWindow);
  end
  else
    if PView(OrderWindow) <> Desktop^.TopView then
      OrderWindow^.Select;
  ShowOrder(0);                         { ShowOrder sets the controls }
end;

procedure TTutorApp.ShowOrder(AOrderNum: Integer);
begin
  OrderWindow^.SetData(OrderInfo);              { set control values }
end;
```

Reading control values

Reading the values of the controls in a dialog box is the inverse process of setting them, using the complementary method *GetData* instead of *SetData*. The dialog box's *GetData* calls the *GetData* methods of each of its subviews in Z-order, giving each a chance to write its value into the given data record.

Add a method to the application object that reads the values of the order window's controls back into *OrderInfo* in response to the *cmOrderSave* command bound to the Save button in the order window, as shown in Listing 5.8. Remember to add the method to the object declaration and to add the command to the **case** statement in the application's *HandleEvent* method.

Listing 5.8
Using GetData to read control values
completes TUTOR09.PAS.

```
procedure TTutorApp.SaveOrderData;
begin
  OrderWindow^.GetData(OrderInfo);      { read values into OrderInfo }
end;
```

Now if you run the application and bring up the order window, it still has the default values assigned by the application's constructor. But if you modify the values of the controls in the window, click the Save button, close the window and then reopen it, the controls have the values they held when you closed the window. Saving the data copied new values to the fields in *OrderInfo*, so you have persistent values for the controls in the dialog box.

In Step 11, you'll use the same mechanism to create and update a simple database of inventory records.

Step 10: Validating data entry

Now that you have a working-data entry window where you can display, enter, and change data, you can address the issue of *validating* that data. Validating is the process of assuring that a field contains correct data. Turbo Vision gives you the ability to validate individual fields or entire screens of data.

In general, you need to validate only input line controls—they are the only controls that allow free-form input, other than memo fields, which are assumed to be notes or comments that don't have to be as precise.

Validating a data field takes only two steps:

- Assigning validator objects
- Calling *Valid* methods

Validator objects are all in the unit *Validate*. Be sure to add *Validate* to the **uses** clause of any program or unit that uses validators.

Assigning validator objects

Every input line object has a field that can point to a validator object. Validator objects are simple objects that check the contents of their associated input lines with some sort of criteria for validity, such as a numeric range, a list of values, or a "picture" of how the field should look.

There are two steps to assigning a validator object, although they're usually accomplished in one statement:

- Constructing the validator object
- Assigning the validator to an input line

Constructing a validator object

Validator constructors are very simple, and since they aren't views, they require only enough parameters to tell them how to validate data. For example, a range validator takes only two parameters: the low and high bounds of the valid range. The following example shows how you could construct a validator that allows only four-digit integer numbers:

```
RangeValidator := New(PRangeValidator, Init(1000, 9999));
```

Tutorial uses only two kinds of validators: range and picture validators. All the different supplied validators are described in detail in Chapter 13, "Data validation objects."

Assigning a validator to an input line

Input line objects have a method called *SetValidator* that assigns a validator object to the input line's *Validator* field. Since your program will almost never need to access a particular validator other than to assign it to the input line, you can generally construct and assign the validator in a single statement:

```
SetValidator(New(PRangeValidator, Init(1000, 9999)));
```

Once you've assigned the validator object, you don't have to worry about it. The input line knows when to call the validator, and the validator alerts the user if it detects invalid data. Listing 5.9 shows the changes to *TOrderWindow*'s constructor to add validators to the four input line objects.

Listing 5.9
Adding validators to input lines

```
constructor TOrderWindow.Init;
begin
   ⋮
  R.Assign(13, 2, 23, 3);
  Field := New(PInputLine, Init(R, 8));
  Field^.SetValidator(New(PRangeValidator,
    Init(1, 99999)));          { order number is a positive integer }
  Insert(Field);
   ⋮
  R.Assign(43, 2, 53, 3);
  Field := New(PInputLine, Init(R, 8));
  Field^.SetValidator(New(PPXPictureValidator,
    Init('{#[#]}/{#[#]}/{##[##]}', True)));      { date is MM/DD/YY }
  Insert(Field);
   ⋮
  R.Assign(13, 4, 23, 5);
  Field := New(PInputLine, Init(R, 8));
  Field^.SetValidator(New(PPXPictureValidator,     { Paradox picture }
    Init('&&&-####', True)));    { stock # is 3 letters, -, 4 digits }
    Insert(Field);
     :
    R.Assign(46, 4, 53, 5);
    Field := New(PInputLine, Init(R, 5));
    Field^.SetValidator(New(PRangeValidator,
      Init(1, 99999)));                    { quantity is positive integer }
    Insert(Field);
     :
  end;
```

Now if you run the application and type a number such as 99999 in the order number field and try to close the window, a message box appears informing you that the number is out of range, returning you to the field in which the validation error occurred. Similarly, errors in other fields prevent the closing of the window until all errors are gone.

But you'll also find that you can *save* invalid data. By default, the dialog box validates its fields only when you close the dialog box, so in the next section you'll see how to validate data at other times, such as before saving data.

Calling Valid methods

The two key questions in validating are "What is valid?" and "When do I validate?" You answered the first question by assigning specific types of validator objects to input lines. The second is somewhat more complex, however. Data validation actually takes place in the *Valid* methods of input line objects, and *Valid* can be called at different times.

There are three times when you might call *Valid*:

- When a window closes
- When the focus moves to another field (on *Tab*)
- When the user asks for validation

Validating on close

By default, when you close any view, it calls its *Valid* method to assure that it's allowed to close. The editor windows you created in Step 4, for example, check to see that changes in the editor are saved to disk (or consciously discarded) before they close.

When you close a dialog box (as with any other group), the dialog box's *Valid* method calls the *Valid* methods of all its subviews and only returns *True* if all the subviews return *True*. Since an input line's *Valid* checks with its validator object, closing a window has the effect of validating all fields.

Validating on Tab

You might want to force the user to enter valid data in a certain field before moving to another field. To do that, you need to set the input line's *ofValidate* option flag. If *ofValidate* is set, when the user or program tries to move the focus from the input line, the input line calls its *Valid* method, and if *Valid* returns *False*, it keeps the focus.

You should use such validation only in cases where it's truly necessary, since it's intrusive to the data entry process. However, if it saves someone from entering a whole screen full of useless data, it's worth the intrusion.

Validating on demand

Probably the most useful kind of validation is validation on demand. That is, at some point you just tell the dialog box or an individual field to validate itself. This is the solution for the problem of saving invalid data: Validate the data before you save it. The changes to *SaveOrderData* shown in Listing 5.10 prevent copying the values of the controls into *OrderInfo* unless all controls report they have valid data.

Listing 5.10
Validating data before saving, completing TUTOR10.PAS

```
procedure TTutorApp.SaveOrderData;
begin
  if OrderWindow^.Valid(cmClose) then
    OrderWindow^.GetData(OrderInfo);
end;
```

Note the use of *cmClose* in the call to *Valid*. *Valid* can handle different sorts of validity checks for different commands. By default, Turbo Vision uses two kinds of validity checks. Passing *cmValid* to *Valid* is used to determine whether the object was constructed correctly. Calling *ValidView* uses the *cmValid* check. You've also seen that windows and dialog boxes call *Valid* before closing, passing *cmClose*, the command that indicates they are supposed to close.

Calling *Valid(cmClose)* is like asking "Would you be valid if I asked you to close now?" Calling it before saving acts as a safety check before saving the data.

CHAPTER

6

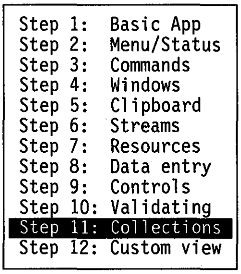
Collecting data

Now that you have a working data-entry window, it makes sense to connect it with a database. Keep in mind that this example is intended to teach you about Turbo Vision, not about database management or inventory control. Some aspects of the program are necessarily simplified to allow you to focus on Turbo Vision without too much attention to the underlying database.

To connect your data-entry window with the database, you'll do the following:

- Load a collection of data records from a stream
- Display, modify, change and add records
- Enable and disable commands as appropriate
- Create a customized view

Step 11: Adding a collection of data



```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

Turbo Vision provides a flexible and powerful object-oriented data management type called a *collection*. A collection is similar to an expandable array of pointers that can point to any sort of data, such as objects or records.

In this step, you'll do the following:

- Create a data object
- Load the data from a stream
- Display data records

- Move from record to record
- Add new records
- Cancel edits

Creating a data object

As you saw in Step 9, the data for setting the controls in a dialog box comes in the form of a record, in this case, of type *TOrder*. But if you want to use your collection of data on a stream, it has to be an object descended from *TObject*. The solution is to create a *wrapper object*, an object that's just an enabling shell around your data.

A wrapper object for use with streams needs four things:

- A field or fields containing the data
- A *Store* method to write the data to the stream
- A *Load* constructor to read the data from the stream
- A registration record

Listing 6.1 shows the declaration of a simple wrapper object, *TOrderObj*, that wraps around *TOrder*.

Listing 6.1
A simple wrapper object

```
type
  POrderObj = ^TOrderObj;
  TOrderObj = object(TObject)
    TransferRecord: TOrder;
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
  end;

constructor TOrderObj.Load(var S: TStream);
begin
  inherited Init;                               { construct the object }
  S.Read(TransferRecord, SizeOf(TransferRecord));  { get stream data }
end;

procedure TOrderObj.Store(var S: TStream);
begin
  S.Write(TransferRecord, SizeOf(TransferRecord));    { write record }
end;
```

As you'll recall from Step 6, all objects used with streams must be registered with streams, so you need to create a *registration record* for *TOrderObj*. By convention, Turbo Vision stream registration records take the name of the object type, but substitute an initial R for the T in the type name. So the registration record for *TOrderObj* would be *ROrderObj*. Listing 6.2 shows the declaration of *ROrderObj*.

Listing 6.2
A stream registration record for the order object

```
const
  ROrderObj: TStreamRec = (
     ObjType: 15000;
     VmtLink: Ofs(TypeOf(TOrderObj)^);
     Load:    @TOrderObj.Load;
     Store:   @TOrderObj.Store
  );
```

The only part of the registration record you have to think about is the *ObjType* field. It must be a unique *Word*-type number. Turbo Vision reserves all the numbers in the range 0..99, but you can use any other numbers for your objects. The only constraint is that they must be unique. All the other fields are always created the same way.

Loading the collection

A collection can deal with any sort of pointer, but when it reads or writes itself on a stream, it assumes that the items in the collection are registered descendants of *TObject* that have *Load* and *Store* methods. If that assumption isn't true, you have to override a few of the collection's methods, as *TStringCollection* does. But since you've created an object that fits what *TCollection* expects, you don't have to modify *TCollection* at all.

Listing 6.3 shows a procedure that loads a collection of order records from a stream. The file ORDERS.DAT on your distribution disks holds a stream of several sample orders. Be sure to add the global variable *OrderColl*, of type *PCollection*, to the program. While you're adding that, also add an integer variable called *CurrentOrder*, which you'll use to keep track of the position of the order in the collection.

Listing 6.3
Loading a collection from a stream

```
procedure LoadOrders;
var OrderFile: TBufStream;
begin
  OrderFile.Init('ORDERS.DAT', stOpenRead, 1024);
  OrderColl := PCollection(OrderFile.Get);
  OrderFile.Done;
end;
```

Listing 6.3 shows some of the advantages of saving data in collections. In a single step, you load all the data, without having to read and allocate individual records, watch for the end of the file, and so on. You just get the collection, and it takes care of loading its items.

Displaying a record

Now that you have a collection of order records in memory, you can use them to provide the data to the order window. Instead of creating an initial record for *OrderInfo* in the application's constructor, you can now copy the first element from the collection, as shown in listing 6.4.

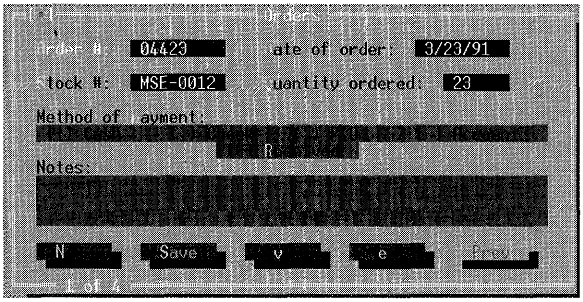
Listing 6.4
Loading the collection of order records

```
constructor TTutorApp.Init;
begin
   ⋮
  LoadOrders;
  CurrentOrder := 0;
  OrderInfo :=
    POrderObj(OrderColl^.At(CurrentOrder))^.TransferRecord;
  DisableCommands([cmOrderNext, cmOrderPrev, cmOrderCancel]);
end;
```

At first glance, this may seem a bit complicated. The collection's *At* method returns the pointer to a particular item in the collection. Since the collection holds untyped pointers, you need to typecast it to a *POrderObj* so that you can access it's *TransferRecord* field. That's the information you want to assign to *OrderInfo*.

Since you don't have responses to the listed commands, you'll disable them for now. Later in this step, as you develop more methods to deal with the database, you'll enable them when appropriate. Now when you open the order window, it holds a record from the database, as shown in Figure 6.1.

Figure 6.1
Displaying the first record in the database



Orders

Order #: 04423   ate of order: 3/23/91

tock #: MSE-0012   uantity ordered: 23

Method of ayment:

Notes:

Saving the record

When you click the Save button, *SaveOrderData* still copies the values of the controls into *OrderInfo*, but you now need to also copy it into the data collection and save the updated collection to the disk, as shown in Listing 6.5.

Listing 6.5
Saving the updated collection to disk, completing TUTOR11A.PAS

```
procedure TTutorApp.SaveOrderData;
begin
  if OrderWindow^.Valid(cmClose) then
  begin
    OrderWindow^.GetData(OrderInfo);
    POrderObj(OrderColl^.At(CurrentOrder))^.TransferRecord :=
      OrderInfo;
    SaveOrders;
  end;
end;

procedure SaveOrders;
var OrderFile: TBufStream;
begin
  OrderFile.Init('ORDERS.DAT', stOpenWrite, 1024);
  OrderFile.Put(OrderColl);
  OrderFile.Done;
end;
```

Moving from record to record

Now that you can edit the first record in the database, you need to be able to move to other records. You've defined the menu and status line commands, so it's time to define responses to those commands. Make the application's *HandleEvent* method call *ShowOrder* and change *ShowOrder* to move to the specified order, as shown in Listing 6.6.

Listing 6.6
Moving among records, which makes TUTOR11B.PAS

```
procedure TTutorApp.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmOrderNext:
        begin
          ShowOrder(CurrentOrder + 1);
          ClearEvent(Event);
        end;
      cmOrderPrev:
        begin
          ShowOrder(CurrentOrder - 1);
          ClearEvent(Event);
        end;
      ⋮
end;
```

```
procedure TTutorApp.ShowOrder(AOrderNum: Integer);
begin
  CurrentOrder := AOrderNum;
  OrderInfo :=
POrderObj(OrderColl^.At(CurrentOrder))^.TransferRecord;
  OrderWindow^.SetData(OrderInfo);
  if CurrentOrder > 0 then EnableCommands([cmOrderPrev])
  else DisableCommands([cmOrderPrev]);
  if OrderColl^.Count > 0 then EnableCommands([cmOrderNext]);
  if CurrentOrder >= OrderColl^.Count - 1 then
    DisableCommands([cmOrderNext]);
end;
```

ShowOrder manipulates the *cmOrderNext* and *cmOrderPrev* commands. By enabling and disabling the commands at the right times, your response methods don't have to check whether it's appropriate to respond. This is good for two reasons:

- Your code is simpler. For example, if *cmNextOrder* is always disabled when you're editing the last order in the collection, your response to *cmNextOrder* doesn't have to check to make sure there is a next order.
- The user knows what's happening. It's much better to disable an inappropriate command than to offer it and then either not do anything with it or flash a message saying the command is inappropriate.

Of course, to make this scheme work properly, you need to set up the Next and Prev commands properly when you open the window initially:

```
procedure TTutorApp.OpenOrderWindow;
var R: TRect;
begin
  if Message(Desktop, evBroadcast, cmFindOrderWindow, nil) = nil then
  begin
    OrderWindow := New(POrderWindow, Init);
    InsertWindow(OrderWindow);
  end
  else OrderWindow^.MakeFirst;
  OrderWindow^.SetData(OrderInfo);
  EnableCommands([cmOrderNew, cmOrderSave, cmOrderCancel]); { always }
  if CurrentOrder < OrderColl^.Count -1 then    { if this is the last }
    EnableCommands([cmOrderNext]);      { ...don't let us go to next }
end;
```

Adding new records

Adding a new record to the database is simple, but you need to create a temporary data object to hold the new data so you can insert it into the database. For this, you can add a global variable, *TempOrder*, of type *POrderObj*. To add the new record, load your empty record into the order window and have the user fill it in and save it:

```
procedure TTutorApp.EnterNewOrder;
begin
  OpenOrderWindow;                          { make sure there's a window }
  CurrentOrder := OrderColl^.Count;           { point past last record }
  TempOrder := New(POrderObj, Init);               { create temp order }
  OrderInfo := TempOrder^.TransferRecord;              { copy the data }
  OrderWindow^.SetData(OrderInfo);                { set control values }
  DisableCommands([cmOrderNext, cmOrderPrev, cmOrderNew]);
end;
```

Saving the new record is slightly different, since you can't just copy the data into an existing item in the colection. Listing 6.7 shows a modified *SaveOrderData* method that handles new records.

Listing 6.7
Saving new or existing records

```
procedure TTutorApp.SaveOrderData;
begin
  if OrderWindow^.Valid(cmClose) then
  begin
    OrderWindow^.GetData(OrderInfo);
    if CurrentOrder = OrderColl^.Count then  { only True if new item }
    begin
      TempOrder^.TransferRecord := OrderInfo;          { copy the data }
      OrderColl^.Insert(TempOrder);           { insert the new order }
    end
    else POrderObj(OrderColl^.At(CurrentOrder))^.TransferRecord :=
      OrderInfo;
    SaveOrders;
    EnableCommands([cmOrderNew, cmOrderPrev]);
  end;
end;
```

Notice that you don't dispose of *TempOrder*—that pointer's still being used by the collection, so you don't want to deallocate the memory.

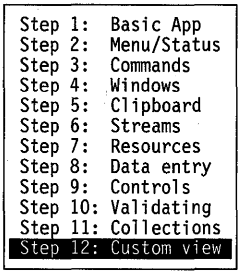
Canceling edits

One last feature that's easy to implement is canceling changes you've made to a record, either when modifying an existing record or when adding a new one. Responding to the command *cmOrderCancel*, you can call *CancelOrder*:

Listing 6.8
Canceling a new order completes TUTOR11C.PAS.

```
procedure TTutorApp.CancelOrder;
begin
  if CurrentOrder < OrderColl^.Count then      { if existing order... }
    ShowOrder(CurrentOrder)                   { ...just reload values }
  else                                                  { otherwise }
  begin
    Dispose(TempOrder, Done);             { dispose of temporary record }
    ShowOrder(CurrentOrder - 1);                 { load in last record }
  end;
end;
```

Step 12: Creating a custom view



```
Step 1:  Basic App
Step 2:  Menu/Status
Step 3:  Commands
Step 4:  Windows
Step 5:  Clipboard
Step 6:  Streams
Step 7:  Resources
Step 8:  Data entry
Step 9:  Controls
Step 10: Validating
Step 11: Collections
Step 12: Custom view
```

One thing you've probably noticed in using this simple database is that you can't tell *which* record you're looking at, unless it happens to be the first or last record, or how many total records there are. A much nicer way to handle this is to show the user the number of the record the window holds and how many total records exist. Since Turbo Vision doesn't provide such a view for you, you'll have to create one yourself.

To create your view, do the following:

- Create the internal counting engine
- Construct the view
- Give the view its appearance
- Add the view to the order window

Those three steps are universal to all views. In fact, there are only four things that every view must be able to do. Briefly, they are:

All these responsibilities are described in detail in Chapter 8, "Views."

- Cover its full rectangular area
- Respond to any events in that area
- Draw itself on the screen when told to
- Perform any internal functions

Creating the counting engine

The internal data for the counter is very simple. It needs to track only two numbers: the current record and the total number of records. For that, you'll give the object two numeric fields of type *Longint*. You then provide methods to set, increment, and decrement the values of each field:

Listing 6.9
The counting engine for the custom view, from the Count unit

```
type
  PCountView = ^TCountView;
  TCountView = object(TView)
    Current: Longint;
    Count: Longint;
    procedure SetCount(NewCount: Longint);
    procedure IncCount;
    procedure DecCount;
    procedure SetCurrent(NewCurrent: Longint);
    procedure IncCurrent;
    procedure DecCurrent;
  end;

procedure TCountView.SetCount(NewCount:Longint);
begin
  Count := NewCount;
  DrawView;
end;

procedure TCountView.IncCount;
begin
  SetCount(Count + 1);
end;

procedure TCountView.DecCount;
begin
  SetCount(Count - 1);
end;

procedure TCountView.SetCurrent(NewCurrent:Longint);
begin
  Current := NewCurrent;
  DrawView;
end;

procedure TCountView.IncCurrent;
begin
  SetCurrent(Current + 1);
end;
```

```
procedure TCountView.DecCurrent;
begin
  SetCurrent(Current - 1);
end;
```

Most of the methods are self-explanatory. After changing or setting the field value, *SetCount* and *SetCurrent* call the inherited method *DrawView*, which tells the view to draw itself if its appearance needs to be updated. You'll add the actual means it uses for that drawing in this step.

Constructing the view

Whenever you derive a new view, you need to make sure its constructor takes care of initializing all the fields inherited from the ancestor type as well as initializing any new fields. Usually, that means you override the constructor but call the inherited constructor first:

```
constructor TCountView.Init(var Bounds:TRect);
begin
  inherited Init(Bounds);
  SetCount(0);
  SetCurrent(1);
end;
```

Bounds in this case is the parameter common to all views, determining the rectangular region the view must cover. Since the ability to handle a rectangular region (and to handle most simple events) is inherited from *TView*, your *TCountView* object can rely on the inherited behavior.

Drawing the view

Every view must have its own method called *Draw* that knows how to represent the contents of the view at any given time. *Draw* methods almost never call their inherited methods, because that would usually result in drawing over areas that have already been drawn, resulting in an annoying flickering.

Drawing takes advantage of the view's *color palette*. The palette maps colors onto the view's owner.

Listing 6.10 shows the *Draw* and *GetPalette* methods for *TCountView*.

Listing 6.10
Drawing the custom view

```
procedure TCountView.Draw;
var
  B: TDrawBuffer;
  C: Word;
  Params: array[0..1] of Longint;
  Start: Word;
  First: String[10];
  Display: String[20];
begin
  C := GetColor(2);                         { Uses same color as frame }
  MoveChar(B, '″', C, Size.X);
  Params[0] := Current;
  Params[1] := Count;
  FormatStr(Display, ' ~%d~ of %d ', Params);
  { If Current is greater than Count, display Current highlighted }
  if Current > Count then C := GetColor($0504)
  else C := GetColor($0202);
  MoveCStr(B, Display, C);
  WriteLine(0, 0, Size.X, Length(Display), B);
end;

function TCountView.GetPalette: PPalette;
const P: string[Length(CCountView)] = CCountView;
begin
  GetPalette := @P;
end;
```

With the addition of *Load* and *Store* methods and a stream registration record, *RCountView,* this completes the file COUNT.PAS.

Using the counter

To add the counter view to the order window, you need to add *Count* to the application's **uses** clause, then do the following tasks:

- Add the view to the window
- Manipulate the counter

Adding the counter to the window

To make it easy to manipulate the counter view, add a field to the order window object that points to the counter and construct the view in the order window constructor. Don't forget to add *RegisterCount* to the application's constructor so you can save counter views on streams.

```
type
  TOrderWindow = object(TDialog)
    Counter: PCountView;                  { add field for the counter }
    constructor Init;
    procedure HandleEvent(var Event: TEvent); virtual;
  end;

constructor TOrderWindow.Init;
begin
  ⋮
  R.Assign(5, 16, 20, 17);        { locate the counter on the frame }
  Counter := New(PCountView, Init(R));                { construct view }
  Counter^.SetCount(OrderColl^.Count);       { set number of orders }
  Insert(Counter);                              { insert into window }
  SelectNext(False);
end;
```

Manipulating the counter

There are only a few times when you have to adjust the counter. The current item needs to be updated when you display a new record, and the item count needs to be updated when you add a new record. Listing 6.11 shows the updated methods in *TTutorApp*.

Listing 6.11 Manipulating the new counter view completes TUTOR12.PAS.

```
procedure TTutorApp.EnterNewOrder;
begin
  OpenOrderWindow;
  CurrentOrder := OrderColl^.Count;
  TempOrder := New(POrderObj, Init);
  OrderInfo := TempOrder^.TransferRecord;
  with OrderWindow^ do
  begin
    SetData(OrderInfo);
    Counter^.SetCurrent(CurrentOrder + 1);
  end;
   ⋮
end;

procedure TTutorApp.SaveOrderData;
begin
  if OrderWindow^.Valid(cmClose) then
  begin
    OrderWindow^.GetData(OrderInfo);
    if CurrentOrder = OrderColl^.Count then
    begin
      TempOrder^.TransferRecord := OrderInfo;
      OrderColl^.Insert(TempOrder);
      OrderWindow^.Counter^.IncCount;
    end
   ⋮
  end;
end;
```

```
procedure TTutorApp.ShowOrder(AOrderNum: Integer);
begin
  CurrentOrder := AOrderNum;
  OrderInfo :=
POrderObj(OrderColl^.At(CurrentOrder))^.TransferRecord;
  with OrderWindow^ do
  begin
    SetData(OrderInfo);
    Counter^.SetCurrent(CurrentOrder + 1);
  end;
   ⋮
end;
```

Where to now?

There are many additions and changes you could make to *Tutorial* to make it more useful. This section suggests some approaches you might use to implement them. A version of the program that incorporates these changes is in the file TUTOR.PAS.

Tutorial contains these changes:

- Supplier and stock item dialog boxes
- Lookup validation

Additional dialog boxes

TUTOR.PAS implements modal dialog boxes for supplier and stock item data bases, much like the order-entry window. The main difference is that both of these new dialog boxes are modal, and therefore respond to commands such as *cmStockNext*, rather than having the application handle them.

The validators for some of the data items in the new dialog boxes also show some additional examples of picture validators.

Lookup validation

Since TUTOR.PAS has databases of stock items and suppliers, it can implement lookup validation of data-entry fields for those items. For example, in Step 10 you validated the stock number field in the order-entry window with a picture validator, which made sure the number had the proper format. By using a lookup validator, TUTOR.PAS can ensure that the items entered not only have the proper format, but also match actual items in the inventory.

PART 2

Using Turbo Vision

CHAPTER

7

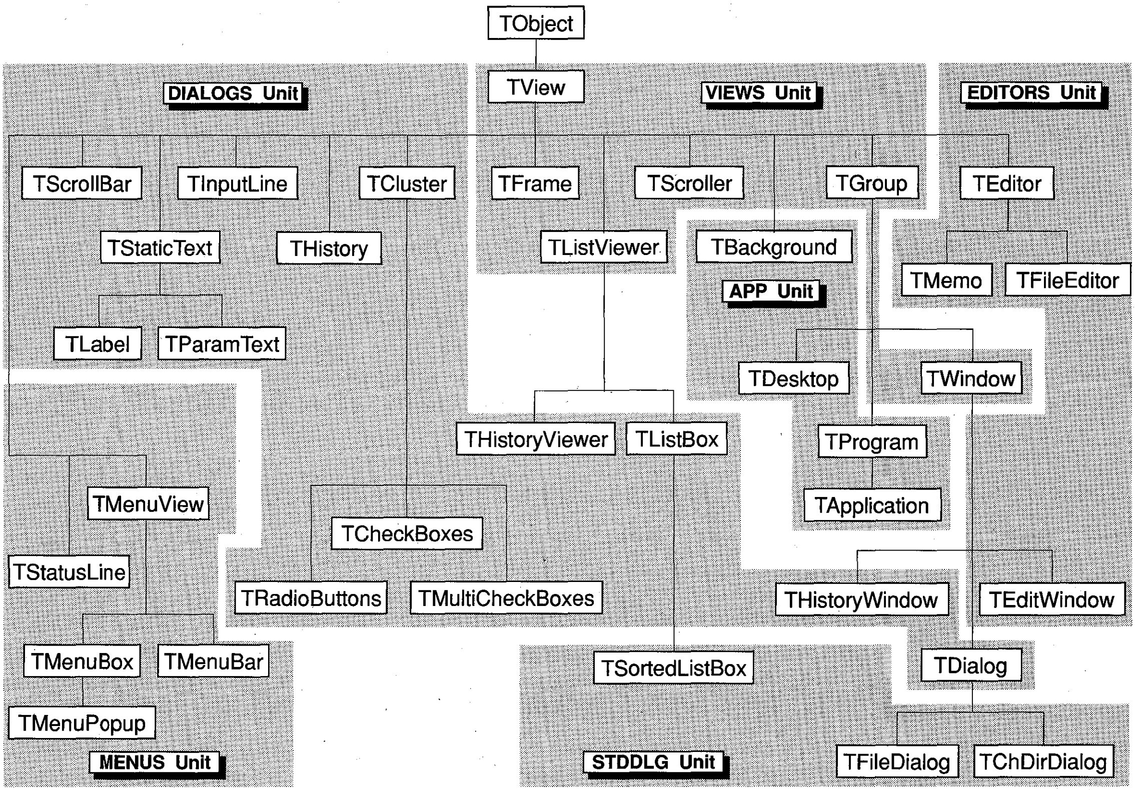
Turbo Vision overview

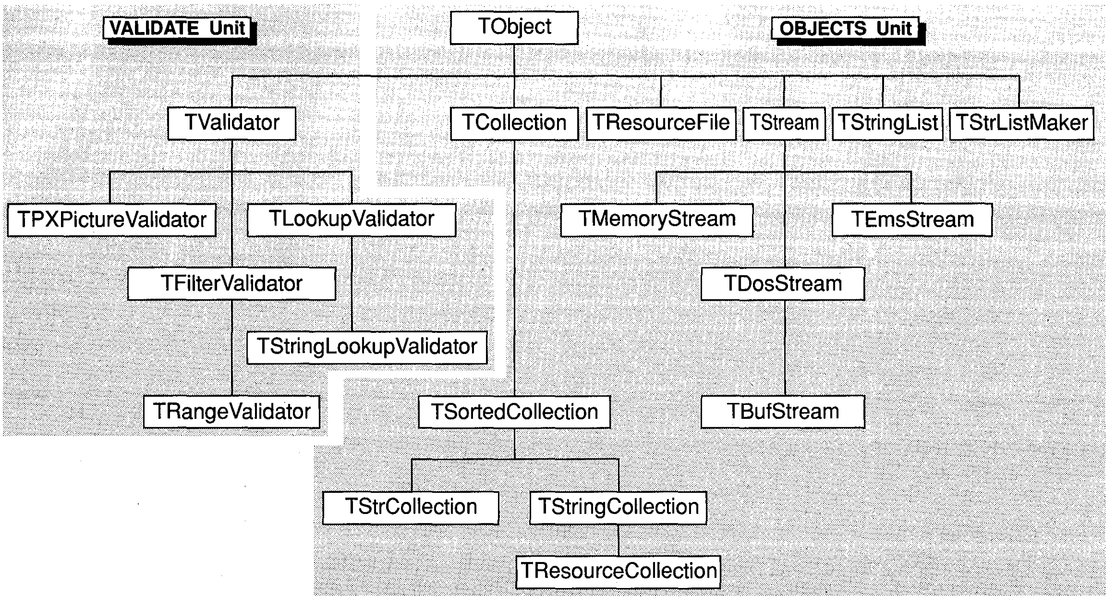
This chapter assumes that you have a good working knowledge of Pascal, especially the object-oriented extensions. It also assumes that you have read Part 1 of this book to get an overview of Turbo Vision's philosophy, capabilities, and terminology.

Chapter 19, "Turbo Vision reference," describes the methods and fields of each standard object type in depth, but until you acquire an overall feel for how the hierarchy is structured, you can easily become overwhelmed by the mass of detail. This chapter presents an informal browse through the hierarchy before you tackle the detail. The remainder of this part will give more detailed explanations of the components of Turbo Vision and how to use them. Chapter 19, "Turbo Vision reference," provides alphabetical reference material.

The view hierarchy tree is shown in Figure 7.1. Study this picture carefully. To know that *TDialog*, for example, is derived from *TWindow*, which is a descendant of *TGroup*, which is a descendant of *TView*, reduces the learning curve considerably. Each new derived object type you encounter already has familiar inherited properties. You simply study whatever additional fields and properties it has over its parent.

Figure 7.1: Turbo Vision object hierarchy





As you develop your own Turbo Vision applications, you'll find that a general familiarity with the standard object types and their mutual relationships is an enormous help. Mastering the minute details will come later, but as with all OOP projects, the initial overall planning of your new objects is the key to success.

Each group is described in a separate section of this chapter. Within each of these groups there are also different sorts of objects. Some are useful objects—you can create instances of them and use them. Others are abstract objects that serve as the basis for deriving related, useful objects. Before looking at the objects in the Turbo Vision hierarchy, it will help to understand a little about object hierarchies.

Working with object hierarchies

This section describes some of the basic properties of objects, specifically applied to the Turbo Vision hierarchy. The topics covered are

- Basic object operations
- Inheriting fields
- Types of methods

Basic object operations

Given any object type there are two basic things you can do: You can

- Derive a descendant object type
- Create an instance of that type ("instantiate" it)

If you derive a descendant object type, you have a new object type on which the previous two operations again apply. The next sections examine both of these operations, then explore the use of abstract objects.

Derivation

When you want to extend or change an existing object type, you derive a new object type from an existing one:

```
PNewScrollBar = ^TNewScrollBar;         { define pointer to new type }
TNewScrollBar = object(TScrollBar)      { derive from existing type }
  constructor Init(...);
end;
```

In defining your new object, you can do three things:

- Add new fields
- Define new methods
- Override existing methods

If you don't do at least one of those things, there is no reason to create a new object type. The new or revised methods and fields you define add functionality to *TScrollBar*. New object types nearly always redefine the *Init* constructor to determine the default values and properties.

Instantiation

Creating an instance of an object is usually accomplished by a variable declaration, either static or dynamic:

```
MyStream: TBufStream;                    { declare a static instance }
SomeButton: PButton;                    { declare a dynamic instance }
```

MyStream would be initialized by *TBufStream.Init* with certain default field values. You can find these by consulting the *TBufStream.Init* entry in Chapter 19, "Turbo Vision reference." Since *TBufStream* is a descendant of *TStream, TBufStream.Init* calls *TStream.Init* to set the fields inherited from *TStream*. Similarly, *TStream.Init* is a descendant of *TObject*, so it calls the *TObject* constructor to allocate memory. *TObject* has no parent, so the buck stops there.

The inheritance diagrams at the beginning of each object's entry in Chapter 19 show you which fields and methods each object type declares or overrides, with overridden methods in ancestor types struck out.

The *MyStream* object now has default field values which you might need to change. It also has all the methods of *TBufStream* plus the methods (possibly overridden) of *TStream* and *TObject*. To make use of *MyStream*, you need to know what its methods *do*. If the required functionality is not defined in *TBufStream*, you need to derive a new descendant type.

Whether you can create a useful instance of an object type depends on what kind of virtual methods the object has. Many of Turbo Vision's standard types have abstract methods that must be defined in descendant types.

Abstract objects

Many object types exist as "abstract" bases from which you can derive more specialized, useful object types. The reason for having abstract types is partly conceptual but serves the practical aim of reducing coding effort.

In general, as you travel down the hierarchy, the types become more specialized and less abstract.

For example, the *TRadioButtons* and *TCheckBoxes* types could each be derived directly from *TView* without difficulty. However, they share a great deal in common. They both represent sets of controls with similar responses. A set of radio buttons is a lot like a set of check boxes in which only one box can be checked, although there are other differences. This commonality warrants creating an abstract object type called *TCluster*. *TRadioButtons* and *TCheckBoxes* are then derived from *TCluster* with the addition of a few specialized methods to provide their individual functionalities.

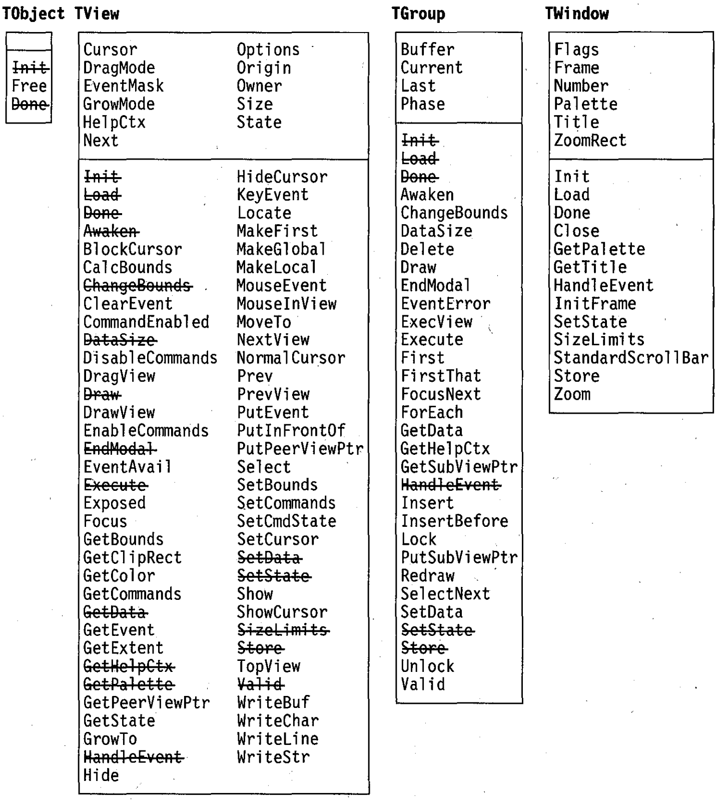
It's never useful, and often not possible, to create an instance of an abstract object type. An instance of *TCluster*, for example, would not have a useful *Draw* method. It inherits *TView.Draw* without overriding, so the cluster's *Draw* would simply display an empty rectangle of the default color.

If you want a fancy cluster of controls with properties different from radio buttons or check boxes, you might try deriving a *TMyCluster* from *TCluster*, or it might be easier to derive your special cluster from *TRadioButtons* or *TCheckBoxes*, depending on which is closer to your needs. In all cases, you add fields, and add or override methods, with the least possible effort. If your plans include a whole family of fancy clusters, you might find it convenient to create an intermediate abstract object type.

Inheriting fields

If you take an important trio of objects: *TView*, *TGroup*, and *TWindow*, a glance at their fields reveals inheritance at work, and also tells you quite a bit about the growing functionality as you move down the hierarchy. Figure 7.2 shows the inheritance of these objects.

Figure 7.2
TWindow inheritance



TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next

Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
~~Awaken~~
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
~~EndModal~~
EventAvail
~~Execute~~
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
~~GetHelpCtx~~
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide

HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
~~SizeLimits~~
~~Store~~
TopView
~~Valid~~
WriteBuf
WriteChar
WriteLine
WriteStr

TGroup

Buffer
Current
Last
Phase

~~Init~~
~~Load~~
~~Done~~
Awaken
ChangeBounds
DataSize
Delete
Draw
EndModal
EventError
ExecView
Execute
First
FirstThat
FocusNext
ForEach
GetData
GetHelpCtx
GetSubViewPtr
~~HandleEvent~~
Insert
InsertBefore
Lock
PutSubViewPtr
Redraw
SelectNext
SetData
~~SetState~~
~~Store~~
Unlock
Valid

TWindow

Flags
Frame
Number
Palette
Title
ZoomRect

Init
Load
Done
Close
GetPalette
GetTitle
HandleEvent
InitFrame
SetState
SizeLimits
StandardScrollBar
Store
Zoom

Table 7.1 shows the fields that each object has, including those inherited.

Table 7.1
Inheritance of view fields

TView fields	TGroup fields	TWindow fields
Owner	Owner	Owner
Next	Next	Next
Origin	Origin	Origin
Size	Size	Size
Cursor	Cursor	Cursor
GrowMode	GrowMode	GrowMode
DragMode	DragMode	DragMode
HelpCtx	HelpCtx	HelpCtx
State	State	State
Options	Options	Options
EventMask	EventMask	EventMask
	Buffer	Buffer
	Phase	Phase
	Current	Current
	Last	Last
		Flags
		Title
		Number
		ZoomRect
		Palette
		Frame

Notice that *TGroup* inherits all the fields of *TView* and adds several more that are pertinent to group operation, such as pointers to the current and last views in the group. *TWindow* in turn inherits all of *TGroup*'s fields and adds yet more which are needed for window operation, such as the title and number of the window.

In order to fully understand *TWindow*, you need to keep in mind that a window is a *group* and also a *view*.

Types of methods

Turbo Vision methods can be characterized in four (possibly overlapping) ways:

- Abstract methods
- Pseudo-abstract methods
- Virtual methods
- Static methods

Static methods

A static method can't be overridden *per se*. A descendant type can define a method with the same name using entirely different arguments and return types, if necessary, but static methods do not operate polymorphically. This is most critical when you call methods of dynamic objects.

For example, if *PGeneric* is a pointer variable of type *PView*, you can assign pointers of any type from the hierarchy to it. However, when you dereference the variable and call a static method, the method called will always be *TView*'s, since that is the type of the pointer as determined at compile time. *PGeneric^.StaticMethod* is *always* equivalent to *TView.StaticMethod*, even if you have assigned a pointer of some other type to *PGeneric*. An example is *TView.Init*.

Virtual methods

Virtual methods use the **virtual** directive in their prototype declarations. A virtual method can be redefined (overridden) in descendants but the redefined method must itself be virtual and match the original method's header exactly. Virtual methods need not be overridden, but the usual intention is that they will be overridden sooner or later. An example of this is *TView.DataSize*.

Abstract methods

Abstract methods are always virtual methods. In the base object type, an abstract method has an empty body or a body containing the statement *Abstract* set to trap illegal calls. Abstract methods *must* be defined by a descendant before they can be used. Objects with abstract methods are truly abstract—you *must* derive a new type and override the abstract methods before you can create a useful instance of that object type. An example is *TStream.Read*.

Pseudo-abstract methods

Unlike truly abstract methods that generate a run-time error, pseudo-abstract methods offer minimal default actions or no actions at all. They serve as placeholders, where you can insert code in your derived objects.

For example, the *TView* type introduces a virtual method called *Awaken*. Awaken contains no code, as shown in Listing 7.1.

Listing 7.1
A pseudo-abstract method

```
procedure TView.Awaken;
begin
end;
```

By default, *Awaken* therefore does nothing. *Awaken* is called when a group object has finished loading itself from a stream. Once it loads all its subviews, the group calls each subview's *Awaken* method. So if you create a view object that needs to initialize itself when loaded from a stream, you can override *Awaken* to perform that initialization.

Object typology

Not all object types are created equal in Turbo Vision. You can separate their functions into four distinct groups:

- Primitive objects
- Views
- Group views
- Engines

Primitive object types

Turbo Vision provides three simple object types that exist primarily to be used by other objects or to act as the basis of a hierarchy of more complex objects. They are

- *TPoint*
- *TRect*
- *TObject*

TPoint and *TRect* are used by all the visible objects in the Turbo Vision hierarchy. *TObject* is the basis of the hierarchy. Objects of these types are not displayable. *TPoint* is simply a screen-position object (X, Y coordinates). *TRect* sounds like a visible object, but it just supplies upper left, lower right rectangle bounds and several non-display utility methods.

TPoint

This object represents a point. Its fields, *X* and *Y*, define the Cartesian (X,Y) coordinates of a screen position. The point (0,0) is the top left corner of the screen. *X* increases horizontally to the right; *Y* increases vertically downwards. *TPoint* has no methods.

TRect

This object represents a rectangle. Its fields, *A* and *B*, are *TPoint* objects defining the rectangle's upper left and lower right points. *TRect* has methods *Assign, Copy, Move, Grow, Intersect, Union, Contains, Equals,* and *Empty*. *TRect* objects are not visible views and can't draw themselves. However, all views are rectangular: Their *Init* constructors all take a *Bounds* parameter of type *TRect* to determine the region they will cover.

TObject

TObject is an abstract base type with no fields. It is the ancestor of all Turbo Vision objects except *TPoint* and *TRect*. *TObject* provides three methods: *Init, Free,* and *Done*. The constructor, *Init*, forms the base for all Turbo Vision constructors by providing memory allocation. *Free* disposes of this allocation. *Done* is a pseudo-abstract destructor that should be overriden by descendants. Any objects you intend to use with Turbo Vision's streams must be derived ultimately from *TObject*.

TObject's descendants fall into one of two families: views or non-views. Views are descendants of *TView*, which gives them special properties not shared by non-views. Views can draw themselves and handle events sent to them. The non-view objects provide a host of utilities for handling streams and collections of other objects, including views, but they are not directly "viewable."

Views

The displayable descendants of *TObject* are known as *views*, and are derived from *TView*, an immediate descendant of *TObject*. You should distinguish "visible" from "displayable," since there may be times when a view is wholly or partly hidden by other views.

A view is any object that can be displayed in a rectangular portion of the screen. All view objects descend from the type *TView*. *TView* itself is an abstract object representing an empty rectangular screen area. Having *TView* as an ancestor, though, ensures that each derived view has at least a rectangular portion of the screen and a pseudo-abstract *Draw* method that just fills the rectangle with a default color.

Turbo Vision includes the following standard views:

- Frames
- Buttons
- Clusters
- Menus
- Histories
- Input lines
- List viewers
- Scrollers
- Scroll bars
- Text devices
- Static text
- Labels
- Status lines

Frames

TFrame provides the displayable frame (border) for a *TWindow* object together with icons for moving and closing the window. *TFrame* objects are never used on their own, but always in conjunction with a *TWindow* object.

Buttons

A *TButton* object is a titled box used to generate a specific command event when "pushed." They are usually placed inside (owned by) dialog boxes, offering such choices as "OK" or "Cancel." The dialog box is usually the modal view when it appears, so it traps and handles all events, including its button events. The event handler offers several ways of pushing a button:

mouse-clicking in the button's rectangle, typing the shortcut letter, or selecting the default button with the *Enter* key.

Clusters

TCluster is an abstract type used to implement check boxes and radio buttons. A cluster is a group of controls that all respond in the same way. Cluster controls are often associated with *TLabel* objects, letting you select the control by selecting on the adjacent text label.

Radio buttons are special clusters in which only one control can be selected. Each subsequent selection deselects the current one (as with a car radio station selector). Check boxes are clusters in which any number of controls can be marked (selected).

Menus

TMenuView and its two descendants provide the basic objects for creating pull-down menus and submenus nested to any level. You supply text strings for the menu selections (with optional highlighted shortcut letters) together with the commands associated with each selection.

By default, Turbo Vision applications reserve the top line of the screen for a menu bar, from which menu boxes drop down. You can also create menu boxes that pop up in response to mouse clicks. Menus are explained in Chapter 10, "Application objects."

Histories

The abstract type *THistory* implements a generic pick-list mechanism. *THistory* works in conjunction with *THistoryWindow* and *THistoryViewer*. Histories are explained in Chapter 12, "Control objects."

Input lines

TInputLine provides a basic input line string editor. It handles all the usual keyboard entries and cursor movements. It offers deletes and inserts, selectable insert and overwrite modes, and automatic cursor shape control. Input lines are explained in Chapter 12, "Control objects."

Input lines support data validation with validator objects.

List viewers

The *TListViewer* object type is an abstract base type from which to derive list viewers such as *TListBox*. *TListViewer's* fields and methods let you display linked lists of strings with control over one or two scroll bars. *TListBox*, derived from *TListViewer*, implements the most commonly used list boxes, namely those displaying lists of strings such as file names. List viewers and list boxes are explained in Chapter 12, "Control objects."

Scrolling views

A *TScroller* object is a scrollable view that serves as a portal onto another larger "background" view. Scrolling occurs in response to keyboard input or actions in the associated *TScrollBar* objects. Scrollers are explained in Chapter 8, "Views."

Scroll bars

TScrollBar objects provide either vertical or horizontal control. Windows containing scrolling interiors use scroll bars to control the scroll position. List viewers also use scroll bars. Scroll bars are explained in Chapter 12, "Control objects."

Text devices

TTextDevice is a scrollable TTY-type text viewer/device driver. Apart from the fields and methods inherited from *TScroller*, *TTextDevice* defines virtual methods for reading and writing strings from and to the device. *TTextDevice* exists solely as a base type for deriving real terminal drivers. *TTerminal* implements a "dumb" terminal with buffered string reads and writes. It is essentially a text file device driver that writes to a view. Text devices are explained in Chapter 15, "Editor and text views."

Static text

TStaticText objects are simple views used to display fixed strings provided by the field *Text*. They ignore any events sent to them. The *TLabel* type adds the property that the view holding the text, known as a label, can be selected (highlighted) by mouse click, cursor key, or shortcut *Alt*+letter keys. Labels are associated with another view, usually a control view. Selecting the label selects the linked control and selecting the linked control highlights the label as well. Static text and labels are explained in Chapter 12, "Control objects."

Status lines

A *TStatusLine* object is intended for various status and hint displays, usually at the bottom line of the screen. A status line is a one-character high strip of any length up to the screen width. The object offers dynamic displays reacting with events in the unfolding application. Status lines are explained in Chapter 10, "Application objects."

Group views

The importance of *TView* is apparent from the hierarchy chart shown in Figure 7.1. Everything you can see in a Turbo Vision application derives in some way from *TView*. But some of those visible objects are also important for another reason: They allow objects to act in concert.

Turbo Vision includes the following standard group views:

- The abstract group
- Applications
- Desktops
- Windows
- Dialog boxes

The abstract group

TGroup lets you handle dynamically chained lists of related, interacting *subviews* via a designated view called the *owner* of the group. Since a group is a view, there can be subviews that are in turn groups owning their own subviews, and so on. The state of the chain is constantly changing as the user clicks and types during an application. New groups can be created and subviews can be added to (inserted) and deleted from a group. Groups and subviews are explained in Chapter 8, "Views."

Applications

TProgram is an abstract type that provides a set of virtual methods for its descendant, *TApplication*. *TApplication* provides a program template object for your Turbo Vision application. It is a descendant of *TGroup* (via *TProgram*). Typically, it owns *TMenuBar*, *TDesktop* and *TStatusLine* subviews. *TApplication* has methods for creating and inserting these three subviews. The key method of *TApplication* is *TApplication.Run* which executes the application's code. Application objects are explained in Chapter 10, "Application objects."

Desktops

TDesktop is the normal startup background view, providing the familar user's desktop, usually surrounded by a menu bar and status line. Other views (such as windows and dialog boxes) are created, displayed, and manipulated in the desktop in response to user actions (mouse and keyboard events). Most of the actual work in an application goes on inside the desktop. Destop objects are explained in Chapter 10, "Application objects."

Windows

TWindow objects, with help from *TFrame* objects, are the bordered rectanglar displays that you can drag, resize, and hide using methods inherited from *TView*. A window object can also zoom and close itself using its own methods. Numbered windows can be selected with *Alt+n* hot keys. Window objects are explained in Chapter 11, "Window and dialog box objects."

Dialog boxes

TDialog is a descendant of *TWindow* used to create dialog boxes that handle a variety of user interactions. Dialog boxes typically contain controls such as buttons and check boxes. The main difference between dialog boxes and windows is that dialog boxes are specialized for modal operation. Dialog boxes are explained in Chapter 11, "Window and dialog box objects."

Engines

Turbo Vision includes five groups of non-view objects derived from *TObject*:

- Streams
- Resource files
- Collections
- String lists
- Validators

Streams

A stream is a generalized object for handling input and output. In traditional device and file I/O, separate sets of functions must be devised to handle the extraction and conversion of different data types. With Turbo Vision streams, you can create polymorphic I/O methods such as *Read* and *Write* that know how to process their own particular stream contents.

TStream is the base abstract object providing polymorphic I/O to and from a storage device. Turbo Vision also includes a number of specialized streams, including DOS file streams, buffered DOS streams, memory streams, and EMS streams. Streams are explained in Chapter 17, "Streams."

Resources

A resource file is a special kind of stream where generic objects can be indexed via string keys. Rather than derive resource files from *TStream, TResouceFile* has a field, *Stream,* associating a stream with the resource file. Resources are explained in Chapter 18, "Resources."

Collections

TCollection implements a general set of items, including arbitrary objects of different types. Unlike the arrays, sets, and linked lists, Turbo Vision collections allow for dynamic sizing. *TCollection* is an abstract base for more specialized collections. Turbo Vision includes several specialized collection types, including an abstract sorted collection and collections of strings. Collections are explained in detail in Chapter 16, "Collections."

String lists

TStringList implements a special kind of string resource in which strings can be accessed via a numerical index. *TStringList* simplifies internationalization and multilingual text applications. *TStringList* offers access only to existing numerically indexed string lists. *TStrListMaker* supplies the *Put* method for adding a string to a string list, and a *Store* method for saving string lists on a stream.

Validators

TValidator is an abstract validator object that's the basis for a family of objects used to validate the contents of input lines. The useful validators *TFilterValidator, TRangeValidator, TLookupValidator, TStringLookupValidator,* and *TPXPictureValidator* all derive their basic behavior from *TValidator,* but provide different forms of validation. All the validator objects and their use are explained in Chapter 13, "Data validation objects."

Turbo Vision coordinates

Turbo Vision's method of assigning coordinates might be different from what you're used to. Unlike coordinate systems that designate the character spaces on the screen, Turbo Vision coordinates specify the grid *between* the characters. If this seems odd, you'll soon see that the system works very well for specifying the boundaries of view objects.

Specifying points

A point in the coordinate system is designated by its x- and y-coordinates. The *TPoint* object type encapsulates the coordinates in its fields, *X* and *Y*. *TPoint* has no methods, but it makes it easy to deal with both coordinates in a single item.

Specifying boundaries

Every item on a Turbo Vision screen is rectangular, defined by a rectangle object of type *TRect*. *TRect* has two fields, *A* and *B*, each of which is a *TPoint*, with *A* representing the upper left corner and *B* holding the lower right corner. When specifying the boundaries of a view object, you pass those boundaries to the view's constructor in a *TRect* object.

For example, if *R* is a *TRect* object, `R.Assign(0,0,0,0)` designates a rectangle with no size—it is only a point. The smallest rectangle that can actually contain anything would be created with `R.Assign(0,0,1,1)`.

Figure 7.3 shows a *TRect* created by `R.Assign(2,2,5,4)`.

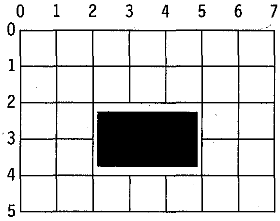


Figure 7.3
Turbo Vision coordinate system

`R.Assign(2,2,5,4)` produces a rectangle that contains six character spaces. This makes it easy to calculate such things as the sizes of rectangles and the coordinates of adjacent rectangles.

Local and global coordinates

In some cases, you have to be aware of *which* coordinate system you're working in. Most of the time, a view only deals with its own local coordinate system, which has its origin at the top left corner of the view. When you place a control in a dialog box, for example, you specify its location relative to the origin of the dialog box. That way, when you move the dialog box, the control moves with the dialog box.

Positional events are explained fully in Chapter 9, "Event-driven programming."

The only time you have to worry about any other coordinate system is when handling *positional events* such as mouse clicks. Mouse clicks are handled by the application, and it records the position of the click in the global coordinate system for the application. The origin for global coordinates is the top left corner of the screen. By determining where on the screen the user clicked the mouse, the application can decide which view on the screen should respond to the event.

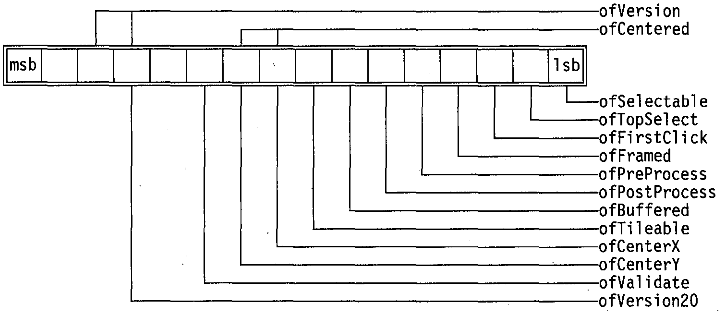
When a view needs to respond to such an event, it has to convert from global coordinates to local coordinates. Every view inherits a method called *MakeLocal* that converts a point from global screen coordinates to local view coordinates. If necessary, it can also convert from local to global coordinates, using another method, *MakeGlobal.*

Using bitmapped fields

Turbo Vision's views use several fields which are *bitmapped.* That is, they use the individual bits of a byte or word to indicate different properties. The individual bits are usually called *flags,* since by being set (equal to 1) or cleared (equal to 0), they indicate whether the designated property is activated.

For example, each view has a bitmapped *Word*-type field called *Options.* Each of the individual bits in the word has a different meaning to Turbo Vision. Figure 7.4 shows the definitions of the bits in the *Options* word.

Figure 7.4
Options bit flags



Flag values

In Figure 7.4, *msb* indicates the "most significant bit," also called the "high-order bit" because in constructing a binary number, that bit has the highest value (2^{15}). The bit at the lowest end of the binary number is marked *lsb*, for "least significant bit," also called the "low-order bit."

So, for example, the fourth bit is called *ofFramed*. If the *ofFramed* bit is set to 1, it means the view has a visible frame around it. If the bit is a 0, the view has no frame.

You generally don't have to worry about what the values of the flag bits are unless you plan to define your own, and even in that case, you only need to make sure that your definitions are unique. The highest-order bits in the *Options* word are presently undefined by Turbo Vision.

Bit masks

A *mask* is a convenient way of dealing with a group of bit flags together. For example, Turbo Vision defines masks for different kinds of events. The *evMouse* mask simply contains all four bits that designate different kinds of mouse events, so if a view needs to check for mouse events, it can compare the event type to see if it's in the mask, rather than having to check for each of the individual kinds of mouse events.

Bitwise operations

Turbo Pascal provides quite a number of useful operations to manipulate individual bits. Rather than giving a detailed explanation of *how* each one works, this section will simply tell you what to do to get the job done.

Setting bits

To set a bit, use the **or** operator. For instance, to set the *ofPostProcess* bit in the *Options* field of a button called *MyButton*, you use this code:

```
MyButton.Options := MyButton.Options or ofPostProcess;
```

You should *not* use addition to set bits unless you are absolutely sure what you are doing. For example, if instead of the preceding code, you used

***Don't** do this!*

```
MyButton.Options := MyButton.Options + ofPostProcess;
```

your operation would work *if and only if* the *ofPostProcess* bit was not already set. If the bit was set before you added another one, the binary add would carry over into the next bit (*ofBuffered*), setting or clearing it, depending on whether it was clear or set to start with.

☞ In other words: *adding* bits can have unwanted side effects. Use the **or** operation to set bits instead.

Before leaving the topic of setting bits, note that you can set several bits in one operation by **or**ing the field with several bits at once. The following code sets two different grow mode flags at once in a scrolling view called *MyScroller*:

```
MyScroller.GrowMode := MyScroller.GrowMode or gfGrowHiX or gfGrowHiY;
```

Clearing bits

Clearing a bit is just as easy as setting it. You just use a different operation. The best way to do this is a combination of two bitwise operations, **and** and **not**. For instance, to clear the *dmLimitLoX* bit in the *DragMode* field of a label called *ALabel*, you use

```
ALabel.DragMode := ALabel.DragMode and not dmLimitLoX;
```

As with setting bits, multiple bits can be set in a single operation.

Toggling bits

Sometimes you'll want to toggle a bit, meaning set it if it's clear and clear it if it's set. To do this, use the **xor** operator. For example, to toggle the horizontal centering of a dialog box *ADialog* on the desktop, toggle the *ofCenterX* flag like this:

```
ADialog.Options := ADialog.Options xor ofCenterX;
```

Checking bits

Quite often, a view will want to check to see if a certain flag bit is set. This uses the **and** operation. For example, to see if the window *AWindow* may be tiled by the desktop, you need to check the *ofTileable* option flag like this:

```
if AWindow.Options and ofTileable = ofTileable then ...
```

Using masks

Much like checking individual bits, you can use **and** to check to see if one or more masked bits are set. For example, to see if an event record contains some sort of mouse event, check

```
if Event.What and evMouse <> 0 then ...
```

Summary

Table 7.2 summarizes the bitmap operations:

Table 7.2 Manipulating bitmapped fields

To do this	**Use this code**
Set a bit	`field := field or flag;`
Clear a bit	`field := field and not flag;`
Toggle a bit	`field := field xor flag;`
Check if a flag is set	`if field and flag = flag then ...`
Check for a flag in a mask	`if flag and mask <> 0 then ...`

CHAPTER

8

Views

One of the keys to Turbo Vision is the system used to present information on the screen, using *views*. Views are objects that represent rectangular regions on the screen, and they are the only way Turbo Vision applications display information to users.

In this chapter, you'll learn the following:

- What is a view?
- What is a group?
- How to use views
- How to use groups

Because views are objects, they all inherit their basic properties from a common ancestor object type, *TView*. Turbo Vision also defines specialized objects descended from *TView*, such as windows, dialog boxes, applications, desktops, menus, and so on.

Other chapters in this part of the manual describe how to use these specific views, but this chapter focuses on the principles common to all views.

What is a view?

Unlike Pascal programs you're probably used to, Turbo Vision applications don't generally use *Write* and *Writeln* statements to display information to the screen. Instead, Turbo Vision applications use views, which are objects that know how to represent themselves on the screen.

Definition of a view

The basic building block of a Turbo Vision application is the *view*. A view is a Pascal object that manages a rectangular area of the screen. For example, the menu bar at the top of the screen is a view. Any program action in that area of the screen (for example, clicking the mouse on the menu bar) will be dealt with by the view that controls that area.

In general, anything that shows up on the screen of a Turbo Vision program *must* be a view, which means it is a descendant of the object type *TView*. There are three things that all views must do:

- Manage a rectangular region
- Draw itself on demand
- Handle events in its boundaries

The standard views provided with Turbo Vision handle these things automatically, and the views you create for your applications will either inherit these abilities or you'll have to add them to your objects. Let's look at each of these properties in more detail.

Defining a region

When you construct a view object, you assign it boundaries, usually in the form of a rectangle object of type *TRect*. Boundary rectangles and the Turbo Vision coordinate system are explained in detail in Chapter 11, starting on page 107, but it's important when you think about the other two properties of a view that you remember that a view is limited to the area defined by its boundaries.

Drawing on demand

The most important visual property of a view is that it knows how to represent itself on the screen. For example, when you want to put a menu bar across the top of the application screen, you construct a menu bar view, giving it the boundaries of the top line of the screen and defining for it a list of menu items. The menu bar view knows how to represent those items in the designated space.

You don't have to concern yourself with *when* the view appears. You define a virtual method for the view called *Draw* that fills in the entire area within its bounding rectangle. Turbo Vision calls *Draw* when it knows that the view needs to show itself, such as when a window is uncovered because the window in front of it closes.

The two important things to remember about *Draw* methods are these:

- The view must fill its entire rectangle.
- The view must be able to draw itself at any time.

Draw methods are explained completely starting on page 119.

Handling events

The third property of any view object is that it must handle events that occur inside its boundaries, such as mouse clicks and keystrokes. Event handling is explained in detail in Chapter 9, "Event-driven programming," but for now just remember that a view is responsible for any events within its boundaries, just as it must draw everything within its boundaries.

What is a group?

Sometimes the easiest way for a view to manage its area is to delegate certain parts of the job to other views, known as *subviews*. A view that has subviews is called a *group*. Any view can be a subview, but groups must be descendants of *TGroup*, which is itself a descendant of *TView*. A group with subviews is said to *own* the subviews, because it manages those subviews. Each subview is said to have an *owner view*, which is the group that owns it.

The most visible example of a group view, but one you might not ordinarily think of as a view, is the application itself. It controls the entire screen, but you don't notice that because the program sets up three other subviews—the menu bar, the status line, and the desktop—to handle its interactions with the user. As you will see, what appears to the user as a single object (like a window) is often a group of related views.

Delegating to subviews

Since a group is a view, all the normal rules of views still apply. A group covers a rectangle, draws itself on demand, and handles events within its boundaries. The main difference with groups is that they handle most of their tasks by delegating them to subviews.

For example, the *Draw* method of a group generally doesn't draw anything itself, but instead calls on each of the group's subviews in turn to draw itself. The result of the *Draw* methods of all the subviews, therefore, must result in covering the group's entire rectangle.

Using view objects

All Turbo Vision views have *TObject* as an ancestor. *TObject* is little more than a common ancestor for all the objects, ensuring that all the objects can operate polymorphically with streams, for example. The visible parts of Turbo Vision start with *TView*.

TView itself is an abstract object type that serves as a common ancestor for all the views. There is little reason to create an instance of *TView* unless you want to create a blank rectangle on the screen for prototyping purposes. Although *TView* is visually simple, it contains all of Turbo Vision's basic screen management methods and fields. This section describes the following tasks you'll need to perform on views:

- Constructing view objects
- Managing view boundaries
- Drawing the view
- Handling the cursor
- Setting state flags
- Validating the view

Constructing view objects

Although you will probably never construct an instance of *TView*, all view objects, which descend from *TView*, call the *TView* constructor as part of their constructors, so it's important that you understand what the constructor does.

By convention, all Turbo Vision objects' constructors are called *Init*. *TView*'s *Init* constructor takes a single parameter, the bounding rectangle of the view:

```
constructor TView.Init(var Bounds: TRect);
```

Calling the inherited constructor

Before doing anything else, *TView.Init* calls the *Init* constructor it inherits from *TObject*, which fills all fields of the view with zeros. Since all other views' constructors eventually result in a call to *TObject.Init*, be sure you don't initialize any fields before calling the inherited constructor.

Init takes the *Bounds* parameter passed to it and sets two important fields based on it: *Origin* and *Size*. *Origin* is the upper right corner of the bounding rectangle. *Size* holds the width and height of the rectangle.

Managing view boundaries

The location of a view is determined by two points: its top left corner (called its origin) and its bottom right corner. Each of these points is represented in the object by a field of type *TPoint*. *Origin* indicates the top left corner of the view, and *Size* represents the position of the lower right corner relative to *Origin*.

Origin is a point in the coordinate system of the owner view. If you open a window on the desktop, its *Origin* field indicates the x- and y-coordinates of the window relative to the origin of the desktop. The *Size* field, on the other hand, is a point relative to the origin of its own object. It tells you how far the lower right corner is from the origin point, but unless you know where the view's origin is located within another view, you can't tell where that corner really is.

Once you've constructed a view, there are a number of methods for manipulating the boundaries of the view. In particular, you can do the following:

- Get the view's coordinates
- Move the view
- Resize the view

Getting the view's coordinates

You'll find that there are many times when you need to get the boundaries of a view, either because you want to change those boundaries, or because you want to construct another view based on those boundaries. *TView* has a method called *GetExtent* that takes a single **var** parameter of type *TRect* and sets the rectangle to the boundaries of the view.

For example, Listing 8.1 shows an application object method that constructs and inserts a window that covers the left half of the desktop view.

Listing 8.1
Constructing a view based on the size of another

```
procedure TYourApplication.AddLeftWindow;
var
  R: TRect;
  LeftWindow: PWindow;
begin
  Desktop^.GetExtent(R);                  { get coordinates of desktop }
  R.B.X := R.B.X div 2;               { move right side halfway to left }
  LeftWindow := New(PWindow, Init(R,             { use R as window size }
    'Left window', wnNoNumber));        { give window title and number }
  InsertWindow(LeftWindow);           { insert left window into desktop }
end;
```

The rectangle set by *GetExtent* always has its *A* field set to the point (0,0), and *B* set to the size of the view. In other words, *GetExtent* returns the view's coordinates in its own local coordinate system.

To get the view's coordinates relative to its owner view, use the method *GetBounds* instead of *GetExtent*. *GetBounds* returns the view's coordinates in the owner view's coordinate system, setting the *A* field of its parameter to the view's *Origin* field, and *B* to the size of the view offset from the origin.

Moving a view

To change the position of a view without affecting its size, call the view's *MoveTo* method. *MoveTo* takes two parameters, the x- and y-coordinates of the new origin of the view. For example, the following statement moves a view two spaces to the left and one space down:

```
MoveTo(Origin.X - 2, Origin.Y + 1);
```

Resizing a view

To change the size of a view without moving it (that is, without changing the position of its upper left corner), you call the view's *GrowTo* method. *GrowTo* takes two parameters, which determine the x- and y-coordinates of the bottom right corner of the view, relative to the origin.

For example, the following code causes a view to double both its width and height:

```
GrowTo(Size.X, Size.Y);
```

Moving and resizing at the same time

To set the size and position of a view in a single step, you call the view's *Locate* method. *Locate* takes a rectangle as its single parameter, setting that rectangle as the boundary of the view.

For example, the following code sets the boundaries of a view to the given rectangle, regardless of the original size and position of the view:

```
R.Assign(1, 3, 27, 6);
Locate(R);
```

Fitting views into owners

One of the most common manipulations of a view's coordinates involves fitting one view into another. For example, creating the interior of a window involves making sure the interior doesn't cover any part of the window's frame. To do that, you assign the interior's boundaries relative to the window, without having to worry about the actual size and position of the window.

Grow is a *TRect* method that increases (or with negative parameters, decreases) the horizontal and vertical sizes of a rectangle. Used in conjunction with a view's *GetExtent* method, *Grow* makes it easy to fit one view into another, as shown in Listing 8.2. The types *TThisWindow* and *PInsideView* are made up just for this example.

Listing 8.2
Fitting a view inside another

```
procedure TThisWindow.MakeInside;
var
  R: TRect;
  Inside: PInsideView;
begin
  GetExtent(R);               { sets R to boundaries of TThisWindow }
  R.Grow(-1, -1);            { shrinks the rectangle by 1, both ways }
  Inside := New(PInsideView, Init(R));         { creates inside view }
  Insert(Inside);             { insert the new view into the window }
end;
```

Drawing a view

The appearance of a view object is determined by its *Draw* method. Nearly every new type of view will need to have its own *Draw*, since it is, generally, the appearance of a view that distinguishes it from other views.

There are a couple of rules that apply to all views with respect to appearance. A view must

- Cover the entire area for which it is responsible
- Be able to draw itself at any time

Both of these properties are very important and deserve some discussion. For information on actually writing *Draw* methods, see the section "Writing Draw methods," starting on page 128.

Drawing on demand

In addition, a view must *always* be able to represent itself on the screen. That's because other views may cover part of it but then be removed, or the view itself might move. In any case, when called upon to do so, a view must always know enough about its present state to show itself properly.

Note that this might mean that the view does nothing at all. It might be entirely covered, or it might not even be on the screen, or the window that holds it might have shrunk to the point that the view is not visible at all. Most of these situations are handled automatically, but it is important to remember that your view must always know how to draw itself.

This is different from other windowing schemes, where the writing on a window, for example, is persistent: You write it there and it stays, even if something covers it up then moves away. In Turbo Vision, you can't assume that a view you uncover is displayed correctly—after all, something may have told it to change while it was covered.

Changing view option flags

All views inherit four fields from *TView* that contain bitmapped information. That is, each bit in each field has a special meaning, setting some option in the view. You can think of each bit as being a Boolean value, but stored in a much more compact form.

To learn how to manipulate bitmapped options, see "Using bitmapped fields" in Chapter 7, "Turbo Vision overview."

The values of these option flags get set once, when you first construct the view, and normally stay set, although you can change the values at any time. For complete information on each flag, you should check Chapter 19, the entries for *ofXXXX* constants, *dmXXXX* constants, and *gfXXXX* constants.

Options is a bitmapped word in every view. Various descendants of *TView* have different *Options* set by default. The *GrowMode* and *DragMode* flags, although present in every view, don't take effect until the view gets inserted in a group, so they are explained in the part of this chapter on groups. The fourth field, *EventMask*, is described in Chapter 9, "Event-driven programming."

Customizing selection

The *Options* word has three bits that govern selection of the view by the user: *ofSelectable*, *ofTopSelect*, and *ofFirstClick*.

Most views have *ofSelectable* set by default, meaning the user can select the view with the mouse. If the view is in a group, the user can also select it with the *Tab* key. You might not want the user to select purely informational views, so you can clear their *ofSelectable* bits. Static text objects and window frames, for example, are not selectable by default.

The *ofTopSelect* bit, if set, causes the view to move to the top of the owner's subviews when selected. This option is designed primarily for windows on the desktop, so don't use it for views in a group.

The *ofFirstClick* bit controls whether the mouse click that selects the view is also passed to the view for processing. For example, if the user clicks a button, you want to both select the button *and* press it with just one click, so buttons have *ofFirstClick* set by default. But if the user clicks on an inactive window, you probably only want to select the window and not process the click as an action on the window once it's activated. This makes it less likely that a user will accidentally close or zoom a window when just trying to activate it.

Framing the view

If you set the *ofFramed* bit, the view has a visible frame around it. This is useful if you create multiple "panes" within a window, or if you want to emphasize a particular view. *ofFramed* does not affect the frame of window and dialog box objects. Those are separate views controlled by a field in the window object. The *ofFramed* bit only affects views inserted into windows or dialog boxes.

Special event handling

The bits *ofPreProcess* and *ofPostProcess* allow a view to process focused events before or after the focused view sees them. The "Phase" section in Chapter 9, "Event-driven programming," explains how to use these bits.

Centering the view

Views have two bits that control the centering of the view within its owner. The *ofCenterX* bit centers the view horizontally, and *ofCenterY* centers it vertically. If you want to center both horizontally and vertically, you can use the mask *ofCentered*, which contains both of the centering bits.

Setting the view's state

Every view maintains a bitmapped field of type *Word* called *State*, which contains information on the state of the view. Unlike option flags and mode bits, which you set when you construct a view (if a window is resizable, it is *always* resizable, for example), state flags often change during the lifetime of a view as the state of the view changes. State information includes whether the view is visible, has a cursor or shadow, is being dragged, or has the input focus.

The meaning of each state flag is covered in Chapter 19, "Turbo Vision reference," under "*sfXXXX* state flag constants." This section focuses on the mechanics of manipulating the *State* field.

Rather than manipulate state flags directly, you use a method called *SetState*, which involves two separate kinds of activities:

- Setting or clearing state flags
- Responding to state changes

Setting and clearing state flags

For the most part, you don't need to change state bits manually, since the most common state changes are handled by other methods. For example, the *sfCursorVis* bit controls whether the view has a visible text cursor. Rather than manipulating that bit directly, you can call either *ShowCursor* or *HideCursor*, which take care of toggling the *sfCursorVis* bit for you. Table 8.1 shows the state flags and the methods that manipulate them.

Table 8.1
Methods that change state flags

State flag(s)	Methods
sfVisible	*Show, Hide*
sfCursorVis	*ShowCursor, HideCursor*
sfCursorIns	*BlockCursor, NormalCursor*
sfShadow	None
sfActive, sfSelected, sfFocused	*Select*
sfDragging	*DragView*
sfModal	*Execute*
sfExposed	*TGroup.Insert*

SetState works on only one bit at a time.

In order to change a state flag that doesn't have a specific method dedicated to it, you need to call the view's *SetState* method, passing two parameters: the bit to change, and a Boolean flag indicating whether to set the bit. For example, to set the *sfShadow* flag, you'd do the following:

```
SetState(sfShadow, True);
```

Responding to state changes

Whenever a view gets the focus, gives up the focus, or becomes selected, Turbo Vision calls *SetState* to change the appropriate state flags. But changing state flags often requires that the view make some other changes in response to the new state, such as redrawing the view. If you want a view to respond in some special way to a state change, you need to override *SetState*, calling the inherited *SetState* to make sure the change occurs, then responding to the new state.

A button, for example, watches *State* and changes its color to cyan when it gets the focus. Listing 8.3 shows how *TButton* overrides *SetState*:

Listing 8.3 Overriding SetState to respond to state changes

```
procedure TButton.SetState(AState: Word; Enable: Boolean);
begin
  inherited SetState(AState, Enable);        { set/clear state bits }
  if AState and (sfSelected + sfActive) <> 0 then DrawView;
  if AState and sfFocused <> 0 then MakeDefault(Enable);
end;
```

You should always call the inherited *SetState* from within a new *SetState* method, because *TView.SetState* does the actual setting or clearing of the state flags. You can then define any special actions based on the state of the view. *TButton* checks to see if it is in an active window in order to decide whether to draw itself. It also checks to see if it has the focus, in which case it calls its *MakeDefault* method, which grabs or releases the focus, depending on the *Enable* parameter.

The programmer and Turbo Vision often cooperate when the state changes. For example, *TEditor* toggles the state of the cursor when the user enters or leaves insertion mode. In response to the user's pressing *Ins*, the editor calls a private method called *ToggleInsMode* that in turn calls *SetState*:

```
procedure TEditor.ToggleInsMode;
begin
  Overwrite := not Overwrite;                { toggle Overwrite mode }
  SetState(sfCursorIns, not GetState(sfCursorIns)); { toggle cursor }
end;
```

Dragging a view

One way to move a view is to let the user position or resize it with a mouse. Moving a view with the mouse is called *dragging*. Each view has a bitmapped field called *DragMode* that provides the default limits to where the user can drag the view. To drag views, you need to understand two tasks:

- Setting drag limits
- Calling *DragView*

Setting drag limits

The bits in *DragMode* determine whether parts of the view can move outside its owner. When you drag some views, such as windows on the desktop, moving the view beyond the boundary of the owner just causes the subview to be clipped at the owner's boundary. In other words, it can move there, even if you can't see it. The bits with names starting with *dmLimit* restrict a view from being dragged outside its owner.

The mask *dmLimitAll* contains all the drag mode limit bits. Setting *dmLimitAll* in a view means that the user won't be able to drag any part of the view outside its owner. The individual bits are *dmLimitLoX*, *dmLimitLoY*, *dmLimitHiX*, and *dmLimitHiY*, which restrict dragging beyond the left, top, right, and bottom boundaries of the owner, respectively. By default, views have *dmLimitLoY* set, meaning the user can't drag the top of a view beyond the top of its owner.

Calling DragView

The actual dragging of the view is handled by a method called *DragView*. Normally a view calls *DragView* in response to a mouse click. For example, when you click on the top bar of a window frame, the window calls *DragView* to move the window. Similarly, when you click on the bottom right corner of a window, it also calls *DragView*, this time to resize the window.

DragView takes five parameters. The first is the event record that initiated the dragging (usually a mouse down event). The second is the dragging mode, which is a combination of either *dmDragMove* or *dmDragGrow* with the limit flags in the view's *DragMode* field. The three remaining parameters provide a rectangle in which the view is allowed to move and the minimum and maximum sizes for the view.

The code in Listing 8.4, from the example program DRAGS.PAS, shows a typical use of *DragView*.

Listing 8.4
A typical use of DragView

```
procedure TDragBlock.HandleEvent(var Event: TEvent);
var
  R: TRect;
  Min, Max: TPoint;
begin
  inherited HandleEvent(Event);
  if Event.What and evMouseDown = evMouseDown then
  begin
    if Event.Double then ChangeFlags
    else
    begin
      Owner^.GetExtent(R);
      R.Grow(-1, -1);
      SizeLimits(Min, Max);
      case Event.Buttons of
        mbLeftButton:
          begin
            DragView(Event, dmDragMove or DragMode, R, Min, Max);
            ClearEvent(Event);
          end;
        mbRightButton:
          begin
            DragView(Event, dmDragGrow or DragMode, R, Min, Max);
            ClearEvent(Event);
          end;
      end;
    end;
  end;
end;
```

Handling the cursor

Any visible view can have a cursor, although the cursor only shows up when the view has the input focus. The cursor provides a visual indication to the user of where keyboard input will go, but it is up to the programmer to make sure the program actually matches the cursor position to the input location.

TView has a field called *Cursor*, of type *TPoint*, that indicates the position of the cursor within the view, relative to the origin of the view. Views have several methods devoted to handling the cursor, which enable you to do the following:

- Show or hide the cursor
- Change the cursor style
- Move the cursor

Showing and hiding the cursor

Views have two methods, *ShowCursor* and *HideCursor*, which handle showing and hiding the text cursor, respectively. By default, the cursor is hidden, although some descendants of *TView* (notably input lines and editors) override this and show their cursors by default.

One of the bits in every view's *State* field (*sfCursorVis*) controls whether the view has a visible cursor. *ShowCursor* and *HideCursor* set and clear the *sfCursorVis* bit. When the view gets the input focus, Turbo Vision shows the cursor at the position indicated by *Cursor* if *sfCursorVis* is set.

Changing the cursor style

Turbo Vision supports two styles of text cursor, an underline character and a solid block. The *TView* methods *NormalCursor* and *BlockCursor* set the cursor style to underline or block, respectively. One style usually indicates an insert mode, the other a typeover mode.

By default, the cursor style is normal, or underline. The *sfCursorIns* bit in the view's *State* word controls which style cursor the view uses. *BlockCursor* and *NormalCursor* set and clear the *sfCursorIns* bit.

Moving the cursor

To change the position of the text cursor in a view, you call the view's *SetCursor* method. *SetCursor* takes two parameters, representing the x- and y-coordinates of the new position for the cursor, relative to the origin of the view.

Avoid modifying the *Cursor* field directly. Instead, use *SetCursor*, which changes the cursor location and also updates the display.

Validating a view

Every view has a virtual method called *Valid* that takes a command constant as its one parameter and returns a Boolean value. In general, calling *Valid* is a way of querying the view, asking "If I sent you this command, would you approve?" If *Valid* returns *True*, it's saying that it is valid for that command.

Valid is used for three different kinds of validation, although you can override it to perform other sorts of operations. This section covers the following uses of *Valid*:

- Checking for proper construction
- Checking for safe closing
- Data validation

Checking view construction

Turbo Vision reserves a special command *cmValid*, which it uses to ensure that views construct themselves correctly. The application object method *ValidView* calls a view's *Valid* method, passing *cmValid* as the parameter. Views should respond to such calls by making sure that anything done during construction, such as memory allocation, succeeded.

For example, the constructor for an input line view takes a maximum length for the text string as one of its parameters, and tries to allocate memory on the heap to hold a string of that many characters. The constructor itself doesn't check to make sure the allocation succeeded, but relies on *Valid* to make that determination.

The input line's *Valid* method checks to see if the parameter passed was *cmValid*, and returns *True* only if the allocation for its data buffer succeeded.

Checking for safe closing

Other than the initial test to make sure a view constructed properly, the most common time to check *Valid* is when closing a view. For example, when you call a window object's *Close* method, it calls *Valid*, passing *cmClose*, to make sure it's safe to close the window. Essentially, *Valid(cmClose)* asks the view, "If I told you to close now, would that be all right?" If *Valid* returns *False*, the view should not close.

When passed *cmClose*, therefore, *Valid* methods should ensure that information is saved, buffers are flushed, and so on. A file editor view, for example, should check to make sure that any changes have been saved to the file before *Valid* returns *True.*

When writing *Valid* methods, you have two options when you detect a reason why the view is not valid: have *Valid* return *False,* or perform some action that makes the view valid and then return *True.* For example, when a file editor with unsaved changes is asked to validate on closing, it puts up a dialog box that asks the user whether to save the changes. The user then has three options: save changes and close, abandon changes and close, or don't close. In the first two cases, *Valid* returns *True,* in the third, *False.*

Data validation

Input line views can use *Valid* to determine whether the contents of the text string contain legal values by checking with validator objects. Chapter 13, "Data validation objects," explains this mechanism in detail. The important thing to note here is that data validation can take place when the user closes a window, but you can use the exact same mechanism to validate at any other time.

For example, input line objects check the validity of their contents when *Valid* is called with the *cmClose* command. But you can just as easily check the input as the user types it, calling *Valid(cmClose)* after each keystroke. That's essentially asking the input line, "If I told you to close now, would your contents be valid?" This is precisely the method used by a number of the validator objects described in Chapter 13.

Writing Draw methods

The appearance of any view is determined by its *Draw* method. When you write *Draw* methods, you need to keep in mind the principles outlined in the "Drawing a view" section that starts on page 119. Turbo Vision and its views provide several tools you can use to write a view's information to the screen.

Writing *Draw* methods involves the following tasks:

- Selecting colors
- Writing directly to the view
- Writing through buffers

Selecting colors

When you write data to the screen in Turbo Vision, you don't specify the color of an item directly, but rather rely on the entries in the view's color palette. Palettes and color selection are described in detail in Chapter 14, "Palettes and color selection," but this section covers a few of the basics.

When you specify a color to a function that writes directly to a view, you'll pass it an index into its color palette.

So, for example, if a view has two kinds of text in it, normal and highlighted, it's palette probably has two entries, one for normal text and one for highlighted text. In the *Draw* method, you'd pass the appropriate index to *GetColor* depending on the attribute you want. Listing 8.5 shows a simple draw method that writes two strings in a view in different colors.

Listing 8.5
A simple Draw method with two colors of text

```
procedure TColorView.Draw;
begin
  WriteStr(0, 0, 'Normal', 1);     { write 'Normal' in normal color }
  WriteStr(1, 0, 'Hilite', 2);  { write 'Hilite' in highlight color }
end;
```

Writing directly

Views have two similar methods for writing characters and strings to the view. In each case, you specify the coordinates within the view where the text should start, the text to display, and the palette index of the text color.

Writing characters

The *WriteChar* method takes five parameters: the x- and y-coordinates of the first character to write, the character, the palette index of the desired color, and the number of consecutive characters to write. For example, the following code fills the third line of a view with the letter W in the color specified by the second palette entry:

```
WriteChar(0, 2, 'W', 2, Size.X);
```

Writing strings

The *WriteStr* method takes four parameters: the x- and y-coordinates of the first character, the string to write, and the palette index for the string's color. For example, the following code writes the string 'Turbo Vision' in the lower left corner of a view, in the color specified by the third palette entry:

```
WriteStr(0, Size.Y - 1, 'Turbo Vision', 3);
```

Writing through buffers

The most efficient way to handle drawing large or complex views is to write the text to a buffer, then display the buffer all at once. Using buffers improves the speed of drawing, and reduces flicker cause by large numbers of individual writes to the screen. You'll usually use the buffer to write entire lines or entire views all at once.

A buffer for drawing is an array of words, with each word representing a character and its color attribute, the same way the video screen represents each character. The type *TDrawBuffer*, defined in the *Views* unit, provides a convenient array of words you can use for draw buffers.

Drawing with a buffer takes two steps:

- Setting the text color
- Moving text into the buffer
- Writing the buffer to the screen

Setting the text color

When writing to a buffer, you need to pass a color attribute for the text you're putting in the buffer. To obtain the color attribute, you call the *GetColor* method. *GetColor* returns a color attribute when passed a palette entry number. For example, to get the color attribute for the third entry in a view's palette, do the following:

```
ColorAttribute := GetColor(3);
```

GetColor and color mapping are explained in more detail in Chapter 14, "Palettes and color selection."

Moving text into buffers

There are four procedures in the *Drivers* unit you can use to put text into a draw buffer: *MoveBuf*, *MoveChar*, *MoveCStr*, and *MoveStr*. Each works in much the same way, but each moves different kinds of text into the buffer.

In general, you want to fill the buffer with spaces, then move text into the places where you want it, assuring that you don't leave gaps in the buffer. Listing 8.6 shows two uses of procedures that move text into a buffer.

Listing 8.6
A Draw method that uses a text buffer

```
procedure TCountView.Draw;
var
  B: TDrawBuffer;
  C, Start: Word;
  Params: array[0..1] of Longint;
  First: String[10];
  Display: String[20];
begin
  C := GetColor(2);                          { Uses same color as frame }
  MoveChar(B, '″', C, Size.X);                     { fill buffer with = }
  Params[0] := Current;
  Params[1] := Count;
  FormatStr(Display, ' ~%d~ of %d ', Params);            { format string }
     { If Current greater than Count, display Current as highlighted }
  if Current > Count then C := GetColor($0504)
  else C := GetColor($0202);
  MoveCStr(B, Display, C);                  { move string into buffer }
  WriteLine(0, 0, Size.X, Length(Display), B);       { write string }
end;
```

Writing buffers to the screen

Views have two methods that copy a draw buffer to the screen: *WriteBuf* and *WriteLine*. Both take the same five parameters: the x- and y-coordinates of the upper left corner of the area to write to, the width of that region, the height of the region, and the buffer that contains the text to write.

The difference between *WriteBuf* and *WriteLine* is that *WriteLine* assumes that everything in the buffer is a single line, while *WriteBuf* wraps the buffer around to multiple lines if it exceeds the width of the writing region. If the height of the writing region is greater than 1, *WriteLine* copies the beginning of the same text to each line; *WriteBuf* writes continuously from the buffer.

For example, if a buffer called *Buffer* contains the characters 'ABCDEFGHIJ', the statement

```
WriteLine(0, 0, 5, 2, Buffer);
```

produces this text:

```
ABCDE
ABCDE
```

On the other hand, using *WriteBuf*, the equivalent statement

```
WriteBuf(0, 0, 5, 2, Buffer);
```

produces this output:

```
ABCDE
FGHIJ
```

Using group objects

You've already learned something about the most important immediate descendant of *TView*, the *TGroup*. *TGroup* and its descendants are collectively referred to as *groups*. Views not descended from *TGroup* are called *terminal* views.

Basically a group is just an empty box that contains and manages other views. Technically, it is a view, and therefore responsible for all the things that any view must be able to do: manage a rectangular area of the screen, visually represent itself at any time, and handle events in its screen region. The difference is really in *how* it accomplishes these things. Most of it is handled by *subviews*.

This section covers the following topics regarding group views:

- Groups, subviews, and owners
- Inserting subviews
- Understanding subviews
- Selecting and focusing subviews
- Groups and option flags
- Drawing groups
- Executing modal groups
- Managing subviews

Although you need to understand them, you should never need to change the basic behavior of groups, such as inserting, drawing, and executing. Most of that behavior is simple and straightforward. You will certainly find yourself changing some properties of some descendants of *TGroup*, such as *TWindow* and *TApplication*, but you should never need to change basic group methods.

For example, it might not be apparent, but the processes of adding a menu bar to an application, a window to the desktop, and a control to a dialog box are *exactly* the same. In each case, you're inserting a subview into a group, executing the same code inherited from *TGroup*.

Groups, subviews, and owners

A group is a holder for other views. You can think of a group as a composite view. Instead of handling all its responsibilities itself, it divides its duties among various *subviews*. A subview is a view that is owned by another view, and the group that owns it is called the owner view.

An excellent example is *TApplication*. *TApplication* is a view that controls a region of the screen—the whole screen, in fact. *TApplication* is also a group that owns three subviews: the menu bar, the desktop, and the status line. The application delegates a region of the screen to each of these subviews. The menu bar gets the top line, the status line gets the bottom line, and the desktop gets all the lines in between. Figure 8.1 shows a typical *TApplication* screen.

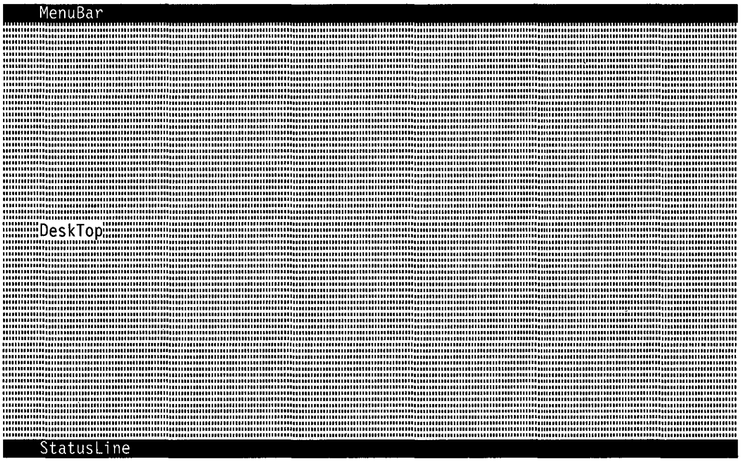


Figure 8.1
TApplication screen layout

Notice that the application itself has no screen representation—you don't *see* the application. Its appearance is entirely determined by the views it owns.

Inserting subviews

To attach a subview to an owner, you *insert* the subview into the owner using a method called *Insert.* Any group can have any number of subviews, and any view can be a subview. The owner treats its subviews as a linked list of views, keeping a pointer to only one subview, using the *Next* field in each subview to point to the next subview.

At a minimum, the subviews of a group must cover the entire area of the group's boundaries. There are two ways to handle this:

- Dividing the group
- Providing a background

In general, the first approach is used in cases where subviews don't overlap, such as the application or a window divided into separate panes. The background method is used in cases where subviews need to overlap and move, such as the desktop, or cases where the important subviews are separated, such as the controls in a dialog box.

The following sections explain each of these cases.

Dividing the group area

Some groups, such as the application object, just divide their rectangular region into parts and permanently assign views to each part.

To create the screen shown in Figure 8.1, the constructor *TApplication.Init* creates three objects and inserts them into the application:

```
InitDeskTop;
InitStatusLine;
InitMenuBar;
if DeskTop <> nil then Insert(DeskTop);
if StatusLine <> nil then Insert(StatusLine);
if MenuBar <> nil then Insert(MenuBar);
```

Application objects and their subviews are explained in Chapter 10, "Application objects."

Only when they have been inserted are the newly created views part of the group. In this particular case, *TApplication* has divided its region into three separate pieces and delegated one to each of its subviews. This makes the visual representation fairly straightforward, as the subviews don't overlap at all.

Providing a background

There is no reason that views can't overlap. Indeed, one of the big advantages of a windowed environment is the ability to have multiple, overlapping windows on the desktop. Luckily, groups (including the desktop) know how to handle overlapping subviews.

The desktop and its background are explained in Chapter 10, "Application objects," starting on page 182.

The basic idea of a background is to assure that *something* is drawn over the entire area of the group, letting other subviews cover only the particular area they need to. One obvious example is the desktop, which provides a halftone backdrop behind any windows. If a window or group of windows covers the entire desktop, the background is hidden, but if moving or closing windows causes some part of the desktop to be uncovered, the background ensures that something is drawn there.

A less obvious example is that of window and dialog box objects, which use their frame objects as backgrounds. Since the frame object has to cover all the edges of the window or dialog box, it also fills in any areas that aren't covered by any other subview. This is particularly useful when designing dialog boxes, where you typically want to insert controls without having to worry about the spaces between them.

Any time you're dealing with a background or other overlapping views, you need to understand how Turbo Vision decides which views are "in front of" or "behind" others. The front-to-back positioning of objects is determined by the objects' Z-order, which is the topic of the next section.

Understanding subviews

There are two important aspects to the relationship between an owner view and its subviews: the actual links between the views, and the order of the views. This section answers two important questions:

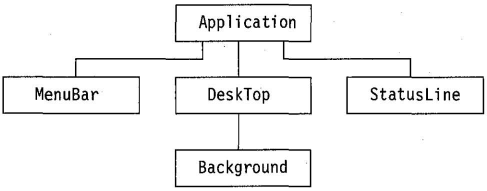
- What is a view tree?
- What is Z-order?

What is a view tree?

When you insert subviews into a group, the views create a kind of *view tree*, with the owner as the "trunk" and the subviews as "branches." The ownership linkages of all the views in a complex application can get fairly complex, but if you visualize them as a single branching tree, you can grasp the overall structure.

For example, the application object owns three subviews, as shown in Figure 8.1. The visible part of the desktop is its background subview, so it's also an owner with a branch. The corresponding view tree looks something like this:

Figure 8.2
Basic Turbo Vision view tree



In a typical application, the user clicks with the mouse or uses the keyboard and creates more views. These views normally appear on the desktop, forming further branches of the tree. Say, for instance, that the user clicks a menu item that opens a file viewer window. The application constructs the window and inserts it into the desktop, making the window another subview of the desktop, and another branch of the view tree.

This same kind of object is depicted somewhat differently in Figure 8.7.

The window itself owns a number of subviews: a frame, a scroller (the interior view that holds a scrollable array of text), and a couple of scroll bars. The application now looks something like Figure 8.3, with the corresponding view tree in Figure 8.4.

Figure 8.3
Desktop with file viewer added

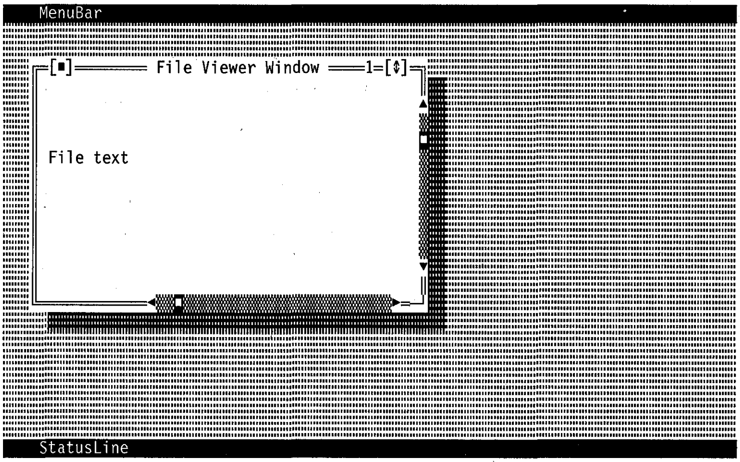
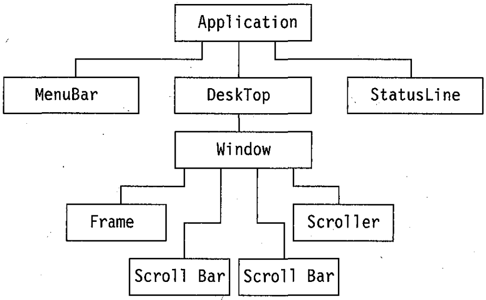
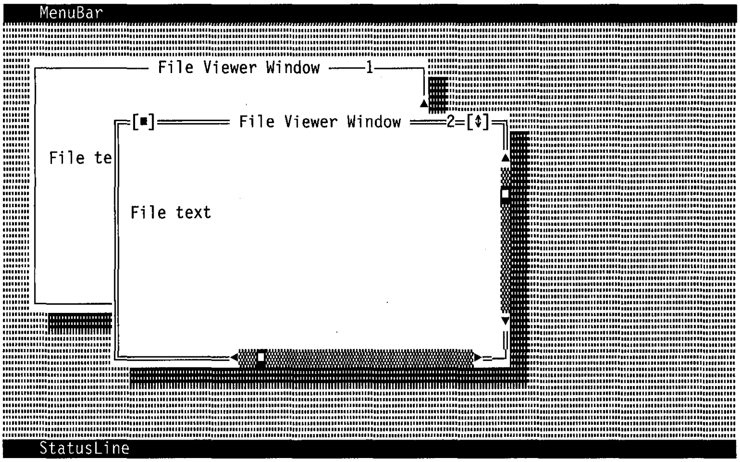


Figure 8.4
View tree with file viewer added



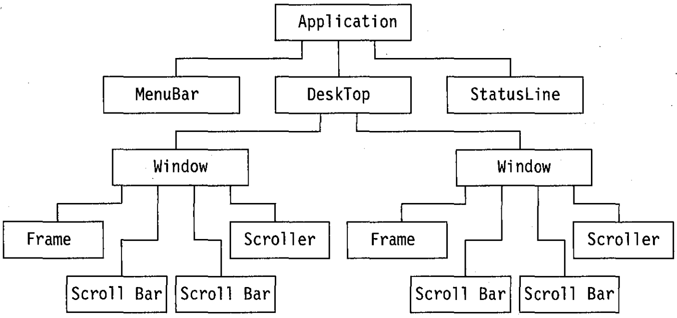
If the user clicks the same menu item again, opening another file viewer window, Turbo Vision constructs a second window and inserts it into the desktop, as shown in Figure 8.5.

Figure 8.5
Desktop with file viewer added



The view tree also becomes correspondingly more complex, as shown in Figure 8.6.

Figure 8.6
View tree with two file viewers added



As you'll see, the order of insertion determines the order in which subviews get drawn and the order in which events get passed to them.

If the user clicks on the second file viewer's close icon or on a Close Window menu item, the second file viewer will close. Turbo Vision then takes it off the view tree and disposes it. The window will dispose of all its subviews, then dispose of itself.

Eventually, the user trims the views down to just the original four, and indicates, by pressing *Alt+X* or by selecting Exit from a menu, that the program should terminate. *TApplication* disposes of its three subviews, then disposes of itself.

What is Z-order?

Groups keep track of the order in which subviews are inserted. That order is referred to as *Z-order*. The term *Z-order* refers to the fact that subviews have a three-dimensional spatial relationship. As you've already seen, every view has a position and size within the plane of the view as you see it (the X and Y dimensions), determined by its *Origin* and *Size* fields. But views and subviews can overlap, and in order for Turbo Vision to know which view is in front of which others, we have to add a third dimension, the Z-dimension.

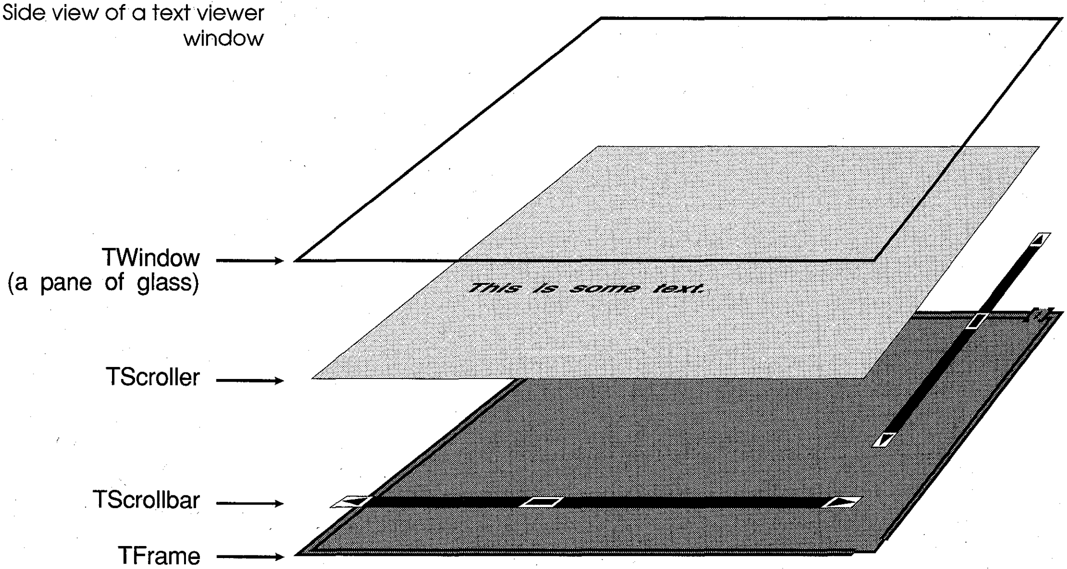
Z-order is the opposite of insertion order.

Z-order, then, refers to the order in which you encounter views as you start closest to you and move back "into" the screen. The last view inserted is the "front" view. Think of X-order as going from left to right, Y-order from top to bottom, and Z-order from front to back.

Visualizing Z-order

Rather than thinking of the screen as a flat plane with things written on it, consider it a pane of glass providing a portal onto a three-dimensional world of views. Indeed, every group may be thought of as a "sandwich" of views, as illustrated in Figure 8.7.

Figure 8.7
Side view of a text viewer window



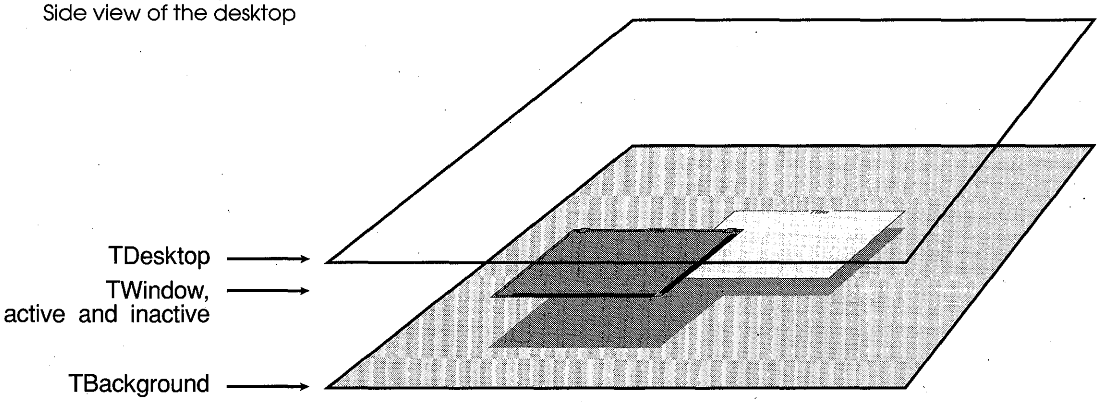
The window itself is just a pane of glass covering a group of views. Since all you see is a flat projection of the views behind the glass on the screen, you can't see which views are in front of others unless they overlap.

By default, a window has a frame, which is inserted before any other subviews. It is therefore the background view. In creating a scrolling interior, two scroll bars are overlaid on the frame. To you, in front of the whole scene, they look like part of the frame, but from the side, you can see that they actually float "above" the frame, obscuring part of the frame from view.

Finally, the scroller itself is inserted, covering the entire area inside the border of the frame. Text is written on the scroller, not on the window, but you can see it when you look through the window.

On a larger scale, you can see the desktop as just a larger pane of glass, covering a larger sandwich, many of the contents of which are also smaller sandwiches, as shown in Figure 8.8.

Figure 8.8
Side view of the desktop



Again, the group (this time the desktop) is a pane of glass. Its first subview is a *TBackground* object, so that view is "behind" all the others. This view also shows two windows with scrolling interior views on the desktop.

Selecting and focusing subviews

Within each group of views, one and only one subview is *selected*. For example, when your application sets up its menu bar, desktop, and status line, the desktop is the selected view, because that is where further work will take place.

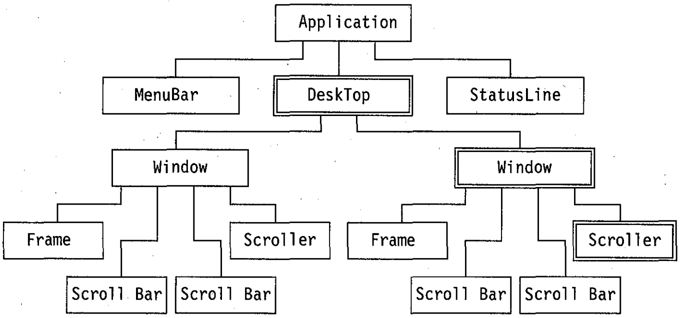
When you have several windows open on the desktop, the selected window is the one in which you're currently working. This is also called the *active* window (typically the topmost window).

The focused view is the end of the chain of selected views that starts at the application.

Within the active window, the selected subview is called the *focused* view. You can think of the focused view as being the one you're looking at, or the one where action will take place. In an editor window, the focused view is the interior view with the text in it. In a dialog box, the focused view is the highlighted control.

In the application diagrammed in Figure 8.6, *Application* is the modal view, and *DeskTop* is its selected view. Within the desktop, the second (more recently inserted) window is selected, and therefore active. Within that window, the scrolling interior is selected, and because it is a terminal view (that is, it's not a group), it is the end of the chain, the focused view. Figure 8.9 depicts the same view tree with the chain of focused views highlighted by double-lined boxes.

Figure 8.9
The focus chain



Among other things, knowing which view is focused tells you which view gets information from the keyboard. For more information, see the section on focused events in Chapter 9, "Event-driven programming."

Finding the focused view

The currently focused view is usually highlighted in some way on the screen. For example, if you have several windows open on the desktop, the active window is the one with the double-lined frame. The others' frames are single-lined. Within a dialog box, the focused control is brighter than the others, indicating that it is the one acted upon if you press Enter. The focused control is therefore the default control as well.

On monochrome displays, Turbo Vision adds arrow characters to indicate the focus.

How does a view get the focus?

A view can get the focus in two ways, either by default when it is created, or by some action by the user.

When a group of views is created, the owner view specifies which of its subviews is to be focused by calling that subview's *Select* method. This establishes the *default focus.*

The user usually determines which view currently has the focus by clicking a particular view. For instance, if the application has several windows open on the desktop, the user can select different ones simply by clicking them. In a dialog box, the user can move the focus among views by pressing *Tab,* which cycles through all the selectable views, by clicking a particular view, or by pressing a hot key.

Note that some views are not selectable, including the background of the desktop, frames of windows, and scroll bars. When you construct a view, you can clear the view's *ofSelectable* option flag, after which the view won't let itself be selected. If you click the frame of a window, for example, the frame does not get the focus, because the frame object knows it can't be selected.

Changing grow modes

A view's *GrowMode* field determines how the view changes when its owner group is resized. The individual bits in *GrowMode* allow you to "anchor" a side of your view to its owner, so that resizing the owner also moves and/or resizes the subview, based on its grow mode.

The *gfGrowLoX* bit anchors the left side of the view to its owner's left side, meaning the view stays a constant distance from its owner's left side. The bits *gfGrowLoY, gfGrowHiX,* and *gfGrowHiY* anchor the top, right side, and bottom of the view to the corresponding parts of the owner. The mask *gfGrowAll* anchors all four sides, resizing the view as the owner's lower right corner moves. Window interiors often use *gfGrowAll* to keep them properly sized within their frames.

The flag *gfGrowRel* is special, and intended only for use with windows on the desktop. Setting *gfGrowRel* causes windows to retain their relative sizes when the user switches the application between different video modes.

The example program DRAGS.PAS on your distribution disks demonstrates how the different *GrowMode* flags affect an object in a window.

Drawing groups

Groups are an exception to the rule that views must know how to draw themselves, because a group does not draw itself *per se.* Rather, a *TGroup* tells its subviews to draw themselves. The cumulative effect of drawing the subviews must cover the entire area assigned to the group.

A dialog box, for example, is a group, and its subviews—frame, interior, controls, and static text—must combine to fully "cover" the full area of the dialog box view. Otherwise, "holes" in the dialog box appear, with unpredictable results.

You will rarely, if ever, need to change the way groups draw themselves, but you do need to understand the following aspects of group drawing:

- Drawing in Z-order
- Using cache buffers
- Locking and unlocking draws
- Clipping subviews

Drawing in Z-order

The group calls on its subviews to draw themselves in Z-order, meaning that the last subview inserted into the group is the first one drawn. If subviews overlap, the one most recently inserted will be in front of any others.

Using cache buffers

All views have a bit in their *Options* word called *ofBuffered,* but only groups make use of it. When this bit is set, groups can speed their output to the screen by writing to a cache buffer. By default, all groups have *ofBuffered* set and use buffered drawing.

☞ The Turbo Vision memory management subsystem allocates cache buffers for groups in the unallocated part of the heap, so if your application also makes use of unallocated memory, you could have conflicts with group buffers. The safest practice is to use only memory you have allocated from the heap.

When a buffered group draws itself, it automatically stores its screen image in a buffer if enough memory is available. The next time the group is asked to draw itself, it copies the cached image to the screen instead of asking all its subviews to draw themselves. Obviously, copying the existing image is much faster than regenerating the image.

Turbo Vision's memory manager disposes of these group buffers whenever other memory allocations need the space. That is, if another memory allocation would otherwise fail, Turbo Vision will try to free enough memory by disposing of group cache buffers. No information is lost when the buffer is disposed of, but the group will have to redraw itself by calling all subviews the next time it needs to draw itself.

You can also force a group to completely draw itself without copying from the buffer by calling its *Redraw* method.

Locking and unlocking draws

Complicated group views can sometimes cause flickering when drawn, particularly when a number of views overlap. In order to avoid flickering, you can lock the group while subviews draw, then unlock the view when the buffer holds a complete group image, at which point the group copies the buffer to the screen.

Calling a group's *Lock* method will stop all writes of the group to the screen until a corresponding call to the method *Unlock*. When *Unlock* is called, the group's buffer is written to the screen. Locking can significantly decrease flicker during complicated updates to the screen. For example, the desktop locks itself while it tiles or cascades its subviews.

Clipping subviews

When the subviews of a group draw themselves, drawing is automatically clipped at the borders of the group. Because subviews are clipped, when you initialize a view and give it to a group, the view needs to reside at least partially within the group's boundaries. (You can grab a window and move it off the desktop until only one corner remains visible, for example, but something must remain visible for the view to be useful.) Only the part of a subview that is within the bounds of its owner group will be visible.

You can use this clipping to your advantage when writing complicated *Draw* methods. Normally, when you fill a draw buffer to write to the screen, you fill enough characters to draw the entire view. If the view is clipped, however, you might need to draw only a few characters, or even skip whole lines.

To find out the area that requires redrawing (that is, the part of the view that is not clipped), call the method *GetClipRect* instead of *GetExtent*. Like *GetExtent*, *GetClipRect* returns a rectangle in local coordinates, but it includes only the part of the view not clipped by its owner's boundaries.

Executing modal groups

Most complex programs have several different *modes* of operation, where a mode is some distinct way of functioning. The integrated development environment, for example, has an editing and debugging mode, a compiler mode, and a run mode. Depending on which mode is active, keys on the keyboard might have varying effects (or no effect at all).

Almost any Turbo Vision view can define a mode of operation, in which case it is called a *modal view*, but modal views are nearly always groups. The classic example of a modal view is a dialog box. Usually, when a dialog box is active, nothing outside it functions. You can't use the menus or other controls not owned by the dialog box. In addition, clicking the mouse outside the dialog box has no effect. The dialog box has control of your program until the user closes it.

In order to use modal views, you need to understand four things:

- What is modality?
- Executing a view
- Finding the modal view
- Ending a modal state

There is *always* a modal view when a Turbo Vision application is running. When you start the program, and often for the duration of the program, the modal view is the application itself, the *TApplication* object at the top of the view tree.

What is modality?

When you make a view modal, only that view and its subviews can interact with the user. You can think of a modal view as defining the "scope" of a portion of your program. When you create a block in a Pascal program (such as a function or a procedure), any identifiers declared within that block are only valid within that block. Similarly, a modal view determines what behaviors are valid within it—events are handled only by the modal view and its subviews. Any part of the view tree that is not the modal view or owned by the modal view is inactive.

The status line is always "hot," no matter what view is modal.

There is one exception to this rule: The status line is available at all times. That way you can have active status line items, even when your program is executing a modal dialog box that does not own the status line. Events and commands generated by the status line, however, will be handled as if they were generated within the modal view.

Making a group modal

The most common kind of modal view (other than the application itself, which is the default modal view in a Turbo Vision program) is a dialog box, so Turbo Vision's application object provides an easy way to execute modal dialog boxes on the desktop, the *ExecuteDialog* method. *ExecuteDialog* is explained in detail in Chapter 11, "Window and dialog box objects."

Event loops are explained in detail in Chapter 9, "Event-driven programming."

In a more general case, you can make a group the current modal view by *executing* it; that is, calling its *Execute* method. *TGroup.Execute* implements an event loop, interacting with the user and dispatching events to the proper subviews. In most cases, you won't call *Execute* directly, but rather rely on *ExecView*.

ExecView is a group method that works much like *Insert* and *Delete* surrounding *Execute*. It inserts a view into the group, executes the new subview, then deletes the subview when the user terminates the modal state (such as closing a modal dialog box).

Finding the modal view

Every view has a method called *TopView* that returns a pointer to the current modal view. There are a couple of times when you might need that information, including ending the current modal state (as described in the next section) and broadcasting an event to all the currently available views (described in Chapter 9, "Event-driven programming").

Ending a modal state

Any view can end the current modal state by calling the method *EndModal*. *EndModal* takes a command constant as its only argument, and passes it to the current modal view. This ends its modal state, returning the command value as the result of the *Execute* method that made the view modal. The previously modal view then becomes the current modal view. If there is no other modal view, such as when you end the application object's modal state, the application terminates.

For example, Listing 8.7 shows part of the *HandleEvent* method of *TDialog*. Modal dialog boxes end their modal state when they see the command *cmOK*, *cmCancel*, *cmYes*, or *cmNo*. That command is then returned by the *Execute* or *ExecuteDialog* call that made the dialog box modal.

Listing 8.7
Ending a modal state on a command event

```
procedure TDialog.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  case Event.What of
      ⋮
    evCommand:
      case Event.Command of
        cmOk, cmCancel, cmYes, cmNo: { for any of these commands... }
          if State and sfModal <> 0 then    { if dialog box is modal }
          begin
            EndModal(Event.Command); { end modal state with command }
            ClearEvent(Event);              { and mark event handled }
          end;
      end;
  end;
end;
```

The example program ENDCMD.PAS on your distributions disks shows how to check the return value from a modal dialog box.

Managing subviews

Once you've inserted a subview into a group, the group handles nearly all the management of the subview for you, making sure it's drawn, moved, and so on. When you dispose of a group object, it automatically disposes of all its subviews, so you don't have to dispose of them individually. So, for example, although a dialog box's constructor is often rather lengthy and complicated, constructing and initializing numerous controls as subviews, the destructor is often not overridden at all, as the default dialog box object uses the *Done* destructor it inherits from *TGroup*, which disposes of each subview before disposing of itself.

Aside from the automatic subview management, you'll sometimes need to perform the following tasks on a group's subviews:

- Deleting subviews
- Iterating subviews
- Finding a particular subview

Deleting subviews

Although a group automatically disposes of all its subviews before disposing of itself, you sometimes want to remove a subview while you're still using the group. An obvious example is closing a window on the desktop: disposing of the desktop disposes of all windows inserted into the desktop, but you'll often need to remove a window in the course of running an application.

To remove a subview from its owner, use the owner's *Delete* method. *Delete* is the inverse of *Insert*: it removes the subview from the owner's list of subviews, but doesn't dispose of the deleted view.

Iterating subviews

As you've seen several times in this chapter, groups handle a number of their duties, such as drawing, by calling on all their subviews in Z-order. The process of calling each subview in order is called *iteration*. In addition to the standard iterations, *TGroup* provides iterator methods that let you define your own actions to be performed by or on each subview.

Finding a particular subview

Sometimes you need to locate a particular subview within a group, such as finding an editor window among the windows on the desktop, or picking out the OK button in a dialog box. Group objects provide a useful method for searching through subviews, testing each one until a certain test is satisfied.

The *TGroup* method *FirstThat* takes a pointer to a Boolean function as its parameter and applies that function to each of the groups subviews in Z-order until the function returns *True*, at which point *FirstThat* returns a pointer to the subview that tested true.

CHAPTER

9

Event-driven programming

The purpose of Turbo Vision is to provide you with a working framework for your applications so you can focus on creating the "meat" of your applications. The two major Turbo Vision tools are built-in windowing support and handling of events. Chapter 8 explained views, and this chapter discusses how to build your programs around events.

Bringing Turbo Vision to life

We have already described Turbo Vision applications as being event-driven, and briefly defined events as being occurrences to which your application must respond.

Reading the user's input

In a traditional Pascal program, you typically write a loop of code that reads the user's keyboard, mouse, and other input, then make decisions based on that input within the loop. You call procedures or functions, or branch to a code loop somewhere else that again reads the user's input:

```
repeat
  B := ReadKey;
  case B of
    'i': InvertArray;
    'e': EditArrayParams;
    'g': GraphicDisplay;
    'q': Quit := true;
  end;
until Quit;
```

An event-driven program is not structured very differently from this. In fact, it is hard to imagine an interactive program that doesn't work this way. However, an event-driven program looks different to you, the programmer.

In a Turbo Vision application, you no longer have to read the user's input because Turbo Vision does it for you. It packages the input into Pascal records called *events*, and dispatches the events to the appropriate views in the program. That means your code only needs to know how to deal with relevant input, rather than sorting through the input stream looking for things to handle.

For instance, if the user clicks an inactive window, Turbo Vision reads the mouse action, packages it into an event record, and sends the event record to the inactive window.

If you come from a traditional programming background, you might be thinking at this point, "O.K., so I don't need to read the user's input anymore. What I'll be doing instead is learning how to read a mouse click event record and how to tell an inactive window to become active." In fact, there's no need for you to write even that much code.

Views can handle much of a user's input all by themselves. A window knows how to open, close, move, be selected, resize, and more. A menu knows how to open, interact with the user, and close. Buttons know how to be pushed, how to interact with each other, and how to change color. Scroll bars know how to be operated. The inactive window can make itself active without any attention from you.

So what is your job as programmer? You define new views with new actions, which need to know about certain kinds of events that you define. You also teach your views to respond to standard commands, and even to generate their own commands ("messages") to other views. The mechanism is already in place.

All you have to do is generate commands and teach views how to respond to them.

But what exactly do events look like to your program, and how does Turbo Vision handle them for you?

The nature of events

Events can best be thought of as little packets of information describing discrete occurrences to which your application needs to respond. Each keystroke, each mouse action, and any of certain conditions generated by other components of the program, constitute a separate event. Events cannot be broken down into smaller pieces. Therefore, the user typing in a word is not a single event, but a series of individual keystroke events.

In the object-oriented world of Turbo Vision, you probably expect events to be objects, too. But they're not. Events themselves perform no actions. They only convey information to your objects, so they are record structures.

At the core of every event record is a single *Word*-type field named *What*. The numeric value of the *What* field describes the kind of event that occurred, and the remainder of the event record holds specific information about that event: the keyboard scan code for a keystroke event, information about the position of the mouse and the state of its buttons for a mouse event, and so on.

Because different kinds of events get routed to their destination objects in different ways, we need to look first at the different kinds of events recognized by Turbo Vision.

Kinds of events

Let's look at the possible values of *Event.What* a little more closely. There are basically four classes of events: mouse events, keyboard events, message events, and "nothing" events. Each class has a mask defined, so your objects can determine quickly which general type of event occurred without worrying about what specific sort it was. For instance, rather than checking for each of the four different kinds of mouse events, you can simply check to see if the event flag is in the mask. Instead of

```
if Event.What and (evMouseDown or evMouseUp or evMouseMove or
  evMouseAuto) <> 0 then...
```

you can use

```
if Event.What and evMouse <> 0 then ...
```

The masks available for separating events are *evNothing* (for "nothing" events), *evMouse* for mouse events, *evKeyboard* for keyboard events, and *evMessage* for messages.

The event mask bits are defined in Figure 9.1.

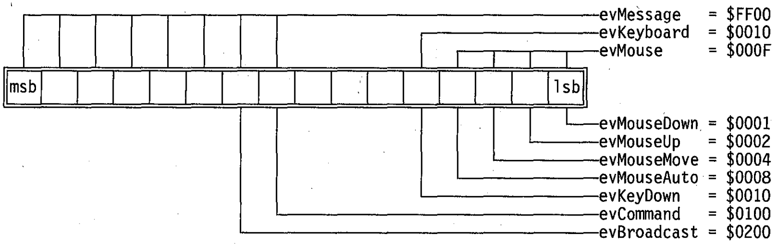


Figure 9.1 *TEvent.What* field bit mapping

Mouse events

There are basically four kinds of mouse events: an up or down click with either button, a change of position, or an "auto" mouse event. Pressing down a mouse button results in an *evMouseDown* event. Letting the button back up generates an *evMouseUp* event. Moving the mouse produces an *evMouseMove* event. And if you hold down the button, Turbo Vision will periodically generate an *evMouseAuto* event, allowing your application to perform such actions as repeated scrolling. All mouse event records include the position of the mouse, so an object that processes the event knows where the mouse was when it happened.

Keyboard events

Keyboard events are even simpler. When you press a key, Turbo Vision generates an *evKeyDown* event, which keeps track of which key was pressed.

Message events

Message events come in three flavors: commands, broadcasts and user messages. The difference is in how they are handled, which is explained later. Basically, commands are flagged in the *What* field by *evCommand*, broadcasts by *evBroadcast*, and user-defined messages by some user-defined constant.

"Nothing" events

A "nothing" event is really a dead event. It has ceased to be an event, because it has been completely handled. If the *What* field in an event record contains the value *evNothing*, that event record contains no useful information that needs to be dealt with.

When a Turbo Vision object finishes handling an event, it calls a method called *ClearEvent*, which sets the *What* field back to *evNothing*, indicating that the event has been handled. Objects should simply ignore *evNothing* events.

Events and commands

Ultimately, most events end up being translated into commands. For example, clicking an item in the status line generates a mouse event. When it gets to the status line object, that object responds to the mouse event by generating a command event, with the *Command* field value determined by the command bound to the status line item. Clicking `Alt-X Exit` generates the *cmQuit* command, which the application interprets as an instruction to shut down and terminate.

Routing of events

Turbo Vision's views operate on the principle "Speak only when spoken to." That is, rather than actively seeking out input, they wait passively for the event manager to tell them that an event has occurred to which they need to respond.

In order to make your Turbo Vision programs act the way you want them to, you not only have to tell your views what to do when certain events occur, you also need to understand how events get to your views. The key to getting events to the right place is correct *routing* of the events. Some events get broadcast all over the application, while others are directed rather narrowly to particular parts of the program.

Where do events come from?

As noted in Chapter 10, "Application objects," the main processing loop of a *TApplication*, the *Run* method, calls *TGroup.Execute*, which is basically a repeat loop that looks something like this:

```
var E: TEvent;
E.What := evNothing;                { indicate no event has occurred }
repeat
  if E.What <> evNothing then EventError(E);
  GetEvent(E);                             { pack up an event record }
  HandleEvent(E);                  { route the event to the right place }
until EndState <> Continue;               { until the quit flag is set }
```

GetEvent, HandleEvent and EventError are all described in greater detail on pages 164, 161, and 163, respectively.

Essentially, *GetEvent* looks around and checks to see if anything has happened that should be an event. If it has, *GetEvent* creates the appropriate event record. *HandleEvent* then routes the event to the proper views. If the event is not handled (and cleared) by the time it gets back to this loop, *EventError* is called to indicate an abandoned event. By default, *EventError* does nothing.

Where do events go?

Events *always* begin their routing with the current modal view. For normal operations, this usually means your application object. When you execute a modal dialog box, that dialog box object is the modal view. In either case, the modal view is the one that initiates event handling. Where the event goes from there depends on the nature of the event.

Events are routed in one of three ways, depending on the kind of event they are. The three possible routings are positional, focused, and broadcast. It is important to understand how each kind of event gets routed.

Positional events

Positional events are virtually always mouse events (*evMouse*).

Z-order is explained in Chapter 8, "Views."

The modal view gets the positional event first, and starts looking at its subviews in Z-order until it finds one that contains the position where the event occurred. The modal view then passes the event to that view. Since views can overlap, it is possible that more than one view will contain that point. Going in Z-order guarantees that the topmost view at that position will be the one that receives the event. After all, that's the one the user clicked.

This process continues until an object cannot find a view to pass the event to, either because it is a terminal view (one with no subviews) or because there is no subview in the position where the event occurred (such as clicking open space in a dialog box). At that point, the event has reached the object where the positional event took place, and that object handles the event.

Focused events

For details on focused views and the focus chain, see "Selected and focused views" in Chapter 8, "Views."

Focused events are generally keystrokes (*evKeyDown*) or commands (*evCommand*), and they are passed down the focus chain.

The current modal view gets the focused event first, and passes it to its selected subview. If that subview has a selected subview, it passes the event to it. This process continues until a terminal view is reached: This is the focused view. The focused view receives and handles the focused event.

Non-focused views may handle focused events. See the "Phase" section in this chapter.

If the focused view does not know how to handle the particular event it receives, it passes the event back up the focus chain to its owner. This process is repeated until the event is handled or the event reaches the modal view again. If the modal view does not know how to handle the event when it comes back, it calls *EventError*. This situation is an *abandoned event.*

Keyboard events illustrate the principle of focused events quite clearly. For example, in the Turbo Pascal integrated environment, you might have several files open in editor windows on the desktop. When you press a key, you know which file you want to receive the character. Let's see how Turbo Vision ensures it actually gets there.

Your keystroke produces an *evKeyDown* event, which goes to the current modal view, the *TApplication* object. *TApplication* sends the event to its selected view, the desktop (the desktop is always *TApplication*'s selected view). The desktop sends the event to its selected view, which is the active window (the one with the double-lined frame). That editor window also has subviews—a frame, a scrolling interior view, and two scrollbars. Of those, only the interior is selectable (and therefore selected, by default), so the keyboard event goes to it. The interior view, an editor, has no subviews, so it gets to decide how to handle the character in the *evKeyDown* event.

Broadcast events

Broadcast events are generally either broadcasts (*evBroadcast*) or user-defined messages.

Broadcast events are not as directed as positional or focused events. By definition, a broadcast does not know its destination, so it is sent to *all* the subviews of the current modal view.

The current modal view gets the event, and begins passing it to its subviews in Z-order. If any of those subviews is a group, it too passes the event to its subviews, also in Z-order. The process continues until all views owned (directly or indirectly) by the modal view have received the event, or until a view clears the event.

Broadcasts can be directed to an object with the Message function.

Broadcast events are commonly used for communication between views. For example, when you click on a scroll bar in a file viewer, the scroll bar needs to let the text view know that it should show some other part of itself. It does that by broadcasting a view saying "I've changed!" which other views, including the text, will receive and react to. For more details, see the "Inter-view communication" section in this chapter.

User-defined events

As you become more comfortable with Turbo Vision and events, you may wish to define whole new categories of events, using the high-order bits in the *What* field of the event record. By default, Turbo Vision will route all such events as broadcast events. But you may wish your new events to be focused or positional, and Turbo Vision provides a mechanism to allow this.

Manipulating bits in masks is explained in Chapter 11, "Overview of the run-time library."

Turbo Vision defines two masks, *Positional* and *Focused*, which contain the bits corresponding to events in the event record's *What* field that should be routed by position and by focus, respectively. By default, *Positional* contains all the *evMouse* bits, and *Focused* contains *evKeyboard*. If you define some other bit to be a new kind of event that you want routed either by position or focus, you simply add that bit to the appropriate mask.

Masking events

Every view object has a bitmapped field called *EventMask* which is used to determine which events the view will handle. The bits in the *EventMask* correspond to the bits in the *TEvent.What* field. If the bit for a given kind of event is set, the view will accept that kind of event for handling. If the bit for a kind of event is cleared, the view will ignore that kind of event.

For example, by default a view's *EventMask* excludes *evBroadcast*, but a group's *EventMask* includes it. Therefore, groups receive broadcast events by default, but views don't.

Phase

There are certain times when you want a view other than the focused view to handle focused events (especially keystrokes). For example, when looking at a scrolling text window, you might want to use keystrokes to scroll the text, but since the text window is the focused view, keystroke events go to it, not to the scroll bars that can scroll the view.

Turbo Vision provides a mechanism, however, to allow views other than the focused view to see and handle focused events.

Although the routing described in the "Focused events" section of this chapter is essentially correct, there are two exceptions to the strict focus-chain routing.

When the modal view gets a focused event to handle, there are actually three "phases" to the routing:

- The event is sent to any subviews (in Z-order) that have their *ofPreProcess* option flags set.
- If the event isn't cleared by any of them, the event is sent to the focused view.
- If the event still hasn't been cleared, the event is sent (again in Z-order) to any subviews with their *ofPostProcess* option flags set.

So in the preceding example, if a scroll bar needs to see keystrokes that are headed for the focused text view, the scroll bar should be initialized with its *ofPreProcess* option flag set.

Notice also that in this particular example it doesn't make much difference whether you set *ofPreProcess* or *ofPostProcess*: Either one will work. Since the focused view in this case doesn't handle the event (*TScroller* itself doesn't do anything with keystrokes), the scroll bars may look at the events either before *or* after the event is routed to the scroller.

In general, however, use *ofPostProcess* in a case like this; it provides greater flexibility. Later on you might want to add functionality to the interior that checks keystrokes. However, if the keystrokes have been taken by the scroll bar before they get to the focused view (*ofPreProcess*), your interior never gets to act on them.

☞ Although there are times when you *need* to grab focused events before the focused view can get at them, it's a good idea to leave as many options open as possible. In that way, you (or someone else) can derive something new from this object in the future.

The Phase field

Every group has a field called *Phase*, which has any of three values: *phFocused*, *phPreProcess*, and *phPostProcess*. By checking its owner's *Phase* flag, a view can tell whether the event it is handling is coming to it before, during, or after the focused routing. This is sometimes necessary, because some views look for different events, or react to the same events differently, depending on the phase.

Consider the case of a simple dialog box that contains an input line and a button labeled "**A**ll right," with *A* being the shortcut key for the button. With normal dialog box controls, you don't really have to concern yourself with phase. Most controls have *ofPostProcess* set by default, so keystrokes (focused events) will get to them and allow them to grab the focus if it is their shortcut letter that was typed. Pressing *A* moves the focus to the "All right" button.

But suppose the input line has the focus, so keystrokes get handled and inserted by the input line. Pressing the *A* key puts an "A" in the input line, and the button never gets to see the event, since the focused view handled it. Your first instinct might be to have the button check for the *A* key preprocess, so it can snag the shortcut key before the focused view handles it. Unfortunately, this would always preclude your typing the letter "A" in the input line!

The solution is to have the button check for different shortcut keys before and after the focused view handles the event. By default, a button will look for its shortcut key in *Alt*+letter form preprocess, and in letter form postprocess. That's why you can always use the *Alt*+letter shortcuts in a dialog box, but you can only use regular letters when the focused control doesn't "eat" keystrokes.

This is easy to do. By default, buttons have both *ofPreProcess* and *ofPostProcess* set, so they get to see focused events both before *and* after the focused view does. But within its *HandleEvent*, the button checks only certain keystrokes if the focused control has already seen the event:

```
evKeyDown:                         { this is part of a case statement }
  begin
    C := HotKey(Title^);
    if (Event.KeyCode = GetAltCode(C)) or
      (Owner^.Phase = phPostProcess) and (C <> #0) and
        (Upcase(Event.CharCode) = C) or
      (State and sfFocused <> 0) and (Event.CharCode = ' ') then
    begin
      PressButton;
      ClearEvent(Event);
    end;
  end;
```

Commands

Most positional and focused events are translated into commands by the objects that handle them. That is, an object often responds to a mouse click or a keystroke by generating a command event.

For example, by clicking on the status line in a Turbo Vision application, you generate a positional (mouse) event. The application determines that the click was positioned in the area controlled by the status line, so it passes the event to the status line object, *StatusLine*.

StatusLine determines which of its status items controls the area where you clicked, and reads the status item record for that item. That item usually has a command bound to it, so *StatusLine* creates a pending event record with the *What* field set to *evCommand* and the *Command* field set to the command bound to that status item. It then clears the mouse event, meaning that the next event found by *GetEvent* will be the command event just generated.

Defining commands

Turbo Vision has many predefined commands, and you will define many more yourself. When you create a new view, you also create a command to invoke the view. Commands can be called anything, but Turbo Vision's convention is that a command identifier should start with "cm." Creating a command is simple—you just create a constant:

```
const
  cmConfuseTheCat = 100;
```

Turbo Vision reserves commands 0 through 99 and 256 through 999 for its own use. Your applications can use the numbers 100 through 255 and 1,000 through 65,535 for commands.

The reason for having two ranges of commands is that only commands 0 through 255 may be disabled. Turbo Vision reserves some of the commands that can be disabled and some of the commands that cannot be disabled for its standard commands and internal workings. You have complete control over the remainder of the commands.

The ranges of available commands are summarized in Table 9.1.

Table 9.1
Turbo Vision command ranges

Range	Reserved	Can be disabled
0..99	Yes	Yes
100..255	No	Yes
256..999	Yes	No
1000..65535	No	No

Binding commands

When you create a menu item or a status line item, you bind a command to it. When the user chooses that item, an event record is generated, with the *What* field set to *evCommand,* and the *Command* field set to the value of the bound command. The command may be either a Turbo Vision standard command or one you have defined. At the same time you bind your command to a menu or status line item, you may also bind it to a hot key. That way, the user can invoke the command by pressing a single key as a shortcut to using the menus or the mouse.

☞ Remember that defining the command does not specify the action to be taken when that command appears in an event record. You have to tell the appropriate objects how to respond to that command.

Enabling and disabling commands

There are times when you want certain commands to be unavailable to the user for a period of time. For example, if you have no windows open, it makes no sense for the user to be able to generate *cmClose,* the standard window closing command. Turbo Vision provides a way to disable and enable sets of commands.

To enable or disable a group of commands, use the global type *TCommandSet,* which is a set of numbers 0 through 255. (This is why only commands in the range 0..255 can be disabled.) The following code disables a group of five window-related commands:

```
var
  WindowCommands: TCommandSet;
begin
  WindowCommands := [cmNext, cmPrev, cmZoom, cmResize, cmClose];
  DisableCommands(WindowCommands);
end;
```

Handling events

Once you have defined a command and set up some kind of control to generate it—for example, a menu item or a dialog box button—you need to teach your view how to respond when that command occurs.

Every view inherits a *HandleEvent* method that already knows how to respond to much of the user's input. If you want a view to do something specific for your application, you need to override its *HandleEvent* and teach the new *HandleEvent* two things—how to respond to new commands you've defined, and how to respond to mouse and keyboard events the way you want.

A view's *HandleEvent* method determines how it behaves. Two views with identical *HandleEvent* methods will respond to events in the same way. When you derive a new view type, you generally want it to behave more or less like its ancestor view, with some changes. By far the easiest way to accomplish this is to call the ancestor's *HandleEvent* as part of the new object's *HandleEvent* method.

The general layout of a descendant's *HandleEvent* would look like this:

```
procedure TNewDescendant.HandleEvent(var Event: TEvent);
begin
  { code to change or eliminate parental behavior }
  inherited HandleEvent(Event);
  { code to perform additional functions }
end;
```

In other words, if you want your new object to handle certain events differently than its ancestor does (or not at all!), you would trap those particular events *before* passing the event to the ancestor's *HandleEvent* method. If you want your new object to behave just like its ancestor, but with certain additional functions, you would add the code to do that *after* the call to the ancestor's *HandleEvent* procedure.

The event record

Up to this point, this chapter has discussed events in a fairly theoretical fashion. We have talked about the different kinds of events (mouse, keyboard, message, and "nothing") as determined by the event's *What* field. We have also discussed briefly the use of the *Command* field for command events.

Now it's time to discuss what an event record actually looks like. The DRIVERS.TPU unit of Turbo Vision defines the *TEvent* type as a variant record:

```
TEvent = record
  What: Word;
  case Word of
    evNothing: ();
    evMouse: (
      Buttons: Byte;
      Double: Boolean;
      Where: TPoint);
    evKeyDown: (
      case Integer of
        0: (KeyCode: Word);
        1: (CharCode: Char;
            ScanCode: Byte));
    evMessage: (
      Command: Word;
      case Word of
        0: (InfoPtr: Pointer);
        1: (InfoLong: Longint);
        2: (InfoWord: Word);
        3: (InfoInt: Integer);
        4: (InfoByte: Byte);
        5: (InfoChar: Char));
end;
```

TEvent is a variant record. You can tell what is in the record by looking at the field *What*. Thus, if *TEvent.What* is an *evMouseDown*, *TEvent* will contain:

```
Buttons: Byte;
Double: Boolean;
Where: TPoint;
```

If *TEvent.What* is an *evKeyDown*, the compiler will let you access the data either as

```
KeyCode: Word;
```

or as

```
CharCode: Char;
ScanCode: Byte;
```

Communication among views and the InfoPtr field are covered in the "Interview communication" section of this chapter.

The final variant field in the event record stores a *Pointer*, *Longint*, *Word*, *Integer*, *Byte* or *Char* value. This field is used in a variety of ways in Turbo Vision. Views can actually generate events themselves and send them to other views. When they do, they often use the *InfoPtr* field.

Clearing events

When a view's *HandleEvent* method has handled an event, it finishes the process by calling its *ClearEvent* method. *ClearEvent* sets the *Event.What* field equal to *evNothing* and *Event.InfoPtr* to *@Self*, which are the universal signals that the event has been handled. If the event then gets passed to another object, that object should ignore this "nothing" event.

Abandoned events

Normally, every event will be handled by some view in your application. If no view can be found that handles an event, the modal view calls *EventError*. *EventError* calls the view owner's *EventError* and so forth up the view tree until *TApplication.EventError* is called.

TApplication.EventError by default does nothing. You may find it useful during program development to override *EventError* to bring up an error dialog box or issue a beep. Since the end user of your software isn't responsible for the failure of the software to handle an event, such an error dialog box in a shipping version would probably just be irritating.

ClearEvent also helps views communicate with each other. For now, just remember that you haven't finished handling an event until you call *ClearEvent*.

Modifying the event mechanism

At the heart of the current modal view is a loop that looks something like this:

```
var
  E: TEvent;
begin
  E.What := evNothing;
  repeat
    if E.What <> evNothing then EventError(E);
    GetEvent(E);
    HandleEvent(E);
  until EndState <> Continue;
end;
```

Centralized event gathering

One of the greatest advantages of event-driven programming is that your code doesn't have to know where its events come from. A window object, for example, just needs to know that when it sees a *cmClose* command in an event, it should close. It doesn't care whether that command came from a click on its close icon, a menu selection, a hot key, or a message from some other object in the program. It doesn't even have to worry about whether that command is intended for it. All it needs to know is that it has been given an event to handle, and since it knows how to handle that event, it does.

The key to these "black box" events is the application's *GetEvent* method. *GetEvent* is the only part of your program that has to concern itself with the source of events. Objects in your application simply call *GetEvent* and rely on it to take care of reading the mouse, the keyboard, and the pending events generated by other objects.

If you want to create new kinds of events (for example, reading characters from a serial port device driver), you would simply override *TApplication.GetEvent* in your application object. As you can see from the *TProgram.GetEvent* code in APP.PAS, the *GetEvent* loop scans among the mouse and the keyboard and then calls *Idle*. To insert a new source of events, you either override *Idle* to look for characters from the serial port and generate events based on them, or override *GetEvent* to add a *GetComEvent(Event)* call to the loop, where *GetComEvent* returns an event record if there is a character available from the serial port.

Overriding GetEvent

The current modal view's *GetEvent* calls its owner's *GetEvent*, and so on, all the way back up the view tree to *TApplication.GetEvent*, which is where the next event is always actually fetched.

Because Turbo Vision always uses *TApplication.GetEvent* to actually fetch events, you can modify events for your entire application by overriding this one method. For example, to implement keystroke macros, you could watch the events returned by *GetEvent*, grab certain keystrokes, and unfold them into macros. As far as the rest of the application would know, the stream of events would be coming straight from the user.

```
procedure TMyApp.GetEvent(var Event: TEvent);
begin
  inherited GetEvent(Event);                 { call TApplication method }
  :                                           { special processing here }
end;
```

Using idle time

Another benefit of *TApplication.GetEvent's* central role is that it calls a method called *TApplication.Idle* if no event is ready. *TApplication.Idle* is a dummy (empty) method that you can override in order to carry out processing concurrent with that of the current view.

An example of a heap viewer is included in the example programs on your distribution disks.

Suppose, for example, you define a view called *THeapView* that uses a method called *Update* to display the currently available heap memory. If you override *TApplication.Idle* with the following, the user will be able to see a continuous display of the available heap memory.

```
procedure TMyApp.Idle;
begin
  inherited Idle;
  HeapViewer.Update;
end;
```

Inter-view communication

A Turbo Vision program is encapsulated into objects, and you write code only within objects. Suppose an object needs to exchange information with another object within your program? In a traditional program, that would probably just mean copying information from one data structure to another. In an object-oriented program, that may not be so easy, since the objects may not know where to find one another.

Inter-view communication is not as easy as sending data between equivalent parts of a traditional Pascal program. (Although two parts of a traditional Pascal application can never achieve the functionality of two Turbo Vision views.)

If you need to do inter-view communication, the first question to ask is if you have divided the tasks up between the two views properly. It may be that the problem is one of poor program design. Perhaps the two views really need to be combined into one view, or part of one view moved to the other view.

Intermediaries

If indeed the program design is sound, and the views still need to communicate with each other, it may be that the proper path is to create an intermediary view.

For example, suppose you have a spreadsheet object and a word processor object, and you want to be able to paste something from the spreadsheet into the word processor, and vice versa. In a Turbo Vision application, you can accomplish this with direct view-to-view communication. But suppose that at a later date you wanted to add, say, a database to this group of objects, and to paste to and from the database. You will now need to duplicate the communication you established between the first two objects between all three.

A better solution is to establish an intermediary view—in this case, say, a clipboard. An object would then need to know only how to copy something to the clipboard, and how to paste something from the clipboard. No matter how many new objects you add to the group, the job will never become any more complicated than this.

Messages among views

If you've analyzed your situation carefully and are certain that your program design is sound and that you don't need to create an intermediary, you can implement simple communication between just two views.

Before one view can communicate with another, it may first have to find out where the other view is, and perhaps even make sure that the other view exists at the present time.

First, a straightforward example. The *StdDlg* unit contains a dialog box called *TFileDialog* (it's the view that opens in the integrated environment when you want to load a new file).

TFileDialog has a *TFileList* that shows you a disk directory, and above it, a *FileInputLine* that displays the file currently selected for loading. Each time the user selects another file in the *FileList*, the *FileList* needs to tell the *FileInputLine* to display the new file name.

In this case, *FileList* can be sure that *FileInputLine* exists, because they are both initialized within the same object, *FileDialog*. How does *FileList* tell *FileInputLine* that the user just selected a new name?

FileList creates and sends a message. Here's *TFileList.FocusItem*, which sends the event, and *FileInputLine's HandleEvent*, which receives it:

TopView points to the current modal view.

```
procedure TFileList.FocusItem(Item: Integer);
var Event: TEvent;
begin
  inherited FocusItem(Item);           { call inherited method first }
  Message(TopView, evBroadcast, cmFileFocused, List^.At(Item));
end;

procedure TFileInputLine.HandleEvent(var Event: TEvent);
var Name: NameStr;
begin
  inherited HandleEvent(Event);
  if (Event.What = evBroadcast) and (Event.Command = cmFileFocused)
    and (State and sfSelected = 0) then
  begin
     if PSearchRec(Event.InfoPtr)^.Attr and Directory <> 0 then
        Data^ := PSearchRec(Event.InfoPtr)^.Name + '\'+
          PFileDialog(Owner)^.WildCard
     else Data^ := PSearchRec(Event.InfoPtr)^.Name;
     DrawView;
  end;
end;
```

Message is a function that generates a message event and returns a pointer to the object (if any) that handled the event.

Note that *TFileList.FocusItem* uses the Turbo Pascal extended syntax (the **$X+** compiler directive) to use the *Message* function as a procedure, since it doesn't care about any results that come back from *Message*.

Who handled the broadcast?

Suppose you need to find out if there is a window open on the desktop before you perform some action. How can you find this out? The answer is to have your code send off a broadcast event that windows know how to respond to. The "signature" left by the object that handles the event will tell you who, if anyone, handled it.

Is anyone out there?

Here's a concrete example. In the Turbo Pascal IDE, if the user asks to open a watch window, the code which opens watch windows needs to check to see if there is already a watch window open. If there isn't, it opens one; if there is, it brings it to the front.

Sending off the broadcast message is easy:

```
AreYouThere := Message(DeskTop, evBroadcast, cmFindWindow, nil);
```

There is a test in the code for a watch window's *HandleEvent* method that responds to *cmFindWindow* by clearing the event:

```
case Event.Command of
   ⋮
  cmFindWindow: ClearEvent(Event);
   ⋮
end;
```

ClearEvent not only sets the event record's *What* field to *evNothing*; it also sets the *InfoPtr* field to *@Self*. *Message* reads these fields, and if the event has been handled, it returns a pointer to the object who handled the message event. In this case, that would be the watch window. So following the line that sends the broadcast, we include

```
if AreYouThere = nil then
  CreateWatchWindow                   { if there is none, create one }
else AreYouThere^.Select;           { otherwise bring it to the front }
```

As long as a watch window is the only object that knows how to respond to the *cmFindWindow* broadcast, your code can be assured that when it finishes, there will be one and only one watch window at the front of the views on the desktop.

Who's on top?

Using the same techniques outlined earlier, you can also determine which window is the topmost view of its type on the desktop. Because a broadcast event is sent to each of the modal view's subviews in Z-order (reverse insertion order), the most recently inserted view is the view "on top" of the desktop.

Consider for a moment the situation encountered in the IDE when the user has a watch window open on top of the desktop while stepping through code in an editor window. The watch window can be the active window (double-lined frame, top of the stack), but the execution bar in the code window needs to keep tracking the executing code. If you have multiple editor windows open on the desktop, they might not overlap at all, but the IDE needs to know which one of the editors it is supposed to bé tracking in.

The answer, of course, is the front, or topmost editor window, which is defined as the last one inserted. In order to figure out which one is "on top," the IDE broadcasts a message that only editor windows know how to respond to. The first editor window to receive the broadcast will be the one most recently inserted. It handles the event by clearing it, and the IDE will then know which window to use for code tracking by reading the result returned by *Message*.

Calling HandleEvent

You can also create or modify an event, then call a *HandleEvent* directly. You can make three types of calls:

"Peer" views are subviews with the same owner.

1. You can have a view call a peer subview's *HandleEvent* directly. The event won't propagate to other views. It goes directly to the other *HandleEvent*, then control returns to you.
2. You can call your owner's *HandleEvent*. The event will then propagate down the view chain. (If you are calling the *HandleEvent* from within your own *HandleEvent*, your *HandleEvent* will be called recursively.) After the event is handled, control returns to you.
3. You can call the *HandleEvent* of a view in a different view chain. The event will travel down that view chain. After it is handled, control will return to you.

CHAPTER

11

Application objects

At the heart of any Turbo Vision program is the application object. This chapter describes in detail all the different things application objects do and how you customize them. It covers the following topics:

- Understanding application objects
- Constructing an application object
- Customizing the desktop
- Shelling to DOS
- Customizing the status line
- Customizing the menu bar
- Using idle time
- Creating context-sensitive Help

Understanding application objects

An application object has two critical roles in your Turbo Vision application. It is a view that manages the entire screen, and it is an event-handling engine that interacts with the mouse, keyboard, and other parts of the computer. There is interaction between these two roles, but you can understand them separately. This section explains these roles by looking at

- The application's role as a view
- The application's role as a group
- The three critical methods: *Init*, *Run*, and *Done*

The application is a view

At first, it might seem strange to think of an application as a view. After all, a view is something visible, while an application is an intangible concept. But the basic principle of a view is that it occupies a rectangular area of the screen, and the application is responsible for the entire screen.

Actually, the application object has a lot more to do than just managing the screen, but that is one of its important duties, and as such there are times when it's important to remember that the application object is a view.

The application is a group

Not only is an application a view, it is also a group. Group views have two special properties: the ability to own subviews, and the ability to be modal. Application objects take advantage of both of these.

The application owns subviews

The boundaries of an application view encompass the entire screen, but the application itself isn't visible. It divides the screen into three distinct areas and assigns a subview to handle each one. By default, the application assigns a menu bar object to the top line of the screen, a status line object to the bottom line, and a desktop object to all the lines in between.

It's easy to remember the application's three subviews, since you see them all during the running of the program, but it's sometimes easy to forget that there's an application object owning all of them. Other sections of this chapter deal specifically with menu bars, status lines, and desktops, but it's important to remember that behind them all is an application object. Most of an application's behavior is easily understandable when you stop to think that it's just another view, or just another group.

You can think of the application view as the ultimate owner of all views in the program. If you follow the ownership chain from any given view, it leads back to the application object.

The application is modal

Most of the time, the application object is the modal view in a running Turbo Vision application. The only exception is when you execute another view (usually a dialog box), which becomes the current modal view until its *EndModal* method gets called, and the application again becomes modal.

As the modal view, the application handles or dispatches most events, so it is an active participant in the running of the program.

Init, Run, Done

The main block of a Turbo Vision application *always* consists of three statements, calling the three main methods of the application object: *Init, Run,* and *Done,* as shown in Listing 10.1.

Listing 10.1
The main loop of a Turbo Vision program

```
var AnyApp: TApplication;
begin
  AnyApp.Init;
  AnyApp.Run;
  AnyApp.Done;
end.
```

You should never need to put any other statements into the main block. Application-specific behavior should be set up in the *Init* constructor and shut down in the *Done* destructor.

The Init constructor

Because the application object (like all views) contains virtual methods, you have to call its constructor before using the object. By convention, all Turbo Vision objects have a constructor called *Init*. The application's constructor sets up the application views and initializes the application's *subsystems*, including the mouse and video drivers, the memory manager, and the error handler. If you override *Init* to add specific items to your application, be sure to call the *Init* constructor inherited from *TApplication*.

The Run method

Run is a simple method, but an extremely important one. After *Init* sets up the application object, *Run* executes the application object, making it modal and setting the application in motion. The bulk of *Run*'s activity is in a simple **repeat..until** loop, which looks something like this pseudo-code:

```
repeat
  Get an event;
  Handle the event;
until Quit;
```

This isn't the actual code, but it shows the concepts. *Run* gets pending events from the mouse or the keyboard or other parts of the application, then handles the events either directly or by routing them to the appropriate views. Eventually, some event comes along that generates a "quit" command, the loop terminates, and the application finishes running.

The Done destructor

Once the *Run* method terminates, the *Done* destructor disposes of any objects owned by the application—the menu bar, status line, and desktop and any objects you've added—then shuts down Turbo Vision's error handler and drivers.

In general, your application's *Done* destructor should undo anything set up by the *Init* constructor, then call the *Done* destructor inherited from *TApplication,* which handles disposing the standard application subviews and shutting down the application subsystems. If you override the application's *Init,* you'll probably have to override *Done,* too.

Constructing an application object

The application constructor is generally simple, taking no parameters, but it performs a number of important functions. When you define your application's constructor, there are relatively few things you *must* do, but one of them is calling the *Init* constructor inherited from *TApplication.* The *TApplication* constructor does two important things that you need to understand:

- Calling the inherited constructor
- Initializing subsystems

Calling the *Init* constructor inherited from *TApplication* takes care of these things completely. If you derive your application from *TProgram* instead of *TApplication,* make sure your constructor calls its inherited constructor and sets up any subsystems you want to use.

Calling the inherited constructor

In most cases, when you override the *Init* constructor in your application object, you include a call to the inherited constructor and then add code. You can achieve most customizations by overriding virtual methods, so you should rarely need to replace the inherited constructor. The next section describes all the behavior inherited from *TProgram* you need to replace if you don't call the inherited constructor.

The TProgram constructor

The *TProgram* constructor does several important things:

- Sets the variable *Application* to point to your application object
- Calls the virtual method *InitScreen* to set up the screen mode variables
- Calls the constructor inherited from *TGroup*
- Sets *State* and *Options* flags
- Sets its video buffer
- Calls the virtual methods *InitDesktop*, *InitStatusLine*, and *InitMenuBar*

These virtual methods are described in the sections of this chapter dealing with the desktop, status line and menu bar.

Note that initialization of screen mode variables, the desktop object, the status line object, and the menu bar object all take place in virtual method calls, so you can override the appropriate methods in your application object, and the inherited constructor will call your redefined methods.

Knowing when to call

When redefining the application constructor, the order in which you call the inherited constructor is very important. As a general rule, you should call the inherited constructor *first*, then define anything specific to your application:

```
constructor TNewApplication.Init;
begin
  inherited Init;                       { call TApplication.Init }
  ⋮                               { Your initialization code goes here }
end;
```

Remember that application objects are views, and that the ultimate ancestor object, *TObject*, clears all fields in the object to zeros and **nil**s. Since calling *TApplication.Init* results in a call to *TObject.Init*, any changes you make to the application object's fields prior to calling the inherited *Init* will be lost.

In general, the only time you *must* do something before calling the inherited application constructor is if you use file editor objects. You must allocate file editor buffers before constructing the application object, as described in Chapter 15.

Initializing subsystems

The main difference between *TApplication* and its ancestor type *TProgram* is that *TApplication* redefines the object's constructor and destructor to initialize and then shut down five major subsystems that make Turbo Vision applications work. The five subsystems are

- The memory manager
- The video manager
- The event manager
- The system error handler
- The history list manager

Turbo Vision sets up each subsystem by calling a procedure in the *App* unit. The *TApplication* constructor calls each before calling the *Init* constructor it inherits from *TProgram*:

```
constructor TApplication.Init;
begin
  InitMemory;                          { set up the memory manager }
  InitVideo;                            { set up the video manager }
  InitEvents;                           { set up the event manager }
  InitSysError;                   { set up the system error handler }
  InitHistory;                    { set up the history list manager }
  inherited Init;                               { call TProgram.Init }
end;
```

Although it's possible to create working applications derived directly from *TProgram*, you should still use at least *some* of the standard application subsystems. For example, to create an application type that doesn't use the history list system, you could derive a new type from *TProgram* much like *TApplication*, but without calling *InitHistory* in the constructor.

The memory manager

The memory manager does three important things for Turbo Vision:

- Sets up a safety pool
- Manages discardable draw buffers
- Manages relocatable file editor buffers

The *safety pool* is an integral part of Turbo Vision. When you allocate memory for a Turbo Vision object, the memory manager checks to make sure the allocation hasn't eaten into the safety pool at the end of memory. This protection keeps your application from simply running out of memory, and gives you a chance to free memory and recover gracefully.

If there is free memory above the stack, group objects allocate discardable draw buffers in that space. By keeping a copy of its screen image, the group can save time when called upon to draw itself. If another allocation needs that space, the group discards its buffer and draws itself completely the next time.

If you use file editors in your applications, you need to set aside memory above the heap for relocatable buffers, as explained in Chapter 15, "Editor and text views." The memory manager subsystem manages those buffers for you.

The video manager

The procedure *InitVideo* sets up Turbo Vision's video manager. The video manager keeps track of the screen mode at application startup so it can restore the screen when the application terminates. *InitVideo* also sets the values of Turbo Vision's internal video variables *ScreenHeight*, *ScreenWidth*, *ScreenBuffer*, *CheckSnow*, *CursorLines*, and *HiResScreen*.

The corresponding procedure *DoneVideo* restores the screen to its startup mode, clears the screen, and restores the cursor.

The event manager

The procedure *InitEvents* checks to see if the system has a mouse installed and, if a mouse is present, sets the variable *MouseEvents* to *True*, enables the mouse interrupt handler, and shows the mouse cursor. If Turbo Vision doesn't detect a mouse at startup, the event manager ignores the mouse completely.

The corresponding procedure *DoneEvents* shuts down the event manager, disabling the mouse interrupt handler and hiding the mouse cursor.

The system error handler

The system error handler does two things for your application:

- Traps DOS critical errors
- Intercepts *Ctrl+Break* keystrokes

By default, the critical-error handler traps DOS critical errors and displays a warning message across the status line of the Turbo

Vision application, giving the user a chance to recover. The error handler allows for user-installable error procedures as well.

The error handler also manages the trapping of the *Ctrl+Break* key, enabling your program to react in some way other than terminating.

See Chapter 20 in the Language Guide for information on the default handling of errors.

If you don't call *InitSysError* to install the Turbo Vision error handler, your application will handle critical errors and *Ctrl+Break* just like any other Pascal application: Critical errors will produce run-time errors, and *Ctrl+Break* will be handled according to the system settings.

The history list manager

The procedure *InitHistory* allocates a block of memory to hold history lists for input lines. The variable *HistorySize* determines the amount of memory allocated, which is 1K by default. If you want to allocate a different amount, you must change the value of *HistorySize* before calling *InitHistory*, which means before calling *TApplication.Init*.

If the memory allocation succeeds, *InitHistory* sets the variable *HistoryBlock* to point to the allocated memory. If the allocation fails, *HistoryBlock* is **nil**, and all attempts to add to or read from history lists will be ignored.

The corresponding procedure *DoneHistory* frees the memory block allocated to *HistoryBlock*. *DoneHistory* uses *HistorySize* to determine how much memory to free, so it is important that you not change the value of *HistorySize* after calling *InitHistory*.

Changing screen modes

Turbo Vision keeps track of the current screen mode in a bitmapped variable called *ScreenMode*. *ScreenMode* contains a combination of the screen mode constants *smMono*, *smBW80*, *smCO80*, and *smFont8x8*. By default, a Turbo Vision application assumes the screen mode that your DOS environment used when you started up the application. If you were in 25-line color mode, that's what the Turbo Vision application uses. If you were in 50-line VGA text mode, the Turbo Vision application also starts up in that mode.

In most cases, you won't switch among the monochrome, black-and-white, and color modes, since they're usually dependent on the user's hardware. More commonly, you'll toggle between 25-line normal mode and 43- or 50-line high resolution mode. To do that, toggle the *smFont8x8* bit in *ScreenMode* by calling *SetScreenMode*. Listing 10.2 shows part of an application's *HandleEvent* method that responds to a command *cmVideo* by toggling the 8x8 pixel font mode.

Listing 10.2
Toggling high-resolution video mode

```
procedure TSomeApplication.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then
  case Event.Command of
    cmVideo: SetScreenMode(ScreenMode xor smFont8x8);
       ⋮
  end;
end;
```

Customizing the desktop

You will rarely change the desktop object. The default desktop object covers the entire screen, other than the top line and bottom line of the screen, and knows how to manage windows and dialog boxes inserted in it. You might, however, need to change its size or position or want to change the default background pattern.

This section describes how to

- Construct a desktop object
- Insert and execute windows
- Arrange windows
- Change the background pattern

Constructing a desktop object

Application objects call a virtual method called *InitDesktop* to construct a desktop object and assign it to the global variable *Desktop*. By default, *InitDesktop* gets the boundary rectangle of the application and constructs a desktop view of type *TDesktop* that covers all but the first and last lines of the application view.

To construct a desktop that covers a different area, you need to override *InitDesktop*. For example, if your application has no status line, you need to make sure the desktop object covers the line that would normally belong to the status line. You can do this either of two ways: calling the inherited method and modifying the result, or replacing the inherited method entirely.

Using the inherited method

Since you know what the inherited *InitDesktop* method does, you can call that method and then modify the resulting object, changing the desktop boundaries, as shown in Listing 10.3.

Listing 10.3
Modifying the default desktop object

```
procedure TMyApplication.InitDesktop;
var R: TRect;
begin
  inherited InitDesktop;                { construct default desktop }
  Desktop^.GetExtent(R);                       { get its boundaries }
  Inc(R.B.Y);                            { move bottom down one line }
  Desktop^.Locate(R);                   { set boundaries to new size }
end;
```

Replacing the inherited method

You can also create an entirely separate desktop object, rather than relying on the inherited method. Listing 10.4 shows an *InitDesktop* method that constructs a desktop object that covers the same area as that created in Listing 10.3.

Listing 10.4
Replacing the inherited desktop object

```
procedure TMyApplication.InitDeskTop;
var R: TRect;
begin
  GetExtent(R);                  { get the application's boundaries }
  Inc(R.A.Y);                 { move top line to allow for menu bar }
  New(DeskTop, Init(R));      { construct desktop with those bounds }
end;
```

The advantage to this approach is that it is a bit quicker, but it relies on knowledge of the inherited method. That is, using the inherited method assures you that any actions the inherited method performs are still performed. If you replace the method, you must ensure that you duplicate all the actions of the method you're replacing.

Inserting and executing windows

In nearly all cases, the desktop object owns all window and dialog box objects in an application. Since the desktop is a group, you can use the usual *Insert* and *Execute* methods to insert non-modal and modal views, respectively. However, the application offers a better, safer way to handle inserting and executing.

Inserting non-modal windows

The application object inherits a method called *InsertWindow* that takes a window object as its parameter, and makes sure the window is valid before inserting it into the desktop. Using *InsertWindow* rather than inserting windows directly into the desktop ensures that any windows in the desktop have passed two tests of validity, so you can be reasonably sure you've avoided problems.

InsertWindow performs two tests on the window object:

- Calls *ValidView* to make sure constructing the window didn't cause the memory manager to reach into its safety pool.
- Calls the window's *Valid* method, passing the parameter *cmValid*, which returns *True* only if the window and all its subviews constructed correctly.

If both *Valid* and *ValidView* indicate that the window is viable, *InsertWindow* calls the desktop object's *Insert* method to insert the window. If the window fails either test, *InsertWindow* does not insert the window, disposes of the window, and returns *False*.

Executing modal views

The application's *ExecuteDialog* method is much like *InsertWindow*. The difference is that after determining the validity of the window object, *ExecuteDialog* calls the desktop's *Execute* method to make the window modal, rather than inserting it. As the name implies, *ExecuteDialog* is designed with dialog boxes in mind, but you can pass any window object you want to make modal.

GetData and SetData are explained in Chapter 12, "Control objects."

ExecuteDialog also takes a second parameter, a pointer to a data buffer for use by *GetData* and *SetData*. If the pointer is **nil**, *ExecuteDialog* skips the *GetData*/*SetData* process. If the pointer is non-**nil**, *ExecuteDialog* calls *SetData* before executing the window and calls *GetData* if the user didn't cancel the dialog box.

Arranging windows

Desktop objects know how to arrange the windows they own in two different ways: *tiling* and *cascading*. Tiling means arranging and resizing the windows like tiles, so that none overlap. Cascading means arranging the windows in descending size from the top left corner of the desktop. The first window covers the entire desktop, the next moves to the right and down one space, and so on. The result is a stack of windows that cascade down the desktop, with the title bar and left side of each window visible.

Tiling or cascading are handled by *TApplication* methods called *Tile* and *Cascade*, respectively. By default the *HandleEvent* method in *TApplication* binds *Tile* and *Cascade* to the commands *cmTile* and *cmCascade*. Those commands come from the Tile and Cascade items on the standard Window menu.

☞ In order to automatically tile or cascade windows, the windows must have their *ofTileable* bit set. By default, window objects have *ofTileable* set; dialog box objects do not. If you're going to use modeless dialog boxes that you will want to tile or cascade, be sure to set *ofTileable* in the object's constructor.

Setting the arrangement region

By default, tiling and cascading use the entire desktop. If you want to change the region used for tiling, your application object must override the virtual method *GetTileRect*.

For example, if you have a non-tileable message window that always covers the last four lines of the desktop, you can arrange the other windows to cover only the area above the message window:

```
procedure TTileApp.GetTileRect(var R: TRect);
begin
  Desktop^.GetExtent(R);                { get the area of the desktop }
  R.B.Y := R.B.Y - 4;                 { but exclude the last four lines }
end;
```

Setting tile direction

The desktop enables you to control which way it tiles windows, horizontally or vertically. By default, windows tile vertically, meaning that if have two windows and tile them, one appears above the other. If you set the desktop's *TileColumnsFirst* field to *True*, the desktop will favor horizontal tiling. With *TileColumnsFirst* set *True*, tiling two windows places them side-by-side.

Changing the background

The desktop object owns one other view by default, even before you insert any windows, and that's the background view. The background view is a very simple view that doesn't *do* anything, but it draws itself in any otherwise uncovered portion of the desktop. In Z-order, the background is behind all other views, and since it's not selectable, it always stays there. The desktop stores a pointer to its background view in a field called *Background*.

Probably the only thing you'd ever want to do to the background is change its background pattern. The default desktop object displays a single character repeatedly over it's entire area. Changing that single character is simple. Changing the background to draw more than the single character is slightly more complicated.

Changing the pattern character

The easiest way to change the background's pattern character is to wait until the desktop creates its default background. You can then change the background object's *Pattern* field, which holds the repeating character. The following example replaces the default background character with the letter C.

```
procedure TMyApplication.InitDesktop;
begin
  inherited InitDesktop;                   { construct default desktop }
  Desktop^.Background^.Pattern := 'C';     { change pattern character }
end;
```

The initial value of the background's pattern character is passed as a parameter to the background view's constructor, which is called by the desktop object's virtual method *InitBackground*. If you derive your own desktop object, you can override *InitBackground* to pass the desired character when you construct the background, rather than changing it later. However, since the only reason you'd be defining a new desktop object is to create a more complex background, you should probably just plug a new value into *Pattern* from within *InitDesktop*.

Drawing a complex background

Drawing a background with a pattern of more than one character requires you to derive two new objects: a background object that draws itself the way you want, and a desktop object that uses your specialized background instead of the standard *TBackground*.

The program in Listing 10.5 implements a background object that repeats a given string over the entire desktop.

Listing 10.5
Creating a complex desktop background

```
program NewBack;

uses Objects, Drivers, Views, App;

type
  PMyBackground = ^TMyBackground;
  TMyBackground = object(TBackground)
    Text: TTitleStr;
    constructor Init(var Bounds: TRect; AText: TTitleStr);
    procedure Draw; virtual;
  end;
  PMyDesktop = ^TMyDesktop;
  TMyDesktop = object(TDesktop)
    procedure InitBackground; virtual;
  end;
  TMyApplication = object(TApplication)
    procedure InitDesktop; virtual;
  end;

constructor TMyBackground.Init(var Bounds: TRect; AText: TTitleStr);
begin
  inherited Init(Bounds, ' ');                      { construct the view }
  Text := AText;                                              { get text }
  while Length(Text) < SizeOf(TTitleStr) - 1 do
    Text := Text + AText;                       { fill the entire string }
end;

procedure TMyBackground.Draw;
var DrawBuffer: TDrawBuffer;
begin
  MoveStr(DrawBuffer, Text, GetColor(1));   { put string into buffer }
  WriteLine(0, 0, Size.X, Size.Y, DrawBuffer);           { write text }
end;

procedure TMyDesktop.InitBackground;
var R: TRect;
begin
  GetExtent(R);                                { get desktop rectangle }
  Background := New(PMyBackground, Init(R, 'Turbo Vision '));
end;

procedure TMyApplication.InitDesktop;
var R: TRect;
begin
  GetExtent(R);                            { get application rectangle }
  R.Grow(0, -1);                    { allow for menu bar, status line }
  Desktop := New(PMyDesktop, Init(R));    { construct custom desktop }
end;

var MyApp: TMyApplication;
begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.
```

The key to the background object is its *Draw* method. You can accomplish fairly dazzling effects if you work at it. Keep in mind, however, that the usual purpose of a background is to provide a neutral background behind the work your users will do, so you don't want it to be too distracting.

Shelling to DOS

TApplication provides an easy way to allow users of your application to start a DOS shell. In response to the *cmDosShell* command from the standard File menu, *TApplication* calls its *DosShell* method.

DosShell shuts down some of the application's subsystems before actually starting the shell, then restarts them when the user exits the shell. The command interpreter used by the shell is the one specified by the COMSPEC environment variable.

Customizing the shell message

Before executing the command interpreter, *DosShell* calls a virtual method called *WriteShellMsg* to display the following message:

```
Type EXIT to return...
```

You can customize the message by overriding *WriteShellMsg* to display any other text. You should use the *PrintStr* procedure instead of *Writeln*, however, to avoid linking in unneeded code. The following code displays a different message:

```
procedure TShellApp.WriteShellMsg;
begin
  PrintStr('Leaving Turbo Vision for DOS. Type EXIT to return.');
end;
```

Customizing the status line

The default application object constructor calls a virtual method *InitStatusLine* to construct and initialize a status line object. To create a custom status line, you need to override *InitStatusLine* to construct a new status line object and assign it to the global variable *StatusLine*. The status line serves three important functions in the application:

- Showing commands the user can click with the mouse
- Binding hot keys to commands
- Providing context-sensitive hints to the user

The first two functions are set up when you construct the status line object. Context-sensitive hints, on the other hand, are controlled by a status line object method called *Hint*.

The status line object constructor takes two parameters: a boundary rectangle and a pointer to a linked list of *status definitions*. A status definition is a record that holds a range of help contexts and the list of *status keys* the status line displays when the application's current help context falls within that range. Status keys are records that hold commands and the text strings and hot keys that generate the commands.

Constructing a status line consists of three steps:

- Setting the boundaries of the view
- Defining status definitions
- Defining status keys

Defining status line boundaries

The status line nearly always appears on the bottom line of an application, but you can put it anywhere you want. You can even make an invisible status line that the user can't see or click, but which still binds hot keys to commands.

The easiest way to place a status line on the bottom line of the screen is to base its location on the bounding rectangle of the application object, as shown in Listing 10.6.

Listing 10.6
Setting the status line boundaries

```
procedure TYourApplication.InitStatusLine;
var R: TRect;
begin
  GetExtent(r);                     { get the application's boundaries }
  R.A.Y := R.B.Y - 1;                  { set top one line above bottom }
   ⋮
                    { use R as the bounding rectangle of status line }
end;
```

Using invisible status lines

To use an invisible status line object, you can either assign a bounding rectangle that's off the screen (for example, one line below the bottom of the application's bounds) or to an empty rectangle. For example, if you change the assignment in Listing 10.6 to

```
R.A.Y := R.B.Y;
```

the status line has no height, and therefore doesn't appear on the screen. Be sure to adjust the desktop object's boundaries to cover the area the status line would normally cover.

Creating status definitions

Status definition records are normally created using the function *NewStatusDef*, which makes it easy to create the linked list of records by nesting calls to *NewStatusDef*. *NewStatusDef* takes four parameters:

- The low boundary of the help context range
- The high boundary of the help context range
- A pointer to a linked list of status keys
- A pointer to the next status definition record, if any

The default status line object created by *TProgram's InitStatusLine* method is very simple. It has only a single status defintion which gets its list of status keys from the function *StandardStatusKeys*:

```
procedure TProgram.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);                           { get application boundaries }
  R.A.Y := R.B.Y - 1;                           { use only bottom line }
  New(StatusLine, Init(R,               { construct StatusLine using R }
    NewStatusDef(0, $FFFF,          { cover all possible help contexts }
      NewStatusKey('~Alt+X~ Exit', kbAltX, cmQuit,      { show Alt+X }
      StdStatusKeys(nil)), nil)));           { include standard keys }
end;
```

For simple applications, a single status line for the entire range of help contexts is probably enough. If your application has different views that might need different commands available on the status line, you can provide them by giving those views different help contexts and creating appropriate status definitions for each.

The simple program in Listing 10.7 (included on your distribution disks in the file TWOSTAT.PAS) shows how status lines change with help contexts.

Listing 10.7 TWOSTAT.PAS shows status lines changing with help contexts.

```
program TwoStat;
uses Objects, Drivers, Views, App, Menus;
type
  TStatApp = object(TApplication)
    constructor Init;
    procedure InitStatusLine; virtual;
  end;
```

```
constructor TStatApp.Init;
var
  R: TRect;
  Window: PWindow;
begin
  inherited Init;
  Desktop^.GetExtent(R);
  R.B.X := R.B.X div 2;
  Window := New(PWindow, Init(R, 'Window A', 1));
  InsertWindow(Window);
  Desktop^.GetExtent(R);
  R.A.X := R.B.X div 2;
  Window := New(PWindow, Init(R, 'Window B', 2));
  Window^.HelpCtx := $8000;
  InsertWindow(Window);
end;

procedure TStatApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  New(StatusLine, Init(R,
    NewStatusDef(0, $7FFF,
      NewStatusKey('~F6~ Go to B', kbF6, cmNext,
      StdStatusKeys(nil)),
    NewStatusDef($8000, $FFFF,
      NewStatusKey('~F6~ Go to A', kbF6, cmNext,
      StdStatusKeys(nil)), nil))));
end;

var StatApp: TStatApp;
begin
  StatApp.Init;
  StatApp.Run;
  StatApp.Done;
end.
```

Creating status keys

Once you've set up status definitions, each of them needs a list of status keys. A status key record consists of four fields:

- A text string that appears on the status line
- A keyboard scan code for a hot key
- A command to generate
- A pointer to the next status key record, if any

Using the NewStatusKey function

The easiest way to create a list of status keys is to make nested calls to the function *NewStatusKey*. Creating a simple, single-item status key list takes only one such call:

```
NewStatusKey('~Alt-Q~ Quit', kbAltQ, cmQuit, nil);
```

To create a longer list, replace **nil** with another call to *NewStatusKey*:

```
NewStatusKey('~Alt-Q~ Quit', kbAltQ, cmQuit,
NewStatusKey('~F10~ Menu', kbF10, cmMenu, nil));
```

Using status key functions

If you use the same set of status keys for several different status definitions, or even in several different applications, you'll probably want to group them together in a function. The *App* unit provides one such function for the common comands you use most, called *StdStatusKeys*. Listing 10.8 shows the declaration of *StdStatusKeys*.

Listing 10.8
The StdStatusKeys function

```
function StdStatusKeys(Next: PStatusItem): PStatusItem;
begin
  StdStatusKeys :=
    NewStatusKey('', kbAltX, cmQuit,
    NewStatusKey('', kbF10, cmMenu,
    NewStatusKey('', kbAltF3, cmClose,
    NewStatusKey('', kbF5, cmZoom,
    NewStatusKey('', kbCtrlF5, cmResize,
    NewStatusKey('', kbF6, cmNext,
    Next))))));
end;
```

Notice that by providing a pointer to a next item, you can use a function like *StdStatusKeys* in the middle of a list of keys, rather than just at the end.

Adding status line hints

The status line object type provides a virtual method called *Hint* that you can override to provide context-sensitive status line information to the right of any displayed status keys. *Hint* takes a help context number as its single parameter and returns a string based on that number. The default *Hint* inherited from *TStatusLine* returns a null string for any input, so you have to override *Hint* to get any meaningful messages.

Listing 10.9
A program that gives context-sensitive status line hints

```
program Hinter;
uses Objects, Drivers, Menus, Views, App;

const
  hcFile = 1001; hcFileNew = 1002; hcFileOpen = 1003;
  hcFileExit = 1004; hcTest = 1005; hcWindow = 1100;
  cmFileNew = 98; cmFileOpen = 99;

type
  PHintStatusLine = ^THintStatusLine;
  THintStatusLine = object(TStatusLine)
    function Hint(AHelpCtx: Word): String; virtual;
  end;
  THintApp = object(TApplication)
    constructor Init;
    procedure InitMenuBar; virtual;
    procedure InitStatusLine; virtual;
  end;

function THintStatusLine.Hint(AHelpCtx: Word): String;
begin
  case AHelpCtx of
    hcFile: Hint := 'This is the File menu';
    hcFileNew: Hint := 'Create a new file';
    hcFileOpen: Hint := 'Open an existing file';
    hcFileExit: Hint := 'Terminate the application';
    hcTest: Hint := 'This is a test. This is only a test.';
    hcWindow: Hint := 'This is a window';
  else Hint := '';
  end;
end;

constructor THintApp.Init;
var
  R: TRect;
  Window: PWindow;
begin
  inherited Init;
  Desktop^.GetExtent(R);
  Window := New(PWindow, Init(R, 'A window', wnNoNumber));
  Window^.HelpCtx := hcWindow;
  InsertWindow(Window);
end;

procedure THintApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcFile, NewMenu(
      NewItem('~N~ew', '', kbNoKey, cmFileNew, hcFileNew,
      NewItem('~O~pen...', 'F3', kbF3, cmFileOpen, hcFileOpen,
      NewLine(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcFileExit,
      nil))))),
    NewItem('~T~est', '', kbNoKey, cmMenu, hcTest, nil)))));
end;

procedure THintApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);  R.A.Y := R.B.Y - 1;
  StatusLine := New(PHintStatusLine, Init(R,
    NewStatusDef(0, $FFFF, StandardStatusKeys(nil), nil)));
end;

var HintApp: THintApp;
begin
  HintApp.Init;
  HintApp.Run;
  HintApp.Done;
end.
```

In a complex application that shows a lot of different hints, you should use a string list resource to supply the strings instead of the lengthy **case** statement in *Hint*.

Updating the status line

You should never need to update the status line manually. The application object's *Idle* method calls the status line object's *Update* method, so the status line keys and hints should never get out of date.

Customizing menus

A menu in Turbo Vision has two parts: a *menu list* that holds the descriptions of the menu items and the commands they generate, and a *menu view* that shows those items on the screen.

Turbo Vision defines two kinds of menu views: *menu bars* and *menu boxes*. Both views use exactly the same underlying lists of menu items. In fact, the same menu items can show up in either a bar or a box. The main difference is that a menu bar can only be a top level menu, usually permanently located across the top line of the application screen. A menu box can either be the primary menu (usually a pop-up, or local menu) or more often a submenu brought up by an item on a menu bar or another menu box.

The application's *Init* constructor calls a virtual method called *InitMenuBar* to construct a menu bar and assign it to the variable *MenuBar*. To define your own menu bar, you need to override *InitMenuBar* to create your special menu bar and assign it to *MenuBar*.

Creating a menu bar takes two main steps:

- Setting the menu bar boundaries
- Defining the menu items

Setting menu bar boundaries

Menu bars nearly always occupy the top line of the application screen. The best way to assure that your menu bar covers this top line is to set its boundaries based on those of the application, as shown in listing 10.10.

Listing 10.10
Ensuring that the menu bar is the top line

```
procedure TYourApplication.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);                       { get application's boundaries }
  R.B.Y := R.A.Y +1;                 { set bottom one line below top }
   ⋮                                  { use R to initialize menu bar }
end;
```

Unlike menu bars, menu boxes adjust their boundaries to accommodate their contents, so you don't have to worry about setting the sizes of each submenu. You just set the boundaries of the menu bar, and the menu objects take care of the rest.

Defining menu items

The menu system uses two different kinds of records to define a menu structure. Each of the record types is designed for use in a linked list, having a pointer field to the next record.

- *TMenu* defines a list of menu items and keeps track of the default, or selected, item. Each main menu and submenu holds one *TMenu* record. The list of items is a linked list of *TMenuItem* records.

- *TMenuItem* defines the text, hot key, command, and help context of a menu item. Every item in a menu, whether a command or a submenu, has its own *TMenuItem* record.

When the menu is displayed as a bar, the hot key string is hidden, although the hot key is still active.

Using the NewItem function

The usual way to allocate and initialize a menu item record is with the function *NewItem*. You can easily create lists of items by nesting calls to *NewItem*.

Using the NewSubMenu function

A submenu is a menu item that brings up another menu instead of generating a command. Generally, you create submenus by calling the function *NewSubMenu* in place of *NewItem*. There are really only two differences between *NewSubMenu* and *NewItem*:

- The submenu has no associated command, so *NewSubMenu* sets the item's *Command* field to zero, and there is no hot key assigned or described.
- In addition to pointing to the next item in its menu, the submenu points to a *TMenu* record, which contains the list of items in the submenu.

Using idle time

The application object's event loop calls a virtual method called *Idle* whenever it finds no pending events in the event queue. That means that you can use Turbo Vision to animate background processes when it's not responding to user input.

To create a background process, you just override *Idle* and have it perform whatever task you want to do in the background. Be sure, however, to call the inherited *Idle* method, because the default *Idle* takes care of such things as updating the status line and notifying views that commands have been enabled or disabled.

☞ Make certain that any background processing you put into *Idle* doesn't take too much time, or it will cause the application to respond sluggishly to the user.

The *TVDemo* program on your distribution disks uses two views from a unit called *Gadgets*. One of them is a clock view that updates the time when told to do so by the application's *Idle* method. The other is an indication of the amount of heap space available, which is also updated by *Idle.*

Context-sensitive Help

Turbo Vision has built-in tools that help you implement context-sensitive help within your application. You can assign a help context number to a view, and Turbo Vision ensures that whenever that view becomes focused, its help context number will become the application's current help context number.

To create global context-sensitive help, you can implement a *HelpView* that knows about the help context numbers that you've defined. When *HelpView* is invoked (usually by the user pressing *F1* or some other hot key), it should ask its owner for the current help context by calling the method *GetHelpCtx*. *HelpView* can then read and display the proper help text. An example *HelpView* is included on your Turbo Pascal distribution disks.

Context-sensitive help is probably one of the last things you'll want to implement in your application, so Turbo Vision objects are initialized with a default context of *hcNoContext*, which is a predefined context that doesn't change the current context. When the time comes, you can work out a system of help numbers, then plug the right number into the proper view by setting the view's *HelpCtx* field right after you construct the view.

Help contexts are also used by the status line to determine which views to display. Remember that when you create a status line, you call *NewStatusDef*, which defines a set of status items for a given range of help context values. When a new view receives the focus, the help context of that item determines which status line is displayed.

CHAPTER

11

Window and dialog box objects

Window objects are specialized group views that provide the distinctive framed, overlapping, titled windows that Turbo Vision applications have on the desktop. Dialog boxes are specialized windows, so anything described in this chapter that specifies windows applies equally to dialog boxes. This chapter also describes the properties unique to dialog box objects.

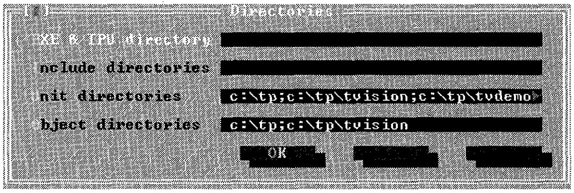
This chapter covers the following topics:

- Understanding windows and dialog boxes
- Working with windows
- Working with dialog boxes
- Using controls with dialog boxes
- Using standard dialog boxes

Understanding windows

Windows in Turbo Vision are different than windows in other systems you might have used in the past. Instead of being a subset of the screen that you can read from and write to, a Turbo Vision window is a group view. This distinction is made clearest by looking at a simple descendant of the window, the dialog box.

Figure 11.1 shows a typical dialog box that contains various controls. It's pretty clear from looking at the dialog box how the user interacts with it by typing in the input line, clicking buttons, and so on. The user doesn't expect to be able to type on the background areas.



Directories

XE & TPU directory

nclude directories

nit directories c:\tp;c:\tp\tvision;c:\tp\tvdemo

bject directories c:\tp;c:\tp\tvision

Figure 11.1
A typical dialog box

In that respect, a dialog box is no different from any other Turbo Vision window. It's not something you write on, but rather it's a holder for other views. If you want text to show up in a window, you insert a text view into the window.

How windows and dialog boxes differ

For the most part, window and dialog box objects are interchangeable. Dialog boxes, however, have a few additional refinements that make them particularly suited for use as modal views. Remember that *any* group view can be modal, including windows, dialog boxes, and applications. However, dialog boxes include some behavior by default that users expect from modal views.

Dialog box objects handle events slightly differently than other windows. They

- Convert *Esc* keystrokes into *cmCancel* commands
- Convert *Enter* keystrokes to *cmDefault* broadcasts
- Close the dialog box (end the modal state) in response to standard commands *cmOK*, *cmCancel*, *cmYes*, and *cmNo*

Working with windows

This section describes the various tasks you perform on all window objects, including dialog box objects:

- Constructing window objects
- Inserting windows into the desktop
- Working with modal windows
- Changing window object defaults
- Managing window sizes
- Creating window scroll bars

Constructing window objects

Window objects provide a certain amount of flexibility that allows you to customize their behavior without having to derive new window types.

This section covers the following topics:

- Constructing the default window object
- Changing window flags

Constructing default windows

The default window object constructor takes three parameters: a bounding rectangle, a title string, and a window number. The default window creates a group view with the given boundaries, sets its title field to point to a copy of the title string, stores the window number, and sets its state and options flags to give the window a shadow and make it selectable.

Once you've called the window's constructor, you can modify any of its fields as you would any other object. For example, to force the window to be centered when you insert it on the desktop, set the *ofCentered* flag in its *Options* field:

```
    ⋮
Window := New(PWindow, Init(R, 'A window title', wnNoNumber));
Window^.Options := Window^.Options or ofCentered;
Application^.InsertWindow(Window);
    ⋮
```

Changing window flags

In addition to the standard view option flags, window objects have a bitmapped *Flags* field that governs certain kinds of moving and resizing behavior. The bits in the window flags field are identified by constants starting with *wf*. Table 11.1 describes the purpose of each of the flags.

Table 11.1
Window flag meanings

Flag	Meaning
wfMove	User can move the window by dragging the title bar.
wfGrow	User can resize the window by dragging the bottom right corner.
wfClose	User can close the window by clicking an icon in the top left corner.
wfZoom	User can zoom or unzoom the window by clicking an icon in the top right corner.

By default, windows have all four of the window flags set.

Inserting windows into the desktop

Windows are normally inserted into the application's desktop group, since you usually want the window to appear in the area between the menu bar and status line views without overlapping either of them. Inserting into the desktop ensures that windows will be clipped at the desktop boundary.

For information on the safety checks performed by InsertWindow, see Chapter 10, "Application objects."

The best way to insert a window into the desktop is to call the application object's method *InsertWindow*. *InsertWindow* performs two validity checks on the window object before inserting it. This ensures that when it does insert a window, the user will be able to use that window.

InsertWindow is a function. It returns a pointer to the window passed as a parameter if the window was valid, or **nil** if the window wasn't valid. If the window wasn't valid, *InsertWindow* disposes of it, so you don't need to access the pointer again. In fact, in many case you'll probably not even bother to check the function result. Because *InsertWindow* takes care of both valid and invalid windows completely, you can take advantage of extended syntax (the **$X+** compiler directive) to treat *InsertWindow* like a procedure.

The program in Listing 11.1 shows a typical use of *InsertWindow* as a procedure. The file INSWIN.PAS on your distribution disks contains the same program.

Listing 11.1
Inserting windows with InsertWindow

```
program InsWin;
uses Objects, App, Drivers, Views, Menus;

const cmNewWin = 2000;

type
  TInsApp = object(TApplication)
    WinCount: Integer;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitMenuBar; virtual;
  end;

procedure TInsApp.HandleEvent(var Event: TEvent);
var R: TRect;
begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    if Event.Command = cmNewWin then
```

```
      begin
        Inc(WinCount);
        Desktop^.GetExtent(R);
        InsertWindow(New(PWindow, Init(R, 'Test window', WinCount)));
      end;
    end;
  end;

  procedure TInsApp.InitMenuBar;
  var R: TRect;
  begin
    GetExtent(R);
    R.B.Y := R.A.Y + 1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
      NewItem('~A~dd window', 'F3', kbF3, cmNewWin, hcNoContext,
        nil))));
  end;

  var InsApp: TInsApp;
  begin
    InsApp.Init;
    InsApp.Run;
    InsApp.Done;
  end.
```

Executing modal windows

Executing a modal window is similar to inserting a window into the desktop. The two exceptions are that the window becomes the application's current modal view, and that you can pass a data record to the window for initializing its controls.

Using modal windows requires you to understand three tasks:

- Making the window modal
- Ending the modal state
- Handling a data record

Making a window modal

Executing a window is simple. Once you've constructed the window object, you pass it to an application object method called *ExecuteDialog*. As the name implies, you'll usually use *ExecuteDialog* with dialog boxes, but you can execute any window object.

ExecuteDialog takes two parameters, a pointer to the window object, and a pointer to a data record for initializing the window controls, as described in the next section. A simple use of *ExecuteDialog* looks something like this:

```
MyWindow := New(PWindow, Init(R, 'This will be modal', wnNoNumber));
ExecuteDialog(MyWindow, nil);
```

Passing **nil** as the data record pointer bypasses the automatic setting and reading of control values.

Ending the modal state

The only "trick" to dealing with modal windows, actually, is making sure you provide a way to end the modal state. All window objects inherit an *EndModal* method from *TGroup,* but you have to make sure your object calls *EndModal* in response to some event or events. Dialog box objects have that capacity built in to their *HandleEvent* methods by default, but if you want to execute other window objects, you need to add that yourself.

Handling data records

ExecuteDialog automatically supports setting and reading the window's controls. The second parameter passed to *ExecuteDialog* points to a data record for the controls in the window. Data records are explained in the section "Manipulating controls" on page 206.

After it executes a window, *ExecuteDialog* calls the window's *SetData* method, passing the data record pointed to by the second parameter. When the user terminates the window's modal state without canceling (in other words, calling *EndModal* with any command other than *cmCancel*), *ExecuteDialog* calls *GetData* to read the values of the controls back into the data record.

Changing window defaults

Once you've constructed a window object, there are several aspects of its appearance and behavior you can change. This section explains the following:

- Using standard window palettes
- Changing the window title
- Altering the window frame
- Using window numbers

Using standard window palettes

Turbo Vision uses three standard color schemes for window objects. The default color scheme is a blue window with a white frame, yellow text, green frame icons, and cyan scroll bars. The alternate color schemes are for gray windows (the default used by dialog boxes), and cyan windows (which the IDE uses for message and watch windows).

For complete information on color mapping, see Chapter 14, "Palettes and color selection."

The color scheme for a given window is controlled by the window object's *Palette* field. By default, the window object's constructor sets *Palette* to *wpBlueWindow*. To change to one of the other palettes, set *Palette* to either *wpCyanWindow* or *wpGrayWindow*. The window object's *GetColor* method uses the value of *Palette* to determine how to map colors onto the application object's palette.

For example, the constructor in Listing 11.2 creates a window that uses the cyan window palette.

Listing 11.2 Changing the window palette

```
constructor TCyanWindow.Init(var Bounds: TRect; ATitle: TTitleStr;
  ANumber: Integer);
begin
  inherited Init(Bounds, ATitle, ANumber);           { default window }
  Palette := wpCyanWindow;                    { change window palette }
end;
```

Changing the window title

The window object stores the title string passed to its constructor in a field called *Title*. In general, however, you should access the title string through the window object's *GetTitle* method, which provides the opportunity to limit the length of the title string. In general, the only part of the program that ever needs to access the window title is the window's frame object, which calls the window's *GetTitle* when it draws itself.

GetTitle takes a single integer-type parameter that you can use to limit the length of the returned string. By default, *GetTitle* ignores the length parameter and returns the entire *Title* string, which the frame then truncates if it exceeds the specified length. In many cases, it doesn't matter what part of the string gets truncated. However, if you want to preserve certain information, you can override *GetTitle* to return a string of the appropriate length that retains the crucial information.

You can also use *GetTitle* to return different titles depending on certain circumstances. For example, the type *TEditWindow* normally displays the full path name of the file in the editor. If the file doesn't yet have a name, *GetTitle* returns the string 'Untitled' instead.

Altering the window frame

By default, a window object constructs an instance of type *TFrame* to serve as its frame. Frame objects are very simple, and you will rarely need to alter them. However, Turbo Vision makes it easy to change the frame if you want to.

The default window object constructor *Init* calls a virtual method *InitFrame* to construct a frame object and assign it to the window's *Frame* field. After calling *InitFrame, Init* checks to make sure *Frame* is non-**nil**, and inserts it if it can.

To construct a different frame, override *InitFrame* to construct an instance of some type derived from *TFrame* and assign that object to *Frame. Init* will then insert your derived frame into the window.

Using window numbers

The last parameter passed to the default window constructor is a number, which the window stores in its *Number* field. If the number is between 1 and 9, the number appears on the window's frame, to the right of the title, near the zoom icon. By default, the keystrokes *Alt-1* through *Alt-9* select (activate and bring to the front) the windows with the corresponding numbers.

Turbo Vision provides no mechanism for tracking which numbers you have assigned and which are available. If you want to take advantage of window numbers, your application must manage the numbers itself. Turbo Vision only handles assigning the passed numbers to the *Number* field and selecting the windows when selected with the *Alt* keystrokes.

Turbo Vision also supplies the mnemonic constant *wnNoNumber*, which you can pass to a window's constructor to indicate that the window has no specific number.

Managing window size

By default, users can resize windows by dragging the bottom right corner to the desired position or zoom the window to fill the desktop by clicking the zoom icon. Turbo Vision gives you a measure of control over both of these behaviors, allowing you to put limits on the size of windows and set the "unzoomed" size of the window.

Limiting window size

Like all views, the minimum and maximum sizes of a window are determined by the virtual method *SizeLimits. TWindow* makes one important change to *SizeLimits*, however. By default, a view's minimum size is zero. *TWindow* overrides this to set the minimum window size to the value stored in the variable *MinWinSize.*

By default, *MinWinSize* restricts windows to a minimum of 16 columns wide and 6 lines tall, which ensures that the size corner, close icon, and zoom icon are all visible, plus at least some of the title. You might want to override *SizeLimits* for special types of windows, such as to make sure that scroll bars on the frame are still usable.

Zooming windows

Every window object has a virtual method called *Zoom* that toggles the size of the window between filling the desktop entirely and a particular "unzoomed" size specified by the window object field *ZoomRect*. *ZoomRect* initially holds the boundaries of the window when it was constructed. When you zoom a window to fill the desktop, *ZoomRect* records the size the window had before zooming.

If you want to change the zooming behavior of a particular window type (for instance, to always set the unzoomed size to a particular value), you can override *Zoom*. You will probably *not* want to call the *Zoom* inherited from *TWindow* in your descendant's method, since *TWindow.Zoom* sets the value of *ZoomRect* to the current size of the window if the window is not filling the desktop.

Creating window scroll bars

The *TWindow* object type provides a function for generating window scroll bars. If you have a windows whose entire contents need to scroll, calling the method *StandardScrollBar* constructs, inserts, and returns a pointer to a scroll bar object on the frame of the window.

StandardScrollBar takes a single parameter that specifies the kind of scroll bar you want. If you pass *sbVertical*, the method returns a vertical scroll bar on the left side of the window frame. Passing *sbHorizontal* produces a horizontal scroll bar on the bottom of the window frame.

You can combine *sbHandleKeyboard* with either *sbVertical* or *sbHorizontal* (using the **or** operator) to enable the resulting scroll bar to respond to arrow and page keys in addition to mouse clicks.

The window constructor in Listing 11.3 uses *StandardScrollBar* to create scroll bars for a scrolling interior that fills a window. Notice that you don't have to insert the window scroll bars as you would normally.

Listing 11.3
Creating standard window scroll bars

```
constructor TScrollWindow.Init(var Bounds: TRect; ATitle: TTitleStr;
  ANumber: Integer);
var
  R: TRect;
  Interior: PScroller;
begin
  inherited Init(Bounds, ATitle, ANumber);        { construct window }
  GetExtent(R);                               { get window boundaries }
  R.Grow(-1, -1);                                  { shrink rectangle }
  Interior := New(PScroller, Init(R,        { construct scroller in R }
    StandardScrollBar(sbHorizontal or sbHandleKeyboard),
    StandardScrollBar(sbVertical or sbHandleKeyboard)));
  Insert(Interior);                             { insert the scroller }
end;
```

Working with dialog boxes

Dialog boxes can do anything any other window object can do. The main differences between dialog box objects and window objects are that dialog box objects have different default attributes, built-in support for modal operation, and adaptations for handling control objects. This section discusses dialog box attributes and modal operation. Use of controls has its own section, starting on page 205.

Dialog box default attributes

The default properties of a dialog box object differ only slightly from those of other window objects. The dialog box constructor takes only two parameters instead of three, since dialog boxes default to having no window number.

The following are the differences between default dialog boxes and default window objects:

- Gray color scheme (*Palette* is *wpGrayWindow*)
- No window number
- Fixed size, so *GrowMode* is zero and *Flags* excludes *wfGrow* and *wfZoom*

These differences affect dialog boxes whether you use them as modeless windows or execute them as modal dialog boxes.

Modal dialog box behavior

Dialog box objects have two methods that streamline their use as modal views: *HandleEvent* and *Valid.*

Handling dialog box events

Dialog box objects handle most events just like regular window objects, but make two changes you'll only notice when you use the dialog box as a modal view:

- The *Enter* and *Esc* are handled specially.

  Enter broadcasts a *cmDefault* message to the dialog box, causing the default button to act as if it had been pressed. *Esc* is translated into a *cmCancel* command.
- Certain commands automatically end the modal state.

  The *cmOk, cmCancel, cmYes,* and *cmNo* commands all produce calls to *EndModal,* with the command passed as the parameter.

Using controls in a dialog box

A common use of dialog box objects is as a holder for *controls.* Controls are specialized views that allow user interaction such as push buttons, list boxes, and scroll bars. Although you can insert controls into a window object, dialog boxes are specifically adapted to handle them.

Adding controls to a dialog box

Adding controls to a window is just like adding any other subviews, and it's normally part of the window's constructor. After calling the inherited window constructor, you can construct and insert control objects, as shown in Listing 11.4.

Listing 11.4
Adding controls in a dialog box's constructor

```
constructor TCtlWindow.Init(var Bounds: TRect; ATitle: TTitleStr;
  ANumber: Integer);
var R: TRect;
begin
  inherited Init(Bounds, ATitle, ANumber);    { construct the window }
  R.Assign(5, 5, 20, 7);
  Insert(New(PInputLine, Init(R, 15)));           { insert a control }
  R.Assign(10, 8, 20, 10);
  Insert(New(PButton, Init(R, 'O~k~', cmOK, bfDefault)));   { button }
end;
```

You need to be conscious of the order in which you insert the controls. The order of insertion establishes the Z-order of the views, which in turn determines the *tab order* of the controls. Tab order is the order in which controls receive focus in a window when the user presses *Tab.*

Tab order is important because it determines

- The order of user interaction
- The order of control initialization

How users see tab order

A good example of how users see tab order is a data entry form. When the user finishes typing in one field and presses *Tab* to move to the next field, focus should move to the next logical control. If the programmer hasn't carefully considered the order of the data entry fields, it annoys users and makes them less productive.

There are no accepted rules governing the order of controls in a dialog box, but in general, it's a good idea to *have* an order. Whether the order goes top-to-bottom or left-to-right, there should be a discernible pattern.

How the programmer sees tab order

As noted earlier, the order of control insertion into a window determines the tab order, so when you write the initialization code for a window, be aware of the order in which you create and insert controls.

An important consideration is not only the actual code that creates and inserts the controls, but also code that sets and reads the controls' values, as described in the next section, "Manipulating controls."

Manipulating controls

To learn about GetData and SetData methods for individual controls, see Chapter 12, "Control objects."

At any time after you construct a window object with controls, you can set or read the values of all the controls using the methods *SetData* and *GetData*. These methods differ from the corresponding methods in controls and other views. All groups, including windows and dialog boxes, inherit *GetData* and *SetData* methods that iterate through their subviews in Z-order, calling the subviews' *GetData* or *SetData* methods.

In the case of a window that contains controls, calling its *SetData* calls the *SetData* method of each control, in order, so that instead of having to manually initialize each control, you can have the window do it for you. The parameter you pass to *SetData* is a record that contains a field for each control in the window.

Defining window data records

To define a data record for a window or dialog box, do the following:

- List each control in Z-order
- Determine the data record for each control
- Create a record with a field for each control

Setting control values

A window object's *SetData* method calls each of its subview's *SetData* methods in Z-order. The data record passed to each subview is a subset of the record passed to the window's *SetData*. The first control in Z-order gets the entire record. If it reads a number of bytes from the record (as reported by its *DataSize* method), *SetData* passes only the remaining part of the record to the next subview. So if the first control reads 4 bytes, the window's *SetData* gives the second subview a record starting four bytes into the original record.

Reading control values

Reading the values of a dialog box's controls is the exact counterpart of setting the values. The dialog box object's *GetData* calls *GetData* for each subview in Z-order. Each subview gets a chance to write a number of bytes (determined by its *DataSize* method) into the data record for the dialog box.

Handling controls in modal dialog boxes

If the second parameter to *ExecuteDialog* is non-**nil**, the application sets the initial values of controls in the dialog box and reads their values when the modal dialog box closes.

The second parameter to *ExecuteDialog* is assumed to be a pointer to a data record for the dialog box. As with all data records for setting and reading control values, you are responsible for ensuring that the indicated record includes the data in the correct order.

After constructing the dialog box and performing validity checks, *ExecuteDialog* calls the window's *SetData* method if the data record pointer is non-**nil**. When the user terminates the window's modal state, *ExecuteDialog* reads the values of the controls back into the same data record by calling *GetData*, unless the modal state terminated with the command *cmCancel*. In that case, no data transfer takes place.

Using standard dialog boxes

Turbo Vision provides three special kinds of dialog boxes you can incorporate into your programs. This section explains how to use each of the following:

- Message boxes
- File dialog boxes
- Change directory dialog boxes

Using message boxes

The Turbo Vision unit *MsgBox* provides two useful functions that display messages on the screen in a dialog box. Although message boxes are not elegant, they are useful for showing error messages or showing information while you debug an application.

The two functions, *MessageBox* and *MessageBoxRect*, differ only in that *MessageBoxRect* takes a bounding rectangle as one of its parameters, while *MessageBox* always uses a 40-column, 9-line box for its messages. For most uses, *MessageBox* is easier to use. You'll probably only need *MessageBoxRect* if you have to show a very large message.

In order to use message boxes, you need to understand two kinds of parameters:

- The message string and its parameters
- Message box flag options

Message strings and parameters

The first two strings passed to *MessageBox* are the string to display and a pointer to an array of parameters for the string. *MessageBox* passes those two parameters directly to the procedure *FormatStr*, which generates an output string by substituting values from the parameter list into the message string. In most cases, you'll probably pass a simple string with no parameters, passing **nil** as the parameter list.

A simple use of a message box is as an About box:

```
procedure TMyApplication.ShowAboutBox;
begin
  MessageBox('My Program version 1.0', nil,
    mfInformation or mfOkButton);
end;
```

The file STRMERR.PAS on your distribution disks gives a more complex example of using a message box to display detailed error messages in a message box.

Setting message box flags

The last parameter to *MessageBox* is a bitmapped word describing the title of the message box and the buttons that should appear in the box. Turbo Vision defines mnemonic constants for each of the flags. You almost always pass a combination of two constants, one setting the title, the other the buttons. For example, the following code generates a confirmation box with buttons labeled Yes, No, and Cancel:

```
MessageBox('Shall I reformat your hard drive now?', nil,
  mfConfirmation or mfYesNoCancel);
```

All the possible flag values are listed in Chapter 19, "Turbo Vision reference."

Using file dialog boxes

One common dialog box type is the file dialog box, used to specify the name of a file to open or save. Turbo Vision's *StdDlg* unit provides a standard dialog box you can use for both loading and saving files.

Using change directory dialog boxes

Another commonly used dialog box is the change directory dialog box, which enables the user to see the disk's directory structure and navigate among subdirectories. Turbo Vision's *StdDlg* unit provides a standard dialog you can use to let users change the current directory.

CHAPTER

12

Control objects

Control objects are specialized views that perform standard user interface functions. This chapter describes the operations common to using all control object types, and provides information on each of the particular controls:

- Using control objects
- Using text controls
- Using scroll bars
- Using check boxes and radio buttons
- Picking from lists
- Displaying an outline
- Getting user input
- Using history lists
- Labeling controls

Using control objects

You can use control objects like any other views. Most often, you put controls in dialog boxes for user input, but you can also use them in windows. Although each kind of control has certain unique properties, there are three general tasks you need to understand for all controls:

- Constructing and inserting controls
- Initializing control values
- Setting and reading control values

Constructing control objects

In general, constructing control objects takes three steps:

- Assigning the bounding rectangle
- Calling the constructor
- Inserting into the owner

You can often combine the second and third steps into a single statement, depending on whether you assign the control object to a variable. For example, Listing 12.1 shows two ways to construct the same static text control.

Listing 12.1
Two ways to construct a control object

```
R.Assign(10, 2, 20, 3);
Control := New(PStaticText, Init(R, 'Borland'));
Insert(Control);

R.Assign(10, 2, 20, 3);
Insert(New(PStaticText, Init(R, 'Borland')));
```

In many cases, the second form, without assigning the control to a variable, is all you need. If you need to access the particular control (for instance, to assign it a label object or to manipulate that particular control from within the program), you should assign it to a variable.

Initializing control objects

Once you've constructed a control object, you can alter its properties or set its initial value. For example, the following code fragment shows how you can set a button object's *ofCenterX* flag to assure that the button stays horizontally centered in its owner window:

```
CenterButton := New(PButton, Init(R, 'O~k~', cmOK, bfDefault));
CenterButton^.Options := CenterButton^.Options or ofCenterX;
Insert(CenterButton);
```

Similarly, you might want to give an input line an initial string value, as this code fragment shows:

```
InitText := New(PInputLine, Init(R, 30));
InitText^.Data^ := Copy('Borland International', 1, 30);
Insert(InitText);
```

☞ You generally don't assign initial values to controls this way when you're using modal dialog boxes. In that case, use a data record to initialize all the controls at once.

Setting and reading control values

In addition to setting initial values for controls (as discussed in the preceding section), there are two situations in which you need to be able to set or read the values of controls:

- When opening or closing a modal dialog box
- At any point in the life of a modeless window or dialog box

Controls take advantage of three methods built into all views to enable your application to set or read the values of a control on demand. Using *DataSize*, *GetData*, and *SetData*, you can change the values of controls or read the current settings as needed.

This section describes in detail how to do the following:

- Set control values
- Read control values
- Customize data transfer

Setting control values

You can set the value of a control object at any time by calling its *SetData* method. *SetData* reads data from the record passed as its argument and sets the control's value accordingly. Since each kind of control needs somewhat different information, the data records vary depending on the type of control.

SetData's parameter is an untyped **var** parameter, so you can pass virtually anything to the control, but there are some limits. First, *SetData* (and the corresponding *GetData* method) expects the record to contain the number of bytes specified by the *DataSize* method. For example, the type *TCheckBoxes* has a data size of 2, because it expects a data record that holds a *Word*-type number (two bytes).

Table 12.1 shows the data size and data records for each of the standard Turbo Vision controls.

Table 12.1
Data transfer records for control objects

Control type	Data size (bytes)	Data interpretation
Button	0	None
Check boxes	2	One bit per check box
Input line	*MaxLen* + 1	A Pascal string with the length byte preceding the text
Label	0	None
List box	6	A pointer to the list of items and the number of the selected item
Multi-state check boxes	4	Varies depending on flags
Param text	*ParamCount* * 4	The parameters to substitute into the text
Radio buttons	2	The ordinal number of the checked box
Scroll bar	0	None
Static text	0	None

Reading control values

Reading the value of a control is the exact inverse of setting the value. You call the control object's *GetData* method, passing a data record, and *GetData* fills the record with a representation of its value. The amount and type of data transferred is the same as for *SetData*, as shown in Table 12.1.

For example, to find out which item in a list box is currently selected, use code similar to that in Listing 12.2.

Listing 12.2
Reading a list box's values

```
type
  TListBoxRec = record          { define data record for a list box }
    ListPtr: PCollection;            { pointer to the list of items }
    SelectedItem: Word;                   { number of selected item }
  end;

function GetSelectedItem: Word;
var ListInfo: TListBoxRec;
begin
  ListBox.GetData(ListInfo);                { set record from control }
  GetSelectedItem := ListInfo.SelectedItem;   { return selected item }
end;
```

Most of the time, you won't need to read the values of individual controls. More likely, you'll read the values of all the controls in a window or dialog box, using the window or dialog box object's *GetData* method.

Customizing data transfer

Turbo Vision's control types are designed for general purpose use, so they might not be the most efficient tools for a particular application. You can derive control objects that use more specialized data records for setting and reading their values.

For example, if you have a program that uses input lines for numeric input, it's not very efficient to have to transfer an entire string to and from the object. It makes much more sense to use a numeric value. Listing 12.3 shows the data transfer methods for a simple numeric input line that handles *Word*-type values.

Listing 12.3
Customizing data transfer for an input line

```
type
  TWordInputLine = object(TInputLine)
    function DataSize: Word; virtual;
    procedure GetData(var Rec); virtual;
    procedure SetData(var Rec); virtual;
  end;

function TWordInputLine.DataSize: Word;
begin
  DataSize := SizeOf(Word);
end;

procedure TWordInputLine.GetData(var Rec);
var ErrCode: Integer;
begin
  Val(Data^, Word(Rec), ErrCode);
end;

procedure TWordInputLine.SetData(var Rec);
begin
  Str(Word(Rec), Data^);                          { set Data from Rec }
end;
```

You can also customize data transfer for input lines by using data validation objects. Chapter 13, "Data validation objects," explains how to use validators.

Displaying static text

Static text objects are the simplest kind of controls. The type *TStaticText* encapsulates a text string in a view, displaying the specified text in the bounding rectangle of the view. A static text object is not designed to display text that changes often, but rather to show a fixed string in a fixed position. Chapter 15, "Editor and text views," describes how to display large amounts of dynamic text. By default, static text controls are not selectable, so the user never actively interacts with them.

Turbo Vision also provides an object, *TParamText*, that displays static text in a view, but allows you to substitute parameters into the text for simple text formatting.

The rest of this section describes how to use the two static text controls:

- Displaying plain text
- Displaying parameterized text

Displaying plain text

The basic *TStaticText* object handles strings that contain only standard ASCII characters and two formatting control characters. There are only two tasks you need to understand to use static text objects:

- Formatting static text
- Constructing static text controls
- Setting and reading static text

Formatting static text

Static text objects allow two kinds of formatting. A *Ctrl+M* character (#13) in the text indicates a line break, so you can specify multiple text lines in a single string. Lines that begin with *Ctrl+C* (#3) center themselves horizontally within the view.

For example, the string 'Turbo Text'#13'Version 0.9' displays as

```
Turbo Text
Version 0.9
```

The string #3'Turbo Text'#13'#3Version 0.9' displays as

```
                            Turbo Text
                           Version 0.9
```

Constructing static text views

The constructor for static text controls takes only two parameters: the bounding rectangle and the text for the control.

```
constructor TStaticText.Init(var Bounds: TRect; AText: String);
```

The most important thing to remember is that the bounding rectangle must be large enough to display the entire text string, since text that goes outside the bounds will be *clipped*, or hidden.

That means that multiple-line static text controls need to include enough lines on the screen for all the lines of text, and each of the lines must be long enough to hold all of its text.

For example, the smallest rectangle that can display the string 'Borland' is assigned with

```
R.Assign(0, 0, 7, 1);
```

To display 'Borland'#13'International', you need to assign a rectangle with at least two lines and thirteen columns:

```
R.Assign(0, 0, 13, 2);
```

Static text controls determine the text to draw by calling a virtual method called *GetText*. You can therefore change the way it displays text by overriding that one method (as, for example, parameterized text controls do).

Setting and reading static text

By default, static text controls can't set or read new values. The static text string is set at initialization, and neither the *GetData* nor *SetData* methods transfers any data. *DataSize* returns zero.

Displaying parameterized text

The type *TParamText* allows you a bit more flexibility than the plain static text control. In addition to displaying text strings, a parameterized static text control lets you pass it varying parameters, which it formats into its text using the *FormatStr* procedure.

The only two tasks you handle differently with parameterized text controls than with static text controls are

- Formatting the text
- Setting and reading the control

Formatting parameterized text

Parameterized text uses the procedure *FormatStr* to substitute parameters into an otherwise static text string. Each parameter is four bytes, passed either in an array or in a record. Special formatting characters in the text string tell *FormatStr* how to interpret what each parameter means. Each parameter can represent a *Longint*-type number, a pointer to a string, or a character.

For example, suppose you have a record with two fields: a pointer to the name of a file, and the size of that file in bytes:

```
type
  TFileInfoRec = record
    FileName: PString;
    FileLength: Longint;
  end;

var FileInfo: TFileInfoRec;
```

Using *FormatStr*, you can format a string that includes both the file name and size, based on the values in the record:

```
FormatStr(ResultStr, 'File: %-12s, Size: %9d', FileInfo);
```

To use this formatting in a parameterized text control, assign 'File: %-12s, Size: %9d' as the control's text string, and tell it to substitute two parameters.

Constructing parameterized text controls

Parameterized text control constructors take only one more parameter than static text controls. In addition to the bounding rectangle and text string, you also pass the number of parameters to substitute into the text:

```
constructor TParamText.Init(var Bounds: TRect; AText: String;
  AParamCount: Integer);
```

Init allocates enough space for *AParamCount* parameters. The parameters get their values in the method *SetData*. *SetData* copies *DataSize* bytes from the passed data record into the parameter list of the control. Be sure *SetData* is called before drawing the control.

The substitution of parameters into text takes place in the virtual *GetText* method. The *Draw* method inherited from *TStaticText* calls *GetText* to determine what text to display. *TParamText*'s *GetText* calls *FormatStr* to merge the parameters in the parameter list into the text string and returns the result.

Setting and reading parameterized text

Users never get a chance to change the values of the parameters to a parameterized text control, so there's no reason to ever read the values from a *TParamText* object. *TParamText* therefore uses the *GetData* method it inherits from *TStaticText*, which does nothing.

On the other hand, it's important that you be able to *set* the parameters, since that's what gets displayed. *SetData* therefore copies enough data for all the parameters from the given data record into its parameter list.

Using scroll bars

A scroll bar is a visual representation of a range of numbers. The user manipulates the current value within that range (represented by the indicator, or "thumb") by clicking the arrows at the ends of the scroll bars, clicking the "page" areas between the arrows, or directly moving the indicator.

Scroll bars give the user the ability to move quickly through a large amount of information, such as scrolling through the text of a document. Most scroll bars in Turbo Vision act closely with another view, such as a scrolling view or a list box. Most of the time you only need to construct a scroll bar object and connect it to the other view; the other view takes care of everything else.

There are only three kinds of tasks you perform on scroll bar controls:

- Constructing scroll bar controls
- Manipulating scroll bar controls
- Responding to scroll bars

Constructing scroll bar controls

Constructing a scroll bar control object is very simple. All you have to specify is a single parameter, the bounding rectangle. If the rectangle has a width of one character, the resulting scroll bar is a vertical scroll bar. Any other size produces a horizontal scroll bar.

All the other parameters of the scroll bar control are set after construction, usually by an associated view. For example, when you construct a scroll bar for a list box, you insert the scroll bar into the owning window, then pass a pointer to the scroll bar as a parameter to the list box constructor. The list box object then takes care of setting the range of the scroll bar to values appropriate to the size of the list box and its list of items.

Manipulating scroll bar controls

Scroll bar objects provide three methods you can call to directly manipulate the settings of a scroll bar: *SetRange*, *SetStep*, and *SetValue*. *SetRange* assigns the minimum and maximum values for the scroll bar's range. *SetStep* sets the amounts the value changes when the user clicks the page areas and arrows. *SetValue* sets the value of the scroll bar indicator.

All three methods call a more general method, *SetParams*, which takes care of setting all the parameters for the scroll bar, then redraws the view and broadcasts a message to alert other views if the scroll bar position changed.

Responding to scroll bars

Other views rarely just poll a scroll bar to find out its value. The scroll bar object itself broadcasts a message to its owner when its value changes, and other objects respond to that broadcast by asking for the scroll bar value.

When the scroll bar value changes, the scroll bar object calls its *ScrollDraw* method, which broadcasts the following message:

```
Message (Owner, evBroadcast, cmScrollBarChanged, @Self);
```

Objects responding to scroll bar changes need to check the *InfoPtr* field of the broadcast event that notifies them of the scroll bar change. If *InfoPtr* points to a scroll bar to which that view is supposed to respond, it should then ask the scroll bar for its value and use that value to update its own appearance. Listing 12.4 shows how *TScroller* checks to see if either of its associated scroll bars has changed.

Listing 12.4
TScroller's response to scroll bar changes

```
procedure TScroller.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);
  with Event do
  if (What = evBroadcast) and (Command = cmScrollBarChanged) and
     ((InfoPtr = HScrollBar) or (InfoPtr = VScrollBar)) then
       ScrollDraw;
end;

procedure TScroller.ScrollDraw;
var D: TPoint;
begin
  if HScrollBar <> nil then D.X := HScrollBar^.Value
  else D.X := 0;
  if VScrollBar <> nil then D.Y := VScrollBar^.Value
  else D.Y := 0;
   ⋮
end;
```

Using cluster objects

Turbo Vision provides three kinds of controls that appear in *clusters*, or groups of related controls: check boxes, radio buttons, and multi-state check boxes. Although you can have a cluster that holds only one check box, in most cases you have more than one. In fact, although you can have a single radio button, it serves no purpose, since the only way to "unpress" a radio button is to press another button in the same cluster.

Working with cluster objects

There are several tasks that apply equally to all cluster objects:

- Constructing a cluster
- Pressing a button
- Telling if a button is checked
- Disabling individual buttons

Constructing cluster objects

Cluster object constructors take only two parameters: a bounding rectangle and a linked list of string items. When you assign the bounding rectangle, be sure to allow room for the box to the left of the text and for all the strings in the list.

The linked list of string items usually comes from a series of nested calls to *NewSItem*, a function which allocates a dynamic string on the heap with a pointer to the next such item. For example, to construct a set of three check boxes, use the following code:

```
var Control: PView;
     ⋮
R.Assign(11, 6, 22, 9);                           { set boundaries }
Control := New(PCheckBoxes, Init(R,             { construct cluster }
  NewSItem('~R~ed',                                    { first item }
  NewSItem('~G~reen',                                   { next item }
  NewSItem('~B~lue', nil)))));                          { last item }
     ⋮
```

Pressing a button

Although you rarely need to do so, you can simulate pressing an individual item in the cluster by calling the cluster object's *Press* method. *Press* is a virtual method, so each kind of cluster can react appropriately. *Press* takes a single parameter, the number of the item you want to press. The items have sequential numbers, starting with zero. Calling *Press* has the same effect as clicking the item with the mouse.

Telling if a button is checked

Clusters have a virtual method called *Mark* that is the converse of *Press. Mark* takes a single parameter indicating the item you want to know about and returns *True* if the item is marked. *Mark* is not meaningful for multi-state check boxes.

Disabling individual buttons

Cluster objects contain a mask indicating whether individual items in the cluster are disabled. *EnableMask* is a *Longint*-type bitmapped field, with each bit representing the state of one of the clustered buttons. Since different descendants of *TCluster* interpret their *Value* fields differently, there isn't a one-to-one correspondence between the bits in *EnableMask* and the bits in *Value.*

Each bit in *EnableMask* represents one of the first thirty-two items in the cluster, with the low-order bit being the first item (item number 0), and the high-order bit being the 32nd item (item number 31). If the bit is set, the item is enabled. By default, cluster objects set all the bits (*EnableMask* is $FFFFFFFF), so all items are enabled. You can therefore disable any or all items by toggling the appropriate bit or bits.

If *EnableMask* is 0, meaning that none of the items are enabled, the cluster sets its state so that the whole control is not selectable.

Using check boxes

Check box clusters interpret the lower half of the *Value* field as 16 separate bits, each one corresponding to one check box. If the bit is set, the box is checked. If the bit is clear, the box is unchecked. If you need more than 16 check boxes in a single cluster, you need to derive a new type from *TCheckBoxes* that uses all 32 bits in *Value.*

Using radio buttons

Since only one of a set of radio buttons in a cluster is checked at a time, clusters interpret the *Value* field as an integer number, the value of which indicates the ordinal number of the checked item. That means that in theory, you could have over 65,000 radio buttons per cluster. Since you wouldn't be able to fit that many in a view, the practical limit is considerably smaller.

Using multi-state check boxes

Multi-state check boxes work just like regular check boxes, except they can have states other than checked and unchecked. For example, you can represent checked, unchecked, and indeterminate states, or cycle through several possible values.

Because representing more than two states requires more than one bit, the interpretation of the *Value* field is more complicated for multi-state check boxes. Unlike check boxes and radio buttons, multi-state check boxes override the constructor inherited from *TCluster*:

```
constructor Init(var Bounds: TRect; AStrings: PSItem;
  ASelRange: Byte; AFlags: Word; const AStates: String);
```

When you construct a set of multi-state check boxes, you must indicate how many states each box can have (*ASelRange*), and how many bits in *Value* represent each item (*AFlags*). Turbo Vision provides the constants *cfOneBit*, *cfTwoBits*, *cfFourBits*, and *cfEightBits* that you can pass to represent one, two, four, or eight bits per check box, respectively. The string passed in *AStates* must contain a character to represent each state of the check box.

Picking from lists

Turbo Vision provides a number of objects for managing lists, including several views that allow the user to choose items from lists. This section describes the abstract list viewer object, *TListViewer*, and the list box control object, *TListBox*. The next section describes a more specialized list view, the outline viewer.

This section describes the following tasks:

- Working with list viewers
- Using list box controls

Working with list viewers

You'll never actually create an instance of *TListViewer*, but Turbo Vision's list box control inherits much of its behavior from the abstract list viewer, and it's likely that you'll want to create list viewers of your own.

The abstract list viewer in *TListViewer* has everything you need to view and pick from a list, except the list. That is, it knows how to display a list of items in a view, scroll through the list, select items, and so on, but it doesn't know anything about the items it would display. Instead, it defines a number of virtual methods you'll override to customize the generic object for specific lists.

The simplest example of a working list viewer is a list box control that uses a collection object as its list (usually a string collection). The list you display can be any sort of list, such as an array.

To use list viewer objects, you need to understand the following tasks:

- Constructing a list viewer
- Getting list item text
- Responding to list selections

Constructing a list viewer

Constructing a list viewer object is quite simple. The default list viewer takes four parameters: a bounding rectangle, the number of columns to display, and pointers to two scroll bar objects.

```
constructor TListViewer.Init(var Bounds: TRect; ANumCols: Integer;
  AHScrollBar, AVScrollBar: PScrollBar);
```

When you derive a usable list viewer from *TListViewer*, you'll probably redefine the constructor to make some assumptions for you. For example, consider the object *TFileList*, which supplies the two-column list of files in the file dialog box in the *StdDlg* unit. *TFileList's* constructor takes only two parameters, the bounding rectangle and one scroll bar, because it will always have two columns a **nil** vertical scroll bar.

Getting list item text

TListViewer provides an abstract method called *GetText*. The list viewer's *Draw* method calls *GetText* for each item it needs to display, passing the number of the item. When you create a list viewer and supply it with a list, you are responsible for overriding *GetText* to return the string you want displayed in the list.

For example, suppose you were building a list viewer to show the members of a small club. Your list consists of an array of records, with each record consisting of information about one member, such as this:

```
type
  TMemberInfo = record
    FirstName, LastName: string[25];
    PhoneNumber: string[20];
      ⋮
  end;

var
  MemberArray: array[0..10] of TMemberInfo;
```

To display the names of the members in the list in the form "Last name, first name" you have to override *GetText*:

```
function TMemberList.GetText(Item: Integer; MaxLen: Integer): String;
begin
  with MemberArray[Item] do
    GetText := Copy(LastName + ', ' + FirstName, 1, MaxLen);
end;
```

Responding to list selections

When the user selects an item in a list viewer, either by double-clicking an item or by moving the focus with arrow keys and pressing *Spacebar*, the list viewer object broadcasts a command event to its owner window with the command *cmListItemSelected*, passing its address in the *InfoPtr* field. Any other object that needs to know what item the user just selected can then query the list viewer as to which item is focused.

For example, to have an input line that displays the currently selected item from a list viewer, the input line's *HandleEvent* method needs to respond to *cmListItemSelected* broadcasts, as shown in Listing 12.5. The code in Listing 12.5 comes from the file PICKLIST.PAS included on your distribution disks.

Listing 12.5
Responding to a list box broadcast

```
procedure TPickLine.HandleEvent(var Event: TEvent);
begin
  inherited HandleEvent(Event);       { perform as normal input line }
  if Event.What = evBroadcast then          { watch for broadcast... }
    if Event.Command = cmListItemSelected then   { ...of the command }
    begin
      with PListBox(Event.InfoPtr)^ do      { talk to broadcaster... }
        Data^ := GetText(Focused, 30);     { ...and get focused text }
      DrawView;                                { update the input line }
      ClearEvent(Event);          { indicate that event was handled }
    end;
end;
```

Note that controls don't get broadcast messages by default. You need to enable the receiving of broadcast messages by including broadcasts in the control's event mask:

```
R.Assign(5, 2, 35, 3);
Control := New(PPickLine, Init(R, 30));
Control^.EventMask := Control^.EventMask or evBroadcast;
Insert(Control);
```

Using list box controls

The list box object, *TListBox,* is a useful descendant of *TListViewer* that stores its list of items in a string collection. For many uses, you can use the default list box object with no modifications. If you have a list that doesn't store its information in simple strings, you can derive a different kind of list box.

To use the default list box object, do the following:

- Build the list of strings
- Construct the list box
- Assign the list to the box
- Set and read the control values

Building the list of strings

By default, list box objects expect string collections as their lists of items. The *List* field that points to the associated list is of type *PCollection,* though, so you can assign the list box any sort of collection.

If you use anything other than a string collection, however, you have to override the list box's *GetText* method to return a string based on the specified item in the collection. For example, if your collection contains numbers, you override *GetText* to convert the numbers to strings so the list box can display them.

Constructing the list box

Constructing a list box object takes three parameters: a bounding rectangle, the number of columns in the box, and a pointer to a vertical scroll bar:

```
constructor TListBox.Init(var Bounds: TRect; ANumCols: Word;
  AScrollBar: PScrollBar);
```

The list box's list is not part of the constructor. You're only constructing the view part. Assigning the list is a separate step.

Assigning a list to a list box

Once you have a list box view, you can assign it a list of items. List box objects have a virtual method *NewList* that enables you to assign a list to the list box at any time. *NewList* disposes of any list already assigned, then assigns the new collection as the current list, adjusting the list box's range and focusing the first item in the list.

Listing 12.6 Contructing a list box and assigning the list, from PICKLIST.PAS

```
constructor TPickWindow.Init;
var
  R: TRect;
  Control: PView;                     { generic pointer for controls }
  ScrollBar: PScrollBar;         { pointer for list box's scroll bar }
begin
  R.Assign(0, 0, 40, 15);                     { assign window bounds }
  inherited Init(R, 'Pick List Window');          { construct window }
  Options := Options or ofCentered;                  { center window }
  R.Assign(5, 2, 35, 3);                   { bounds for display line }
  Control := New(PPickLine, Init(R, 30));           { construct line }
  Control^.EventMask := Control^.EventMask or evBroadcast;
  Insert(Control);                               { insert input line }
  R.Assign(4, 1, 13, 2);                        { bounds for label }
  Insert(New(PLabel, Init(R, 'Picked:', Control)));   { insert label }
  R.Assign(34, 5, 35, 11);                 { bounds for scroll bar }
  New(ScrollBar, Init(R));                  { construct scroll bar }
  Insert(ScrollBar);                             { insert scroll bar }
  R.Assign(5, 5, 34, 11);                     { bounds for list box }
  Control := New(PListBox, Init(R, 1, ScrollBar));  { construct box }
  Insert(Control);                                 { insert list box }
  PListBox(Control)^.NewList(New(PCityColl, Init));    { assign list }
  R.Assign(4, 4, 12, 5);                        { bounds for label }
  Insert(New(PLabel, Init(R, 'Items:', Control)));    { insert label }
  R.Assign(15, 12, 25, 14);                    { bounds for button }
  Insert(New(PButton, Init(R, '~Q~uit', cmQuit, bfDefault)));
end;
```

Setting and reading list box values

The data record for setting or reading a list box comes in two parts, a pointer to the collection holding the list box's items, and a number indicating the number of the selected item. For example, you could define a record type:

```
type
  TListBoxRec = record
    Items: PStringCollection;
    Selected: Integer;
  end;
```

By default, the *SetData* for *TListBox* calls *NewList*, passing the string list pointer from the data record to replace any existing list, then sets the *Focused* field to the selected item from the record.

Displaying outlines

The program OUTDIR.PAS on your distribution disks creates and displays an outline of the directories on a disk.

Turbo Vision provides two useful view objects for displaying outlines. One is an abstract outline viewer, the other displays an outline of branching nodes. The relationship of *TOutlineViewer*, the abstract view, and *TOutline*, the useful control, is much like that of the abstract *TListViewer* and *TListBox*. *TOutlineViewer* defines all the behavior necessary to display an outline, but knows nothing about the items it might display. *TOutline* is a specialized outline viewer that displays strings in a branching tree of nodes.

To use outline views, you need to understand the following things:

- The basic behavior of an outline view
- Specific outline tasks

Basic outline use

Like a list box, most of the behavior of an outline view is automatic, handled by the methods inherited from the abstract outline viewer object type. The basic actions include the following:

- Graphical hierarchy
- Expanding and contracting
- Iterators
- Focus and selection behavior
- Updating

Graphical hierarchy

The outline viewer knows how to display three different kinds of graphic lines to show the hierarchical relationships between the items in the outline. You can change the way the outline displays this graphic by overriding the method *CreateGraph*.

Expanding and contracting

The user can hide or reveal detail about items with subitems using the mouse or keyboard. In some cases, you might want to adjust the display under program control, by calling the methods *Adjust* and *ExpandAll*.

Iterating items

Outline views have iterator methods called *FirstThat* and *ForEach* that operate much like the like-named *TGroup* methods. Outline viewers use the iterators internally, but you might find uses for them in your applications.

Focus and selection behavior

When an item in the outline gets focused, the event handler calls a method called *Focused*. When the user selects an item, the event handler calls *Selected*. If you need to perform a particular action when focus changes or the user selects an item, you can override *Focused* or *Selected*.

Updating the outline

Whenever the information in the outline changes, you need to call the *Update* method. *Update* makes sure as many lines as possible show in the view and that scroll bars stay synchronized with the displayed data.

Using the outline views

To use outline views, you need to understand the following tasks:

- Creating the outline tree
- Constructing the outline view
- Getting the selected node
- Disposing of an outline

Creating the outline tree

In order to display an outline, you have to have a *tree* of items for the outline. A tree is much like a linked list, but it branches, so that each node in the tree has not only a pointer to the next item in the list, but also the possibility of subentries. Subentries are called *child nodes*, and a node with child nodes is called a *parent node*. The base of an outline tree is the *root* node, the node which has no parent. The *TOutline* object displays an outline of a tree made of nodes of type *TNode*.

TNode is a simple record. It contains a pointer to the next node at its same level, the text string it displays in the outline, a pointer to the first of its child nodes, and a Boolean flag indicating whether those children are visible.

To create a node, call the function *NewNode*. *NewNode* allocates a node, given a string and pointers to the child list and next node. To construct a tree of nodes, you can nest calls to *NewNode*.

Constructing the outline view

Constructing an outline view is quite simple. The constructor takes only four parameters: a bounding rectangle, horizontal and vertical scroll bars, and a pointer to the root node of an outline tree.

Getting the selected node

The outline viewer's *Foc* field holds an integer number indicating how far down from the top of the outline the focused item is currently located. If *Foc* is 0, the root is focused. The *GetNode* method takes an integer number and returns a pointer to the node at that position. *GetNode(Foc)*, therefore, returns the focused node.

Disposing of an outline

TOutline's *Done* destructor takes care of disposing of the associated outline tree before disposing of the view object. Nodes allocated with *NewNode* need to be disposed of with *DisposeNode*. *TOutline.Done* calls *DisposeNode(Root)*, which recursively disposes of the entire list.

Reading user input

Input line objects enable the user to type and edit single lines of text. To get more than a single line of input, use the memo field control, described in Chapter 15, "Editor and text views."

Input line objects support full user editing with the mouse and keyboard, cutting and pasting to the clipboard (if any), and data validation of various kinds. This section explains basic use of input line controls. Input validation is explained in Chapter 13, "Data validation objects."

To use an input line control, you need to

- Construct the input line view
- Set and read the control value
- Manipulate the control values

Constructing input line controls

Constructing an input line requires only two parameters: the bounding rectangle for the view and the maximum length of the input string:

```
   constructor TInputLine.Init(var Bounds: TRect; AMaxLen: Integer);
```

The input line allocates space on the heap for the maximum string size and points it's *Data* field to that memory. You can manipulate *Data^* as a normal string. If the full length of the string won't fit in the view, the user can click left or right scrolling arrows to scroll the input line text. Note that you need to allow for those two extra characters in the size of the view.

Setting and reading input lines

The data record for setting or reading an input line control is a string. The size of the string must be the same as the maximum size of the text for the input line. Thus, if you construct an input line control with a maximum length of 10, the data record must be of type **string**[10].

Manipulating input lines

You can directly manipulate certain fields of an input line object. For example, you can change the contents of the text string pointed to by *Data*, although it must not exceed the length specified by *MaxLen*. *MaxLen* must never change after constructing the object, because the destructor uses *MaxLen* to deallocate the memory assigned to *Data*.

You can also directly change *CurPos*, the position of the cursor in the string, and *FirstPos*, the index of the first character displayed in the view (which varies as the user scrolls the text). You should not change the *SelStart* and *SelEnd* fields. If you need to change the selected text, use the method *SelectAll*.

Using history lists

History lists enable the user to easily recall previous entries into an input line. The history control itself, of type *THistory*, is a small view located next to the input line. When the user clicks the history view, the history object displays the list of previous entries in a history viewer in a history window. *THistoryViewer* and *THistoryWindow* are handled automatically by the history object, so you don't need to work with them directly.

The standard application object initializes the history list system during its own initialization when it calls the procedure *InitHistory*. Once that's done, you can link history lists to any input lines in your application.

To use history lists, you need to understand the following tasks:

- Defining a history list
- Constructing a history view

Defining history lists

The application's history list subsystem sets aside an area in memory called the *history block* to store history items for all history lists. A history list consists of string items and their associated ID numbers within the history block.

When the user clicks the history view to display the history for a given input line, Turbo Vision scans the history block for all entries associated with the particular history ID for that input line and displays them in a history window.

Managing the history block

The history block acts as a first in, first out (FIFO) list of items. As long as there is room in the history block, new items are added to the block. When the block fills up, the oldest history items are deleted to make room for new ones.

The history block variables and the procedures that manipulate them are in the HistList unit.

Because all history lists share the same block, a frequently-used list could cause the items from other lists to scroll from the history block. If you plan to use a lot of different history lists in an application, increase the value of the variable *HistorySize* before initializing the application. *HistorySize* determines the size of the history block.

You can save and restore the history block along with your program's objects on a stream. The procedures *StoreHistory* and *LoadHistory* take care of all the details for you.

Constructing a history view

Constructing a history view takes only three parameters: a bounding rectangle, a pointer to the associated input line, and the ID of a history list.

```
constructor THistory.Init(var Bounds: TRect; ALink: PInputLine;
  AHistoryID: Word);
```

The linked input line is usually the line located adjacent to the history view. The history ID is the number you want to associate with items entered in this input line. Different input lines can share the same history list by using the same history ID number. If you want to ensure that each input line has its own history list, be sure to assign unique history IDs.

Once you construct and insert the history view, the management of the history list and the association with the input line are automatic. The history list is persistent. If you destroy the history object and construct another history view with the same ID, all items previously entered in that history list will still be there, as long as they haven't been scrolled out of the history block.

Labeling controls

Label objects, using type *TLabel*, serve two functions. They provide a description of another control, such as a group of radio buttons or an input line, and they enable the user to select the control with the mouse or keyboard without affecting the selected control.

Label objects always serve as partners to other controls. For example, if you have an input line in a dialog box, there's no indication of what the user is supposed to type in the line if the line has no label. Label objects provide the identifying text for the other control and also enable the user to select the associated control by selecting the label. If you just want to display some text in a window or dialog box, use a *TStaticText* or *TParamText* object instead of a label.

In order to make use of label objects, you need to understand three things:

- Constructing label objects
- Establishing focus
- Assigning shortcut characters

Constructing label objects

The constructor for label objects is very simple. It takes only three parameters: a bounding rectangle, a text string for the label, and a pointer to the control object being labeled. A typical constructor for a label object follows the constructor for its associated control, as shown in Listing 12.7.

Listing 12.7
Constructing a label object

```
constructor TYourDialog.Init(var Bounds: TRect; ATitle: TTitleStr);
var
  R: TRect;
  Control: PView;
begin
  inherited Init(Bounds, ATitle);
  R.Assign(10, 2, 40, 3);
  Control := New(PInputLine, Init(R, 30));        { construct control }
  Insert(Control);                                { insert the control }
  R.Assign(1, 2, 10, 3);
  Insert(New(PLabel, Init(R, 'Flavor:', Control)));   { link a label }
end;
```

Note that you don't need to assign the label object to a variable, since inserting it into the dialog box makes it a subview which the dialog box object will dispose when you call its destructor. The associated control, however, you assign to a temporary variable so you can establish the link to the label.

Selecting controls with labels

Labels themselves never get the input focus, since the user can't really interact with the label. When the user clicks the label or selects the label by its shortcut key, the associated control actually gets focus, and the associated label shows itself as active. Conversely, if the user selects the associated control, giving it the focus, the associated label sees the *cmReceivedFocus* broadcast, recognizes that it came from its associated control, and again shows itself as active.

Labels and their associated controls, therefore, always show up as selected or unselected together. From the user's point of view, they are a single unit.

Assigning shortcut characters

When you assign the text for a label object, you can highlight a character as a shortcut. Be careful to avoid asigning several controls with the same shortcut, as only the first one in Z-order will get selected when the user presses the shortcut key.

CHAPTER

13

Data validation objects

Turbo Vision gives you several flexible ways to validate the information a user types into an input line by associating validator objects with the input line objects. Using validator objects makes it easy to add validation to existing Turbo Vision applications or to change the way a field validates its data.

This chapter covers three topics related to data validation:

- The three kinds of data validation
- Using data validator objects
- How validators work

Data validation is handled by the *Valid* method of view objects. You can validate the contents of any particular input line or data screen at any time by calling the object's *Valid*, but Turbo Vision also provides mechanisms to automate data validation. Most of the time, you'll find that data validation takes almost no effort on your part as programmer.

The three kinds of data validation

There are three distinct types of data validation, and Turbo Vision supports all three directly. The three kinds of data validation are

- Filtering input
- Validating each item
- Validating complete screens

Note that these methods are not mutually exclusive. A number of the standard validators combine the different techniques in a single validator.

☞ It's important to remember that the validation is handled by the validator object, *not* by the input line object. If you've already created a customized input line object for a specialized purpose, you've probably already duplicated capability that's built into input lines and their validators.

The "How validators work" section of this chapter describes the various ways in which input line objects automatically call on validator objects.

Filtering input

The simplest way to ensure that a field contains valida data is to ensure that the user can only type valid data. Turbo Vision provides *filter* validators that enable you to restrict the characters the user can type. For example, a numeric input field might restrict the user to typing only numeric digits.

Turbo Vision's filter validator object provides a generic mechanism for limiting the characters a user can type in a given input line. Picture validator objects can also control the formatting and types of characters a user can type.

Validating each field

Sometimes you'll find it necessary to ensure that the user types valid input in a particular field before moving to the next field. This approach is often called "validate on Tab," since pressing *Tab* is the usual way to move the input focus in a data entry screen.

An example is an application that performs a lookup from a database, where the user types in some kind of key information in a field, and the application responds by retrieving the appropriate record and filling the rest of the fields. In such a case, your application needs to check that the user has typed the proper information in the key field before acting on that key.

The *Options* field of Turbo Vision views has a bit that control individual field validation. If a view's *ofValidate* bit is set, it validates its contents when the view loses the input focus. If the data in the field is invalid, the validator alerts the user and keeps the focus in the field until the user provides valid data.

Validating full screens

You can handle validation of full data screens in three different ways:

- Validating modal windows
- Validating on focus change
- Validating on demand

Validating modal windows

When a user closes a modal window, the window automatically validates all its subviews before closing, unless the closing command was *cmCancel*. To validate its subviews, the window calls each subview's *Valid* method, and if each returns *True*, the window can close. If any of the subviews returns *False*, the window is not allowed to close.

A modal window with invalid data can only be canceled until the user provides valid data.

Validating on focus change

As with any view, you can set a window's *ofValidate* option flag. If you use a modeless data entry window, you can force it to validate its subviews when the window loses focus, such as when you select another window with the mouse. Setting a window's *ofValidate* flag prevents you from moving to another window that might act on the data entry window's data before you've validated those data.

Validating on demand

You can tell a window to validate all its subviews at any time by calling its *Valid* method, usually passing *cmClose* as the parameter. Calling *Valid(cmClose)* essentially asks the window "If I told you to close right now, would all your fields be valid?" The window calls the *Valid* methods of all its subviews in Z-order and returns *True* if all of them return *True*.

Calling *Valid* does not obligate you to actually close the window. For example, you might call *Valid(cmClose)* when the user presses a Save button, ensuring the validity of the data before saving it.

You can validate any window, modal or modeless, at any time. Only modal windows have automatic validation on closing, however. If you use modeless data entry windows, you need to ensure that your application calls the window's *Valid* method before acting on entered data.

Using a data validator

Using a data validator object with an input line takes only two simple steps:

- Constructing the validator object
- Assigning the validator to an input line

Once you've constructed the validator and associated it with an input line, you never need to interact with the validator object directly. The input line knows when to call validator methods at the appropriate times.

Constructing validator objects

Since validators are not views, their constructors require only enough information to establish the validation criteria. For example, a numeric range validator object takes two parameters: the minimum and maximum values in the valid range, as show in Listing 13.1.

Listing 13.1
A typical validator constructor

```
constructor TRangeValidator.Init(AMin, AMax: Integer);
```

Adding validation to input lines

Every input line object has a field called *Validator*, set to **nil** by default, that can point to a validator object. If you don't assign an object to *Validator*, the input line behaves as described in Chapter 12, "Control objects." Once you assign a validator to *Validator*, the input line automatically checks with the validator when processing key events and when called on to validate itself.

Normally you construct and assign the validator in a single statement, as shown in Listing 13.2.

Listing 13.2
Adding data validation to an input line

```
  ⋮
InputLine := New(PInputLine, Init(R, 3));    { make 3-char input line }
InputLine^.SetValidator(New(PRangeValidator, Init(100, 999)));
Insert(InputLine);                           { insert into owner window }
  ⋮
```

How validators work

Turbo Vision supplies several kinds of validator objects that cover most data validation needs. You can also derive your own validators from the abstract validator types.

This section covers the following topics:

- The virtual methods of a validator
- The standard validator types

The methods of a validator

Every validator object inherits four important methods from the abstract validator object type *TValidator*. By overriding these methods in different ways, the various descendant validators perform their specific validation tasks. If you're going to modify the standard validators or write your own validation objects, you need to understand what each of these methods does and how input lines use them.

The four validation methods are

- *Valid*
- *IsValid*
- *IsValidInput*
- *Error*

The only methods called from outside the object are *Valid* and *IsValidInput*. *Error* and *IsValid* are only called by other validator methods.

Checking for valid data

The main external interface to data validator objects is the method *Valid*. Like the view method of the same name, *Valid* is a Boolean function that returns *True* only if the string passed to it is valid data. One component of an input line's *Valid* method is calling the validator's *Valid* method, passing the input line's current text.

When using validators with input lines, you should never need to either call or override the validator's *Valid* method. By default, *Valid* returns *True* if the method *IsValid* returns *True*; otherwise it calls *Error* to notify the user of the error and returns *False*.

Validating a complete line

Validator objects have a virtual method called *IsValid* that takes a string as its only parameter and returns *True* if the string represents valid data. *IsValid* is the method that does the actual validation, so if you create your own validator objects, you'll override *IsValid* in most cases.

☞ You don't call *IsValid* directly. Use *Valid* to call *IsValid*, because *Valid* calls *Error* to alert the user if *IsValid* returns *False*. Be sure to keep the validation role separate from the error reporting role.

Validating keystrokes

When an input line object recognizes a keystroke event meant for it, it calls its validator's *IsValidInput* method to ensure that the typed character is a valid entry. By default, *IsValidInput* methods always return *True*, meaning that all keystrokes are acceptable. However, some descendant validators override *IsValidInput* to filter out unwanted keystrokes.

For example, range validators, which are used for numeric input, return *True* from *IsValidInput* only for numeric digits and the characters '+' and '-'.

IsValidInput takes two parameters. The first parameter is a **var** parameter, holding the current input text. The second parameter is a Boolean value indicating whether the validator should apply filling or padding to the input string before attempting to validate it. *TPXPictureValidator* is the only one of the standard validator objects that makes use of the second parameter.

Reporting invalid data

The virtual method *Error* alerts the user that the contents of the input line don't pass the validation check. The standard validator objects generally present a simple message box notifying the user that the contents of the input are invalid and describing what proper input would be.

For example, the *Error* method for a range validator object creates a message box indicating that the value in the input line is not between the indicated minimum and maximum values.

Although most descendant validator objects override *Error*, you should never call it directly. *Valid* calls *Error* for you if *IsValid* returns *False*, which is the only time *Error* needs to be called.

The standard validators

Turbo Vision includes six standard validator object types, inlcuding an abstract validator and the following five specific validator types:

- Filter validator
- Range validator
- Lookup validator
- String lookup validator
- Picture validator

The abstract validator

The abstract type *TValidator* serves as the base type for all the validator objects, but it does nothing useful by itself. You'll never create an instance of *TValidator*. Essentially, *TValidator* is a validator to which all input is always valid. *IsValid* and *IsValidInput* always return *True*, and *Error* does nothing. Descendant types override *IsValid* and/or *IsValidInput* to define which values actually are valid.

You can use *TValidator* as a starting point for your own validator objects if none of the other validation types are appropriate starting points.

Filter validators

Filter validators are a simple implementation of validators that only check input as the user types it. The filter validator constructor takes one parameter, a set of valid characters:

```
constructor TFilterValidator.Init(AVaildChars: TCharSet);
```

TFilterValidator overrides *IsValidInput* to return *True* only if all characters in the current input string are contained in the set of characters passed to the constructor. The input line only inserts characters if *IsValidInput* returns *True*, so there is no need to override *IsValid*. Because the characters made it through the input filter, the complete string is valid by definition.

Descendants of *TFilterValidator*, such as *TRangeValidator*, can combine filtering of input with other checks on the completed string.

Range validators

The range validator *TRangeValidator* is a straightforward descendant of *TFilterValidator* that accepts only numbers and adds range checking on the final results. The constructor takes two parameters that define the minimum and maximum valid values:

```
constructor TRangeValidator.Init(AMin, AMax: Integer);
```

The range validator constructs itself as a numeric filter validator, accepting only the digits '0'..'9' and the plus and minus characters. The inherited *IsValidInput* therefore ensures that only numbers filter through. *TRangeValidator* then overrides *IsValid* to return *True* only if the entered numbers are a valid integer within the range defined in the constructor. The *Error* method displays a message box indicating that the entered value is out of range.

Lookup validators

The abstract lookup validator *TLookupValidator* provides the basis for a common type of data validator, one which compares the entered value with a list of acceptable items in order to determine validity.

One example of a working lookup validator is the string lookup validator.

TLookupValidator is an abstract type that you'll never use as it stands, but it makes one important change and one addition to the standard abstract validator.

The one new method introduced by *TLookupValidator* is called *Lookup*. By default, *Lookup* returns *False*, but when you derive a descendant lookup validator type, you override *Lookup* to compare the passed string with a list, returning *True* if the string contains a valid entry.

☞ *TLookupValidator* overrides *IsValid* to return *True* only if *Lookup* returns *True*. In descendant lookup validator types, do not override *IsValid*, but rather override *Lookup*.

String lookup validators

TStringLookupValidator is a working example of a lookup validator. It compares the string passed from the input line with the items in a string list. If the passed string occurs in the list, the string lookup validator's valid method returns *True*. The constructor takes only one parameter, the list of valid strings:

```
constructor TStringLookupValidator.Init(AStrings: PStringCollection);
```

To use a different string list after constructing the string lookup validator, pass the new list to the validator's *NewStringList* method. It disposes of the old list and installs the new list.

TStringLookupValidator overrides *Lookup* and *Error*, so that *Lookup* returns *True* if the passed string is in the string collection and *Error* displays a message box indicating that the string wasn't in the list.

Picture validators

Picture validators compare the string typed by the user with a *picture* or template that describes the format of valid input. The pictures used are compatible with those used by Borland's Paradox relational database to control user input. Constructing a picture validator takes two parameters: a string holding the template image and a Boolean value indicating whether to automatically fill in literal characters in the picture:

```
constructor TPXPictureValidator.Init(const APic: String;
  AAutoFill: Boolean);
```

TPXPictureValidator overrides *Error*, *IsValidInput*, and *IsValid*, and adds a new method, *Picture*. The changes to *Error* and *IsValid* are simple. *Error* displays a message box indicating what format the string should have, and *IsValid* returns *True* only if the function *Picture* returns *True*. This allows you to derive new kinds of picture validators by overriding only the *Picture* method. *IsValidInput* checks characters as the user types them, allowing only those allowed by the picture format, and optionally filling in literal characters from the picture.

The *Picture* method tries to format the given input string according to the picture format and returns a value indicating the degree of its success: complete, incomplete, or error.

CHAPTER

14

Palettes and color selection

No one ever seems to agree on what colors are "best" for any computer screen. Rather than dictating the colors of screen items, Turbo Vision enables both programmers and users to vary the colors of views. This chapter covers the two features of Turbo Vision you need to understand to work with colors: color palettes and color selection objects.

Using color palettes

Instead of making you specify the color of every view in your application, Turbo Vision uses a *color palette* in the application object to map all the colors of all the views. For example, when you create a menu bar, you don't have to tell it what color you want it to be. It gets that information from the application's color palette. You can change that color by putting different information in the palette, which will change the color of every menu in the application. If you want to have a single menu that's a different color from all the other menus, you can map it onto a different entry in the palette, which allows it to be special without affecting any other views.

The only time you have to concern yourself with color palettes (or colors, for that matter) is when writing *Draw* methods. *Draw* is the only method that puts information on the screen.

The remainder of this section covers the following topics:

- Understanding color palettes
- Using default colors
- Changing default colors
- Defining new colors
- Defining palettes for new views

Understanding color palettes

When you write a *Draw* method for a Turbo Vision view, you don't specify the actual color it will use. Instead, you'll call a method called *GetColor* that asks the view's owner what color it should be. The view, therefore, only has to know what kinds of colors it needs to ask its owner about.

The palette for a view has entries for each distinct part of the view that might have a different color. A window object's palette, for example, has entries for the frame when it's the active window, another for the frame when it's not active, one for the icons on the frame, two for scroll bars, and two for scroller text. As you'll see, none of these entries is actually a *color*, but rather an indication of where to find the color.

Looking at a simple palette

Before examining the more complex palettes, look at a simple one. The palette for *TScroller* looks like this:

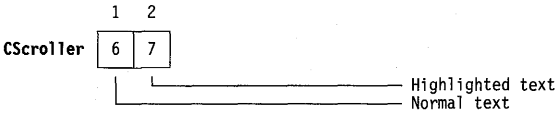
```
CScroller   = #6#7;
```

Color palettes are actually stored in Pascal strings. This allows them to be flexible arrays of varying length. *CScroller*, then, is a two-character string, which you can think of as two palette entries. The layout of the *TScroller* palette is defined as

```
{ Palette layout }
{ 1 = Normal     }
{ 2 = Highlight  }
```

but it's more useful to look at it this way:

Figure 14.1
TScroller's default color palette



A scroller object only knows how to draw two things: normal and highlighted text. The default color of each is determined by the palette entries. When displaying normal text, the *Draw* method uses the color indicated by the first palette entry. To show highlighted text, it uses the second entry.

Getting colors from the palette

To retrieve the color attribute indicated by a palette entry, views have a virtual method called *GetColor*. *GetColor* takes a single parameter, which is the number of an entry in the palette, and returns the color attribute for that entry. To get the color attribute for normal text, then, a scroller's *Draw* method would call *GetColor(1)*.

Understanding color attributes

The color attributes used by Turbo Vision are the same standard video attribute bytes used by all DOS text depicted in Figure 14.2. The lower four bits represent the foreground color, the next three bits the background color, and the highest-order bit the blink attribute.

Figure 14.2
Text color attribute mapping

bit →	7	6	5	4	3	2	1	0
	B	b	b	b	f	f	f	f

The only place you actually find these attributes in Turbo Vision is in the application's palette. All other palettes are indexes into other palettes that eventually point to an entry in the application palette. The next section explains this color mapping.

Mapping colors with palettes

A view's palette entries are not actually *colors*, but indexes into the view's owner's palette. That's an important point: Color mapping has nothing to do with inheritance, but it has everything to do with *ownership*. When you insert a view into a group, its color mapping is determined by that group, and whatever group that group is in, and so on. Inserting it into a different kind of group (say, a window instead of a dialog box) could change its eventual color completely.

Keep in mind that in normal use, you don't have to think at all about how colors get mapped or what color a view will be. When you write *Draw* methods you get colors from the view's palette by calling *GetColor*, so you don't have to worry about how *GetColor* returns an attribute.

A simple example of color mapping

To understand how *GetColor* returns a color from a palette entry, it would help to trace what happens when you call the method. The default *Draw* method for *TScroller* is the method inherited from *TView*. It draws all text in the color indicated by the first palette entry. *TView.Draw* is shown in Listing 14.1.

Listing 14.1
The Draw method for TView, also used by TScroller

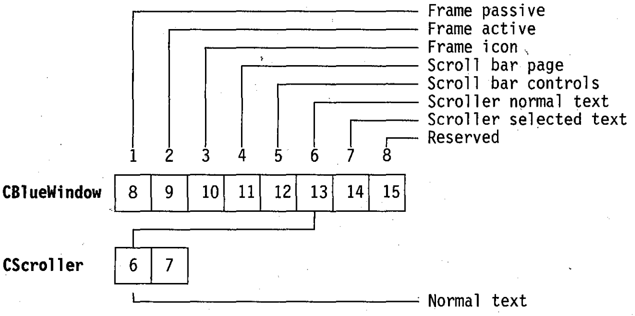
```
procedure TView.Draw;
var B: TDrawBuffer;
begin
  MoveChar(B, ' ', GetColor(1), Size.X);      { fill line with spaces }
  WriteLine(0, 0, Size.X, Size.Y, B);    { fill whole view with lines }
end;
```

If you don't understand Draw methods, see Chapter 8, "Views."

Assume for the moment that the scroller object is inserted into a normal blue window, which is in turn inserted into the desktop, which is, of course, inserted in the application object. The color mapping follows that chain of ownership.

Draw gets normal text color, the first entry in the scroller's palette, by calling *GetColor(1)*. *GetColor* takes its parameter and uses it as an index into the palette; that first entry in the palette contains the number 6. That 6 isn't a color attribute, though: it's a palette index. Because a view knows that it has to map through its owner's palette, it calls its owner's *GetColor*, to find the sixth entry in the owner's palette. Figure 14.3 shows *TWindow*'s palette and the mapping of the scroller's normal text.

Figure 14.3
Mapping a scroller's palette onto a window



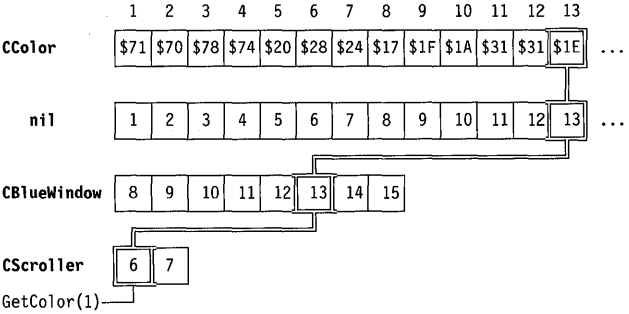
The same process of calling owner views' *GetColor* methods continues until it reaches the one view that has no owner: the application object.

The sixth entry in *TWindow*'s palette is 13, which is an index into the palette of the window's owner (the desktop), which in turn indexes into the palette of its owner, the application. *TDeskTop* has a **nil** palette, meaning that it doesn't change any mappings. You can think of it as a "straight" or "transparent" palette. The first entry is the number 1, the second is 2, and so on.

The application *does* have a palette, a large one containing entries for all the elements you might insert into a Turbo Vision application. Its 13th element is $1E for color screens. Since the application has no owner, the mapping stops there. Figure 14.4

depicts the mapping of the scroller through the window and the desktop to the application.

Figure 14.4
Mapping the normal text color of a scroller view



	1	2	3	4	5	6	7	8	9	10	11	12	13	
CColor	$71	$70	$78	$74	$20	$28	$24	$17	$1F	$1A	$31	$31	$1E	...
nil	1	2	3	4	5	6	7	8	9	10	11	12	13	...
CBlueWindow	8	9	10	11	12	13	14	15						
CScroller	6	7												

So now you have $1E, a text attribute byte that corresponds to background color 1 and foreground color $E (or 14), which produces yellow characters on a blue background. Again, don't think of this in terms of yellow-on-blue. Rather say that you want your text in the normal color for window text.

A different view of mapping

GetColor uses some intricate assembly language to perform the color mappings. Listing 14.2 shows an equivalent Pascal method that shows more clearly how views perform color mapping.

Listing 14.2
The color mapping algorithm used by views

```
function TView.NotGetColor(Color: Byte): Byte;
var
  P: PPalette;
  Curr: PView;
begin
  Curr := @Self;                            { start with current view }
  while Curr <> nil do                       { continue while non-nil }
  begin
    P := Curr^.GetPalette;                            { get the palette }
    if P <> nil then                       { if the view has a palette... }
      Color := Byte(P^[Color]);   { ...get index into owner's palette }
    Curr := Curr^.Owner;                         { point to owner view }
  end;
  NotGetColor := Color;                   { return the last color found }
end;
```

Changing the default colors

The obvious way to change a view's color is to change its palette. If you don't like your scroller's normal text color, your first instinct might be to change entry 1 (the normal text entry) in the scroller's palette, perhaps from 6 to 5. Normal scroller text is then mapped onto the window entry for scroll bar controls (blue on cyan, by default). Remember, *5 is not a color!* All you've done is tell the scroller that its normal text should look like the scroll bars around it.

Color attributes appear only in the application's palette, so that's the only place you can change them.

So what if you don't want bright yellow on blue? Colors are not absolute. They are mapped through the owner's palettes, so change the palette entry for normal window text in the application object. Since that is the last non-**nil** palette, the entries in the application palette determine the colors that will appear in all views within a window.

Palettes centralize color information

Controlling color attributes from a central location makes sense. Presumably you want all your windows to look similar. You certainly don't want to have to tell every single window what color it should be. If you change your mind later (or you allow users to customize colors), you don't want to have to change the entries for each and every window.

Also, a scroller or other interior does not have to worry about its colors if it is inserted into some window other than the one you originally intended. If you put a scroller into a dialog box instead of a window, for example, it will not (by default) come up in the same colors, but rather in the colors of normal text in a dialog box.

Changing a view's palette

To change a view's palette, override its *GetPalette* method. To create a new scroller object type that draws its normal text in the window's frame color instead of the window's normal text color, the declaration and implementation of the object would include the following:

```
type
  TMyScroller = object(TScroller)
     function GetPalette: PPalette; virtual;
  end;

function TMyScoller.GetPalette: PPalette;
const
  CMyScroller = #1#7;
  P: string[Length(CMyScroller)] = CMyScroller;
begin
  GetPalette := @P;
end;
```

The only change made by *TMyScroller* is that it changes the first entry in the scroller's palette from #6 to #1. In other words, it maps the scroller's normal text onto the first entry in the window's palette ("Frame Passive") instead of the sixth entry ("Scroller Normal Text").

The types TPalette and String are completely interchangeable.

Note that the palette constant is a string constant because Turbo Vision uses the *String* type to represent the palettes. This makes it easier to manipulate palettes, since all the string functions and operators can also be used with palettes.

Extending a palette

When you derive a new view type, you sometimes need to define new palette entries for new kinds of visual elements.

For example, you could create a new kind of scroller that understands not only normal and highlighted text, but also some kind of specially emphasized text.

Extending the palette takes three steps:

- Adding elements to the view's palette
- Ensuring that owner object types have the needed palette entries
- Revising *Draw* to use the new color

Adding a palette entry

Extending a view's palette is quite simple. Since the palette is a string, you append as many entries to as you need the end of the existing palette.

There are two cases you have to consider when adding a new palette entry:

- Using a color that already exists in the owner view
- Creating a new kind of color item

Reusing an existing color

Using an already-existing color is the easiest because you don't have to change the owner palette. For example, to make the scroller's third palette entry use the owner window's scroll bar color (the fourth entry in the window palette), you'd append #4 to the existing scroller palette:

```
const
  CMyScroller = CScroller + #4;
```

Because this uses colors the owner window already knows about, you can now go ahead and rewrite *Draw* to use the third scroller color.

Adding a new color

If you want your new view palette to use a color not already defined in the owner view's palette, you still append an item to the view's palette, but instead of pointing to an existing entry, you point to a new entry that you'll add to the owner view. For example, you can point to the ninth entry in the window's palette:

Listing 14.3
Adding an entry to the scroller palette

```
function TMyScroller.GetPalette: PPalette;
const
  CMyScroller = CScroller + #9;         { append to existing palette }
  P: string[Length(CMyScroller)] = CMyScroller;
begin
  GetPalette := @P;                    { point to the typed new palette }
end;
```

The catch, of course, is that the window object only has eight entries in it palette, so you have to override the owner window's *GetPalette* method to return a palette with the new ninth entry:

Adding entries to owner palettes

Any time you create a view that accesses more entries than its owner view's palette has by default, make sure that you also extend the owner view's palette. Accessing nonexistent palette entries produces undefined results.

Extending the window palette to accomodate the new scroller palette entry defined in Listing 14.3 is slightly more complicated than extending the scroller view's palette, because there are actually three standard window palettes, for blue, gray, and cyan windows. Listing 14.3 shows how to add a ninth entry to all three palettes.

Figure 14.5
adding entries to the window palettes

```
function TMyWindow.GetPalette: PPalette;
const
  CMyBlueWindow = CBlueWindow + #64;            { add a ninth entry }
  CMyCyanWindow = CCyanWindow + #65;
  CMyGrayWindow = CGrayWindow + #66;
  P: array[wpBlueWindow..wpGrayWindow] of
       string[Length(CMyBlueWindow)] =
         (CBlueWindow, CCyanWindow, CGrayWindow);
begin
  GetPalette := @P[Palette];      { return is based on Palette field }
end;
```

This new ninth entry points to the 64th, 65th, or 66th entry in the application's palette, depending on the color scheme of the window. But again, those are new entries: the default application palette has only 63 entries, so you have to add entries to the application palette as well.

But like the window palette, the application palette is actually three different palettes, one each for color, black-and-white, and monochrome systems. So modifying the application palette in this case means adding three new entries to each of three palettes, as shown in Listing 14.4.

Listing 14.4
Adding entries to the application palettes

```
function TMyApplication.GetPalette: PPalette;
const
  CMyColor = CColor + #$25#$50#$0F;
  CMyBlackWhite = CBlackWhite + #$0F#$0F#$0F;
  CMyMonochrome = CMonochrome + #$0F#$0F#$7F;
  P: array[apColor..apMonochrome] of string[Length(CMyColor)] =
    (CMyColor, CMyBlackWhite, CMyMonochrome);
begin
  GetPalette := @P[AppPalette];
end;
```

The scroller's palette entry 3 is now the new extra-highlight color. If you use this new *GetPalette* using the *CMyScroller* that accesses the ninth element in its owner's palette, be sure that the owner uses the extended palette, and that the application uses the corresponding extensions. If you try to access the ninth element in an eight-element palette, the results are undefined.

Rewriting Draw methods

Once you have a palette that contains additional entries, you can rewrite your *Draw* method to take advantage of the new color. In the example in this section, *TMyScroller.Draw* can now pass any value in the range 1..3 to *GetColor* and get a valid color. Passing any other value returns the error attribute, blinking white on red.

Letting users change colors

Turbo Vision's *ColorSel* unit provides a dialog box you can include in your applications to enable users to alter the application's color palette. This section describes

- Using the color selection dialog box
- Saving and restoring colors

Using the TColorDialog

The color selection dialog box, *TColorDialog*, gives users of your programs easy access to the application's color palette by letting they specify which kinds of items they want to alter. You control which items they can change by passing lists of color groups and color items to the dialog box.

Using the color selection dialog box takes two steps:

- Defining color groups and items
- Executing the color selection dialog box

Defining color groups and items

Color items are just names you give to various color palette entries. Color groups are linked lists of color items which in turn have names. The Turbo Vision functions *ColorItem* and *ColorGroup* make it easy to create such lists. In addition, The *ColorSel* unit provides functions that return lists of the standard items, so you only have to define new items and groups when you create new views or extend the palettes of existing views. Listing 14.5 shows one such function, *MenuColorItems*, which defines color items for all the palette entries associated with menu views.

Listing 14.5
The MenuColorItems function

```
function MenuColorItems(const Next: PColorItem): PColorItem;
begin
  MenuColorItems :=
    ColorItem('Normal',           2,
    ColorItem('Disabled',         3,
    ColorItem('Shortcut',         4,
    ColorItem('Selected',         5,
    ColorItem('Selected disabled', 6,
    ColorItem('Shortcut selected', 7,
    Next))))));
  end;
```

By combining lists of color items, you create a list of color groups, which you can then pass to the color selection dialog box constructor, as shown in Listing 14.6.

Listing 14.6
Passing groups of color items to a color selection dialog box

```
D := New(PColorDialog, Init('',
  ColorGroup('Desktop',       DesktopColorItems(nil),
  ColorGroup('Menus',         MenuColorItems(nil),
  ColorGroup('Dialog boxes',  DialogColorItems(dpGrayDialog, nil),
  nil)))));
```

Executing the dialog box

Once you've defined your groups of color items, you pass that list, and the palette to modify, to the constructor of *TColorDialog*. Often, rather than passing the palette to the constructor, you'll instead pass it as the data record for the dialog box when you execute it. For example, to modify the palette currently being used by the application, you do this:

```
if ExecuteDialog(D, Application^.GetPalette) <> cmCancel then
begin
  DoneMemory;                          { Dispose of all group buffers }
  ReDraw;                          { Redraw application with new palette }
end;
```

Saving and restoring colors

A Turbo Vision application stores the current state of the color selection dialog box in a variable called *ColorIndexes*. To save the user's current color choices, you call the procedure *SaveIndexes*. To restore a previous state, you call *LoadIndexes*. Both *SaveIndexes* and *LoadIndexes* take a single parameter, specifying the stream that holds the color indexes.

The example program *TVDemo* shows the use of *LoadIndexes* and *StoreIndexes*.

CHAPTER

15

Editor and text views

Turbo Vision provides two kinds of objects for handling text in your applications. Text views display text in a flexible fashion, while editor objects enable the user to enter and modify text. This chapter covers the following uses of text views:

- The basic text view
- The "dumb" terminal view
- The basic text editor
- The memo editor
- The file editor
- The editor clipboard
- The edit window

What is a text view?

For details on scroller objects, see Chapter 8, "Views."

Text views are simple descendants of *TScroller* that link a text file device with a scrollable view. Turbo Vision defines an abstract text device in type *TTextDevice*, which adds virtual methods for reading strings from the text file and writing strings to the file. The type *TTextDevice* doesn't do anything useful by itself, and you'll never have reason to create an instance of it, but it provides the foundation for more useful text views, specifically the terminal view, *TTerminal*.

Using the terminal view

The terminal view defined by type *TTerminal* is the only text view type provided in Turbo Vision. It provides a scrolling "write only" view onto a text file device. You'll probably find terminal views most useful for debugging purposes or for monitoring the contents of a text file.

Most of the behavior of a terminal view is handled for you. If you're accustomed to dealing with text file devices, you'll have no trouble dealing with *TTerminal.*

Using a terminal view takes three steps:

- Constructing the terminal view
- Assigning the text device
- Writing to the view

Note that although you can read from the terminal view, it always returns an empty string.

Constructing the terminal view

Constructing the terminal view is only slightly different than constructing any other scrolling view. In addition to the boundary rectangle and scroll bar parameters, the constructor takes a *Word*-type parameter specifying the size of the terminal's buffer.

Managing the buffer

The actual management of the text buffer is handled for you by *TTerminal.* When you specify the size of the buffer, *TTerminal* allocates that many bytes to a zero-based array of characters of type *TTerminalBuffer*. All characters written to the terminal view are placed in the buffer. Upon reaching the end of the buffer, the terminal view automatically wraps around to the beginning of the buffer again, keeping track of the beginning point of displayable text.

TTerminal has several methods you can use to find out the status of the buffer. The Boolean function *CanInsert* indicates whether inserting a given number of characters will cause the top line of the buffer to be discarded. *QueEmpty* indicates whether the buffer contains any characters. *CalcWidth* returns the length of the longest line in the text buffer.

Assigning the text device

Before a terminal view can interact with a text device, you have to assign a device to the view. Turbo Vision provides a procedure called *AssignDevice* that does for your text view exactly what the standard procedure *Assign* does for a text file. It associates the given text file with a terminal view, meaning that all future input or output operations on the text file will read or write from the terminal view.

For example, given a terminal view called *Terminal* and a text file called *TermText*, you assign the text device to the terminal view as follows:

```
AssignDevice(TermText, Terminal);
```

Writing to the terminal view

Writing to the terminal view is just like writing to any text file device. You use the *Write* and *Writeln* standard procedures. Once you call *AssignDevice* to associate a text file device with a terminal view, all output to the text file device appears in the terminal view.

Note that as with any text file device, you need to call *Rewrite* or *Reset* to open the text file. After that, you call *Write* or *Writeln*, specifying the text device's identifier. For example:

```
AssignDevice(TermText, Terminal);
Rewrite(TermText);
Writeln(TermText, 'This appears in the view.');
```

The simple program in Listing 15.1 traps mouse events and writes the coordinates of mouse clicks to a terminal view in a window. The file TERMTEST.PAS on your distribution disks contains the same program.

Listing 15.1
Using a simple terminal view

```
program TermTest;

uses Objects, Views, App, Drivers, TextView;

type
  PTermWin = ^TTermWin;
  TTermWin = object(TWindow)
    TermText: Text;
    Terminal: PTerminal;
    constructor Init;
    procedure HandleEvent(var Event: TEvent); virtual;
  end;
  TTermApp = object(TApplication)
    constructor Init;
  end;

constructor TTermWin.Init;
var
  R: TRect;
  HScrollBar, VScrollBar: PScrollBar;
begin
  Desktop^.GetExtent(R);
  inherited Init(R, 'Terminal test window', wnNoNumber);
  R.Grow(-1, -1);
  HScrollBar := StandardScrollBar(sbHorizontal or sbHandleKeyboard);
  Insert(HScrollBar);
  VScrollBar := StandardScrollBar(sbVertical or sbHandleKeyboard);
  Insert(VScrollBar);
  New(Terminal, Init(R, HScrollBar, VScrollBar, 8192));
  if Application^.ValidView(Terminal) <> nil then
  begin
    AssignDevice(TermText, Terminal);
    Rewrite(TermText);
    Insert(Terminal);
  end;
end;

procedure TTermWin.HandleEvent(var Event: TEvent);
begin
  if Event.What and evMouseDown <> 0 then
  begin
    if Event.Buttons and mbLeftButton <> 0 then
      Write(TermText, 'Left  ')
    else Write(TermText, 'Right ');
    Writeln(TermText, '(',Event.Where.X, ',', Event.Where.Y, ')');
  end;
  inherited HandleEvent(Event);
end;

constructor TTermApp.Init;
var TextWin: PTermWin;
begin
  inherited Init;
  New(TextWin, Init);
  if ValidView(TextWin) <> nil then InsertWindow(TextWin);
end;

var TermApp: TTermApp;
begin
  TermApp.Init;
  TermApp.Run;
  TermApp.Done;
end.
```

Using the editor object

Turbo Vision defines an editor object type, *TEditor*, that implements a small, fast, 64K editor with full mouse support, clipboard operations, undo, autoindent and overwrite modes, WordStar key bindings, and search and replace functions.

This section explains the following:

- How the editor works
- Using the Edit menu
- WordStar key bindings
- Editor options
- Searching and replacing
- Using scroll bars and indicators

How the editor works

You should rarely need to get at the internal workings of the editor object. It's most common uses, as a file editor and as a memo field in a window, are handled by two descendant types, *TFileEditor* and *TMemo*, both described in this chapter.

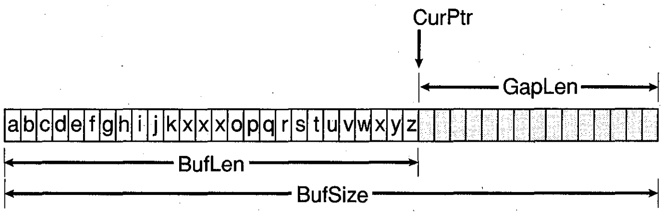
TEditor implements a "buffer gap" editor, meaning that it stores its text in two pieces with a *gap* between them. Text before the cursor is stored at the beginning of the buffer, and text after the cursor at the end. The space between the text is the gap.

Characters inserted into the editor go at the beginning of the gap. Deleted characters remain in the buffer, but at the end of the gap. The editor supports undoing of insertions and deletions by maintaining the number of characters inserted and deleted. When asked to perform an undo, the editor deletes the characters that were inserted, moves the deleted characters to the beginning of the gap, and positions the cursor after the formerly deleted text.

Understanding the buffer

To see how the buffer operates, look at Figure 15.1, which shows an edit buffer with the text 'abcdefghijkxxxopqrstuvwxyz' newly inserted:

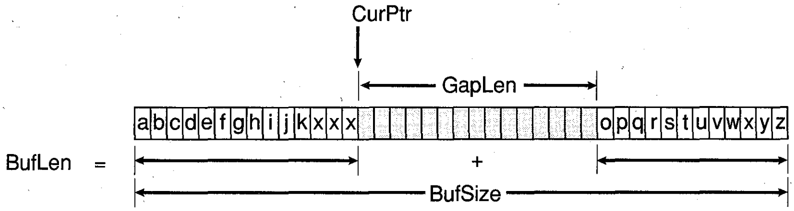
Figure 15.1
Buffer with inserted text



CurPtr, GapLen, BufLen, and BufSize are all fields of TEditor.

BufSize is the size of the buffer, set when you construct the editor object. *CurPtr* indicates the position of the cursor, *GapLen* is the length of the gap, and *BufLen* is the total number of characters in the buffer. The sum of *GapLen* and *BufLen* is always *BufSize*. If you move the cursor to just after the 'xxx' characters, the buffer looks like Figure 15.2.

Figure 15.2
Buffer after cursor movement



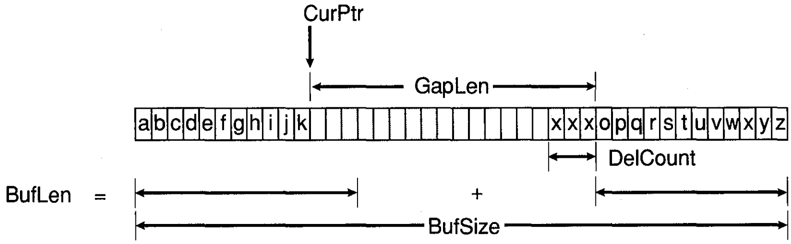
Note that the gap is kept in front of the cursor, allowing for quick insertion of characters without moving any text.

Deleting text

The user can delete text either by backspacing over it, deleting the character ahead of the cursor with *Delete*, or by selecting a block of text and pressing *Delete*. Your program can delete a selected block by calling the method *DeleteSelect*.

If you delete 'xxx' using *Backspace*, the editor moves the characters to the end of the gap and the cursor moves backward. The *DelCount* field records the number of characters deleted. Figure 15.3 shows the state of the buffer after deleting 'xxx'.

Figure 15.3
Buffer after deleting 'xxx'

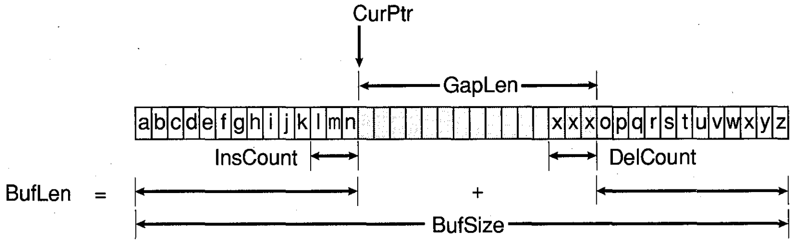


Inserting text

Text insertions normally come either from keystrokes or from pasting text from the clipboard. The editor has two methods, *InsertText* and *InsertFrom*, that handle text insertion. *InsertText* takes a given number of characters and inserts them into the buffer. *InsertFrom* inserts the selected text from a specified editor object's buffer. Both insertion methods call a lower-level insertion method called *InsertBuffer*, but you shouldn't ever have to call that directly.

When you insert characters, the editor increments the insertion count, *InsCount*, to record how many characters to delete with an undo. If you now type 'lmn', the buffer looks like Figure 15.4.

Figure 15.4
Buffer after inserting 'lmn'



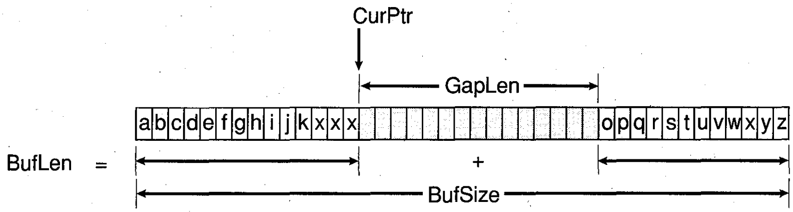
InsCount records the number of characters inserted.

Undoing edits

Editor objects have a limited undo function, usually accessed by the user through an option on the Edit menu, which calls the editor object's *Undo* method.

If you now request an undo, the editor deletes 'lmn' and moves 'xxx' to where they were, restoring the buffer to what it was before the edits, as shown in Figure 15.5.

Figure 15.5
Buffer after undo



Undo can only undo operations done between cursor movements. As soon as the user or the program moves the cursor, all edits are accepted. All undo information is lost because the gap moves. Note that undo information takes space inside the buffer, which could prevent the user from inserting text. Moving the cursor reclaims that space.

Handling blocks

The *selection* or marked block of text is always either before or after the cursor. The fields *SelStart* and *SelEnd* indicate the beginning and ending of the selection. Normally, the user selects text with the mouse or keyboard, but your program can set the selection by calling *SetSelection*, which also moves the cursor.

Inserting text into the editor, either through a key press, or with *InsertText*, replaces the selected text with the inserted text. If there is no selection, the text is just inserted at the cursor.

Using the Edit menu

All editor objects know how to respond to several commands from the standard edit menu: *cmCut*, *cmCopy*, *cmClear*, *cmPaste*, and *cmUndo*. The cut, copy, and clear commands act on the selected text in the editor. The paste command inserts the contents of the clipboard editor at the cursor position. The undo command undoes all edits since the last cursor movement.

Other editing commands, such as search and replace, are handled by the window that owns the editor object by displaying an editor dialog box that prompts the user for search and replace text and options. The owner then calls the editor's *Search* method with the appropriate options.

Updating the active commands

Not all editing commands are valid at all times. For example, there's no point in sending the *cmCut* command if the user hasn't selected any characters to cut. Editor objects have a method called *UpdateCommands* that enables and disables commands based on the current state of the editor and clipboard. It is called whenever the state of the editor changes.

The commands to find, replace, and search again are always active, while those to cut, copy, clear, and paste depend on whether the user has selected text. The undo command is active only if the user has inserted or deleted text since the last cursor movement.

Editor key bindings

By default, editor objects bind commands to many of the familiar WordStar key sequences used in the IDE, including cursor movements and text deletions. The main exceptions are the block commands.

You can change these key bindings by overriding the *ConvertEvent* method, which translates certain keyboard events into command events.

Manipulating blocks

Since *TEditor* does not use persistent blocks, it simulates the block commands by copying text to and from the clipboard. For example, *Ctrl+K Ctrl+B* begins selecting text. *Ctrl+K Ctrl+K* copies the text to the clipboard. *Ctrl+K Ctrl+C* pastes the text from the clipboard to the editor. This simulates quite closely the WordStar keystrokes. You can also select a block of text by holding down the *Shift* key with any of the cursor movement commands.

Editor options

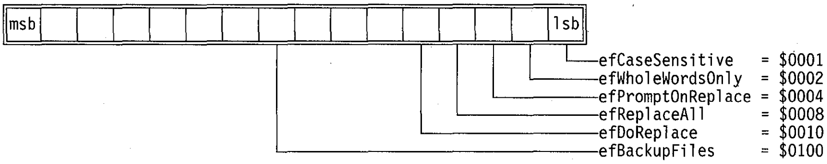
Editor objects provide several options, selectable using Boolean fields:

- *CanUndo* indicates whether the editor records undo information. Since undo information temporarily takes space from inserts, you might find it advantageous to disable undo. You should always set *CanUndo* to *False* for clipboard editors.
- *Overwrite* indicates whether the editor is in overwrite or insert mode.

- *AutoIndent* records whether pressing Enter causes the editor to indent the new line to the column of the first nonspace character of the previous line rather than to the leftmost column. This is convenient for editing source code.

Editor objects also use a bitmapped variable in the *Editors* unit, *EditorFlags*, to determine certain options that apply to all editors in the application. *EditorFlags* controls creation of backup files and certain search and replace options. Figure 15.6 shows the mapping of the bits in *EditorFlags*.

Figure 15.6
Editor flag bit mapping



The editor flag bits are reasonably self-explanatory. If you need further details, consult Chapter 19, "Turbo Vision reference."

Searching and replacing

Search and replace operations are handled by command responses. Instead of directly calling a text-search method, for example, the user generates a *cmFind* command, which the editor object responds to by displaying the appropriate dialog box prompting for the search text and options.

Similarly, the *cmReplace* command causes the editor to prompt the user for search text and replacement text, as well as options. The dialog boxes used for these operations are controlled by the *EditorDialogs* function.

Using the memo field

The memo object is a special extension of the editor object designed for use as a control in a dialog box. It has no special editing abilities, but it adds certain facilities needed for use as a control:

- A palette that maps onto the dialog box
- *GetData*, *SetData*, and *DataSize* methods

Memo colors

Most editor objects use the standard scroller palette, which defaults to yellow characters on a white background. Because memo objects generally exist only in dialog boxes, they map onto a more natural black-on-cyan color combination.

Acting like a control

In order to act like a control, memo objects have to handle two things that other editors don't:

- Tabbing between fields
- Reading and writing values to a data record

Handling Tab

Normally, an editor object handles *Tab* characters by inserting a tab into the text. Since a memo acts as a control in a dialog box, it traps keyboard events with a character code of *kbTab* and assures that the dialog box will handle the normal *Tab* behavior, moving the focus to the next field. The memo object passes all other events to the event handler it inherits from *TEditor*.

Setting and reading values

Controls need to set their values from a data record and read their values back into that record. *TMemo* defines the three methods needed to handle that transfer: *DataSize*, *GetData*, and *SetData*.

DataSize returns the size of the editor's buffer, plus the size of a length word. *GetData* and *SetData* treat the data record similar to a long string, treating the first two bytes as the text length, and the remaining bytes as the memo text.

The portion of the data record for the memo field should have two entries. For example, Listing 15.2 shows the data record for a simple dialog box that has only one control, a memo field.

Listing 15.2
Data record for a memo field

```
{ Note that ABufSize is the same constant passed to the editor's
  constructor as its maximum buffer size. }
type
  TDialogData = record
    MemoLength: Word;
    MemoText: array[0..ABufSize] of Char;
  end;
```

Using file editors

A *file editor* is an editor object that's linked to a specific text file. It has no extra editing functions, but it adds the following features:

- File loading and saving
- Flexible buffers

Using a file editor requires only one change in the editor constructor, but there are also some concepts you have to understand to make best use of the file editor objects. This section covers:

- Constructing a file editor
- Working with files
- Working with buffers

Constructing a file editor

The constructor for a file editor is almost identical to that of a regular editor, but instead of the last parameter being a buffer size, you pass the name of the file you want to edit. The file editor will set up its own buffer, as described in the section "Working with buffers" on page 272.

The file editor stores the name of the current file in a field called *FileName*. If you pass an empty string as the file name, the file editor assumes you're creating a new file.

If the global variable *EditorFlags* has its *efBackupFiles* bit set, the file editor automatically keeps a copy of the last saved version of an edited file (with its extension changed to .BAK) when saving a file.

Working with files

The obvious difference between a file editor and a standard editor is the fact that the file editor needs to manage its associated file. Most of this is handled transparently when you construct and destruct the object, but if you want to customize behavior, you'll need to know some detail about the following topics:

- Loading a file
- Saving a file
- Ensuring that changes are saved

Loading a file

When you pass a file name to the file editor's constructor, the object checks to see if the file name represents a valid file, then calls the method *LoadFile* to assign a buffer and read the contents of the file into the editor buffer. If the file is too large for the editor, or if the editor can't allocate a large enough buffer, the file editor displays an "Out of memory" dialog box and *LoadFile* returns *False*.

☞ It's generally not a good idea to call *LoadFile* at any other time. If you want to load a different file into the file editor, you're better off disposing of the existing editor and constructing a new one in its place. That ensures that associations between file names and editor buffers stay valid, and that buffer memory is managed properly.

Saving a file

In addition to the editing commands understood by all editor objects, file editors respond to two additional commands: *cmSave* and *cmSaveAs*. The event handler for *TFileEditor* calls the methods *Save* and *SaveAs*, respectively, in response to those commands. *TFileEditor*'s *UpdateCommands* method calls the inherited *UpdateCommands* and then enables *cmSave* and *cmSaveAs*.

The main difference between *Save* and *SaveAs* is that *Save* assumes you want to use the file named in *FileName* to save the current editor buffer, while *SaveAs* assumes you want to assign a new file. If *FileName* is an empty string, *Save* calls *SaveAs* to assign a name to the file.

The actual saving of the file's text is handled by a method called *SaveFile*. You should never call *SaveFile* directly; instead, you should call *Save* or *SaveAs*. *SaveFile* takes care of keeping backup copies of files if *EditorFlags* has its *efBackupFiles* bit set, then writes the contents of the editor buffer into the file named by *FileName*.

Making sure changes get saved

If the user or another object tries to close a file editor that hasn't had its changes saved (that is, it's *Modified* field is *True*), the file editor's *Valid* method displays a dialog box warning the user that the modifications need to be saved, giving the options of saving or disposing of the changes. The user can then either save the changes, throw away the changes, or abort closing the editor (by having Valid return *False*).

Working with buffers

File editors need somewhat greater flexibility in their buffer handling than most editors, so instead of allocating file edit buffers on the heap, Turbo Vision sets aside file edit buffer space *above* the heap. That allows file edit buffers to grow, shrink, and move. Most of the buffer handling is automatic, but you do have to specify how much memory you want your application to set aside.

Note that what you specify is the size of the regular Pascal heap, with everything remaining made available for your file editors. This ensures that your application always gets the amount of memory it needs, with file editors sharing what remains.

Specifying buffer space

The *Editors* unit declares a global variable called *MaxHeapSize* that you must set if your application uses file editors. *MaxHeapSize* defaults to 640K, meaning that your application's heap get all available memory, with no memory available for file edit buffers.

☞ If you try to use a file editor without setting aside buffer space, your program will crash and possibly hang your system.

There are two things you need to be aware of about *MaxHeapSize*:

- It specifies the size of your application's heap in 16-byte paragraphs. Memory beyond that amount is not available to the rest of your application, so be sure to allocate enough. Setting *MaxHeapSize* to 4096, for example, sets aside 64K for the application's heap, with everything else left for file edit buffers.
- You must set *MaxHeapSize* before allocating any memory from the heap. The safest thing to do is make it the first statement in your application object's constructor, *before* calling the inherited constructor:

```
constructor TMyApp.Init;
begin
  MaxHeapSize := 4096;                      { this must come first }
  inherited Init;                          { this allocates memory }
end;
```

Managing file edit buffers

You should never need to manipulate a file editor's buffer yourself. *TFileEditor* overrides the virtual methods *DoneBuffer, InitBuffer,* and *SetBufSize* to ensure that the editor uses file buffer space rather than heap space, but you should never need to call those directly.

File editor buffers come in 4K increments. That is, when *LoadFile* requests a buffer for its file, it passes the size of the file to *SetBufSize,* which attempts to allocate that many bytes, rounded up to the nearest 4K boundary. If the size of the editor's gap shrinks to zero, the file buffer grows in increments of 4K, if memory is available.

Using the clipboard

Turbo Vision's editor objects all support cutting, copying, and pasting with a clipboard, but to use these features, you have to create a clipboard object. Any editor can serve as the clipboard, but most often it's an unnamed file editor in an editor window so that you can display and edit the clipboard easily.

Using the clipboard requires only two extra steps in your application:

- Constructing the clipboard editor
- Assigning the editor to *Clipboard*

Constructing the clipboard editor

You can use any Turbo Vision editor object as your application's clipboard, but you need to ensure that the clipboard is available at all times. In general, that means having a separate editor dedicated to the clipboard. Using a file editor for the clipboard gives you the benefit of flexible size without devoting a large part of your regular application heap.

Assigning the Clipboard variable

The *Editors* unit declares a global variable called *Clipboard* of type *PEditor* that your application should assign its clipboard editor to if you plan to allow clipboard operations. File editors behave somewhat differently if they know they are the clipboard, and cut, copy, and paste operations will fail if *Clipboard* is not assigned to a valid editor object.

Listing 15.3 shows a typical creation of a clipboard window, and the assignment of its editor to *Clipboard*.

Listing 15.3
Assigning a clipboard editor

```
type
  TMyApp = object(TApplication)
    ClipWindow: PEditWindow;
    constructor Init;
      ⋮
  end;

constructor TMyApp.Init;
begin
  MaxHeapSize := 4096;                  { allow 64K for file editors }
  inherited Init;                            { construct application }
  New(ClipWindow, Init(R, '', wnNoNumber));       { construct window }
  if ValidView(ClipWindow) <> nil then      { if it's a valid window }
    Clipboard := ClipWindow^.Editor;     { make editor the clipboard }
end;
```

Using an editor window

An editor window (type *TEditWindow*) is a window object designed to hold a file editor. It changes its title to show the name of the file being edited and automatically sets up scroll bars and an indicator for the editor. The editor window keeps a pointer to its associated editor through a field called *Editor*.

Constructing the editor window

Constructing an editor window is exactly like constructing any other window, with the exception that the second parameter passed to the constructor is the name of a file to edit. The title of the window reflects the file being edited: 'Clipboard' if the editor is the application's clipboard or 'Untitled' if the file name is an empty string, or the full path name of the file.

Other editor window considerations

There are only two other ways in which an editor window behaves differently from a plain window: its closing behavior and its response to one broadcast command.

When told to close, an editor window will close like any other window (including calling *Valid* for all its subviews), unless the window holds the clipboard editor, in which case it hides itself instead of closing. This enables you to edit the clipboard in a window without losing it every time you close the window.

Editor windows respond to one broadcast command event that normal windows don't need to handle. When the name of the file in the editor changes (generally after a Save As operation), the window receives a broadcast event with the command *cmUpdateTitle*, which alerts the window that it needs to redraw its frame to include the new file name.

C H A P T E R

16

Collections

Pascal programmers traditionally spend much programming time creating code that manipulates and maintains data structures, such as linked lists and dynamically sized arrays. Virtually the same data structure code tends to be written and debugged again and again.

As powerful as traditional Pascal is, it provides you with only built-in record and array types. Any structure beyond that is up to you.

For example, if you're going to store data in an array, you typically need to write code to create the array, to import data into the array, to extract array data for processing, and perhaps to export data to I/O devices. Later, when the program needs a new array element type, you start all over again.

Wouldn't it be great if an array type came with code that would handle many of the operations you normally perform on an array? An array type that could also be extended without disturbing the original code?

That's the aim of Turbo Vision's *TCollection* type. It's an object that stores a collection of pointers and provides a host of methods for manipulating them.

Collection objects

Besides being an object, and therefore having methods built into it, a collection has two additional features that address shortcomings of ordinary Pascal arrays—it is dynamically sized and polymorphic.

Collections are dynamically sized

The size of a standard Turbo Pascal array is fixed at compile time, which is fine if you know exactly what size your array will always need to be, but it may not be a particularly good fit by the time someone is actually running your code. Changing the size of an array requires changing the code and recompiling.

With a collection, however, you set an initial size, but it can dynamically grow at run time to accommodate the data stored in it. This makes your application much more flexible in its compiled form.

Collections are polymorphic

A second aspect of arrays that can be limiting to your application is the fact that each element in the array must be of the same type, and that type must be determined when the code is compiled.

Collections get around this limitation by using untyped pointers. Not only is this fast and efficient, but a collection can then consist of objects (and even non-objects) of different types and sizes. Just like a stream, a collection doesn't need to know anything about the objects it is handed. It just holds on to them and gives them back when asked.

Type checking and collections

A collection is an end-run around Pascal's traditional strong type checking. That means that you can put anything into a collection, and when you take something back out, the compiler has no way to check your assumptions about what that something is. You can put in a *PHedgehog* and read it back out as a *PSheep*, and the collection will have no way of alerting you.

As a Turbo Pascal programmer, you may rightfully feel nervous about such an end-run. Pascal's type checking, after all, saves hours and hours of hunting for some very elusive bugs. So you should proceed with caution here: You may not even be aware of how difficult a mixed-type bug can be to find, because the compiler has been finding all of them for you! However, if you find that your programs are crashing or locking up, carefully check the types of objects being stored in and read from collections.

Collecting non-objects

You can even add something to a collection that isn't an object at all, but this raises another serious point of caution. Collections expect to receive untyped pointers to something. But some of *TCollection*'s methods act specifically on a collection of *TObject*-derived instances. These include the stream access methods *PutItem* and *GetItem* as well as the standard *FreeItem* procedure.

This means that you can store a *PString* in a collection, for example, but if you try to send that collection to a stream, the results aren't going to be pretty unless you override the collection's standard *GetItem* and *PutItem* methods. Similarly, when you attempt to deallocate the collection, it will try to dispose of each item using *FreeItem*. If you plan to use non-*TObject* items in a collection, you need to redefine the meaning of "item" in *GetItem*, *PutItem*, and *FreeItem*. That is precisely what *TStringCollection*, for example, does.

If you proceed with prudence, you will find collections (and the descendants of collections that you build) to be fast, flexible, dependable data structures.

Creating a collection

Creating a collection is really just as simple as defining the data type you wish to collect. Suppose you're a consultant, and you want to store and retrieve the account number, name, and phone number of each of your clients. First you define the client object (*TClient*) that will be stored in the collection:

Remember to define a pointer for each new object type.

```
type
  PClient = ^TClient;
  TClient = object(TObject)
    Account, Name, Phone: PString;
    constructor Init(NewAccount, NewName, NewPhone: String);
    destructor Done; virtual;
  end;
```

Next you implement the *Init* and *Done* methods to allocate and dispose of the client data. Note that the object fields are of type *PString* so that memory is only allocated for the portion of the string that is actually used. The *NewStr* and *DisposeStr* functions handle dynamic strings very efficiently.

```
constructor TClient.Init(NewAccount, NewName, NewPhone: String);
begin
  Account := NewStr(NewAccount);
  Name := NewStr(NewName);
  Phone := NewStr(NewPhone);
end;

destructor TClient.Done;
begin
  DisposeStr(Account);
  DisposeStr(Name);
  DisposeStr(Phone);
end;
```

TClient.Done will be called automatically for each client when you dispose of the entire collection. Now you just instantiate a collection to store your clients, and insert the client records into it. The main body of the program looks like this:

This is COLLECT1.PAS.

```
var
  ClientList: PCollection;

begin
  ClientList := New(PCollection, Init(50, 10));
  with ClientList^ do
  begin
    Insert(New(PClient, Init('90-167', 'Smith, Zelda',
      '(800) 555-1212')));
    Insert(New(PClient, Init('90-160', 'Johnson, Agatha',
      '(302) 139-8913')));
    Insert(New(PClient, Init('90-177', 'Smitty, John',
      '(406) 987-4321')));
    Insert(New(PClient, Init('91-100', 'Anders, Smitty',
      '(406) 111-2222')));
  end;
  PrintAll(ClientList);
  Writeln; Writeln;
  SearchPhone(ClientList, '(406)');
  Dispose(ClientList, Done);
end.
```

PrintAll and SearchPhone are procedures that will be discussed later.

Notice how easy it was to build the collection. The first statement allocates a new *TCollection* called *ClientList* with an initial size of 50 clients. If more than 50 clients are inserted into *ClientList*, its size will increase in increments of 10 clients whenever needed.

The next 2 statements create a new client object and insert it into the collection. The *Dispose* call at the end frees the entire collection—clients and all.

Nowhere did you have to tell the collection what *kind* of data it was collecting—it just took a pointer.

Iterator methods

Insert and deleting items aren't the only common collection operations. Often you'll find yourself writing **for** loops to range over *all* the objects in the collection to display the data or perform some calculation. Other times, you'll want to find the first or last item in the collection that satisfies some search criterion. For these purposes, collections have three *iterator* methods: *ForEach, FirstThat,* and *LastThat*. Each of these takes a pointer to a procedure or function as its only parameter.

The ForEach iterator

ForEach takes a pointer to a procedure. The procedure has one parameter, which is a pointer to an item stored in the collection. *ForEach* calls that procedure once for each item in the collection, in the order that the items appear in the collection. The *PrintAll* procedure in *COLLECT1* shows an example of a *ForEach* iterator.

```
procedure PrintAll(C: PCollection);     { print info for all clients }

  procedure PrintClient(P: PClient); far;         { local procedure }
  begin
    with P^ do
      Writeln(Account^, '':20-Length(Account^),  { show client info }
        Name^, '':20-Length(Name^),
        Phone^, '':20-Length(Phone^));
  end;                                      { end of local procedure }

begin                                                     { PrintAll }
  Writeln;
  Writeln;
  C^.ForEach(@PrintClient);    { Call PrintClient for each item in C }
end;
```

For each item in the collection passed as a parameter to *PrintAll,* the nested procedure *PrintClient* is called. *PrintClient* simply prints the client object information in formatted columns.

Iterators must call far local procedures.

You need to be careful about what sort of procedures you call with iterators. In this example, *PrintClient* must be a procedure—it cannot be an object's method—and it must be local to (nested in the same block with) the routine that is calling it. It must also be declared as a far procedure, either with the **far** directive or with the **$F+** compiler directive. Finally, the procedure must take a pointer to a collection item as its only parameter.

The FirstThat and LastThat iterators

In addition to being able to apply a procedure to every element in the collection, it is often useful to be able to find a particular element in the collection based on some criterion. That is the purpose of the *FirstThat* and *LastThat* iterators. As their names imply, they search the collection in opposite directions until they find an item meeting the criteria of the Boolean function passed as an argument.

FirstThat and *LastThat* return a pointer to the first (or last) item that matches the search conditions. Consider the earlier example of the client list, and imagine that you can't remember a client's account number or exactly how his last name is spelled. Luckily, you distinctly recall that this was the first client you acquired in the state of Montana. Thus you want to find the first occurrence of a client in the 406 area code (since your list happens to be in chronological order). Here's a procedure using the *FirstThat* method that would do the job:

```
procedure SearchPhone(C: PClientCollection; PhoneToFind: String);

  function PhoneMatch(Client: PClient): Boolean; far;
  begin
    PhoneMatch := Pos(PhoneToFind, Client^.Phone^) <> 0;
  end;

var
  FoundClient: PClient;
begin
  FoundClient := C^.FirstThat(@PhoneMatch);
  if FoundClient = nil then
    Writeln('No client met the search requirement')
  else
    with FoundClient^ do
      Writeln('Found client: ', Account^, '  ', Name^, '  ', Phone^);
end;
```

Again notice that *PhoneMatch* is nested and uses the far call model. In this case, it's a function that returns *True* only if the client's phone number and the search pattern match. If no object in the collection matches the search criteria, a **nil** pointer is returned.

Remember: *ForEach* calls a user-defined procedure, while *FirstThat* and *LastThat* each call a user-defined Boolean function. In all cases, the user-defined procedure or function is passed a pointer to an object in the collection.

Sorted collections

Sometimes you need to have your data in a certain order. Turbo Vision provides a special type of collection that allows you to order your data in any manner you want: the *TSortedCollection*.

TSortedCollection is a descendant of *TCollection* which automatically sorts the objects it is given. It also automatically checks the collection when a new member is added and rejects duplicate members.

TSortedCollection is an abstract type. To use it, you must first decide what type of data to collect and define two methods to meet your particular sorting requirements. To do this, you will need to derive a new collection type from *TSortedCollection*. In this case, call it *TClientCollection*.

Your *TClientCollection* already knows how to do all the real work of a collection. It can *Insert* new client records and *Delete* existing ones—it inherited all this basic behavior from *TCollection*. All you have to do is teach *TClientCollection* which field to use as a sort key and how to compare two clients and decide which one belongs ahead of the other in the collection. You do this by overriding the *KeyOf* and *Compare* methods and implementing them as shown here:

```
PClientCollection = ^TClientCollection;
TClientCollection = object(TSortedCollection)
  function KeyOf(Item: Pointer): Pointer; virtual;
  function Compare(Key1, Key2: Pointer): Integer; virtual;
end;
```

```
function TClientCollection.KeyOf(Item: Pointer): Pointer;
begin
  KeyOf := PClient(Item)^.Name;
end;
```

Keys must be typecast because they are untyped pointers.

```
function TClientCollection.Compare(Key1, Key2: Pointer): Integer;
begin
  if PString(Key1)^ = PString(Key2)^ then
    Compare := 0                          { return 0 if they're equal }
  else if PString(Key1)^ < PString(Key2)^ then
    Compare := -1                     { return -1 if Key1 comes first }
  else
    Compare := 1;              { otherwise return 1; Key2 comes first }
end;
```

KeyOf defines which field or fields should be used as a sort key. In this case, it's the client's *Name* field. *Compare* takes two sort keys and determines which one should come first in the sorted order. *Compare* returns –1, 0, or 1, depending on whether *Key1* is less than, equal to, or greater than *Key2*. This example uses a straight alphabetical sort of the key (*Name*) strings.

Note that since the keys returned by *KeyOf* and passed to *Compare* are untyped pointers, you need to typecast them into *PString*s before dereferencing them.

That's all you have to define! Now if you redefine *ClientList* as a *PClientCollection* instead of a *PCollection* (changing the **var** declaration and the *New* call), you can easily list your clients in alphabetical order:

This is COLLECT2.PAS.

```
var
  ClientList: PClientCollection;
    ⋮
begin
  ClientList := New(PClientCollection, Init(50, 10));
     ⋮
end.
```

Notice also how easy it would be if you wanted the client list sorted by account number instead of by name. All you would have to do is change the *KeyOf* method to return the *Account* field instead of the *Name* field.

String collections

Many programs need to keeping track of sorted strings. For this purpose, Turbo Vision provides a special purpose collection, *TStringCollection*. Note that the elements in a *TStringCollection* are *not* objects—they are pointers to Turbo Pascal strings. Since a string collection is a descendant of *TSortedCollection*, duplicate strings are not stored.

Using a string collection is easy. Just declare a pointer variable to hold the string collection. Allocate the collection, giving it an initial size and an amount to grow by as more strings are added

This is COLLECT3.PAS.

```
var
  WordList: PCollection;
  WordRead: String;
      ⋮
begin
  WordList := New(PStringCollection, Init(10, 5));
      ⋮
```

WordList holds ten strings initially and then grows in increments of five. All you have to do is insert some strings into the collection. In this example, words are read out of a text file and inserted into the collection:

```
repeat
      ⋮
  if WordRead <> '' then
    WordList^.Insert(NewStr(WordRead));
      ⋮
until WordRead = '';
    ⋮
Dispose(WordList, Done);
```

Notice that the *NewStr* function is used to make a copy of the word that was read and the address of the string copy is passed to the collection. When using a collection, you always give it control over the data you're collecting. It will take care of deallocating the data when you're done. And that's exactly what the call to *Dispose* does; it disposes each element in the collection, and then disposes the *WordList* collection itself.

Iterators revisited

The *ForEach* method traverses the entire collection one item at a time, and passes each one to a procedure you provide. Continuing with the previous example, the procedure *PrintWord* is given a pointer to a string to display. Note that *PrintWord* is a nested (or local) procedure. Wrapped around it is another procedure, *Print,* which is given a pointer to a *TStringCollection.* Print uses the *ForEach* iterator method to pass each item in its collecton to the *PrintWord* procedure.

```
procedure Print(C: PCollection);

  procedure PrintWord(P : PString); far;
  begin
    Writeln(P^);                { Display the string }
  end;

begin { Print }
  Writeln;
  Writeln;
  C^.ForEach(@PrintWord);      { Call PrintWord }
end;
```

The CallDraw procedure in COLLECT4.PAS shows how to call a method from inside an iterator call.

PrintWord should look familiar; it's just a procedure that takes a string pointer and passes its value to *Writeln.* Note the **far** directive after *PrintWord*'s declaration. *PrintWord* cannot be a method—it must a procedure. And it must be a nested procedure as well. Think of *Print* as a wrapper around a procedure that has the job of doing something—displaying or modifying data, perhaps—with each item in the collection. You can have more than one procedure like the preceding *PrintWord,* but each has to be nested inside *Print* and each has to be a far procedure (using the **far** directive or {**$F+**}).

Finding an item

Sorted collections (and therefore string collections) have a *Search* method that returns the index of an item with a particular key. But how do you find an item in a collection that may not be sorted? Or when the search criteria don't involve the key itself? The answer, of course, is to use *FirstThat* and *LastThat.* You simply define a Boolean function to test for whatever criteria you want, and call *FirstThat.*

Polymorphic collections

You've seen that collections can store any type of data dynamically, and there are plenty of methods to help you access collection data efficiently. In fact, *TCollection* itself defines 23 methods. When you use collections in your programs, you'll be equally impressed by their speed. They're designed to be flexible and implemented to be fast.

But now comes the *real* power of collections: items can be treated polymorphically. That means you can do more than just store an object type on a collection; you can store many different objects types, from anywhere in your object hierarchy.

If you consider the collection examples you've seen so far, you'll realize that all the items on each collection were of the same type. There was a list of strings in which every item was a string. And there was a collection of clients. But collections can store *any* object that is a descendant of *TObject*, and you can mix these objects freely. Naturally, you'll want the objects to have something in common. In fact, you'll want them to have an abstract ancestor object in common.

As an example, here's a program that puts 3 different graphical objects into a collection. Then a *ForEach* iterator is used to traverse the collection and display each object.

This example uses the *Graph* unit and BGI drivers, so make sure GRAPH.TPU is in the current directory or on your unit path (**O**ptions | **D**irectories | **U**nit directory) when you compile. When you run the program, change to the directory that contains the .BGI drivers or modify the call to *InitGraph* to specify their location (for example, C:\TP\BGI).

The abstract ancestor object is defined first.

This is COLLECT4.PAS.

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    X,Y: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;
```

You can see from this declaration that each graphical object can initialize itself (*Init*) and display itself on the graphics screen

(*Draw*). Now define a point, a circle, and a rectangle, each descended from this common ancestor:

```
PGraphPoint = ^TGraphPoint;
TGraphPoint = object(TGraphObject)
  procedure Draw; virtual;
end;

PGraphCircle = ^TGraphCircle;
TGraphCircle = object(TGraphObject)
  Radius: Integer;
  constructor Init;
  procedure Draw; virtual;
end;

PGraphRect = ^TGraphRect;
TGraphRect = object(TGraphObject)
  Width, Height: Integer;
  constructor Init;
  procedure Draw; virtual;
end;
```

These three object types all inherit the *X* and *Y* fields from *PGraphObject*, but they are all different sizes. *PGraphCircle* adds a *Radius*, while *PGraphRect* adds a *Width* and *Height*. Here's the code to make the collection:

```
    ⋮
List := New(PCollection, Init(10, 5));      { Create collection }

for I := 1 to 20 do
begin
  case I mod 3 of                          { Create an object }
    0: P := New(PGraphPoint, Init);
    1: P := New(PGraphCircle, Init);
    2: P := New(PGraphRect, Init);
  end;
  List^.Insert(P);                          { Add it to collection }
end;
    ⋮
```

As you can see, the **for** loop inserts 20 graphical objects into the *List* collection. All you know is that each object in *List* is some kind of *TGraphObject*. But once inserted, you'll have no idea whether a given item in the collection is a circle, point or rectangle. Thanks to polymorphism, you don't need to know since each object contains the data and the code (*Draw*) it needs. Just traverse the collection using an iterator method and have each object display itself:

```
procedure DrawAll(C: PCollection);

procedure CallDraw(P: PGraphObject); far;
begin
    P^.Draw;                            { Call the Draw method }
end;

begin { DrawAll }
  C^.ForEach(@CallDraw);                { Draw each object }
end;

var
  GraphicsList: PCollection;
begin
    ⋮
  DrawAll(GraphicsList);
    ⋮
end.
```

This ability of a collection to store different but related objects leans on one of the powerful cornerstones of object-oriented programming. In the next chapter, you'll see this same principal of polymorphism applied to streams with equal advantage.

Collections and memory management

A *TCollection* can grow dynamically from the initial size set by *Init* to a maximum size of 16,380 elements. The maximum collection size is stored by Turbo Vision in the variable *MaxCollectionSize*. Each element you add to a collection only takes four bytes of memory, because the element is stored as a pointer.

No library of dynamic data structures would be complete unless it provided some provision for error detection. If there is not enough memory to initialize a collection, a **nil** pointer is returned.

If memory is not available when adding an element to a *TCollection*, the method *TCollection.Error* is called and a run-time heap memory error occurs. You may want to override *TCollection.Error* to provide your own error reporting or recovery mechanism.

You need to pay special attention to heap availability, because the user has much more control of a Turbo Vision program than a traditional Pascal program. If the user is the one who controls the adding of objects to a collection (for example, by opening new windows on the desktop), the possibility of a heap error may not be so easy to predict. You may need to take steps to protect the user from a fatal run-time error, with either memory checks of your own when a collection is being used, or a run-time error handler that lets the program recover gracefully.

CHAPTER

17

Streams

Object-oriented programming techniques and Turbo Vision give you a powerful way of encapsulating code and data, and powerful ways of building an interrelated structure of objects. But what if you want to do something simple, like store some objects on disk?

Back in the days when data sat by itself in a record, writing data to disk was pretty clear-cut, but the data within a Turbo Vision program is largely bound up within objects. You could, of course, separate the data from the object and write the data to a disk file. But you've achieved something important by joining the two together in the first place, and it would be a step backwards to take them apart.

Couldn't OOP and Turbo Vision themselves somehow be enlisted in solving this problem? That's what streams are all about.

A Turbo Vision stream is a collection of objects on its way somewhere: typically to a file, EMS, a serial port, or some other device. Streams handle I/O on the object level rather than the data level. When you extend a Turbo Vision object, you need to provide for handling any additional data fields that you define. All the complexity of handling the object representation is taken care of for you.

The question: Object I/O

As a Pascal programmer, you know that before you can do any file I/O, you must tell the compiler what type of data you will be reading or writing to the file. The file must be typed, and the type must be determined at compile time.

Turbo Pascal implements a very useful workaround to this rule: an untyped file accessed with *BlockWrite* and *BlockRead*. But the lack of type checking creates some extra responsibilities for the programmer, although it does let you perform very fast binary I/O.

A second problem, though, is that you can't use files directly with objects. Turbo Pascal doesn't allow you to create a typed file of objects. And because objects may contain virtual methods who addresses are determined at run time, storing the VMT information outside the program is pointless; reading such information *into* a program is even more so.

Again, you can work around the problem. You can copy the data out of your objects and store the information in some sort of file, then rebuild the objects from the raw data again later. But that is a rather inelegant solution at best, and complicates the construction of objects.

The answer: Streams

Turbo Vision allows you to overcome both of these difficulties, and gives you some side benefits as well. Streams provide a simple, yet elegant, means of storing object data outside your program.

Streams are polymorphic

A Turbo Vision stream gives you the best of both typed and untyped files: type checking is still there, but what you intend to send to a stream doesn't have to be determined at compile time. The reason is that streams know they are dealing with objects, so as long as the object is a descendant of *TObject*, the stream can handle it. In fact, different Turbo Vision objects can as easily be written to the same stream as a group of identical objects.

Streams handle objects

All you have to do is define for the stream which objects it needs to handle, so it knows how to match data with VMTs. Then you can put objects onto the stream and get them back effortlessly.

But how can the same stream read and write such widely differing objects as a *TDeskTop* and a *TDialog*, and not even need to know at compile time what objects it is going to be handed? This is *very* different from traditional Pascal I/O. In fact, a stream can even handle new object types that weren't even created when the stream was compiled.

The answer is *registration*. Each Turbo Vision object type (and any new object types you derive from the hierarchy) is assigned a unique registration number. That number gets written to the stream ahead of the object's data. Then, when you go to read the object back from the stream, Turbo Vision gets the registration number first, and based on that knows how much data to read and what VMT to attach to your data.

Essential stream usage

On a fairly fundamental level, you can think about streams much as you think about Pascal files. At its most basic, a Pascal file can be simply a sequential I/O device: you write things to it, and you read them back. A stream, then, is a *polymorphic* sequential I/O device, meaning that it behaves much like a sequential file, but you can also read or write various types of objects at the current point.

Streams can also (like Pascal files) be viewed as a random-access I/O devices, where you seek to a position in the file, read or write at that point, return the position of the file pointer, and so on. These operations are also available with streams, and are described in the section "Random-access streams."

There are two different aspects of stream usage that you need to master, and luckily they are both quite simple. The first is setting up a stream, and the second is reading and writing objects to the stream.

Setting up a stream

All you have to do to use a stream is initialize it. The exact syntax of the *Init* constructor will vary, depending on what type of stream you're dealing with. For example, if you're opening a DOS stream, you need to pass the name of the DOS file and the access mode (read-only, write-only, read/write) for the file containing the stream.

For example, to initialize a buffered DOS stream for loading the desktop object into a program, all you need to is this:

```
var
  SaveFile: TBufStream;
begin
  SaveFile.Init('SAMPLE.DSK', stOpen, 1024);
     ⋮
```

Once you've initialized the stream, you're ready to go—that's all there is to it.

TStream is an abstract stream mechanism, so you can't actually create an instance of it, but useful stream objects are all derived from *TStream*. These include *TDosStream*, which provides disk I/O, and *TBufStream*, which provides buffered disk I/O (useful if you read or write a lot of small pieces to disk), and *TEmsStream*, a stream that sends objects to EMS memory (especially useful for implementing fast resources).

Turbo Vision also implements an indexed stream, with a pointer to a place in the stream. By relocating the pointer, you can do random stream access.

Reading and writing a stream

TStream, the basic stream object implements three basic methods you need to understand: *Get*, *Put*, and *Error*. *Get* and *Put* roughly correspond to the *Read* and *Write* procedures you would use for ordinary file I/O operations. *Error* is a procedure that gets called whenever a stream error occurs.

Putting it on

Let's look first at the *Put* procedure. The general syntax of a *Put* method is this:

```
SomeStream.Put(PSomeObject);
```

where *SomeStream* is any object descended from *TStream* that has been initialized, and *PSomeObject* is a pointer to any object descended from *TObject* that has been registered with the stream. That's all you have to do. The stream can tell from *PSomeObject*'s VMT what type of object it is (assuming the type has been registered), so it knows what ID number to write, and how much data to write after it.

Of special interest to you as a Turbo Vision programmer, however, is the fact that when you *Put* a group with subviews onto a stream, the subviews are automatically written to the stream as well. Thus, saving complex objects is not complex at all—in fact, it's automatic! You can save the entire state of your program simply by writing the desktop onto a stream. When you restart your program and load the desktop back in, it will be in the same condition it was in when you saved it.

Getting it back

Getting objects back from the stream is just as easy. All you have to do is call the stream's *Get* function:

```
PSomeObject := SomeStream.Get;
```

where again, *SomeStream* is an initialized Turbo Vision stream, and *PSomeObject* is a pointer to any type of Turbo Vision object. *Get* simply returns a pointer to whatever it has pulled off the stream. How much data it has pulled, and what type of VMT it has assigned to that data, is determined not by the type of *PSomeObject*, but by the type of object found on the stream. Thus, if the object at the current position of *SomeStream* is not of the same type as *PSomeObject*, you will get garbled information.

As with *Put*, *Get* will retrieve complex objects. Thus, if the object you retrieve from a stream is a view that owns subviews, the subviews will be loaded as well.

In case of error

Finally, the *Error* procedure determines what happens when a stream error occurs. By default, *TStream.Error* simply sets two fields (*Status* and *ErrorInfo*) in the stream. If you want to do anything fancier, like generating a run-time error or popping up an error dialog box, you'll need to override the *Error* procedure.

Shutting down the stream

When you're finished using a stream, you call its *Done* method, much as you would normally call *Close* for a disk file. As with any Turbo Vision object, you do this as

```
Dispose(SomeStream, Done);
```

so as to dispose of the stream object as well as shutting it down.

Making objects streamable

All standard Turbo Vision objects are ready to be used with streams, and all Turbo Vision streams know about the standard objects. When you derive a new object type from one of the standard objects, it is very easy to prepare it for stream use, and to alert streams to its existence.

Load and Store methods

The actual reading and writing of objects to the stream is handled by methods called *Load* and *Store*. While each object must have these methods to be usable by streams, you never call them directly. (They are called by *Get* and *Put*.) So all you need to do is make sure that your object knows how to send itself to the stream when called upon to do so.

Because of OOP, this job is very easy, since most of the mechanism is inherited from the ancestor object. All your object has to handle is loading or storing the parts of itself that you added; the rest is taken care of by calling the ancestor's method.

For example, let's say you derive a new kind of view from *TWindow*, named after the surrealist painter Rene Magritte, who painted many famous pictures of windows:

```
type
  TMagritte = object(TWindow)
    Painted: Boolean;
    constructor Load(var S: TStream);
    procedure Draw;
    procedure Store(var S: TStream);
  end;
```

All that has been added to the data portion of the window is one Boolean field. In order to load the object, then, you simply read a standard *TWindow*, then read an additional byte to accommodate the Boolean field. The same applies to storing the object: you simply write a *TWindow*, then write one more byte. Typical *Load* and *Store* methods for descendant objects look like this:

```
constructor TMagritte.Load(var S: Stream);
begin
  inherited Load(S);                       { load the ancestor type }
  S.Read(Painted, SizeOf(Boolean));        { read additional fields }
end;

procedure TMagritte.Store(var S: Stream);
begin
  inherited Store(S);                     { store the ancestor type }
  S.Write(Painted, SizeOf(Boolean));      { write additional fields }
end;
```

Warning! It is entirely your responsibility to ensure that the same amount of data is stored as is loaded, and that data is loaded in the same order that it is stored. The compiler will return no errors. This can cause huge problems if you are not careful. If you modify an object's fields, make sure to update *both* the *Load* and *Store* methods.

Stream registration

In addition to defining the *Load* and *Store* methods for a new object, you will also have to register your new object type with the streams. Registration is a simple, two-step process: you define a stream registration record, and you pass it to the global procedure *RegisterType*.

To define a stream registration record, just follow the format. Stream registration records are Pascal records of type *TStreamRec*, which is defined as follows:

```
PStreamRec = ^TStreamRec;
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

By convention, all Turbo Vision stream registration records are given the same name as the corresponding object type, with the initial "T" replaced by an "R." Thus, the registration record for *TDeskTop* is *RDeskTop*, and the registration record for *TMagritte* is *RMagritte*. Abstract types such as *TObject* and *TView* do not have registration records because there should never be instances of them to store on streams.

Object ID numbers

The *ObjType* field is really the only part of the record you need to think about; the rest is mechanical. Each new type you define will need its own, unique type-identifier number. Turbo Vision reserves the registration numbers 0 through 99 for the standard objects, so your registration numbers can be anything from 100 through 65,535.

It is your responsibility to create and maintain a library of ID numbers for all your new objects that will be used in stream I/O, and to make the IDs available to users of your units. As with command constants, the numbers you assign may be completely arbitrary, as long as they are unique.

The automatic fields

The *VmtLink* field is a link to the objects virtual method table (VMT). You simply assign it as the offset of the type of your object:

```
RSomeObject.VmtLink := Ofs(TypeOf(TSomeObject)^);
```

The *Load* and *Store* fields contain the addresses of the *Load* and *Store* methods of your object, respectively.

```
RSomeObject.Load := @TSomeObject.Load;
RSomeObject.Store := @TSomeObject.Store;
```

The final field, *Next*, is assigned by *RegisterType*, and requires no intervention on your part. It simply facilitates the internal use of a linked list of stream registration records.

Register here

Once you have constructed the stream registration record, you call *RegisterType* with your record as its parameter. So, to register your new *TMagritte* object for use with streams, you would include the following code:

```
const
  RMagritte: TStreamRec = (
    ObjType: 100;
    VmtLink: Ofs(TypeOf(TMagritte)^);
    Load: @TMagritte.Load;
    Store: @TMagritte.Store
  );

RegisterType(RMagritte);
```

That's all there is to it. Now you can *Put* instances of your new object type to any Turbo Vision stream and read instances back from streams.

Registering standard objects

Turbo Vision defines stream registration records for all its standard objects. In addition, each Turbo Vision unit defines a *RegisterXXXX* procedure that automatically registers all of the objects in that unit.

The stream mechanism

Now that you've examined the process you go through to use streams, you should probably take a quick look behind the scenes to see just what Turbo Vision does with your objects when you *Get* or *Put* them. It's an excellent example of objects interacting and using the methods built into each other.

The Put process

When you send an object to a stream with the stream's *Put* method, the stream first takes the VMT pointer from offset 0 of the object and looks through the list of types registered with the streams system for a match. When it finds the match, the stream retrieves the object's registration ID number and writes it to the stream's destination. The stream then calls the object's *Store* method to finish writing the object. The *Store* method makes use of the stream's *Write* procedure, which actually writes the correct number of bytes to the stream's destination.

Your object doesn't have to know anything about the stream—it could be a disk file, an chunk of EMS memory, or any other sort of stream—your object merely says "Write me to the stream," and the stream handles the rest.

The Get process

When you read an object from the stream with the *Get* method, its ID number is retrieved first, and the list of registered types is scanned for a match. When the match is found, the registration record provides the stream with the location of the object's *Load* method and VMT. The *Load* method is then called to read the proper amount of data from the stream.

Again, you simply tell the stream to *Get* the next object it contains and stick it at the location of the new pointer you specify. Your object doesn't even care what kind of stream it's dealing with. The stream takes care of reading the proper amount of data by using the object's *Load* method, which in turn relies on the stream's *Read* method.

All this is transparent to the programmer, but it shows you how crucial it is to register a type before attempting stream I/O with it.

Handling nil object pointers

You can write a **nil** object to a stream. However, when you do, a word of 0 is written to the stream. On reading an ID word of 0, the stream returns a **nil** pointer. 0 is therefore reserved, and cannot be used as a stream object ID number.

Collections on streams: A complete example

In Chapter 16, "Collections," you saw how a collection could hold different, but related, objects. The same polymorphic ability applies to streams as well, and they can be used to store an entire collection on disk for retrieval at another time or even by another program. Go back and look at COLLECT4.PAS. What more must you do to make that program put the collection on a stream?

The answer is remarkably simple. First, start at the base object, *TGraphObject*, and "teach" it how to store its data (*X* and *Y*) on a stream. That's what the *Store* method is for. Then, similarly define a new *Store* method for each descendant of *TGraphObject* that adds additional fields (*TGraphCircle* adds a *Radius*; *TGraphRec* adds *Width* and *Height*). Next, build a registration record for each object type that will actually be stored and register each of those types when your program first begins. And that's it. The rest is just like normal file I/O: declare a stream variable; create a new stream; put the entire collection on the stream with one simple statement; and close the stream.

Adding Store methods

Here are the *Store* methods. Notice that *PGraphPoint* doesn't need one, since it doesn't add any fields to those it inherits from *PGraphObject*.

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
      ⋮
    procedure Store(var S: TStream); virtual;
  end;

  PGraphCircle = ^TGraphCircle;
  TGraphCircle = object(TGraphObject)
    Radius: Integer;
      ⋮
    procedure Store(var S: TStream); virtual;
  end;

  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    Width, Height: Integer;
      ⋮
    procedure Store(var S: TStream); virtual;
  end;
```

Implementing the *Store* is quite straightforward. Each object calls its inherited *Store* method, which stores all the inherited data, then the stream's *Write* method to write the additional data:

TGraphObject doesn't call TObject.Store because TObject has no data to store.

```
procedure TGraphObject.Store(var S: TStream);
begin
  S.Write(X, SizeOf(X));
  S.Write(Y, SizeOf(Y));
end;
```

```
procedure TGraphCircle.Store(var S: TStream);
begin
  inherited Store(S);
  S.Write(Radius, SizeOf(Radius));
end;

procedure TGraphRect.Store(var S: TStream);
begin
  inherited Store(S);
  S.Write(Width, SizeOf(Width));
  S.Write(Height, SizeOf(Height));
end;
```

Note that *TStream*'s *Write* method does a binary write. Its first parameter can be a variable of any type, but *TStream.Write* has no way to know how big that variable is. The second parameter provides that information and you should follow the convention of using the standard *SizeOf* function. That way, if you decide to change the coordinate system to use floating point numbers, you won't have to revise your *Store* methods.

Registration records

Defining a registration record constant for each of the descendent types is our last step. It's a good idea to follow the Turbo Vision naming convention of using an R as the initial letter, replacing the type's T.

☞ Remember, each registration record gets a unique object ID number (*Objtype*). Turbo Vision reserves 0 through 99 for its standard objects. It's a good idea to keep track of all your objects stream ID numbers in one central place to avoid duplication.

```
const
  RGraphPoint: TStreamRec = (
    ObjType: 150;
    VmtLink: Ofs(TypeOf(TGraphPoint)^);
    Load: nil;                                  { No load method yet }
    Store: @TGraphPoint.Store);

  RGraphCircle: TStreamRec = (
    ObjType: 151;
    VmtLink: Ofs(TypeOf(TGraphCircle)^);
    Load: nil;                                  { No load method yet }
    Store: @TGraphCircle.Store);

  RGraphRect: TStreamRec = (
    ObjType: 152;
    VmtLink: Ofs(TypeOf(TGraphRect)^);
    Load: nil;                                  { No load method yet }
    Store: @TGraphRect.Store);
```

You don't need a registration record for *TGraphObject* beause it's an abstract type and thus won't ever be instantiated or put onto a collection or stream. Each registration record's *Load* pointer is set **nil** here because this example is only concerned with storing data onto a stream. *Load* methods will be defined and the registration records will be updated in the next example (STREAM2.PAS).

Registering

You must always remember to register each of these records before performing any stream I/O. The easiest way to do this is to wrap them all in one procedure and call it at the very beginning of your program (or in your application's *Init* method):

```
procedure StreamRegistration;
begin
  RegisterType(RCollection);
  RegisterType(RGraphPoint);
  RegisterType(RGraphCircle);
  RegisterType(RGraphRect);
end;
```

Notice that you have to register the *TCollection* (using its *RCollection* record—now you see why naming conventions make programming easier) even though you didn't define *TCollection.* The rule is simple and unforgiving: it's *your* responsibility to register every object type that your program will put onto a stream.

Writing to the stream

All that's left to follow is the normal file I/O sequence of: create a stream; put the data (a collection) onto it; close the stream. You don't have to write a *ForEach* iterator to stream each collection item. You just tell the stream to *Put* the collection on the stream:

This is STREAM1.PAS.

```
var
  GraphicsList: PCollection;
  GraphicsStream: TBufStream;
begin
  StreamRegistration;                        { Register all streams }
    ⋮
  { Put the collection in a stream on disk }
  GraphicsStream.Init('GRAPHICS.STM', stCreate, 1024);
  GraphicsStream.Put(GraphicsList);             { Output collection }
  GraphicsStream.Done;                            { Shut down stream }
    ⋮
end.
```

This creates a disk file that contains all the information needed to "read" the collection back into memory. When the stream is opened and the collection is retrieved (see STREAM2.PAS), all the hidden links between the collection and its items, and objects and their virtual method tables will be magically restored. This same technique is used by the Turbo Pascal IDE to save its desktop file. The next example shows you how to do that. But first you have to learn about streaming objects that contain links to other objects.

Who gets to store things?

An important caution about streams: the owner of an object is the only one that should write that object to a stream. This caution is similar to one with which you have probably become familiar while using traditional Pascal: the owner of a pointer is the one that should dispose of the pointer.

In the midst of the complexity of a real-life application, numerous objects will often have a pointer to a particular structure. When the time arrives for stream I/O, you need to decide who "owns" the structure; that owner alone should be the one to send that structure to the stream. Otherwise, you'll end up with multiple copies in the stream of what was initially just one structure. When you then read the stream, multiple instances of the structure will be created, with each of the original objects now pointing at their own personal copy of the structure instead of at the original single structure.

Subview instances

Many times you'll find it convenient to store pointers to a group's subviews in local instance variables. For example, a dialog box will often store pointers to its control objects in mnemonically named fields for easy access (fields like *OKButton* or *FileInputLine*). When that view is then inserted into the view tree, the owner has *two* pointers to the subview, one in the field and one in the subview list. If you don't make allowances for this, reading back the object from a stream will result in duplicate instances.

The solution is provided in the *TGroup* methods called *GetSubViewPtr* and *PutSubViewPtr*. When storing a field that is also a subview, rather than writing the pointer as if it were just another variable, you call *PutSubViewPtr*, which stores a reference to the ordinal position of the subview in the group's subview list. This way, when you *Load* the group back from the stream, you can call *GetSubViewPtr*, which makes sure the field and the subview list point to the same object.

Here's a quick example using *GetSubViewPtr* and *PutSubViewPtr* in a simple window:

```
type
  TButtonWindow = object(TWindow)
    Button: PButton;
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
  end;

constructor TButtonWindow.Load(var S: TStream);
begin
  inherited Load(S);
  GetSubViewPtr(S, Button);
end;

procedure TButtonWindow.Store(var S: TStream);
begin
  inherited Store(S);
  PutSubViewPtr(S, Button);
end;
```

Let's take a look at how this *Store* method differs from a normal *Store*. After storing the window normally, all you have to do is store a reference to the *Button* field, rather than storing the field itself as you would normally do. The actual button object is stored as a subview of the window when you call *TWindow.Store*. All you have to do in addition is put information on the stream indicating that *Button* is to point to that subview. The *Load* method does the same thing in reverse, first loading the window and its button subview, then restoring the pointer to that subview to *Button*.

Peer view instances

A similar situation can arise when a view has a field that points to one of its peers. A view is called a *peer view* of another if both views are owned by the same group. An excellent example is that of a scroller. Because the scroller has to know about two scroll bars which are also members of the same window that contains the scroller, it has two fields that point to those views.

As with subviews, you can run into problems when reading and writing references to peer views to streams. The solution, however, is also similar. The *TView* methods *PutPeerViewPtr* and *GetPeerViewPtr* provide a means for accessing the ordinal position of another view in the owner object's list of subviews.

The only thing to worry about is loading references to peer views that have not yet been loaded (that is, they come later in the subview list, and therefore later on the stream). Turbo Vision handles this automatically, keeping track of all such forward references and resolving them when all the subviews of the group have been loaded. The part you may need to consider is that peer view references are not valid until the entire *Load* has been completed. Because of this, you should not put any code into *Load* methods that makes use of subviews that depend on their peer subviews, as the results will be unpredictable.

Copying a stream

TStream has a method *CopyFrom(S,Count)*, which copies *Count* bytes from the given stream *S*. *CopyFrom* can be used to copy the entire contents of a stream to another stream. If you repeatedly access a disk-based stream, for example, you may want to copy it to an EMS stream for more rapid access:

```
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

Random-access streams

So far, we have dealt with streams as sequential devices: you *Put* objects at the end of a stream, and *Get* them back in the same order. But Turbo Vision provides more capabilities than that. Specifically, it allows you to treat a stream as a virtual, random-access device. In addition to *Get* and *Put*, which correspond to *Read* and *Write* on a file, streams provide features analogous to a file's *Seek*, *FilePos*, *FileSize*, and *Truncate*.

- The *Seek* procedure of a stream moves the current stream pointer to a specified position (in bytes from the beginning of the stream), just like the standard Turbo Pascal *Seek* procedure.

- The *GetPos* function is the inverse of the *Seek* procedure. It returns a *Longint* with the current position of the stream.
- The *GetSize* function returns the size of the stream in bytes.
- The *Truncate* procedure deletes all data after the current stream position, making the current position the end of the stream.

Resources are discussed in Chapter 18, "Resources."

While these routines are useful, random access streams require you to keep an index, outside the stream, noting the starting position of each object in the stream. A collection is ideal for this purpose, and is, in fact, the means used by Turbo Vision with resource files. If you want to use a random access stream, consider whether using a resource file would do the job for you.

Non-objects on streams

You can write things that are not objects onto streams, but you have to use a somewhat different approach to do it. The standard stream *Get* and *Put* methods require that you load or store an object derived from *TObject*. If you want to create a stream of non-objects, go directly to the lower-level *Read* and *Write* procedures, each of which reads or writes a specified number of bytes onto the stream. This is the same mechanism used by *Get* and *Put* to read and write the data for objects; you're simply bypassing the VMT mechanism provided by *Get* and *Put*.

Designing your own streams

This section summarizes the methods and error-handling capabilities of Turbo Vision streams so that you know what you can use to create new types of streams.

TStream itself is an abstract object that must be extended to create a useful stream type. Most of *TStream*'s methods are abstract and must be implemented in your descendant, and some depend upon *TStream* abstract methods. Basically, only the *Error*, *Get*, and *Put* methods of *TStream* are fully implemented. *GetPos*, *GetSize*, *Read*, *Seek*, *SetPos*, *Truncate*, and *Write* must be overridden. If the descendant object type has a buffer, the *Flush* method should be overridden as well.

Stream error handling

TStream has a method called *Error(Code, Info)*, which is called whenever the stream encounters an error. *Error* simply sets the stream's *Status* field to one of the constants listed in Chapter 19, "Turbo Vision reference," under "*stXXXX* constants."

The *ErrorInfo* field is undefined except when *Status* is *stGetError* or *stPutError*. If *Status* is *stGetError*, the *ErrorInfo* field contains the stream ID number of the unregistered type. If *Status* is *stPutError*, the *ErrorInfo* field contains the VMT offset of the type you tried to put onto the stream. You can override *TStream.Error* to generate any level of error handling, including run-time errors.

Stream versioning

Turbo Vision version 2.0 supports a limited form of stream versioning. Versioning allows applications written with version 2.0 to read objects from streams created with version 1.0. Streams written by version 2.0 applications that include objects that changed between versions are not readable by applications created with version 1.0.

Version flags

Turbo Vision objects that have different fields than their version 1.0 counterparts have the *ofVersion20* bit set in their *Options* field. The *ofVersion20* bit was undefined in version 1.0.

Handling different versions

Versioning is handled transparently by the *Load* constructors of version 2.0 objects. After they call their inherited *Load* constructors, they look for the *ofVersion* bits in the *Options* field just read. Based on the version bits set, *Load* then reads the remainder of the object as it was written, but stores the information internally as a version 2.0 object.

Store methods for version 2.0 objects write only version 2.0 objects.

You can read any standard objects written by version 1.0 Turbo Vision applications with version 2.0 programs without change.

CHAPTER

18

Resources

A resource file is a Turbo Vision object that will save objects handed to it, and can then retrieve them by name. Your application can then retrieve the objects it uses from a resource rather than initializing them. Instead of making your application initialize the objects it uses, you can have a separate program create all the objects and save them to a resource.

The mechanism is really fairly simple: a resource file works like a random-access stream, with objects accessed by *keys*, which are simply unique strings identifying the resources.

Unlike other portions of Turbo Vision, you probably won't need or want to change the resource mechanism. As provided, resources are robust and flexible. You really should only need to learn to use them.

Why use resources?

There are a number of advantages to using a resource file.

Using resources allows you to customize your application without changing the code. For example, the text of dialog boxes, the labels of menu items, and the colors of views can all be altered within a resource, allowing the appearance of your application to change without anyone having to get inside of it.

You can normally save code by putting all your object *Init*s in a separate program. *Init*s often turn out to be fairly complex, containing calculations and other operations that can make the rest of your code simpler. You still have a *Load* in your application for each object, but loads are trivial compared to *Init*s. You can usually expect to save about 8% to 10% of your code size by using a resource.

Using a resource also simplifies maintaining language-specific versions of an application. Your application loads the objects by name, but the language that they display is up to them.

If you want to provide versions of an application with differing capabilities, you can, for example, design two sets of menus, one of which provides access to all capabilities and another which provides access to only a limited set of functions. That way you don't have to rewrite your code at all, and you don't have to worry about accidentally stripping out the wrong part of the code. And you can upgrade the program to full functionality by providing only a new resource, instead of replacing the whole program.

In short, a resource isolates the representation of the objects in your program, and makes it easier for it to change.

What's in a resource?

Before digging into the details of resources, you might want to make sure you're comfortable with streams and collections, because the resource mechanism uses both of them. You can *use* resources without needing to know just how they work, but if you plan to alter them in any way, you need to know what you're getting into.

A *TResourceFile* contains both a sorted string collection and a stream. The strings in the collection are keys to objects in the stream. *TResourceFile* has an *Init* method that takes a stream, and a *Get* method that takes a string and returns an object.

Creating a resource

Creating a resource file is essentially a four-step process. You need to open a stream, initialize a resource file on that stream, store one or more objects with their keys, and close the resource.

The following code creates a simple resource file called MY.TVR containing a single resource: a status line with the key 'Waldo'.

This is RESOURC1.PAS.

```
program Resourc1;

uses Drivers, Objects, Views, App, Menus;

type
  PHaltStream = ^THaltStream;
  THaltStream = object(TBufStream)
    procedure Error(Code, Info: Integer); virtual;
  end;

const cmNewDlg = 1001;
var
  MyRez: TResourceFile;
  MyStrm: PHaltStream;

procedure THaltStream.Error(Code, Info: Integer);
begin
  Writeln('Stream error: ', Code, ' (',Info,')');
  Halt(1);
end;

procedure CreateStatusLine;
var
  R: TRect;
  StatusLine: PStatusLine;
begin
  R.Assign(0, 24, 80, 25);
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
      NewStatusKey('~F3~ Open', kbF3, cmNewDlg,
      NewStatusKey('~F5~ Zoom', kbF5, cmZoom,
      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
      nil)))),
    nil)
  ));
  MyRez.Put(StatusLine, 'Waldo');
  Dispose(StatusLine, Done);
end;
```

```
begin
  MyStrm := New(PHaltStream, Init('MY.TVR', stCreate, 1024));
  MyRez.Init(MyStrm);
  RegisterType(RStatusLine);
  CreateStatusLine;
  MyRez.Done;
end.
```

Reading a resource

Retrieving a resource from a resource file is just as simple as getting an object from a stream: You just call the resource file's *Get* function with the desired resource's key as a parameter. *Get* returns a generic *PObject* pointer.

The status line resource created in the previous example can be retrieved and used by an application in this way:

This is RESOURC2.PAS.

```
program Resourc2;

uses Objects, Drivers, Views, Menus, Dialogs, App;

var
  MyRez: TResourceFile;

type
PMyApp = ^TMyApp;
TMyApp = object(TApplication)
  constructor Init;
  procedure InitStatusLine; virtual;
end;

constructor TMyApp.Init;
const
  MyRezFileName: FNameStr = 'MY.TVR';
begin
  MyRez.Init(New(PBufStream, Init(MyRezFileName, stOpen, 1024)));
  if MyRez.Stream^.Status <> 0 then Halt(1);
  RegisterType(RStatusLine);
  TApplication.Init;
end;

procedure TMyApp.InitStatusLine;
begin
  StatusLine := PStatusLine(MyRez.Get('Waldo'));
end;
```

```
var WaldoApp: TMyApp;
begin
  WaldoApp.Init;
  WaldoApp.Run;
  WaldoApp.Done;
end.
```

When you read an object off a resource, you need to be aware of the possibility of receiving a **nil** pointer. If your index name is invalid (that is, if there is no resource with that key in the file), *Get* returns **nil**. After your resource code is debugged, however, this should no longer be a problem.

You can read an object repeatedly off a resource. It's unlikely that you would want to do so with our example of a status line, but a dialog box, for example, might typically be retrieved many times by a user during the course of an application's running. A resource just repeatedly provides an object when it is requested.

This can potentially produce problems with slow disk I/O, even though the resource file is buffered. You can adjust your disk buffering, or you can copy the stream to an EMS stream using the *SwitchTo* method if you have EMS installed.

String lists

In addition to the standard resource mechanism, Turbo Vision provides a pair of specialized objects that handle string lists. A *string list* is a special resource access object that allows your program to access resourced strings by number (usually represented by an integer constant) instead of a key string. This allows a program to store strings out on a resource file for easy customization and internationalization.

For example, the Turbo Pascal IDE uses a string list object for all its error messages. This means the program can simply call for an error message by number, and different versions in different countries will find different strings in their resources.

The string list object is by design not very flexible, but it is fast and convenient when used as designed.

The *TStringList* object is used to access the strings. To create the string list requires the use of the *TStrListMaker* object. The registration records for both have the same object type number.

The string list object has no *Init* method. The only constructor it has is a *Load* method, because string lists only exist on resource files. Similarly, since the string list is essentially a read-only resource, it has a *Get* function, but no *Put* procedure.

Making string lists

The *TStrListMaker* object type is used to create a string list on a resource file for use with *TStringList*. In contrast to the string list, which is read-only, the string list maker is write-only. Basically, all you can do with a string list maker is initialize a string list, write strings onto it sequentially, and store the resulting list on a stream.

PART

3

Turbo Vision Reference

CHAPTER

19

Turbo Vision reference

This chapter describes all the elements of Turbo Vision, including all the object types, procedures, functions, types, variables, and constants. All items are listed alphabetically.

The purpose of this chapter is not to teach you how to use these items—it is only a reference. To learn to best use each of these elements, consult the appropriate chapters in Part 2, "Using Turbo Vision."

To find information on a specific object, keep in mind that many of the properties of the objects in the hierarchy are inherited from ancestor objects. Rather than duplicate all that information endlessly, this chapter only documents fields and methods that are *new* or *changed* for a particular object. By looking at the inheritance diagram for the object, you can easily determine which of its ancestors introduced a field, and which objects introduce or redefine methods.

Abstract procedure — Objects

Declaration

```
procedure Abstract;
```

Function Calling this procedure terminates the program with run-time error 211. When implementing an abstract object type, call *Abstract* in those virtual methods that must be overridden in descendant types. This ensures that any attempt to use instances of the abstract object type will fail.

See also "Abstract methods" in Chapter 7

Application variable App

Declaration

```
Application: PApplication = nil;
```

Function Throughout the execution of a Turbo Vision program, *Application* points to the application object. The *Init* constructor of *TProgram* sets *Application* to *@Self,* and the *Done* destructor clears it to **nil**. By default, *TApplication*'s constructor calls *TProgram.Init,* so all application objects inherit this behavior.

See also *TProgram.Init*

AppPalette variable App

Declaration

```
AppPalette: Integer = apColor;
```

Function Selects one of the three available application palettes (*apColor, apBlackWhite,* or *apMonochrome*). The *InitScreen* method of *TProgram* sets *AppPalette* depending on the current screen mode. *TProgram*'s *GetPalette* method check's *AppPalette* to determine which of the three available application palettes to return. You can override *TProgram.InitScreen* to change the default palette selection.

See also *TProgram.Getpalette, TProgram.InitScreen, apXXXX* constants

apXXXX constants App

Values The following application palette constants are defined:

Table 19.1 Application palette constants

Constant	Value	Meaning
apColor	0	Use palette for color screen
apBlackWhite	1	Use palette for LCD screen
apMonochrome	2	Use palette for monochrome screen

Function The *apXXXX* constants designate which of three standard color palettes a Turbo Vision application should use for color, black and white, and monochrome displays.

AssignDevice procedure TextView

Declaration

```
procedure AssignDevice(var T: Text; Screen: PTextDevice);
```

Function Associates a text file with a text device. *AssignDevice* works exactly like the *Assign* standard procedure, except that no file name is specified. Instead, the text file is associated with the *TTextDevice* given by *Screen* (by storing *Screen* in the first four bytes of the *UserData* field in *TextRec(T)*).

Subsequent I/O operations on *T* will read from and write to *Screen*, using the *StrRead* and *StrWrite* virtual methods. Since *TTextDevice* is an abstract type, *Screen* should point to an instance of a descendant of *TTextDevice* such as *TTerminal*, which implements a fully functional scrolling view.

See also *TTextDevice*; *TextRec* (in the *Programmer's Reference*)

bfXXXX constants — Dialogs

Values The following button flags are defined:

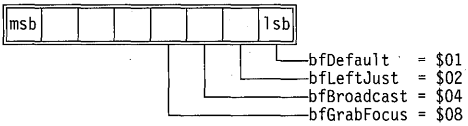


Figure 19.1 Button flags

Table 19.2 Button flags

Constant	Value	Meaning
bfNormal	$00	Button is a normal button
bfDefault	$01	Button is the default button
bfLeftJust	$02	Button label is left-aligned
bfBroadcast	$04	Button notifies its owner when pressed
bfGrabFocus	$08	Button receives input focus when user clicks

Function Button objects have a bitmapped *Flags* field that holds a combination of *bfXXXX* constants that determine the button's style. *bfNormal* indicates a normal, non-default button. *bfDefault* indicates that the button is the default button. It is your responsibility to ensure that there is only one default button in a group. The *bfLeftJust* bit affects the position of the text displayed within the button: If clear, the text is centered; if set, the text is left-aligned.

bfBroadcast controls the way button objects generate events when pressed:

- If *bfBroadcast* is clear (the default setting), the button uses *PutEvent* to generate a command event when pressed:

```
E.What := evCommand;
E.Command := Command;
E.InfoPtr := @Self;
PutEvent(E);
```

- If *bfBroadcast* is set, the button uses *Message* to send a broadcast message to its owner when pressed:

```
Message(Owner, evBroadcast, Command, @Self);
```

Setting *bfGrabFocus* causes the input focus to move to the button when the user clicks it with the mouse. By default, buttons don't take the focus.

See also *TButton.Flags, TButton.MakeDefault, TButton.Draw*

ButtonCount variable — Drivers

Declaration `ButtonCount: Byte = 0;`

Function *ButtonCount* holds the number of buttons on the mouse, or zero if no mouse is installed. You can use this variable to determine whether mouse support is available. The value is set by the initialization code in *Drivers*, and should not be changed.

cdXXXX constants — StdDlg

Values

Constant	Value	Meaning
cdNormal	$0000	Create the dialog box normally, including loading the directory.
cdNoLoadDir	$0001	Initialize the dialog box without loading the directory contents. Used when creating a dialog box to store on a stream.
cdHelpButton	$0002	Put a help button in the dialog box.

Function These constants define the values passed to a change directory dialog box's *Init* constructor in the *AOptions* parameter.

See also *TChDirDialog* object

cfXXXX constants — Dialogs

Values

Constant	Value	Meaning
cfOneBit	$0101	1 bit per checkbox
cfTwoBits	$0203	2 bits per check box
cfFourBits	$040F	4 bits per check box
cfEightBits	$08FF	8 bits per check box

Function Multistate check boxes use the *cfXXXX* constants to specify how many bits in the *Value* field represent the state of each check box. The high-order word of the constant indicates the number of bits used for each check box, and the low-order word holds a bit mask used to read those bits.

For example, *cfTwoBits* indicates that *Value* uses two bits for each check box (making a maximum of 16 check boxes in the cluster), and masks each check box's values with the mask $03.

See also *TMultiCheckBoxes* object

CheckSnow variable — Drivers

Declaration `CheckSnow: Boolean;`

Function *CheckSnow* performs the same function as the flag of the same name in the *Crt* unit. Snow checking is only needed to slow down screen output for some older CGA adapters. *InitVideo* sets *CheckSnow* to *True* only if it detects a CGA adapter. You can set the value to *False* at any time after the *InitVideo* call for faster screen I/O.

See also *InitVideo*

ClearHistory procedure — HistList

Declaration `procedure ClearHistory;`

Function Removes all strings from all history lists.

ClearScreen procedure — Drivers

Declaration `procedure ClearScreen;`

Function Clears the screen. *ClearScreen* assumes that *InitVideo* has been called first. You seldom need to call *ClearScreen*, as explained in the description of *InitVideo*.

See also *InitVideo*

Clipboard variable — Editors

Declaration `Clipboard: PEditor = nil;`

Function *Clipboard* points to an editor object used for transfer of data between other editor objects. Any editor object can serve as the clipboard. The clipboard editor should not support undo (that is, its *CanUndo* field should be *False*).

cmXXXX constants

Function These constants represent Turbo Vision's predefined *commands*. They are passed in the *Command* field of *evMessage* events (*evCommand* and *evBroadcast*), and cause the *HandleEvent* methods of Turbo Vision's standard objects to perform various tasks.

Turbo Vision reserves constant values 0 through 99 and 256 through 999 for its own use. Standard Turbo Vision objects' event handlers respond to these predefined constants. Programmers can define their own constants in the ranges 100 through 255 and 1,000 through 65,535 without conflicting with predefined commands.

Values The following standard commands are defined in the *Views* unit and used by all views:

Table 19.3 Standard command codes

Command	Value	Meaning
cmValid	0	Passed to a view's *Valid* to check the validity of a newly instantiated view.
cmQuit	1	Terminates the application by calling the application object's *EndModal* method, passing *cmQuit*.
cmError	2	Never handled by any object. Can be used to represent unimplemented or unsupported commands.
cmMenu	3	Causes a menu view to call *ExecView* on itself to perform a menu selection process, the result of which might generate a new command through *PutEvent*.
cmClose	4	Closes a window. If the window is modal, a command event with a value of *cmCancel* is generated with *PutEvent*. If the window is modeless, the window's *Close* method is called.
cmZoom	5	Causes a zoomable window to call *Zoom*.
cmResize	6	Causes a resizable window to call *DragView* on itself.
cmNext	7	Selects the next window on the desktop.
cmPrev	8	Selects the previous window on the desktop.

The following standard commands are used to define default behavior of dialog box objects:

Table 19.4
Dialog box standard commands

Command	Value	Meaning
cmOK	10	OK button was pressed.
cmCancel	11	Dialog box was canceled by Cancel button, close icon or *Esc* key.
cmYes	12	Yes button was pressed.
cmNo	13	No button was pressed.
cmDefault	14	Default button was pressed.

An event with one of the commands *cmOK, cmCancel, cmYes,* or *cmNo* causes a modal dialog box to terminate it's modal state (by calling *EndModal*) and return that value. A modal dialog box typically contains at least one button with one of these command values. By default, dialog boxes generate a *cmCancel* command event in response to a *kbEsc* keyboard event.

The *cmDefault* command causes the default button to simulate a button press. By default, dialog boxes generate a *cmDefault* command event in response to a *kbEnter* keyboard event.

The following comands are used for clipboard and window operations, and are generated by the standard Edit and Window menus:

Table 19.5
Standard Edit and Window menu commands

Command	Value	Meaning
cmCut	20	Cut selected text to clipboard
cmCopy	21	Copy selected text to clipboard
cmPaste	22	Paste clipboard text
cmUndo	23	Undo last edit
cmClear	24	Clear selected text
cmTile	25	Tile all tileable windows on desktop
cmCascade	26	Cascade all tileable windows on desktop

See also *StdEditMenuItems* function, *StdWindowMenuItems* function

Function Turbo Vision 2.0 defines new command constants for the items on the standard file menu.

Values The *App* unit defines the following standard application commands:

Table 19.6
Standard application commands

Constant	Value	Meaning
cmNew	30	Open new file, from File | New
cmOpen	31	Open existing file, from File | Open
cmSave	32	Save current file, from File | Save
cmSaveAs	33	Save and rename file, from File | Save As
cmSaveAll	34	Save all open files, from File | Save All
cmChangeDir	35	Change current directory, from File | Change Dir
cmDosShell	36	Shell to DOS, from File | DOS Shell
cmCloseAll	37	Close all open files, from File | Close All

See also *StdFileMenuItems* function

The following standard commands are defined for use by standard views:

Table 19.7 Standard view commands

Command	Value	Meaning
cmReceivedFocus *cmReleasedFocus*	50 51	*TView.SetState* uses the *Message* function to send an *evBroadcast* event with one of these values to its owner whenever *sfFocused* changes. This informs any peer views that the view has received or released focus, and that they should update themselves appropriately. Label objects, for example, respond to these commands by highlighting or unhighlighting themselves when the view they label is focused or unfocused.
cmCommandSetChanged	52	The application's *Idle* method broadcasts an event with this value whenever it detects a change in the current command set. The broadcast goes to every view in the application that accepts broadcast events. Views should react to command set changes by redrawing themselves as needed.
cmScrollBarChanged *cmScrollBarClicked*	53 54	A scroll bar uses the *Message* function to send a broadcast event with one of these values to its owner whenever its value changes or the user clicks the scroll bar. Views connected to the scroll bar, such as scrollers and list viewers, can then react to the broadcast.
cmSelectWindowNum	55	Causes a window to select itself if the *InfoInt* of the event record corresponds to the window's *Number* field. *TProgram*'s *HandleEvent* responds to *Alt+1* through *Alt+9* keyboard events by broadcasting a *cmSelectWindowNum* event with an *InfoInt* of 1 through 9.
cmListItemSelected	56	List viewer objects broadcast events with a *Command* value of *cmListItemSelected* to their owners whenever an item in the list is selected.
cmRecordHistory	60	Causes a history object to record the current contents of the linked input line object. Buttons send these broadcasts to their owners when pressed, causing all history objects in the dialog box to record at that time.

See also *TView.HandleEvent*, *TCommandSet*

Values The following constants are used by *TEditor* objects:

Command	Value	Meaning
cmFind	82	Invoke the text search dialog box
cmReplace	83	Invoke the text search and replace dialog box
cmSearchAgain	84	Repeat the previous text search

TEditor.HandleEvent maps various keystrokes into the following commands:

Command	Value	Command	Value
cmCharLeft	500	*cmNewLine*	512
cmCharRight	501	*cmBackSpace*	513
cmWordLeft	502	*cmDelChar*	514
cmWordRight	503	*cmDelWord*	515
cmLineStart	504	*cmDelStart*	516
cmLineEnd	505	*cmDelEnd*	517
cmLineUp	506	*cmDelLine*	518
cmLineDown	507	*cmInsMode*	519
cmPageUp	508	*cmStartSelect*	520
cmPageDown	509	*cmHideSelect*	521
cmTextStart	510	*cmIndentMode*	522
cmTextEnd	511	*cmUpdateTitle*	523

Values The *StdDlgs* unit defines the following commands for file dialog boxes:

Command	Value	Meaning
cmFileOpen	800	Returned from *TFileDialog* when Open clicked
cmFileReplace	801	Returned from *TFileDialog* when Replace clicked
cmFileClear	802	Returned from *TFileDialog* when Clear clicked

ColorIndexes variable ColorSel

Declaration

```
ColorIndexes: PColorIndex = nil;
```

Function Holds the current state of the application's color selection dialog box, enabling the program to easily save and restore the state for future use.

See also *LoadIndexes* procedure, *StoreIndexes* procedure

ColorGroup function ColorSel

Declaration

```
function ColorGroup(Name: String; Items: PColorItem;
  Next: PColorGroup): PColorGroup;
```

Function Allocates a new group of color items on the heap with the name given by *Name* and the list of color items passed in *Items* and returns a pointer to the group. *Next* points to the next group in a linked list of groups, with **nil** indicating the end of the list.

See also *TColorGroup* type

ColorItem function — ColorSel

Declaration

```
function ColorItem(Name: String; Index: Byte; Next: PColorItem): PColorItem;
```

Allocates a new color item on the heap with the name given by *Name* and the color index given by *Index*. *Next* points to the next color item in a linked list, with **nil** indicating the end of the list.

coXXXX constants — Objects

Function The *coXXXX* constants are passed as the *Code* parameter to *TCollection.Error* when a collection detects an error during an operation.

Values The following standard error codes are defined for all collections:

Table 19.8 Collection error codes

Error code	Value	Meaning
coIndexError	–1	Index out of range. The *Info* parameter passed to the *Error* method contains the invalid index.
coOverflow	–2	Collection overflow. *TCollection.SetLimit* failed to expand the collection to the requested size. The *Info* parameter passed to the *Error* method contains the requested size.

See also *TCollection*

CStrLen function — Drivers

Declaration

```
function CStrLen(S: String): Integer;
```

Function Returns the length of string *S*, where *S* is a control string using tilde characters ('~') to designate shortcut characters. The tildes are excluded from the length of the string, as they will not appear on the screen. For example, given the string '~B~roccoli' as its parameter, *CStrLen* returns 8.

See also *MoveCStr*

CtrlBreakHit variable — Drivers

Declaration `CtrlBreakHit: Boolean = False;`

Function Set *True* by the Turbo Vision keyboard interrupt driver whenever *Ctrl+Break* is pressed. This allows Turbo Vision applications to trap and respond to *Ctrl+Break* as a user control. You can clear the flag at any time by setting it to *False*.

See also *SaveCtrlBreak*

CtrlToArrow function — Drivers

Declaration `function CtrlToArrow(KeyCode: Word): Word;`

Function Converts a WordStar-compatible control key code to the corresponding cursor key code. If the low byte of *KeyCode* matches one of the control key values in Table 19.9, the result is the corresponding *kbXXXX* constant. Otherwise, *KeyCode* is returned unchanged.

Table 19.9 Control-key mappings

Keystroke	Lo(KeyCode)	Result
Ctrl+A	$01	*kbHome*
Ctrl+C	$03	*kbPgDn*
Ctrl+D	$04	*kbRight*
Ctrl+E	$05	*kbUp*
Ctrl+F	$06	*kbEnd*
Ctrl+G	$07	*kbDel*
Ctrl+H	$08	*kbBack*
Ctrl+R	$12	*kbPgUp*
Ctrl+S	$13	*kbLeft*
Ctrl+V	$16	*kbIns*
Ctrl+X	$18	*kbDown*

CursorLines variable — Drivers

Declaration `CursorLines: Word;`

Function Set to the starting and ending scan lines of the cursor by *InitVideo*. The format is that expected by BIOS interrupt $10, function 1 to set the cursor type.

See also *InitVideo, TView.ShowCursor, TView.HideCursor, TView.BlockCursor, TView.NormalCursor*

DefEditorDialog function Editors

Declaration

```
function DefEditorDialog(Dialog: Integer; Info: Pointer): Word;
```

Function *DefEditorDialog* is the default value assigned to the *EditorDialog* variable. For a description of the general use of editor dialog functions, see the entry for the *TEditorDialog* type. *DefEditorDialog* shows no dialog boxes at all, and simply returns the value *cmCancel*, as if any dialog it was called to show had been canceled.

See also *TEditorDialog* type, *EditorDialog* variable

Desktop variable App

Declaration

```
Desktop: PDesktop = nil;
```

Function Stores a pointer to the application's desktop object. Application objects use the virtual method *InitDesktop*, called by the application's *Init* constructor, to construct a desktop object and assign a pointer to it to *Desktop*. To change the default desktop, override *InitDesktop* in your application object to construct a different kind of desktop object and assign it to *Desktop*.

See also *TProgram.InitDesktop*

DesktopColorItems function ColorSel

Declaration

```
function DesktopColorItems(const Next: PColorItem): PColorItem;
```

Function Returns a linked list of *TColorItem* records for the standard desktop object. For programs that allow the user to change desktop colors with the color selection dialog box, *DesktopColorItems* simplifies the process of setting up the color items.

DialogColorItems function ColorSel

Declaration

```
function DialogColorItems(Palette: Word; const Next: PColorItem): PColorItem;
```

Function Returns a linked list of *TColorItem* records for the standard dialog box object. For programs that allow the user to change dialog box colors with the color selection dialog box, *DesktopColorItems* simplifies the process of setting up the color items.

DisposeBuffer procedure Memory

Declaration `procedure DisposeBuffer(P: Pointer);`

Function Disposes of the buffer *P*^. *P* must be a buffer allocated by *NewBuffer*.

See also *NewBuffer* procedure

DisposeCache procedure Memory

Declaration `procedure DisposeCache(P: Pointer);`

Function Disposes of the cache buffer *P*^. *P* must be a cache buffer allocated by *NewCache*.

See also *NewCache* procedure

DisposeMenu procedure Menus

Declaration `procedure DisposeMenu(Menu: PMenu);`

Function Disposes of all the elements of the specified menu (and all its submenus).

See also *TMenu* type

DisposeNode procedure Outline

Declaration `procedure DisposeNode(Node: PNode);`

Function Disposes of an outline node created by *NewNode*, including recursively disposing of any child nodes.

See also *NewNode* function

DisposeStr procedure — Objects

Declaration

```
procedure DisposeStr(P: PString);
```

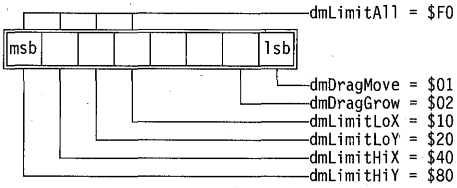
Disposes of a string allocated on the heap by the *NewStr* function.

See also *NewStr*

dmXXXX constants — Views

Values The *DragMode* bits are defined as follows:

Figure 19.2 Drag mode bit flags



Function Drag mode constants serve two purposes. Constants beginning with *dmLimit* are used in a view's *DragMode* field to indicate which parts, if any, of a view should not move outside the owner view when dragged. Those beginning with *dmDrag* specify how the view responds to dragging: by moving or by growing.

DragMode and the drag mode constants combine to form the *Mode* parameter of the *TView.DragView* method. Normally, you combine either *dmDragGrow* or *dmDragMove* with *DragMode* and pass the result in *Mode*. The example program DRAGS.PAS illustrates how changing the drag mode flags affects a view when dragged.

The drag mode constants are defined as follows:

Table 19.10 Drag mode constants

Constant	Meaning
dmDragMove	Move the view when dragged.
dmDragGrow	Change the size of the view when dragged.
dmLimitLoX	The view's left-hand side cannot move outside *Limits*.
dmLimitLoY	The view's top side cannot move outside *Limits*.
dmLimitHiX	The view's right-hand side cannot move outside *Limits*.
dmLimitHiY	The view's bottom side cannot move outside *Limits*.
dmLimitAll	No part of the view can move outside *Limits*.

A view's *DragMode* field contains any combination of the *dmLimitXX* flags. By default, *TView.Init* sets the field to *dmLimitLoY*. Currently, the *DragMode* field is used only in a *TWindow* to construct the *Mode* parameter to *DragView* when a window is moved or resized.

DoneDosMem procedure Memory

Declaration `procedure DoneDosMem;`

Function Frees up memory for DOS shells or execution of another program. *DoneDosMem* releases all cache buffers, then calls *SetMemTop* to the end of the last item on the heap, making th rest of the heap available. When the shell or subprogram returns, you need to call *InitDosMem* to restore the full heap to your application. For an example of the use of *InitDosMem* and *DoneDosMem*, see the implementation of *TApplication.DosShell* in APP.PAS.

See also *InitDosMem* procedure, *SetMemTop* procedure

DoneEvents procedure Drivers

Declaration `procedure DoneEvents;`

Function Terminates Turbo Vision's event manager by disabling the mouse interrupt handler and hiding the mouse. Called by *TApplication.Done.*

See also *TApplication.Done, InitEvents*

DoneHistory procedure HistList

Declaration `procedure DoneHistory;`

Function Frees the history block allocated by *InitHistory*. Called by *TApplication.Done.*

See also *InitHistory* procedure, *TApplication.Done*

DoneMemory procedure Memory

Declaration `procedure DoneMemory;`

Function Terminates Turbo Vision's memory manager by freeing all buffers allocated through *GetBufMem*. Called by *TApplication.Done.*

See also *TApplication.Done, InitMemory*

DoneSysError procedure — Drivers

Declaration `procedure DoneSysError;`

Function Terminates Turbo Vision's system error handler by restoring interrupt vectors 09H, 1BH, 21H, 23H, and 24H and restoring the *Ctrl+Break* state in DOS. Called by *TApplication.Done.*

See also *TApplication.Done, InitSysError*

DoneVideo procedure — Drivers

Declaration `procedure DoneVideo;`

Function Terminates Turbo Vision's video manager by restoring the initial screen mode (given by *StartupMode*), clearing the screen, and restoring the cursor. Called by *TApplication.Done.*

See also *TApplication.Done, InitVideo, StartupMode* variable

DoubleDelay variable — Drivers

Declaration `DoubleDelay: Word = 8;`

Function Defines the time interval (in 1/18.2 parts of a second) between mouse-button presses in order to distinguish a double click from two distinct clicks. Used by *GetMouseEvent* to generate a *Double* event if the clicks occur within this time interval.

See also *TEvent.Double, GetMouseEvent*

dpXXXX constants — Dialogs

Values

Constant	Value	Meaning
dpBlueDialog	1	Dialog box background is blue
dpCyanDialog	2	Dialog box background is cyan
dpGrayDialog	3	Dialog box background is gray

Function Dialog box objects use the *dpXXXX* constants to specify which of the three standard color palettes to use. By default, dialog box objects use *dpGrayDialog*. You can choose one of the other standard palettes by setting the dialog box's *Palette* field to one of the other *dpXXXX* constants after constructing the dialog box object.

EditorDialog variable — Editors

Declaration

```
EditorDialog: TEditorDialog = DefEditorDialog;
```

Function *EditorDialog* is a global procedural variable. It holds the editor dialog function defined for all editors in the application. By default, *EditorDialog* holds the function *DefEditorDialog*, which bypasses the display of the dialog boxes and returns *cmCancel*.

Turbo Vision also provides a usable set of editor dialog boxes through the *StdEditorDialog* function.

See also *StdEditorDialog* function

EditorFlags variable — Editors

Declaration

```
EditorFlags: Word = efBackupFiles + efPromptOnReplace;
```

Function *EditorFlags* is a bitmapped global variable that controls the behavior of editor objects throughout the application. The bits are defined by the *efXXXX* constants. By default, *EditorFlags* causes file editors to save backup versions of edited files and causes search-and-replace operations to prompt before replacing text.

See also *efXXXX* constants

edXXXX constants — Editors

Function Editor objects pass these constants to the *EditorDialog* function to specify which of several possible dialog boxes the function should display. The standard editor dialog boxes provided by *StdEditorDialog* respond to all of these. You should only need to use these constants if you write your own editor dialog boxes.

Values

Constant	Value	Meaning
edOutOfMemory	0	Display an "out of memory" warning.
edReadError	1	Error reading a file.
edWriteError	2	Error writing a file.
edCreateError	3	Could not create a file.
edSaveModify	4	File being closed has unsaved changes.
edSaveUntitled	5	Untitled file being closed; ask to save changes.
edSaveAs	6	Saving file with new name or saving for first time.
edFind	7	Prompt user for text to find.
edSearchFailed	8	Tell user that search string not found.
edReplace	9	Prompt user for text to search for and replace with.
edReplacePrompt	10	Ask whether to replace the located search text.

See also *EditorDialog* variable, *TEditorDialog* type

efXXXX constants

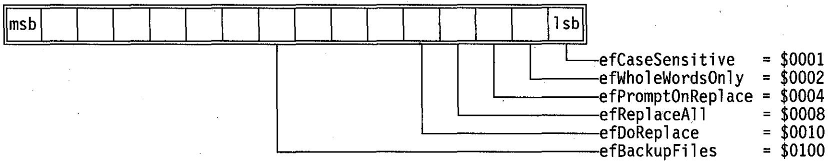
Editors

Function Editor flag constants are used to control the bitmapped global variable *EditorFlags*. Most of the flags affect the way search and replace operations behave, but one flag determines whether file editors create backup files.

Values

Constant	Value	Meaning
efCaseSensitive	$0001	Treat uppercase and lowercase letters differently.
efWholeWordsOnly	$0002	Search only for whole words (separated by spaces, tabs, or end-of-line).
efPromptOnReplace	$0004	Prompt the user before replacing text.
efReplaceAll	$0008	Search for and replace all instances of the search text.
efDoReplace	$0010	Replace search text if found. Used internally by *TEditor*.
efBackupFiles	$0100	Make backup copies of edited files, using the extension .BAK.

Figure 19.3 Editor flag bit mapping



EmsCurHandle variable Objects

Declaration `EmsCurHandle: Word = $FFFF;`

Function Holds the current EMS handle as mapped into EMS physical page 0 by a *TEmsStream*. *TEmsStream* avoids costly EMS remapping calls by caching the state of EMS. If your program uses EMS for other purposes, be sure to set *EmsCurHandle* and *EmsCurPage* to $FFFF before using a *TEmsStream*—this will force the *TEmsStream* to restore its mapping.

See also *TEmsStream.Handle*

EmsCurPage variable Objects

Declaration `EmsCurPage: Word = $FFFF;`

Function Holds the current EMS logical page number as mapped into EMS physical page 0 by a *TEmsStream*. *TEmsStream* avoids costly EMS remapping calls by caching the state of EMS. If your program uses EMS for other purposes, be sure to set *EmsCurHandle* and *EmsCurPage* to $FFFF before using a *TEmsStream*—this will force the *TEmsStream* to restore its mapping.

See also *TEmsStream.Page*

ErrorAttr variable Views

Declaration `const ErrorAttr: Byte = $CF;`

Function Contains a video attribute byte used as the error return value of a call to a view's *GetColor* method. If *GetColor* fails to correctly map a palette index into a video attribute byte (because of an out-of-range index), it returns the value in *ErrorAttr*.

The default *ErrorAttr* value represents blinking high-intensity white characters on a red background. If you see this color combination on the screen, it probably indicates a palette mapping error.

See also *TView.GetColor*

evXXXX constants — Drivers

Function These mnemonics indicate types of events to Turbo Vision event handlers. *evXXXX* constants appear in several places:

- In the *What* field of an event record
- In the *EventMask* field of a view object
- In the *PositionalEvents* and *FocusedEvents* variables

Values The following event flag values designate standard event types:

Table 19.11 Standard event flags

Constant	Value	Meaning
evMouseDown	$0001	Mouse button depressed
evMouseUp	$0002	Mouse button released
evMouseMove	$0004	Mouse changed location
evMouseAuto	$0008	Periodic event while mouse button held down
evKeyDown	$0010	Key pressed
evCommand	$0100	Command event
evBroadcast	$0200	Broadcast event

The following constants mask types of events:

Table 19.12 Standard event masks

Constant	Value	Meaning
evNothing	$0000	Event already handled
evMouse	$000F	Mouse event
evKeyboard	$0010	Keyboard event
evMessage	$FF00	Message (command, broadcast, or user-defined) event

The event mask bits are defined as follows:

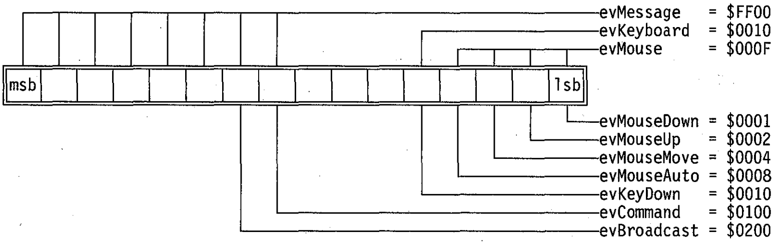


Figure 19.4 Event mask bit mapping

The standard event masks can be used to quickly determine whether an event belongs to a particular "family" of events. For example,

```
if Event.What and evMouse <> 0 then DoMouseEvent(Event);
```

See also *TEvent*, *TView.EventMask*, *GetKeyEvent*, *GetMouseEvent*, *HandleEvent* methods, *PositionalEvents*, *FocusedEvents*

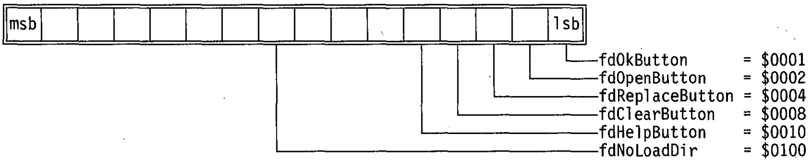
fdXXXX constants StdDlg

Function The *fdXXXX* constants are passed in the *AOptions* parameter to the constructor of *TFileDialog* objects.

Values

Constant	Value	Meaning
fdOkButton	$0001	Put an OK button in the dialog.
fdOpenButton	$0002	Put an Open button in the dialog.
fdReplaceButton	$0004	Put a Replace button in the dialog.
fdClearButton	$0008	Put a Clear button in the dialog.
fdHelpButton	$0010	Put a Help button in the dialog.
fdNoLoadDir	$0100	Do not load the current directory contents into the dialog at *Init*. This means you intend to change the *WildCard* by using *SetData* or store the dialog on a stream.

Figure 19.5 File dialog box option flags



See also *TFileDialog*

FindStr variable Editors

Declaration `FindStr: string[80] = '';`

Function *FindStr* holds the last string searched for in a search operation.

FNameStr type Objects

Declaration `FNameStr = string[79];`

Function DOS file name string

FocusedEvents variable Views

Declaration `FocusedEvents: Word = evKeyboard + evCommand;`

Function Defines the event classes that are focused events. The *FocusedEvents* and *PositionalEvents* variables are used by *TGroup.HandleEvent* to determine how to dispatch events to the group's subviews. If an event class isn't contained in *FocusedEvents* or *PositionalEvents*, it's treated as a broadcast event.

See also *PositionalEvents* variable, *TGroup.HandleEvent*, *TEvent*, *evXXXX* constants

FormatStr procedure — Drivers

Declaration

```
procedure FormatStr(var Result: String; const Format: String; var Params);
```

Function A generalized string formatting routine that works much like the C language's **vsprintf** function. Given a string in *Format* that includes format specifiers and a list of parameters in *Params*, *FormatStr* produces a formatted output string in *Result*.

The *Format* parameter can contain any number of format specifiers directing what format to use to display the parameters in *Params*. Format specifiers are of the form `%[-][nnn]X`, where

- `%` indicates the beginning of a format specifier.
- `[-]` is an optional minus sign (-) indicating the parameter is to be left-aligned (by default, parameters are displayed right-justified).
- `[nnn]` is an optional, decimal-number width specifier in the range 0..255 (0 indicates no width specified, and non-zero means to display in a field of nnn characters).
- `X` is a format character:
  - 's' means the parameter is a pointer to a string.
  - 'd' means the parameter is a Longint to be displayed in decimal.
  - 'c' means the low byte of the parameter is a character.
  - 'x' means the parameter is a Longint to be displayed in hexadecimal.
  - '#' sets the parameter index to `nnn`.

For example, if the parameter points to a string containing `'spiny'` for printing, the following table shows specifiers and their results:

Table 19.13 Format specifiers and their results

Specifier	Result
%6s	`' spiny'`
%-6s	`'spiny '`
%3s	`'iny'`
%-3s	`'spi'`
%06s	`'0spiny'`
%-06s	`'spiny0'`

Params is an untyped **var** parameter containing enough parameters to match each of the format specifiers in *Format*. *Params* must be a zero-based array of *Longint*s or pointers or a record containing *Longint*s or pointers.

For example, to print the error message string `Error in file [file name] at line [line number]`, you could pass the following string in *Format*: 'Error in file %s at line %d'. *Params*, then, needs to contain a pointer to a string with the file name and a *Longint* representing the line number in the file. This could be specifed in an array or in a record.

The following example shows two type declarations and variable assignments that both produce acceptable values to be passed as *Params* to *FormatStr*:

```
type
  ErrMsgRec = record
    FileName: PString;
    LineNo: Longint;
  end;

  ErrMsgArray = array[0..1] of Longint;

const TemplateMsg = 'Error in file %s at line %d';

var
  MyFileName: FNameStr;
  OopsRec: ErrMsgRec;
  DarnArray: ErrMsgArray;
  TestStr: String;

begin
  MyFileName := 'WARTHOG.ASM';
  with OopsRec do
  begin
    FileName := @MyFileName;
    LineNo := 42;
  end;
  FormatStr(TestStr, TemplateMsg, OopsRec);
  Writeln(TestStr);
  DarnArray[0] := Longint(@MyFileName);
  DarnArray[1] := 24;
  FormatStr(TestStr, TemplateMsg, DarnArray);
  Writeln(TestStr);
end.
```

See also *SystemError* function, *TParamText* object

FreeBufMem procedure — Memory

Declaration `procedure FreeBufMem(P: Pointer);`

Function Frees the cache buffer referenced by the pointer *P* by calling *DisposeCache*. *FreeBufMem* is provided for compatibility with earlier versions of Turbo Vision; you should use *DisposeCache* directly instead.

See also *DisposeCache* procedure

GetAltChar function — Drivers

Declaration `function GetAltChar(KeyCode: Word): Char;`

Function Returns the character, *Ch*, for which *Alt-Ch* produces the 2-byte scan code given by the argument *KeyCode*. Gives the reverse mapping to *GetAltCode*.

See also *GetAltCode*

GetAltCode function — Drivers

Declaration `function GetAltCode(Ch: Char): Word;`

Function Returns the 2-byte scan code (keycode) corresponding to *Alt-Ch*. This function gives the reverse mapping to *GetAltChar*.

See also *GetAltChar*

GetBufferSize function — Memory

Declaration `function GetBufferSize(P: Pointer): Word;`

Function Returns the size in bytes of the buffer *P^*. *P* must point to a buffer allocated by *NewBuffer*.

See also *NewBuffer* procedure

GetBufMem procedure — Memory

Declaration `procedure GetBufMem(var P: Pointer; Size: Word);`

Function Allocates a cache buffer of *Size* bytes and stores a pointer to the buffer in *P* by calling *NewCache*. *GetBufMem* is provided for compatibility with earlier versions of Turbo Vision. You should call *NewCache* directly instead.

See also *NewCache* procedure

GetKeyEvent procedure — Drivers

Declaration

```
procedure GetKeyEvent(var Event: TEvent);
```

Function Checks whether a keyboard event is available by calling the BIOS INT 16H service. If a key has been pressed, *Event.What* is set to *evKeyDown* and *Event.KeyCode* is set to the scan code of the key. Otherwise, *Event.What* is set to *evNothing*. *GetKeyEvent* is called by *TProgram.GetEvent*.

See also *TProgram.GetEvent*, *evXXXX* constants, *TView.HandleEvent*

GetMouseEvent procedure — Drivers

Declaration

```
procedure GetMouseEvent(var Event: TEvent);
```

Function Checks whether a mouse event is available by polling the mouse event queue maintained by Turbo Vision's event handler. If a mouse event has occurred, *Event.What* is set to *evMouseDown*, *evMouseUp*, *evMouseMove*, or *evMouseAuto*; *Event.Buttons* is set to *mbLeftButton* or *mbRightButton*; *Event.Double* is set to *True* or *False*; and *Event.Where* is set to the mouse position in global coordinates (corresponding to *TApplication*'s coordinate system). If no mouse events are available, *Event.What* is set to *evNothing*. *GetMouseEvent* is called by *TProgram.GetEvent*.

See also *TProgram.GetEvent*, *evXXXX* events, *HandleEvent* methods

gfXXXX constants — Views

Function These mnemonics are used to set the *GrowMode* field in all *TView* and derived objects. The bits set in *GrowMode* determine how the view will grow in relation to changes in its owner's size.

Values The *GrowMode* bits are defined as follows:

Figure 19.6
Grow mode bit mapping

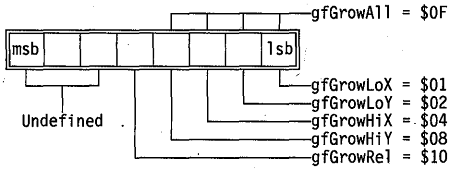


Table 19.14
Grow mode flag definitions

Constant	Meaning if set
gfGrowLoX	The left-hand side of the view maintains a constant distance from its owner's right-hand side.
gfGrowLoY	The top of the view maintains a constant distance from the bottom of its owner.
gfGrowHiX	The right-hand side of the view maintains a constant distance from its owner's right side.
gfGrowHiY	The bottom of the view maintains a constant distance from the bottom of its owner's.
gfGrowAll	The view moves with the lower-right corner of its owner.
gfGrowRel	When used with window objects in the desktop, the view changes size relative to the owner's size. The window maintains its relative size with respect to the owner even when switching between 25 and 43/50 line modes.

Note that *LoX* = left side; *LoY* = top side; *HiX* = right side; *HiY* = bottom side.

See also *TView.GrowMode*

hcXXXX constants App

Function The menu items defined by the standard menu item functions *StdFileMenuItems*, *StdEditMenuItems*, and *StdWindowMenuItems* assign help contexts for each item. The *App* unit defines constants beginning with *hc* for each standard menu item.

☞ Turbo Vision reserves help context ranges 0..999 and $FF00..$FFFF.

Values The *App* unit defines three sets of help contexts, for the standard items on the File, Edit, and Window menus. The following tables show the meaning of each.

Table 19.15
Standard File menu item help contexts

Constant	Value	Meaning
hcNew	$FF01	File | New
hcOpen	$FF02	File | Open
hcSave	$FF03	File | Save
hcSaveAs	$FF04	File | Save As
hcSaveAll	$FF05	File | Save All

Table 19.15: Standard File menu item help contexts (continued)

hcChangeDir	$FF06	File | Change Dir
hcDosShell	$FF07	File | DOS Shell
hcExit	$FF08	File | Exit

Table 19.16 Standard Edit menu item help contexts

Constant	Value	Meaning
hcUndo	$FF10	Edit | Undo
hcCut	$FF11	Edit | Cut
hcCopy	$FF12	Edit | Copy
hcPaste	$FF13	Edit | Paste
hcClear	$FF14	Edit | Clear

Table 19.17 Standard Window menu item help contexts

Constant	Value	Meaning
hcTile	$FF20	Window | Tile
hcCascade	$FF21	Window | Cascade
hcCloseAll	$FF22	Window | Close All
hcResize	$FF23	Window | Resize
hcZoom	$FF24	Window | Zoom
hcNext	$FF25	Window | Next
hcPrev	$FF26	Window | Prev
hcClose	$FF27	Window | Close

hcXXXX constants Views

Values The following help context constants are defined:

Table 19.18 Help context constants

Constant	Value	Meaning
hcNoContext	0	No context specified
hcDragging	1	Object is being dragged

Function The default value of *TView.HelpCtx* is *hcNoContext*, which indicates that there is no help context for the view. *TView.GetHelpCtx* returns *hcDragging* whenever the view is being dragged (as indicated by the *sfDragging* state flag).

Turbo Vision reserves help context values 0..999 and $FF00..$FFFF for its own use. You can define your own constants in the range 1,000..65,280.

See also *TView.HelpCtx, TStatusLine.Update*

HideMouse procedure Drivers

Declaration `procedure HideMouse;`

Function The mouse cursor is initially visible after the call to *InitEvents*. *HideMouse* hides the mouse and increments the internal "hide counter" in the mouse driver. *ShowMouse* decrements this counter, and shows the mouse cursor when the counter becomes zero. Thus, calls to *HideMouse* and *ShowMouse* can be nested but must also always be balanced.

See also *InitEvents, DoneEvents, ShowMouse*

HiResScreen variable Drivers

Declaration `HiResScreen: Boolean;`

Function Set to *True* by *InitVideo* if the screen supports 43- or 50-line mode (EGA or VGA); otherwise, set to *False*.

See also *InitVideo*

HistoryAdd procedure HistList

Declaration `procedure HistoryAdd(Id: Byte; const Str: String);`

Function Adds the string *Str* to the history list indicated by *Id*.

See also *HistoryStr* function, *HistoryCount* function

HistoryBlock variable HistList

Declaration `HistoryBlock: Pointer = nil;`

Function Points to a buffer called the history block used to store history strings. The size of the block is defined by *HistorySize*. The pointer is **nil** until set by *InitHistory*, and its value should not be altered.

See also *InitHistory* procedure, *HistorySize* variable

HistoryCount function HistList

Declaration `function HistoryCount(Id: Byte): Word;`

Function Returns the number of strings in the history list with ID number *Id*.

See also *HistoryAdd* procedure, *HistoryStr* function

HistorySize variable HistList

Declaration

```
HistorySize: Word = 1024;
```

Function Specifies the size of the history block used by the history list manager to store values entered into input lines. The size is fixed by *InitHistory* at program startup. The default size of the block is 1K but can be changed *before InitHistory* is called. The value should not be changed after the call to *InitHistory*.

See also *InitHistory* procedure, *HistoryBlock* variable

HistoryStr function HistList

Declaration

```
function HistoryStr(Id: Byte; Index: Integer): String;
```

Function Returns the *Index*th string in the history list with ID number *Id*.

See also *HistoryAdd* procedure, *HistoryCount* function

HistoryUsed variable HistList

Declaration

```
HistoryUsed: Word = 0;
```

Function Used internally by the history list manager to point to an offset within the history block. The value should not be changed.

InitDosMem procedure Memory

Declaration

```
procedure InitDosMem;
```

Function Reclaims all available heap space for the application following shelling to DOS or executing another program by calling *SetMemTop* to place the end of the heap at the top of available memory. For an example of the use of *InitDosMem* and *DoneDosMem*, see the implementation of *TApplication.DosShell* in APP.PAS.

See also *DoneDosMem* procedure, *SetMemTop* procedure

InitEvents procedure — Drivers

Declaration `procedure InitEvents;`

Function Initializes Turbo Vision's event manager by enabling the mouse interrupt handler and showing the mouse. Called by *TApplication.Init*.

See also *DoneEvents*

InitHistory procedure — HistList

Declaration `procedure InitHistory;`

Function Called by *TApplication.Init* to allocate a block of memory on the heap for use by the history list manager. The size of the block is determined by the *HistorySize* variable. After *InitHistory* is called, the *HistoryBlock* variable points to the beginning of the block.

See also *TProgram.Init*, *DoneHistory* procedure

InitMemory procedure — Memory

Declaration `procedure InitMemory;`

Function Initializes Turbo Vision's memory manager by installing a heap notification function in *HeapError*. Called by *TApplication.Init*.

See also *DoneMemory*

InitSysError procedure — Drivers

Declaration `procedure InitSysError;`

Function Initializes Turbo Vision's system error handler by capturing interrupt vectors 09H, 1BH, 21H, 23H, and 24H and clearing the *Ctrl+Break* state in DOS. Called by *TApplication.Init*.

See also *DoneSysError*

InitVideo procedure — Drivers

Declaration

```
procedure InitVideo;
```

Function Initializes Turbo Vision's video manager. Saves the current screen mode in *StartupMode* and switches the screen to the mode indicated by *ScreenMode.* The *ScreenWidth, ScreenHeight, HiResScreen, CheckSnow, ScreenBuffer,* and *CursorLines* variables are updated accordingly. The screen mode can later be changed using *SetVideoMode. InitVideo* is called by *TApplication.Init.*

See also *DoneVideo, SetVideoMode, smXXXX*

InputBox function — MsgBox

Declaration

```
function InputBox(const Title, ALabel: String; var S: String;
  Limit: Byte): Word;
```

Function Displays a 60-column, 8-line dialog box with the title specified in *Title,* a text label given by *ALabel,* an OK button, a Cancel button, and a single input line that initially contains the string passed in *S.* Returns the value returned by *ExecView* when it finishes executing the dialog box. If the user does not cancel the dialog box, *S* contains the string typed by the user. *Limit* is the maximum number of characters in the input line string.

InputBoxRect function — MsgBox

Declaration

```
function InputBoxRect(var Bounds: TRect; const Title, ALabel: String;
  var S: String; Limit: Byte): Word;
```

Function Works exactly like *InputBox,* but allows you to specify a bounding rectangle for the dialog box.

See also *InputBox* function

kbXXXX constants — Drivers

Function There are two sets of constants beginning with "kb," associated with the keyboard.

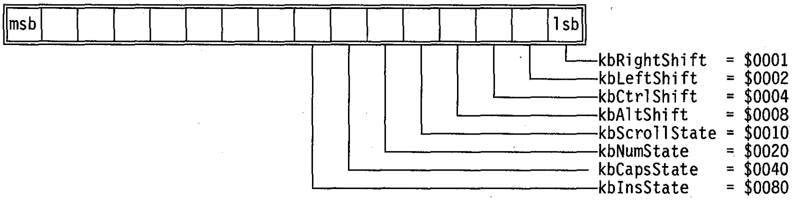
Values The following values define keyboard states and can be used when examining the keyboard shift state which is stored in a byte at absolute address Seg0040:$17. For example,

```
var
  ShiftState: Byte absolute Seg0040:$17;
    ⋮
if ShiftState and kbAltShift <> 0 then AltKeyDown;
```

Table 19.19 Keyboard state and shift masks

Constant	Value	Meaning if set
kbRightShift	$0001	The right *Shift* key is currently down
kbLeftShift	$0002	The left *Shift* key is currently down
kbCtrlShift	$0004	The *Ctrl* key is currently down
kbAltShift	$0008	The *Alt* key is currently down
kbScrollState	$0010	The keyboard is in the *Scroll Lock* state
kbNumState	$0020	The keyboard is in the *Num Lock* state
kbCapsState	$0040	The keyboard is in the *Caps Lock* state
kbInsState	$0080	The keyboard is in the *Ins Lock* state

Figure 19.7 Keyboard state mask flags



The following values define keyboard scan codes and can be used when examining the *TEvent.KeyCode* field of an *evKeyDown* event record:

Table 19.20 Alt-letter key codes

Constant	Value	Constant	Value
kbAltA	$1E00	*kbAltN*	$3100
kbAltB	$3000	*kbAltO*	$1800
kbAltC	$2E00	*kbAltP*	$1900
kbAltD	$2000	*kbAltQ*	$1000
kbAltE	$1200	*kbAltR*	$1300
kbAltF	$2100	*kbAltS*	$1F00
kbAltG	$2200	*kbAltT*	$1400
kbAltH	$2300	*kbAltU*	$1600
kbAltI	$1700	*kbAltV*	$2F00
kbAltJ	$2400	*kbAltW*	$1100
kbAltK	$2500	*kbAltX*	$2D00
kbAltL	$2600	*kbAltY*	$1500
kbAltM	$3200	*kbAltZ*	$2C00

Table 19.21 Special key codes

Constant	Value	Constant	Value
kbAltEqual	$8300	*kbBack*	$0E08
kbAltMinus	$8200	*kbCtrlBack*	$0E7F
kbAltSpace	$0200	*kbCtrlDel*	$0600

Table 19.21: Special key codes (continued)

kbCtrlEnd	$7500	*kbGrayMinus*	$4A2D
kbCtrlEnter	$1C0A	*kbGrayPlus*	$4E2B
kbCtrlHome	$7700	*kbHome*	$4700
kbCtrlIns	$0400	*kbIns*	$5200
kbCtrlLeft	$7300	*kbLeft*	$4B00
kbCtrlPgDn	$7600	*kbNoKey*	$0000
kbCtrlPgUp	$8400	*kbPgDn*	$5100
kbCtrlPrtSc	$7200	*kbPgUp*	$4900
kbCtrlRight	$7400	*kbRight*	$4D00
kbDel	$5300	*kbShiftDel*	$0700
kbDown	$5000	*kbShiftIns*	$0500
kbEnd	$4F00	*kbShiftTab*	$0F00
kbEnter	$1C0D	*kbTab*	$0F09
kbEsc	$011B	*kbUp*	$4800

Table 19.22
Alt-number key codes

Constant	Value	Constant	Value
kbAlt1	$7800	*kbAlt6*	$7D00
kbAlt2	$7900	*kbAlt7*	$7E00
kbAlt3	$7A00	*kbAlt8*	$7F00
kbAlt4	$7B00	*kbAlt9*	$8000
kbAlt5	$7C00	*kbAlt0*	$8100

Table 19.23
Function key codes

Constant	Value	Constant	Value
kbF1	$3B00	*kbF6*	$4000
kbF2	$3C00	*kbF7*	$4100
kbF3	$3D00	*kbF8*	$4200
kbF4	$3E00	*kbF9*	$4300
kbF5	$3F00	*kbF10*	$4400

Table 19.24
Shift-function key codes

Constant	Value	Constant	Value
kbShiftF1	$5400	*kbShiftF6*	$5900
kbShiftF2	$5500	*kbShiftF7*	$5A00
kbShiftF3	$5600	*kbShiftF8*	$5B00
kbShiftF4	$5700	*kbShiftF9*	$5C00
kbShiftF5	$5800	*kbShiftF10*	$5D00

Table 19.25
Ctrl+function key codes

Constant	Value	Constant	Value
kbCtrlF1	$5E00	*kbCtrlF6*	$6300
kbCtrlF2	$5F00	*kbCtrlF7*	$6400
kbCtrlF3	$6000	*kbCtrlF8*	$6500
kbCtrlF4	$6100	*kbCtrlF9*	$6600
kbCtrlF5	$6200	*kbCtrlF10*	$6700

Table 19.26
Alt-function key codes

Constant	Value	Constant	Value
kbAltF1	$6800	*kbAltF6*	$6D00
kbAltF2	$6900	*kbAltF7*	$6E00
kbAltF3	$6A00	*kbAltF8*	$6F00
kbAltF4	$6B00	*kbAltF9*	$7000
kbAltF5	$6C00	*kbAltF10*	$7100

See also *evKeyDown*, *GetKeyEvent*

LoadHistory procedure HistList

Declaration

```
procedure LoadHistory(var S: TStream);
```

Function Reads the application's history block from the stream *S* by reading the size of the block, then the block itself. Sets *HistoryUsed* to the end of the block read. Use *LoadHistory* to restore a history block saved with *StoreHistory*.

See also *HistoryUsed* variable, *StoreHistory* procedure

LoadIndexes procedure ColorSel

Declaration

```
procedure LoadIndexes(var S: TStream);
```

Function Loads a set of color indexes from the stream *S* and stores it in the variable *ColorIndexes*. By storing and reloading *ColorIndexes* on a stream, an application can restore the state of the color selection dialog box, enabling the user to easily modify or undo color changes.

See also *ColorIndexes* variable, *StoreIndexes* procedure

LongDiv function Objects

Declaration

```
function LongDiv(X: Longint; Y: Integer): Integer;
inline($59/$58/$5A/$F7/$F9);
```

Function A fast, inline assembly division routine, returning the integer value `X/Y`.

LongMul function Objects

Declaration

```
function LongMul(X, Y: Integer): Longint;
inline($5A/$58/$F7/$EA);
```

Function A fast, inline assembly coded multiplication routine, returning the long integer value `X * Y`.

LongRec type Objects

Declaration

```
LongRec = record
  Lo, Hi: Word;
end;
```

Function A useful record type for handling double-word length variables.

LowMemory function Memory

Declaration

```
function LowMemory: Boolean;
```

Function Returns *True* if memory is low, otherwise *False*. *True* means that a memory allocation call (for example, by a constructor) was forced to "dip into" the memory safety pool. The size of the safety pool is defined by the *LowMemSize* variable.

See also Chapter 7, "Turbo Vision overview," *InitMemory*, *TView.Valid*, *LowMemSize*

LowMemSize variable Memory

Declaration

```
LowMemSize: Word = 4096 div 16;
```

Function Sets the size of the safety pool in 16-byte paragraphs. The default value is the pratical minimum, but it can be increased to suit your application.

See also *InitMemory*, Safety pool, *TView.Valid*, *LowMemory*

MaxBufMem variable — Memory

Declaration `MaxBufMem: Word = 65536 div 16;`

Function Specifies the maximum amount of memory, in 16-byte paragraphs, that can be allocated to cache buffers.

See also *GetBufMem, FreeBufMem*

MaxCollectionSize variable — Objects

Declaration `MaxCollectionSize = 65520 div SizeOf(Pointer);`

Function *MaxCollectionSize* determines the maximum number of elements a collection can contain, which is essentially the number of pointers that can fit in a 64K memory segment.

MaxHeapSize variable — Memory

Declaration `MaxHeapSize: Word = 655360 div 16;`

Function Defines the maximum size of the buffer heap, in 16-byte paragraphs. The buffer heap is used by file editor objects to allocate movable, resizeable buffers without eating into the application's heap space.

See also *NewBuffer* procedure, *TFileEditor.InitBuffer*

MaxLineLength constant — Editors

Declaration `MaxLineLength = 256;`

Function *MaxLineLength* is the maximum length of a line in an editor object.

MaxViewWidth constant — Views

Declaration `MaxViewWidth = 132;`

Function The maximum width of a view.

See also *TView.Size* field

mbXXXX constants — Drivers

Function These constants can be used when examining the *TEvent.Buttons* field of an *evMouse* event record. For example,

```
if (Event.What = evMouseDown) and
  (Event.Buttons = mbLeftButton) then LeftButtonDown;
```

Values The following constants are defined:

Table 19.27
Mouse button constants

Constant	Value	Meaning
mbLeftButton	$01	Set if left button was pressed
mbRightButton	$02	Set if right button was pressed

See also *GetMouseEvent*

MemAlloc function — Memory

Declaration

```
function MemAlloc(Size: Word): Pointer;
```

Function Allocates *Size* bytes of memory on the heap and returns a pointer to the block. If a block of the requested size cannot be allocated, a value of **nil** is returned. Unlike the *New* and *GetMem* standard procedures, *MemAlloc* will not allow the allocation to dip into the safety pool. Dispose of blocks allocated by *MemAlloc* with *FreeMem.*

See also *New, GetMem, Dispose, FreeMem, MemAllocSeg*

MemAllocSeg function — Memory

Declaration

```
function MemAllocSeg(Size: Word): Pointer;
```

Function Allocates a segment-aligned memory block. Corresponds to *MemAlloc,* except the offset part of the resulting pointer value is always zero.

See also *MemAlloc*

MenuBar variable — App

Declaration

```
MenuBar: PMenuView = nil;
```

Function Stores a pointer to the application's menu bar (a descendant of *TMenuView*). The *MenuBar* variable is initialized by *TProgram.InitMenuBar*, which is called by *TProgram.Init*. A value of **nil** indicates that the application has no menu bar.

MenuColorItems function — ColorSel

Declaration

```
function MenuColorItems(const Next: PColorItem): PColorItem;
```

Function Returns a linked list of *TColorItem* records for the standard menu views. For programs that allow the user to change menu colors with the color selection dialog box, *MenuColorItems* simplifies the process of setting up the color items.

Message function — Views

Declaration

```
function Message(Receiver: PView; What, Command: Word; InfoPtr:
  Pointer): Pointer;
```

Function *Message* sets up an event record with the arguments *What, Command* and *InfoPtr* then, if possible, calls *Receiver^.HandleEvent* to handle the event.

Message returns **nil** if *Receiver* is **nil**, or if the event is not handled successfully. If the receiver handles the event (that is, *HandleEvent* returns *Event.What* as *evNothing*), *Message* returns *Event.InfoPtr*. The latter can be used to determine which view actually handled the dispatched event, because *ClearEvent* sets *InfoPtr* to point to the object that cleared the event.

What is usually *evBroadcast*. For example, the default *TScrollBar.ScrollDraw* sends the following message to the scroll bar's owner:

```
Message(Owner, evBroadcast, cmScrollBarChanged, @Self);
```

The above message ensures that the appropriate views are redrawn whenever the scroll bar's *Value* changes.

See also *TView.HandleEvent*, *TEvent* type, *cmXXXX* constants, *evXXXX* constants

MessageBox function — MsgBox

Declaration

```
function MessageBox(const Msg: String; Params: Pointer; AOptions: Word): Word;
```

Displays a 40-column, 9-line dialog box centered on the screen. The dialog box contains the message passed in *Msg*, inserting any parameters passed in *Params*. *AOptions* contains some combination of the *mfXXXX* message flag constants, determining which buttons appear in the message box. *MessageBox* uses the *FormatStr* procedure to incorporate any parameters passed in *Params* into *Msg*.

See also *mfXXXX* constants, *FormatStr* procedure

MessageBoxRect function — MsgBox

Declaration

```
function MessageBoxRect(var R: TRect; const Msg: String; Params: Pointer;
  AOptions: Word): Word;
```

Function Works exactly like *MessageBox* but allows you to specify a bounding rectangle for the dialog box.

See also *MessageBox* function

mfXXXX constants — MsgBox

Function Turbo Vision's message box functions, *MessageBox* and *MessageBoxRect*, use *mfXXXX* constants to specify the type of message being displayed and the buttons that appear in the box.

Values The following constants designate the type of message box displayed by *MessageBox* when passed in the *AOptions* parameter:

Constant	Value	Meaning
mfWarning	$0000	Display a Warning box
mfError	$0001	Display an Error box
mfInformation	$0002	Display an Information Box
mfConfirmation	$0003	Display a Confirmation Box

The following constants, passed in the *AOptions* parameter of *MessageBox* or *MessageBoxRect*, determine which buttons appear in the message box:

Constant	Value	Meaning
mfYesButton	$0100	Put a Yes button into the dialog
mfNoButton	$0200	Put a No button into the dialog
mfOKButton	$0400	Put an OK button into the dialog
mfCancelButton	$0800	Put a Cancel button into the dialog
mfYesNoCancel	$0B00	Standard Yes, No, Cancel dialog
mfOKCancel	$0C00	Standard OK, Cancel dialog

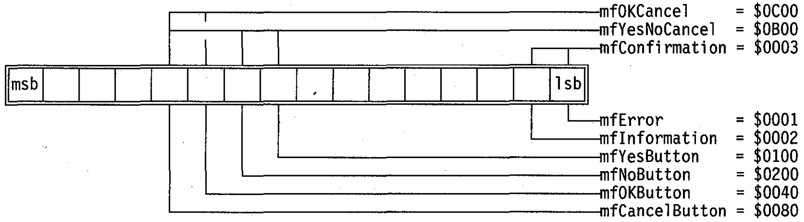


Figure 19.8
Message box flag mapping

See also *MessageBox* function, *MessageBoxRect* function

MinWinSize variable Views

Declaration

```
MinWinSize: TPoint = (X: 16; Y: 6);
```

Function Defines the minimum size of a window object. The value is returned in the *Min* parameter on a call to *TWindow.SizeLimits*. *MinWinSize* is global. Its value affects all windows, unless a particular window type overrides *SizeLimits* to ignore *MinWinSize*.

See also *TWindow.SizeLimits*

MouseButtons variable Drivers

Declaration

```
MouseButtons: Byte;
```

Function Contains the current state of the mouse buttons. *MouseButtons* is updated by the mouse interrupt handler whenever a button is pressed or released. Use the *mbXXXX* constants to examine *MouseButtons*.

See also *mbXXX* constants

MouseEvents variable Drivers

Declaration

```
MouseEvents: Boolean = False;
```

Function Set to *True* if *InitEvents* detects a mouse; otherwise set to *False*. If *False*, the application's event loop bypasses all mouse event routines.

See also *GetMouseEvent*

MouseIntFlag variable Drivers

Declaration `MouseIntFlag: Byte;`

Function Used internally by the Turbo Vision mouse driver and by views. Set whenever a mouse event occurs.

MouseReverse variable Drivers

Declaration `const MouseReverse: Boolean = False;`

Function Setting *MouseReverse* to *True* causes the event manager to reverse the *mbLeftButton* and *mbRightButton* flags in the *Buttons* field of *TEvent* records.

See also *mbXXXX* constants, *TEvent* type

MouseWhere variable Drivers

Declaration `MouseWhere: TPoint;`

Function Contains the current position of the mouse in global coordinates. The mouse interrupt handler updates *MouseWhere* whenever the mouse moves. Use *MakeLocal* to convert to local, window-relative coordinates. *MouseWhere* is passed to event handlers together with other mouse data.

See also *GetMouseEvent*, *GetEvent* methods, *MakeLocal*

MoveBuf procedure Drivers

Declaration `procedure MoveBuf(var Dest; var Source; Attr: Byte; Count: Word);`

Function Moves text and video attributes into a buffer for use with a view's *WriteBuf* or *WriteLine* methods. *Dest* must be *TDrawBuffer* (or an equivalent array of words) and *Source* must be an array of bytes. *Count* bytes are moved from *Source* into the low bytes of corresponding words in *Dest*. The high bytes of the words in *Dest* are set to *Attr* or remain unchanged if *Attr* is zero.

See also *TDrawBuffer* type, *MoveChar*, *MoveCStr*, *MoveStr*

MoveChar procedure Drivers

Declaration `procedure MoveChar(var Dest; C: Char; Attr: Byte; Count: Word);`

Function Moves characters into a buffer for use with a view's *WriteBuf* or *WriteLine*. *Dest* must be a *TDrawBuffer* (or an equivalent array of words). The low bytes of the first *Count* words of *Dest* are set to *C* or remain unchanged if *Ord(C)* is zero. The high bytes of the words are set to *Attr* or remain unchanged if *Attr* is zero.

See also *TDrawBuffer* type, *MoveBuf*, *MoveCStr*, *MoveStr*

MoveCStr procedure Drivers

Declaration `procedure MoveCStr(var Dest; const Str: String; Attrs: Word);`

Function Moves a two-colored string into a buffer for use with a view's *WriteBuf* or *WriteLine*. *Dest* must be a *TDrawBuffer* (or an equivalent array of words). The characters in *Str* are moved into the low bytes of corresponding words in *Dest*. The high bytes of the words are set to *Lo(Attr)* or *Hi(Attr)*. Tilde characters (~) in the string toggle between the two attribute bytes passed in the *Attr* word.

See also *TDrawBuffer* type, *MoveChar*, *MoveBuf*, *MoveStr*

MoveStr procedure Drivers

Declaration `procedure MoveStr(var Dest; const Str: String; Attr: Byte);`

Function Moves a string into a buffer for use with a view's *WriteBuf* or *WriteLine*. *Dest* must be a *TDrawBuffer* (or an equivalent array of words). The characters in *Str* are moved into the low bytes of corresponding words in *Dest*. The high bytes of the words are set to *Attr* or remain unchanged if *Attr* is zero.

See also *TDrawBuffer* type, *MoveChar*, *MoveCStr*, *MoveBuf*

NewBuffer procedure Memory

Declaration `procedure NewBuffer(var P: Pointer; Size: Word);`

Allocates a movable, resizeable buffer of *Size* bytes in the space reserved for file editor buffers above the regular heap and assigns it to *P*. You can later change the amount of memory allocated to *P* by calling *SetBufferSize*. You must dispose of the buffer by calling *DisposeBuffer*, rather than *FreeMem* or *Dispose*.

☞ The memory manager can move the buffer at any time, but it updates the value of *P* when it does so. That means *P* itself is always a valid pointer, but other values based on *P* could become invalid without warning.

See also *DisposeBuffer* procedure, *GetBufferSize* function, *SetBufferSize* function

NewCache procedure — Memory

Declaration

```
procedure NewCache(var P: Pointer; Size: Word);
```

Function Allocates a cache buffer of *Size* bytes to the pointer *P*. If there is no room for a cache buffer of the requested size, *P* is set to **nil**.

If the memory manager later needs to reclaim the cache space for another allocation, it disposes of the memory allocated to *P* and sets *P* to **nil**. Be sure to test cache buffers before dereferencing them, as your application has no control over whether or when the memory manager might reclaim the cache memory.

Turbo Vision uses cache buffers to cache the contents of group objects whenever those objects have the *ofBuffered* flag set, greatly increasing performance of redraw operations.

See also *DisposeCache* procedure

NewItem function — Menus

Declaration

```
function NewItem(Name, Param: TMenuStr; KeyCode: Word; Command: Word;
  AHelpCtx: Word; Next: PMenuItem): PMenuItem;
```

Function Allocates and returns a pointer to a new *TMenuItem* record that represents a menu item (using *NewStr* to allocate the *Name* and *Param* string pointer fields). The *Name* parameter must be a non-empty string, and the *Command* parameter must be non-zero. Calls to *NewItem*, *NewLine*, *NewMenu*, and *NewSubMenu* can be nested to create entire menu trees in one statement. For examples of this, see Chapter 10, "Application objects."

See also *TApplication.InitMenuBar*, *TMenuView* type, *NewLine*, *NewMenu*, *NewSubMenu*

NewLine function Menus

Declaration `function NewLine(Next: PMenuItem): PMenuItem;`

Function Allocates and returns a pointer to a new *TMenuItem* record that represents a separator line in a menu box.

See also *TApplication.InitMenuBar, TMenuView* type, *NewMenu, NewSubMenu, NewItem*

NewMenu function Menus

Declaration `function NewMenu(Items: PMenuItem): PMenu;`

Function Allocates and returns a pointer to a new *TMenu* record. Sets the *Items* and *Default* fields of the record to the value given by the *Items* parameter.

See also *TApplication.InitMenuBar, TMenuView* type, *NewLine, NewSubMenu, NewItem*

NewNode function Outline

Declaration `function NewNode(const AText: String; AChildren, ANext: PNode): PNode;`

Function Creates and allocates a node record of type *TNode* for an outline list and returns a pointer to the new node. *NewNode* sets the new node's *Text, ChildList,* and *Next* fields to *AText, AChildren,* and *ANext,* respectively.

See also *DisposeNode* procedure, *TNode* type

NewSItem function Dialogs

Declaration `function NewSItem(const Str: String; ANext: PSItem): PSItem;`

Function Allocates and returns a pointer to a new *TSItem* record. Sets the *Value* and *Next* fields of the record to *NewStr(Str)* and *ANext,* respectively. The *NewSItem* function and the *TSItem* record type allow easy construction of singly-linked lists of strings. For an example of this, see Chapter 12, "Control objects."

NewStatusDef function — Menus

Declaration

```
function NewStatusDef(AMin, AMax: Word; AItems: PStatusItem;
  ANext: PStatusDef): PStatusDef;
```

Function Allocates and returns a pointer to a new *TStatusDef* record initialized with the given parameter values. Calls to *NewStatusDef* and *NewStatusKey* can be nested to create entire status line definitions in one Pascal statement. For an example of this, see Chapter 10, "Application objects."

See also *TApplication.InitStatusLine, TStatusLine, NewStatusKey*

NewStatusKey function — Menus

Declaration

```
function NewStatusKey(const AText: String; AKeyCode: Word; ACommand: Word;
  ANext: PStatusItem): PStatusItem;
```

Function Allocates and returns a pointer to a new *TStatusItem* record initialized with the given parameter values (using *NewStr* to allocate the *Text* pointer field). If *AText* is empty (which results in a **nil** *Text* field), the status item is invisible but still binds *KeyCode* to the given *Command*.

See also *TApplication.InitStatusLine, TStatusLine, NewStatusDef*

NewStr function — Objects

Declaration

```
function NewStr(const S: String): PString;
```

Function Allocates a dynamic string on the heap. If *S* is null, *NewStr* returns a **nil** pointer. Otherwise, *NewStr* allocates *Length(S)*+1 bytes containing a copy of *S* and returns a pointer to the first byte.

Strings created with *NewStr* should be disposed of with *DisposeStr*.

See also *DisposeStr*

NewSubMenu function — Menus

Declaration

```
function NewSubMenu(Name: TMenuStr; AHelpCtx: Word; SubMenu: PMenu;
  Next: PMenuItem): PMenuItem;
```

N

Function Allocates and returns a pointer to a new *TMenuItem* record which represents a submenu (using *NewStr* to allocate the *Name* pointer field).

See also *TApplication.InitMenuBar*, *TMenuItem* type, *NewLine*, *NewItem*

ofXXXX constants Views

Function These mnemonics refer to the bit positions of a view's *Options* field. Setting a bit position to 1 indicates that the view has that particular attribute. Clearing the bit position means that the attribute is off or disabled. For example,

```
MyWindow.Options := ofTileable + ofSelectable;
```

Values The following option flags are defined:

Table 19.28
Option flags

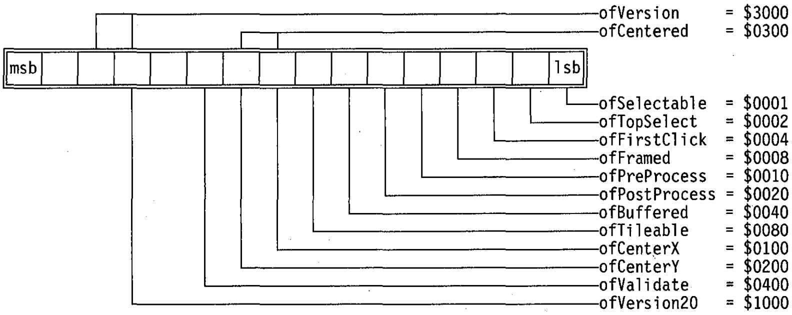
Constant	Meaning if set
ofSelectable	The view should select itself automatically, for example, by a mouse click in the view, or a *Tab* in a dialog box.
ofTopSelect	The view should move in front of all other peer views when selected. When the *ofTopSelect* bit is set, a call to *Select* corresponds to a call to *MakeFirst*. Windows (*TWindow* and descendants), by default, have the *ofTopSelect* bit set, which causes them to move in front of all other windows on the desktop when selected.
ofFirstClick	A mouse click that selects the view is also processed as a normal mouse click after selecting the view. Has no effect unless *ofSelectable* is also set. If clear, the selecting mouse click has no further effect.
ofFramed	The view should have a frame drawn around it. A window, and any descendant of *TWindow*, has a frame object as its last subview. When drawing itself, the frame object also draws a frame line around any other subviews that have *ofFramed* set.
ofPreProcess	The view should receive focused events before they are sent to the focused view.
ofPostProcess	The view should receive focused events if the focused view failed to handle them.
ofBuffered	[for group objects only] A cache buffer should be allocated if sufficient memory is available. The group buffer holds a screen image of the whole group so that group redraws can be speeded up. In the absence of a buffer, *TGroup.Draw* calls on each subview's *DrawView* method. If later *New* and *GetMem* calls cannot gain enough memory, group buffers will be deallocated to make memory available.

Table 19.28: Option flags (continued)

ofTileable	The desktop can tile (or cascade) this view. Used only with window objects.
ofCenterX	The view should be centered on the X-axis of its owner when inserted into a group.
ofCenterY	The view should be centered on the Y-axis of its owner when inserted into a group.
ofCentered	The view should be centered on both axes of its owner when inserted into a group.
ofValidate	The view should call *Valid* before losing the input focus.
ofVersion	The view contains version-dependent fields. See Chapter 17 for details on versioning.
ofVersion10	The view is a version 1.0 view. See Chapter 17 for details on versioning.
ofVersion20	The view is a version 2.0 view. See Chapter 17 for details on versioning.

The *Options* bits are defined as follows:

Figure 19.9 Options bit flags



See also *TView.Options*

ovXXXX constants Outline

Function The *CreateGraph* method of *TOutlineViewer* receives a parameter called *Flags* that holds a combination of *ovXXXX* constants. *Flags* determines how the outline viewer should draw the graphic portion of the outline.

Values The following constants are defined:

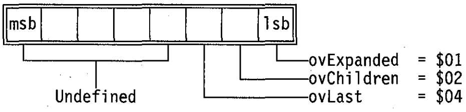


Table 19.29 Outline view constants

Constant	Value	Meaning
ovExpanded	$01	Node is expanded (show child nodes)
ovChildren	$02	Node has child nodes
ovLast	$04	Node is the last child of its parent

PositionalEvents variable

Views

Declaration

```
PositionalEvents: Word = evMouse;
```

Function Defines the event classes that are *positional events*. The *HandleEvent* method of a group object uses *FocusedEvents* and *PositionalEvents* to determine how to dispatch events to the group's subviews. If an event class isn't contained in *FocusedEvents* or *PositionalEvents*, the group treats it as a broadcast event.

See also *TGroup.HandleEvent*, *TEvent* type, *evXXXX* constants, *FocusedEvents* variable

PrintStr procedure

Drivers

Declaration

```
procedure PrintStr(S: String);
```

Function Prints the string *S* on the screen, using DOS function call 40H to write to the DOS standard output handle. Has the same effect as *Write(S)*, except that *PrintStr* doesn't require the file I/O run-time library to be linked into the application.

PString type

Objects

Declaration

```
PString = ^String;
```

Function Defines a pointer to a string.

PtrRec type — Objects

Declaration

```
PtrRec = record
  Ofs, Seg: Word;
end;
```

Function A record holding the offset and segment values of a pointer.

RegisterColorSel procedure — ColorSel

Declaration

```
procedure RegisterColorSel;
```

Function Calls *RegisterType* for each of the object types defined in the *ColorSel* unit: *TColorSelector, TMonoSelector, TColorDisplay, TColorGroupList, TColorItemList*, and *TColorDialog*. After a call to *RegisterColorSel*, any of those type can be read from or written to streams.

See also *RegisterType* procedure

RegisterDialogs procedure — Dialogs

Declaration

```
procedure RegisterDialogs;
```

Function Calls *RegisterType* for each of the standard object types defined in the Dialogs unit: *TDialog, TInputLine, TButton, TCluster, TRadioButtons, TCheckBoxes, TListBox, TStaticText, TParamText, TLabel*, and *THistory*. After a call to *RegisterDialogs*, any of those types can be read from or written to a stream.

See also *TStreamRec, RegisterTypes*

RegisterEditors procedure — Editors

Declaration

```
procedure RegisterEditors;
```

Function Calls *RegisterType* for each of the object types defined in the *Editors* unit: *TEditor, TMemo, TFileEditor, TIndicator*, and *TEditWindow*. After a call to *RegisterEditors*, any of those types can be read from or written to a stream.

See also *RegisterType*

RegisterStdDlg procedure StdDlg

Declaration `procedure RegisterStdDlg;`

Function Calls *RegisterType* for each of the object types defined in the *StdDlg* unit: *TFileInputLine, TFileCollection, TFileList, TFileInfoPane, TFileDialog, TDirCollection, TDirListBox,* and *TChDirDialog*. After a call to *RegisterStdDlg,* any of those type can be read from or written to a stream.

See also *RegisterType*

RegisterType procedure Objects

Declaration `procedure RegisterType(var S: TStreamRec);`

Function Registers an object type with Turbo Vision's streams, creating an entry in a linked list of known objects. Streams can only store and return these known object types. Each registered object needs a unique stream registration record of type *TStreamRec.*

See also *TStream.Get, TStream.Put, TStreamRec*

RegisterValidate procedure Validate

Declaration `procedure RegisterValidate;`

Function Calls *RegisterType* for each of the validator object types defined in the *Validate* unit: *TPXPictureValidator, TFilterValidator, TRangeValidator, TLookupValidator,* and *TStringLookupValidator*. After calling *RegisterValidate,* your application can read or write any of those types with streams.

See also *RegisterType* procedure

RepeatDelay variable Drivers

Declaration `RepeatDelay: Word = 8;`

Function Defines the number of clock ticks (1/18.2 parts of a second) that must transpire before *evMouseAuto* events are generated. The time interval between *evMouseAuto* events is always in increments of one clock tick.

See also *DoubleDelay, GetMouseEvent, evXXXX* constants

ReplaceStr variable Editors

Declaration `ReplaceStr: string[80] = '';`

Function Holds the last replacement string used in a search-and-replace operation.

See also *FindStr* variable, *TEditor.DoSearchReplace*

SaveCtrlBreak variable Drivers

Declaration `SaveCtrlBreak: Boolean = False;`

Function The *InitSysError* routine stores the state of DOS *Ctrl+Break* checking in this variable before it disables DOS *Ctrl+Break* checks. *DoneSysError* restores DOS *Ctrl+Break* checking to the value stored in *SaveCtrlBreak*.

See also *InitSysError, DoneSysError*

sbXXXX constants Views

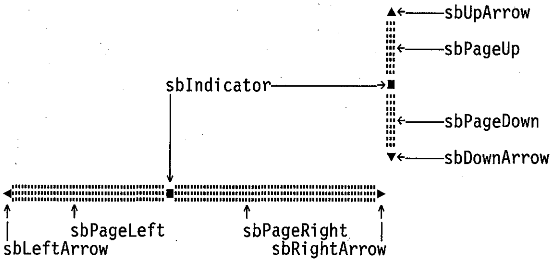
Function These constants define the different areas of a *TScrollBar* in which a mouse click can occur.

The *TScrollBar.ScrollStep* method converts these constants into actual scroll step values. Although defined, the *sbIndicator* constant is never passed to *ScrollStep*.

Table 19.30 Scroll bar part constants

Constant	Value	Meaning
sbLeftArrow	0	Left arrow of horizontal scroll bar
sbRightArrow	1	Right arrow of horizontal scroll bar
sbPageLeft	2	Left paging area of horizontal scroll bar
sbPageRight	3	Right paging area of horizontal scroll bar
sbUpArrow	4	Top arrow of vertical scroll bar
sbDownArrow	5	Bottom arrow of vertical scroll bar
sbPageUp	6	Upper paging area of vertical scroll bar
sbPageDown	7	Lower paging area of vertical scroll bar
sbIndicator	8	Position indicator on scroll bar

Figure 19.10
Scroll bar parts



The following values can be passed to the *TWindow.StandardScrollBar* method:

Table 19.31
StandardScrollBar constants

Constant	Value	Meaning
sbHorizontal	$0000	Scroll bar is horizontal
sbVertical	$0001	Scroll bar is vertical
sbHandleKeyboard	$0002	Scroll bar responds to keyboard commands

See also *TScrollBar, TScrollBar.ScrollStep*

ScreenBuffer variable — Drivers

Declaration `ScreenBuffer: Pointer;`

Function Pointer to the video screen buffer, set by *InitVideo*.

See also *InitVideo*

ScreenHeight variable — Drivers

Declaration `ScreenHeight: Byte;`

Function Set by *InitVideo* and *SetVideoMode* to the screen height (lines of the current video screen).

See also *InitVideo, SetVideoMode, ScreenWidth*

ScreenMode variable — Drivers

Declaration `ScreenMode: Word;`

Function Holds the current video mode. Set initially by the initialization code of the *Drivers* unit, *ScreenMode* can be changed using *SetVideoMode*. *ScreenMode* values are usually set using the *smXXXX* screen mode mnemonics.

See also *InitVideo, SetVideoMode, smXXXX*

ScreenWidth variable Drivers

Declaration

```
ScreenWidth: Byte;
```

Function Set by *InitVideo* to the screen width (number of characters per line).

See also *InitVideo*

SelectMode type Views

Declaration

```
SelectMode = (NormalSelect, EnterSelect, LeaveSelect);
```

Function Used internally by Turbo Vision.

See also *TGroup.ExecView, TGroup.SetCurrent*

SetBufferSize function Memory

Declaration

```
function SetBufferSize(P: Pointer; Size: Word): Boolean;
```

Function Sets the size of the buffer pointed to by *P* to *Size* bytes. *P^* must be a buffer allocated by *NewBuffer*. Returns *True* if the new allocation succeeds; otherwise returns *False*, and the size of the buffer remains unchanged.

See also *NewBuffer* procedure, *GetBufferSize* function

SetMemTop procedure Memory

Declaration

```
procedure SetMemTop(MemTop: Pointer);
```

Function Sets the top of the application's memory block. The initial memory top corresponds to the value stored in the *HeapEnd* variable. *SetMemTop* is typically used to shrink the application's memory block before executing a DOS shell or another program and to expand the memory block afterward.

SetVideoMode procedure — Drivers

Declaration `procedure SetVideoMode(Mode: Word);`

Function Sets the video mode. *Mode* is one of the constants *smCO80*, *smBW80*, or *smMono*, optionally with *smFont8x8* added to select 43- or 50-line mode on an EGA or VGA. *SetVideoMode* initializes the same variables as *InitVideo* (except for the *StartupMode* variable, which isn't affected). *SetVideoMode* is normally not called directly. Instead, you should use your application object's *SetScreenMode* method, which also adjusts the application palette.

See also *InitVideo*, *smXXXX* constants, *TProgram.SetScreenMode*

sfXXXX constants — Views

Function These constants correspond to bits in *TView.State* fields. You should never modify *State* fields directly; instead, use the view's *SetState* method.

Values The following state flags are defined:

Table 19.32
State flag constants

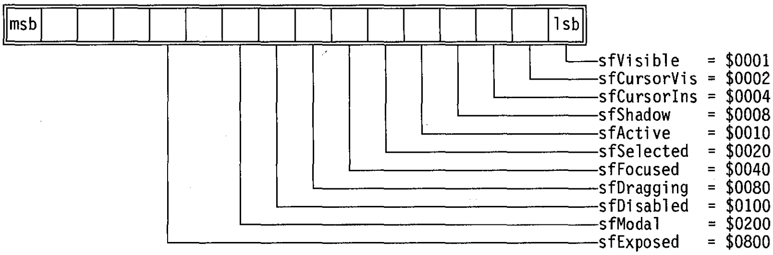
Constant	Meaning if set
sfVisible	The view is visible on its owner. *sfVisible* is set by default. A view's *Show* and *Hide* methods modify *sfVisible*. An *sfVisible* view is not necessarily visible on the screen, since its owner might not be visible. To test for visibility on the screen, call the view's *Exposed* method.
sfCursorVis	The view's cursor is visible. The default is clear. *ShowCursor* and *HideCursor* modify *sfCursorVis*.
sfCursorIns	The view's cursor is a solid block. The default is clear, making the cursor an underline. *BlockCursor* and *NormalCursor* modify *sfCursorIns*.
sfShadow	The view has a shadow.
sfActive	The view is the active window or a subview in that window.
sfSelected	The view is the currently selected subview within its owner. Every group object has a *Current* field that points to its currently selected subview (or is **nil** if no subview is selected). There can be only one currently selected subview in a group.
sfFocused	The view has the input focus. A view is focused if it is selected and all owners above it are also selected, that is, if the view is on the chain that is formed by following each *Current* pointer of all groups starting at *Application*. The last view on the focused chain is the final target of all focused events.
sfDragging	The view is being dragged.

Table 19.32: State flag constants (continued)

sfDisabled	The view is disabled. A disabled view ignores all events sent to it.
sfModal	The view is modal. When a view starts executing (through an *ExecView* call), that view becomes modal. The modal view represents the apex (root) of the active event tree, getting and handling events until its *EndModal* method is called. During this "local" event loop, events are passed down to lower subviews in the view tree. Events from these lower views pass back up the tree, but go no further than the modal view.
sfExposed	The view is owned directly or indirectly by the *Application* object, and therefore possibly visible on the screen. The *Exposed* method uses this flag to determine whether any part of the view might be visible on the screen.

The state flag bits are defined as follows:

Figure 19.11 State flag bit mapping



See also *TView.State*

ShadowAttr variable — Views

Declaration

```
ShadowAttr: Byte = $08;
```

Function This value controls the color of the "shadow" effect available on those views with the *sfShadow* bit set. The shadow is usually a thin, dark region displayed just beyond the view's edges giving a 3-D illusion.

See also *ShadowSize*

ShadowSize variable — Views

Declaration

```
ShadowSize: TPoint = (X: 2; Y: 1);
```

Function This value controls the size of the shadow effect available on those views with the *sfShadow* bit set. The shadow is usually a thin, dark region displayed just beyond the view's right and bottom edges giving a 3-D illusion. The default size is 2 in the X direction, and 1 in the Y direction.

TProgram.InitScreen initializes *ShadowSize* as follows: If the screen mode is *smMono*, *ShadowSize* is set to (0, 0); otherwise, *ShadowSize* is set to (2, 1), unless *smFont8x8* (43- or 50-line mode) is selected, in which case it is set to (1, 1).

See also *TProgram.InitScreen*, *ShadowAttr*

ShowMarkers variable — Views

Declaration `ShowMarkers: Boolean = False;`

Function Specifies whether indicators should appear around focused controls. *TProgram.InitScreen* sets *ShowMarkers* to *True* if the video mode is monochrome. Otherwise, it is *False*. You can also set *ShowMarkers* to *True* in color and black-and-white modes.

See also *TProgram.InitScreen*, *SpecialChars* variable

ShowMouse procedure — Drivers

Declaration `procedure ShowMouse;`

Function *ShowMouse* decrements the "hide counter" in the mouse driver and makes the mouse cursor visible if counter becomes zero.

See also *InitEvents*, *DoneEvents*, *HideMouse*

smXXXX constants — Drivers

Function These mnemonics are used with *SetVideoMode* to set the appropriate video mode value in *ScreenMode*.

Values The following screen modes are defined by Turbo Vision:

Table 19.33 Screen mode constants

Constant	Value	Meaning
smBW80	$0002	Black-and-white mode with color video
smCO80	$0003	Color mode
smMono	$0007	Monochrome mode
smFont8x8	$0100	43-line or 50-line mode

See also *SetVideoMode*, *ScreenMode*

SpecialChars variable — Views

Declaration

```
SpecialChars: array[0..5] of Char = (#175, #174, #26, #27, ' ', ' ');
```

Function Defines the indicator characters used to highlight the focused view in monochrome video mode. The variable *ShowMarkers* controls whether these characters appear.

See also *ShowMarkers* variable

stXXXX constants — Objects

Function There are two sets of constants beginning with *st* used by the Turbo Vision streams system.

Values The following mode constants are used by *TDosStream* and *TBufStream* to determine the file access mode of a file being opened for a Turbo Vision stream:

Table 19.34
Stream access modes

Constant	Value	Meaning
stCreate	$3C00	Create new file
stOpenRead	$3D00	Open existing file with read access only
stOpenWrite	$3D01	Open existing file with write access only
stOpen	$3D02	Open existing file with read and write access

A stream object's *Error* method places one of the following values in the stream's *ErrorInfo* field when a stream error occurs:

Table 19.35
Stream error codes

Error code	Value	Meaning
stOk	0	No error
stError	–1	Access error
stInitError	–2	Cannot initialize stream
stReadError	–3	Read beyond end of stream
stWriteError	–4	Cannot expand stream
stGetError	–5	Get of unregistered object type
stPutError	–6	Put of unregistered object type

See also *TStream*

StartupMode variable — Drivers

Declaration

```
StartupMode: Word = $FFFF;
```

Function *InitVideo* stores the current screen mode in this variable before it switches to the screen mode given by *ScreenMode*. *DoneVideo* restores the screen mode to the value stored in *StartupMode*.

See also *InitVideo, DoneVideo, ScreenMode*

StatusLine variable App

Declaration `StatusLine: PStatusLine = nil;`

Function Points to the application's status line, or **nil** if the application has no status line. The application object's virtual method *InitStatusLine* constructs a status line object and assigns it to *StatusLine*. You can define a customized status line by overriding *InitStatusLine* to construct the desired status line object and set *StatusLine* to point to it.

See also *InitStatusLine*

StdEditMenuItems function App

Declaration `function StdEditMenuItems(Next: PMenuItem): PMenuItem;`

Function Returns a pointer to a list of menu items for a standard Edit menu. You can use the list either as the entire menu, or as part of a larger list of items.

The standard Edit menu items are Undo, Cut, Copy, Paste, and Clear.

StdEditorDialog function Editors

Declaration `function StdEditorDialog(Dialog: Integer; Info: Pointer): Word;`

Function Displays a dialog box based on the value of *Dialog* and the information passed in *Info*. *StdEditorDialog* is intended as a working set of editor dialog boxes to be assigned to *EditorDialog*.

See also *EditorDialog* variable, *TEditorDialog* type

StdFileMenuItems function App

Declaration `function StdFileMenuItems(Next: PMenuItem): PMenuItem;`

Function Returns a pointer to a list of menu items for a standard File menu. You can use the list either as the entire menu, or as part of a larger list of items.

The standard File menu items are New, Open, Save, Save As, Save All, Change Dir, Dos Shell, and Exit.

StdStatusKeys function App

Declaration `function StdStatusKeys(Next: PStatusItem): PStatusItem;`

Returns a pointer to a linked list of commonly used status line keys. The default status line for *TApplication* uses *StdStatusKeys* as its complete list of status keys. You can append *StdStatusKeys* to your user-defined status line definitions to ensure that all your status lines maintain the standard command and key bindings.

The following is the implementation of *StdStatusKeys*:

```
function StdStatusKeys(Next: PStatusItem): PStatusItem;
begin
  StdStatusKeys :=
    NewStatusKey('', kbAltX, cmQuit,
    NewStatusKey('', kbF10, cmMenu,
    NewStatusKey('', kbAltF3, cmClose,
    NewStatusKey('', kbF5, cmZoom,
    NewStatusKey('', kbCtrlF5, cmResize,
    NewStatusKey('', kbF6, cmNext,
    Next))))));
end;
```

StdWindowMenuItems function App

Declaration `function StdWindowMenuItems(Next: PMenuItem): PMenuItem;`

Function Returns a pointer to a list of menu items for a standard Window menu. You can use the list either as the entire menu, or as part of a larger list of items.

The standard Window menu items are Tile, Cascade, Close All, Size/Move, Zoom, Next, Previous, and Close.

StreamError variable Objects

Declaration

```
StreamError: Pointer = nil;
```

Function *StreamError* points to a procedure called by a stream's *Error* method when a stream error occurs. The procedure must be a **far** procedure with one **var** parameter that is a *TStream*. That is, the procedure must be declared as

```
procedure MyStreamErrorProc(var S: TStream); far;
```

StreamError allows you to globally override all stream error handling. If *StreamError* is **nil**, no generalized stream error handling occurs. To change error handling for a particular type of stream, you should override that stream type's *Error* method.

StoreHistory procedure HistList

Declaration

```
procedure StoreHistory(var S: TStream);
```

Function Writes the currently used portion of the history block to the stream *S*, first writing the length of the block then the block itself. Use the *LoadHistory* procedure to restore the history block.

See also *LoadHistory* procedure

StoreIndexes procedure ColorSel

Declaration

```
procedure StoreIndexes(var S: TStream);
```

Function Stores a set of color indexes from the *ColorIndexes* variable on the stream *S*. By storing and reloading *ColorIndexes* on a stream, an application can restore the state of the color-selection dialog box, enabling the user to easily modify or undo color changes.

See also *ColorIndexes* variable, *LoadIndexes* procedure

SysColorAttr variable Drivers

Declaration

```
SysColorAttr: Word = $4E4F;
```

Function The default color used for error message displays by the system error handler. On monochrome systems, *SysMonoAttr* is used in place of *SysColorAttr*. Error messages with a cancel/retry option are displayed on the status line. The previous status line is saved and restored when conditions allow.

See also *SystemError*, *SysMonoAttr*

SysErrActive variable Drivers

Declaration

```
SysErrActive: Boolean = False;
```

Function Indicates whether the system error manager is currently active. Set *True* by *InitSysError*.

SysErrorFunc variable Drivers

Declaration

```
SysErrorFunc: TSysErrorFunc = SystemError;
```

Function *SysErrorFunc* is the system error function, of type *TSysErrorFunc*. The system error function is called whenever a DOS critical error occurs and whenever a disk swap is required on a single floppy system. *ErrorCode* is a value between 0 and 15 as defined in Table 19.36, and *Drive* is the drive number (0=A, 1=B, etc.) for disk-related errors. The default system error function is *SystemError*. You can install your own system error function by assigning it to *SysErrorFunc*. System error functions cannot be overlayed.

Table 19.36 System error function codes

Error code	Meaning
0..12	DOS critical error codes
13	Bad memory image of file allocation table
14	Device access error
15	Drive swap notification

Return values of the function should be as follows:

Table 19.37 System error function return values

Return value	Meaning
0	User requested retry
1	User requested abort

See also *SystemError* function, *TSysErrorFunc* type, *InitSysError* procedure

SysMonoAttr variable Drivers

Declaration

```
SysMonoAttr: Word = $7070;
```

Function The default attribute used for error message displays by the system error handler. On color systems, *SysColorAttr* is used in place of *SysMonoAttr*. Error messages with a cancel/retry option are displayed on the status line. The previous status line is saved and restored when conditions allow.

See also *SystemError, SysColorAttr*

SystemError function Drivers

Declaration

```
function SystemError(ErrorCode: Integer; Drive: Byte): Integer;
```

Function This is the default system error function. It displays one of the following error messages on the status line, depending on the value of *ErrorCode*, using the color attributes defined by *SysColorAttr* or *SysMonoAttr*.

Table 19.38 SystemError function messages

Error code	Message
0	Disk is write-protected in drive X
1	Critical disk error on drive X
2	Disk is not ready in drive X
3	Critical disk error on drive X
4	Data integrity error on drive X
5	Critical disk error on drive X
6	Seek error on drive X
7	Unknown media type in drive X
8	Sector not found on drive X
9	Printer out of paper
10	Write fault on drive X
11	Read fault on drive X
12	Hardware failure on drive X
13	Bad memory image of FAT detected
14	Device access error
15	Insert diskette in drive X

See also *SysColorAtrr, SysMonAttr, SysErrorFunc*

TApplication App

TObject	TView		TGroup	TProgram	TApplication
	Cursor DragMode EventMask GrowMode HelpCtx Next	Options Origin Owner Size State	Buffer Current Last Phase		
~~Init~~ Free ~~Done~~	~~Init~~ ~~Load~~ ~~Done~~ ~~Awaken~~ BlockCursor CalcBounds ~~ChangeBounds~~ ClearEvent CommandEnabled ~~DataSize~~ DisableCommands DragView ~~Draw~~ DrawView EnableCommands ~~EndModal~~ EventAvail ~~Execute~~ Exposed Focus GetBounds GetClipRect GetColor GetCommands ~~GetData~~ ~~GetEvent~~ GetExtent ~~GetHelpCtx~~ ~~GetPalette~~ GetPeerViewPtr GetState GrowTo ~~HandleEvent~~ Hide	HideCursor KeyEvent Locate MakeFirst MakeGlobal MakeLocal MouseEvent MouseInView MoveTo NextView NormalCursor Prev PrevView PutEvent PutInFrontOf PutPeerViewPtr Select SetBounds SetCommands SetCmdState SetCursor ~~SetData~~ ~~SetState~~ Show ShowCursor SizeLimits ~~Store~~ TopView ~~Valid~~ WriteBuf WriteChar WriteLine WriteStr	~~Init~~ Load ~~Done~~ Awaken ChangeBounds DataSize Delete Draw EndModal EventError ExecView Execute First FirstThat FocusNext ForEach GetData GetHelpCtx GetSubViewPtr ~~HandleEvent~~ Insert InsertBefore Lock PutSubViewPtr Redraw SelectNext SetData SetState Store Unlock Valid	~~Init~~ ~~Done~~ CanMoveFocus ExecuteDialog GetEvent GetPalette ~~HandleEvent~~ Idle InitDeskTop InitMenuBar InitScreen InitStatusLine InsertWindow OutOfMemory PutEvent Run SetScreenMode ValidView	Init Done Cascade DosShell GetTileRect HandleEvent Tile WriteShellMsg

TApplication is a simple "wrapper" around *TProgram* and differs from *TProgram* only in its constructor and destructor methods. Normally, you will derive your application objects from *TApplication*. Should you require a different sequence of subsystem initialization and shut down, however, you can derive your application from *TProgram* and manually initialize and shut down the Turbo Vision subsystems along with your own.

☞ In version 2.0, *TApplication* adds several new methods for handling standard application commands. *TApplication* now has a *HandleEvent* method that handles commands from the standard menus, and methods that tile and cascade windows and shell to DOS.

Methods

Init

```
constructor Init;
```

Constructs an application object by first initializing all the Turbo Vision subsystems (the memory, video, event, system error, and history list managers) and then calling the *Init* constructor inherited from *TProgram*.

See also: *InitMemory, InitVideo, InitEvents, InitSysError, InitHistory, TProgram.Init*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the application object by first calling the *Done* destructor inherited from *TProgram* and then shutting down all the Turbo Vision subsystems.

See also: *TProgram.Done, DoneHistory, DoneSysError, DoneEvents, DoneVideo, DoneMemory*

Cascade

```
procedure Cascade;
```

Calls *GetTileRect* to get the region over which windows should cascade, then if *Desktop* is not **nil**, calls the desktop's *Cascade* method, passing the tiling rectangle.

See also: *TApplication.GetTileRect, TDesktop.Cascade*

DosShell

```
procedure DosShell;
```

Starts a DOS shell. *DosShell* first shuts down the system error handler, event manager, video manager, and DOS memory manager subsystems, then calls *WriteShellMsg* to display any user message, then executes the command interpreter indicated by the DOS environment variable COMSPEC.

When the user exits from the DOS shell, *DosShell* restarts the subsystems, then calls *Redraw* to refresh the application views.

See also: *TApplication.WriteShellMsg*

GetTileRect

```
procedure GetTileRect(var R: TRect); virtual;
```

Sets *R* to the rectangle on the desktop that tiled or cascaded windows should cover. By default, *GetTileRect* returns the extent of the entire desktop view. Both the *Cascade* and *Tile* methods call *GetTileRect* to determine the area for rearranging windows.

Your application can override *GetTileRect* to return a different rectangle, for example to exclude areas covered by message windows.

See also: *TApplication.Cascade, TApplication.Tile*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles most events by calling the *HandleEvent* method inherited from *TProgram,* then responds to three of the standard application commands, *cmTile, cmCascade,* and *cmDosShell,* by calling the methods *Tile, Cascade,* and *DosShell,* respectively.

In version 1.0, *TApplication* did not override *TProgram.HandleEvent.*

See also: *TProgram.HandleEvent, TApplication.Cascade, TApplication.DosShell, TApplication.Tile*

Tile

```
procedure Tile;
```

Calls *GetTileRect* to get the region over which windows should tile, then if *Desktop* is not **nil**, calls the desktop's *Tile* method, passing the tiling rectangle.

See also: *TApplication.GetTileRect, TDesktop.Tile*

WriteShellMsg

```
procedure WriteShellMsg; virtual;
```

Prints a message to the user before shelling to DOS. The *DosShell* routine calls *WriteShellMsg* just before executing the command interpreter. By default, *WriteShellMsg* displays the following message:

```
Type EXIT to return...
```

You can override *WriteShellMsg* to display any message to users. You should print the message using the *PrintStr* procedure rather than using *Writeln,* since *PrintStr* does not require the use of the runtime library.

See also *TApplication.DosShell*

TBackground App

TObject
~~Init~~ Free ~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	HandleEvent	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	SetState
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TBackground
Pattern
Init Load Draw GetPalette Store

TBackground is a simple view consisting of a uniformly patterned rectangle. It is usually owned by a *TDesktop*.

Field

Pattern

`Pattern: Char;` **Read only**

The bit pattern giving the view's background.

Methods

Init

```
constructor Init(var Bounds: TRect; APattern: Char);
```

Creates a background object with the given *Bounds* by calling the *Init* constructor inherited from *TView*. Sets *GrowMode* to *gfGrowHiX* + *gfGrowHiY*, and *Pattern* to *APattern*.

See also: *TView.Init, TBackground.Pattern*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a background object from the stream *S* by calling the *Load* constructor inherited from *TView* and then reading the *Pattern* character.

See also: *TView.Load*

Draw `procedure Draw; virtual;`

Override: Seldom Fills the background view rectangle with the current *Pattern* in the default color.

GetPalette `function GetPalette: PPalette; virtual;`

Override: Seldom Returns a pointer to the default background palette, *CBackground*.

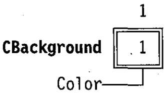
Store `procedure Store(var S: TStream);`

Stores the background view on the stream by calling the *Store* method inherited from *TView* and then writing the *Pattern* character.

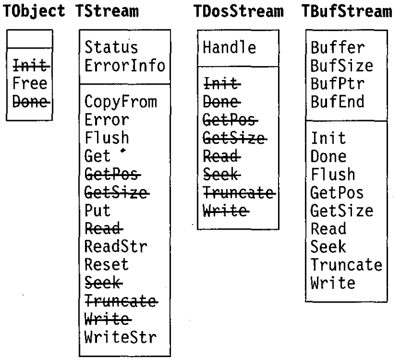
See also: *TView.Store*, *TBackground.Load*

Palette

Background objects use the default palette *CBackground* to map onto the first entry in the application palette.



TBufStream Objects



TBufStream implements a buffered version of *TDosStream*. The additional fields specify the size and location of the buffer, together with the current and last positions within the buffer. In addition to overriding the eight methods of *TDosStream*, *TBufStream* defines the abstract *TStream.Flush* method. The *TBufStream* constructor creates and opens a named file by calling *TDosStream.Init*, then creates the buffer with *GetMem*.

TBufStream is significantly more efficient than *TDosStream* when a large number of small data transfers take place on the stream, such as when loading and storing objects using *TStream.Get* and *TStream.Put*.

Fields

BufEnd

`BufEnd: Word;` **Read only**

If the buffer is not full, *BufEnd* gives an offset from the *Buffer* pointer to the last used byte in the buffer.

Buffer

`Buffer: Pointer;` **Read only**

A pointer to the start of the stream's buffer

BufPtr

`BufPtr: Word;` **Read only**

An offset from the *Buffer* pointer indicating the current position within the buffer.

BufSize

`BufSize: Word;` **Read only**

The size of the buffer in bytes

Methods

Init

```
constructor Init(FileName: FNameStr; Mode, Size: Word);
```

Constructs the object and opens the file named in *FileName* with access mode *Mode* by calling the *Init* constructor inherited from *TDosStream*. Allocates a buffer of *Size* bytes on the heap. Sets *BufPtr* and *BufEnd* to zero. Typical buffer sizes range from 512 bytes to 2,048 bytes.

See also: *TDosStream.Init*

Done

```
destructor Done; virtual;
```

Override: Never

Calls *Flush* to flush buffer contents to disk, then disposes of the buffered stream object by calling the *Done* destructor inherited from *TDosStream*. Frees the memory allocated to the buffer.

See also: *TBufStream.Flush, TDosStream.Done*

Flush

```
procedure Flush; virtual;
```

Override: Never

Flushes the stream's buffer provided the stream's status is *stOK*. The *Done* destructor calls *Flush* to make sure all data get written to the disk before disposing of the stream object.

See also: *TBufStream.Done*

GetPos

```
function GetPos: Longint; virtual;
```

Override: Never

Returns the value of the stream's current position (not to be confused with *BufPtr*, the current location within the buffer).

See also: *TBufStream.Seek*

GetSize

```
function GetSize: Longint; virtual;
```

Override: Never

Flushes the buffer then returns the total size in bytes of the stream.

See also: *TBufStream.Flush*

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Override: Never

If the stream's status is *stOK*, reads *Count* bytes into the *Buf* buffer starting at the stream's current position.

☞ *Buf* is *not* the stream's buffer, but an external buffer to hold the data read in from the stream.

See also: *TBufStream.Write*, *stReadError*

Seek

```
procedure Seek(Pos: Longint); virtual;
```

Override: Never

Flushes the buffer then resets the current position to *Pos* bytes from the start of the calling stream. The start of a stream is position 0.

See also: *TBufStream.Flush*, *TBufStream.GetPos*

Truncate

```
procedure Truncate; virtual;
```

Override: Never

Flushes the buffer then deletes all data on the stream from the current position to the end by calling the *Truncate* method inherited from *TDosStream*. The current position is set to the new end of the truncated stream.

See also: *TBufStream.Flush*, *TDosStream.Truncate*

Write

```
procedure Write(var Buf; Count: Word); virtual;
```

Override: Never

If the stream's status is *stOK*, writes *Count* bytes from the *Buf* buffer to the stream, starting at the current position.

☞ *Buf* is *not* the stream's buffer, but an external buffer to hold the data being written to the stream. When you call *Write*, *Buf* points to the variable whose value is being written to the stream.

See also: *TBufStream.Read*, *stWriteError*

TButton

Dialogs

TObject	TView			TButton
	Cursor	HelpCtx	Owner	Title
~~Init~~	DragMode	Next	Size	Command
Free	EventMask	Options	State	Flags
~~Done~~	GrowMode	Origin		AmDefault

TView			TButton
~~Init~~	GetCommands	Prev	Init
~~Load~~	GetData	PrevView	Load
~~Done~~	GetEvent	PutEvent	Done
Awaken	GetExtent	PutInFrontOf	Draw
BlockCursor	GetHelpCtx	PutPeerViewPtr	DrawState
CalcBounds	~~GetPalette~~	Select	GetPalette
ChangeBounds	GetPeerViewPtr	SetBounds	HandleEvent
ClearEvent	GetState	SetCommands	MakeDefault
CommandEnabled	GrowTo	SetCmdState	Press
DataSize	~~HandleEvent~~	SetCursor	SetState
DisableCommands	Hide	SetData	Store
DragView	HideCursor	~~SetState~~	
~~Draw~~	KeyEvent	Show	
DrawView	Locate	ShowCursor	
EnableCommands	MakeFirst	SizeLimits	
EndModal	MakeGlobal	~~Store~~	
EventAvail	MakeLocal	TopView	
Execute	MouseEvent	Valid	
Exposed	MouseInView	WriteBuf	
Focus	MoveTo	WriteChar	
GetBounds	NextView	WriteLine	
GetClipRect	NormalCursor	WriteStr	
GetColor			

A *TButton* object is a view with a title and a shadow that generates a command when pressed, most often found in dialog boxes. Users can press buttons by pressing the highlighted letter, by tabbing to the button and pressing *Spacebar,* by pressing *Enter* when the button is the default (indicated by highlighting), or by clicking the button with a mouse.

With the color and black-and-white palettes, buttons have a three-dimensional look and appear to move when clicked. On monochrome systems, Turbo Vision borders buttons with brackets, with other ASCII characters to indicate whether the button is default, selected, and so on.

Like the other controls defined in the *Dialogs* unit, *TButton* is a "terminal" object that you can insert into any group without having to override any of its methods.

There can only be one default button in a window or dialog box at any given time. Buttons that are peers in a group grab and release the default state via *evBroadcast* messages. Disabling or enabling the command bound to a button also disables or enables the button itself.

Fields

AmDefault

```
AmDefault: Boolean;                                        Read only
```

If *True*, the button is the default (and therefore "pressed" when the user presses *Enter*). Otherwise, the button is normal.

See also: *bfXXXX* button flag constants

Command

```
Command: Word;                                             Read only
```

The command word of the event generated when the user presses the button.

See also: *TButton.Init*

Flags

```
Flags: Byte;                                               Read/write
```

Flags is a bitmapped field used to indicate whether button text is left-aligned or centered. The individual flags are described under "*bfXXXX* button flag constants" in this chapter.

See also: *TButton.Draw*, *bfXXXX* button flag constants

Title

```
Title: PString;                                            Read only
```

A pointer to the button label's text.

Methods

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr; ACommand: Word;
  AFlags: Byte);
```

Creates a button object with the given bounds by calling the *Init* constructor inherited from *TView*. Allocates a title string *Title* by calling *NewStr(ATitle)*. *AFlags* serves two purposes: If *AFlags* **and** *bfDefault* is nonzero, *AmDefault* is set to *True*; in addition, *AFlags* indicates whether the title should be centered or left-aligned by testing whether *AFlags* **and** *bfLeftJust* is nonzero.

Options is set to (*ofSelectable* + *ofFirstClick* + *ofPreProcess* + *ofPostProcess*). *EventMask* is set to *evBroadcast*. If the given *ACommand* is not enabled, *sfDisabled* is set in the *State* field.

To define a shortcut key for the button, put tildes (~) around one of its characters in *ATitle*, which becomes the shortcut letter.

See also: *TView.Init*, *bfXXXX* button flag constants

Load

```
constructor Load(var S: TStream);
```

Constructs a button object and initializes it from the stream *S* by calling the *Load* constructor inherited from *TView*. Sets other fields by calling *S.Read*, and sets *State* according to whether the command in the *Command* field is enabled. Used in conjunction with *Store* to save and retrieve button objects on a *TStream*.

See also: *TView.Load, TButton.Store*

Done

```
destructor Done; virtual;
```

Override: Never

Disposes of the memory assigned to the button's *Title*, then calls the *Done* destructor inherited from *TView* to dispose of the view.

See also: *TView.Done*

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the button with appropriate palettes for its current state (normal, default, disabled) and positions the label according to the *bfLeftJust* bit in the *Flags* field.

See also: *TButton.DrawState*

DrawState

```
procedure DrawState(Down: Boolean);
```

Draws the button in either a pressed or unpressed state. If *Down* is *True*, *DrawState* draws the button as pressed, otherwise draws the button as unpressed. Draw calls *DrawState* with *Down* set *False* to draw the view. *HandleEvent* calls *DrawState* in response to mouse clicks and drags, depending on the location of the click.

See also: *TButton.Draw*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default palette, *CButton*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Sometimes

Responds to being pressed in any of three ways: mouse clicks on the button, its shortcut key being pressed, or being the default button when a *cmDefault* broadcast arrives. When the user presses a button, the button generates a command event with *PutEvent*, setting *Event.Command* to the value in the button's *Command* field and *Event.InfoPtr* to *@Self*.

Buttons also recognize the broadcast commands *cmGrabDefault* and *cmReleaseDefault*, to become or "unbecome" the default button, as appropriate, and *cmCommandSetChanged*, which causes them to check whether their commands have been enabled or disabled.

Handles all other events by calling the *HandleEvent* method inherited from *TView*.

See also: *TView.HandleEvent*

MakeDefault

```
procedure MakeDefault(Enable: Boolean);
```

This method does nothing if the button is already the default button. Otherwise, notifies its *Owner* of the change in the button's default status with a broadcast command. If *Enable* is *True*, broadcasts the *cmGrabDefault* command, otherwise, sends *cmReleaseDefault*. Redraws the button to reflect the new status.

See also: *TButton.AmDefault, bfDefault*

Press

```
procedure Press; virtual;
```

Override: Sometimes

Called to generate the effect associated with "pressing" a button object. The default method sends an *evBroadcast* event with a command value of *cmRecordHistory* to the button's owner (causing all *THistory* objects to record the contents of the input line objects they control), and then uses *PutEvent* or *Message* to generate an event depending on the value of the *bfBroadcast* flag. You can override *Press* to change the behavior of a button when pressed, but you'll probably want to call the inherited method in your modified *Press* method.

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Override: Seldom

Calls the *SetState* method inherited from *TView* to actually set the state flags, then calls *DrawView* if the button has been made *sfSelected* or *sfActive*. If the button receives focus (*AState* contains *sfFocused*), the button grabs or releases default from the default button by calling *MakeDefault*.

See also: *TView.SetState, TButton.MakeDefault*

Store

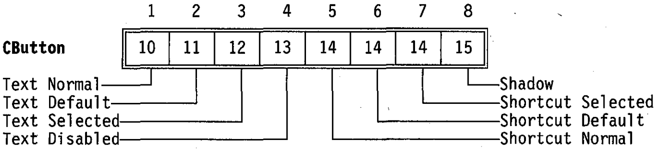
```
procedure Store(var S: TStream);
```

Stores the button object on the stream *S* by calling the *Store* method inherited from *TView*, then calling *S.Write* to store the *Title* and *Command* values. Used in conjunction with *TButton.Load* to save and retrieve *TButton* objects on streams.

See also: *TView.Store, TButton.Load, TStream.Write*

Palette

Button objects use the default palette *CButton* to map onto *CDialog* palette entries 10 through 15.



TByteArray type — Objects

Declaration `TByteArray = array[0..32767] of Byte;`

Function A byte array type for general use in typecasts.

See also *TStringListMaker*

TCharSet type — Objects

Declaration `TCharSet = set of Char;`

Function Filter validator objects use a field of type *TCharSet* to define the legal characters a user can type in a filtered input line.

See also *TFilterValidator.ValidChars*

TChDirDialog object StdDlg

TObject

- ~~Init~~
- Free
- ~~Done~~

TView

- Cursor
- DragMode
- EventMask
- GrowMode
- HelpCtx
- Next
- Options
- Origin
- Owner
- Size
- State

- ~~Init~~
- ~~Load~~
- ~~Done~~
- ~~Awaken~~
- BlockCursor
- CalcBounds
- ~~ChangeBounds~~
- ClearEvent
- CommandEnabled
- ~~DataSize~~
- DisableCommands
- DragView
- ~~Draw~~
- DrawView
- EnableCommands
- ~~EndModal~~
- EventAvail
- ~~Execute~~
- Exposed
- Focus
- GetBounds
- GetClipRect
- GetColor
- GetCommands
- ~~GetData~~
- GetEvent
- GetExtent
- ~~GetHelpCtx~~
- ~~GetPalette~~
- GetPeerViewPtr
- GetState
- GrowTo
- ~~HandleEvent~~
- Hide
- HideCursor
- KeyEvent
- Locate
- MakeFirst
- MakeGlobal
- MakeLocal
- MouseEvent
- MouseInView
- MoveTo
- NextView
- NormalCursor
- Prev
- PrevView
- PutEvent
- PutInFrontOf
- PutPeerViewPtr
- Select
- SetBounds
- SetCommands
- SetCmdState
- SetCursor
- ~~SetData~~
- ~~SetState~~
- Show
- ShowCursor
- ~~SizeLimits~~
- ~~Store~~
- TopView
- ~~Valid~~
- WriteBuf
- WriteChar
- WriteLine
- WriteStr

TGroup

- Buffer
- Current
- Last
- Phase

- ~~Init~~
- ~~Load~~
- ~~Done~~
- Awaken
- ChangeBounds
- ~~DataSize~~
- Delete
- Draw
- EndModal
- EventError
- ExecView
- Execute
- First
- FirstThat
- FocusNext
- ForEach
- ~~GetData~~
- GetHelpCtx
- GetSubViewPtr
- ~~HandleEvent~~
- Insert
- InsertBefore
- Lock
- PutSubViewPtr
- Redraw
- SelectNext
- ~~SetData~~
- ~~SetState~~
- ~~Store~~
- Unlock
- ~~Valid~~

TWindow

- Flags
- Frame
- Number
- Palette
- Title
- ZoomRect

- ~~Init~~
- ~~Load~~
- Done
- Close
- ~~GetPalette~~
- GetTitle
- ~~HandleEvent~~
- InitFrame
- SetState
- SizeLimits
- StandardScrollBar
- ~~Store~~
- Zoom

TDialog

- ~~Init~~
- ~~Load~~
- GetPalette
- ~~HandleEvent~~
- ~~Valid~~

TChDirDialog

- DirInput
- DirList
- OkButton
- ChDirButton

- Init
- Load
- DataSize
- GetData
- HandleEvent
- SetData
- Store
- Valid

TChDirDialog implements a dialog box labeled "Change Directory" that provides an input line to accept a directory name from the user. The input line has a history list, and a directory list box with a vertical scroll bar shows a tree of all available directories.

Fields

ChDirButton

```
ChDirButton: PButton;
```

ChDirButton points to the button object that changes to the directory currently indicated in the input line *DirInput*.

DirInput

```
DirInput: PInputLine;
```

DirInput points to an input line object where the user can type the name of a directory to change to. By default, the input line shows the path name of the directory currently selected in the file tree.

DirList

```
DirList: PDirListBox;
```

DirList points to a list box containing an outline of the directories on the current disk.

OkButton

```
OkButton: PButton;
```

OkButton points to the button object that closes the dialog box.

Methods

Init

```
constructor Init(AOptions: Word; HistoryId: Word);
```

Creates a change-directory dialog box object with the options specified in *AOptions*, and associates the history list designated by *HistoryID* with the directory input line pointed to by *DirInput*. *AOptions* contains a combination of the *cdXXXX* constants.

See also: *cdXXXX* constants

Load

```
constructor Load(var S: TStream);
```

Creates and loads a change-directory dialog box object from the stream *S* by calling the *Load* constructor inherited from *TDialog*, then reading the additional fields defined in *TChDirDialog*.

See also: *TDialog.Load*

DataSize

```
function DataSize: Word; virtual;
```

By default, *DataSize* returns zero. If you derive a new object from *TChDirDialog* that uses *SetData* and *GetData* methods to transfer data to the dialog box, you need to also override *DataSize* to return the size in bytes of the data used by *SetData* and *GetData*.

GetData

```
procedure GetData(var Rec); virtual;
```

By default, *GetData* does nothing. If you derive objects from *TChDirDialog* that have controls whose values need to be read, you need to override *GetData* to copy *DataSize* bytes from *Rec*. If you override *GetData*, you also need to override *DataSize* and *SetData*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles events in the dialog box by first calling its inherited *HandleEvent* from *TDialog* to handle standard dialog box behavior, then processes the commands *cmRevert* and *cmChangeDir* which can be generated by buttons in the dialog box.

SetData

```
procedure SetData(var Rec); virtual;
```

By default, *SetData* does nothing. If you derive objects from *TChDirDialog* that have controls whose values need to be set, you need to override *SetData* to copy *DataSize* bytes into *Rec*. If you override *SetData*, you also need to override *DataSize* and *GetData*.

Store

```
procedure Store(var S: TStream);
```

Stores the dialog box object on the stream *S* by first calling the *Store* method inherited from *TDialog* and then writing the additional fields introduced by *TChDirDialog*.

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

Returns *True* if *Command* is anything other than *cmOK*. If the user pressed the Ok button, generating the *cmOK* command, *Valid* checks the contents of the *DirInput* input line to make sure it names a valid directory. If the directory is valid, *Valid* returns *True*; otherwise, it invokes an "Invalid directory" message box and returns *False*.

TCheckBoxes Dialogs

TObject	TView			TCluster	TCheckBoxes
	Cursor	HelpCtx	Owner	Value	
~~Init~~	DragMode	Next	Size	Sel	Draw
Free	EventMask	Options	State	EnableMask	Mark
~~Done~~	GrowMode	Origin		Strings	Press
	~~Init~~	GetCommands	Prev	Init	
	~~Load~~	~~GetData~~	PrevView	Load	
	~~Done~~	GetEvent	PutEvent	Done	
	Awaken	GetExtent	PutInFrontOf	ButtonState	
	BlockCursor	~~GetHelpCtx~~	PutPeerViewPtr	DataSize	
	CalcBounds	~~GetPalette~~	Select	DrawBox	
	ChangeBounds	GetPeerViewPtr	SetBounds	DrawMultiBox	
	ClearEvent	GetState	SetCommands	GetData	
	CommandEnabled	GrowTo	SetCmdState	GetHelpCtx	
	~~DataSize~~	~~HandleEvent~~	SetCursor	GetPalette	
	DisableCommands	Hide	~~SetData~~	HandleEvent	
	DragView	HideCursor	~~SetState~~	~~Mark~~	
	~~Draw~~	KeyEvent	Show	MovedTo	
	DrawView	Locate	ShowCursor	MultiMark	
	EnableCommands	MakeFirst	SizeLimits	~~Press~~	
	EndModal	MakeGlobal	~~Store~~	SetButtonState	
	EventAvail	MakeLocal	TopView	SetData	
	Execute	MouseEvent	Valid	SetState	
	Exposed	MouseInView	WriteBuf	Store	
	Focus	MoveTo	WriteChar		
	GetBounds	NextView	WriteLine		
	GetClipRect	NormalCursor	WriteStr		
	GetColor				

TCheckBoxes is a specialized cluster of one to 16 controls. Unlike radio buttons, any number of check boxes can be marked independently, so the cluster has boxes checked by default. The user can mark check boxes with mouse clicks, cursor movements, and *Alt+*letter shortcuts. Each check box can be highlighted and toggled on/off (with *Spacebar*). An `X` appears in the box when it is selected.

Other parts of your application typically examine the state of the check boxes to determine which options the user chose (the IDE, for example, has compiler/linker options selected in this way).

Check box clusters often have associated *TLabel* objects to give the user an overview of the clustered options.

Methods

Note that *TCheckBoxes* does not override the *TCluster* constructors, destructor, or event handler. Derived object types, however, might need to override them.

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the *TCheckBoxes* object by calling the inherited *TCluster.DrawBox* method. The default check box is `" [ ] "` when unselected and `" [X] "` when selected.

Note that if the boundaries of the view are sufficiently wide, check boxes may be displayed in multiple columns.

See also: *TCluster.DrawBox*

Mark

```
function Mark(Item: Integer): Boolean; virtual;
```

Override: Seldom

Returns *True* if the *Item*'th bit of *Value* is set, that is, if the *Item*'th check box is marked. You can override this to give a different interpretation of the *Value* field. By default, the items are numbered 0 through 15.

See also: *TCheckBoxes.Press*

Press

```
procedure Press(Item: Integer); virtual;
```

Toggles the *Item*'th bit of *Value*. You can override this to give a different interpretation of the *Value* field. By default, the items are numbered 0 through 15.

See also: *TCheckBoxes.Mark*

Palette

By default, check boxes objects use *CCluster*, the default palette for all cluster objects.

```
                      1    2    3    4
                    ╔════╤════╤════╤════╗
CCluster            ║ 16 │ 17 │ 18 │ 18 ║
                    ╚════╧════╧════╧════╝
Text Normal───────────┘    │    │    └────Shortcut Selected
Text Selected──────────────┘    └─────────Shortcut Normal
```

TCluster

Dialogs

TObject **TView** **TCluster**

TObject
~~Init~~
Free
~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	~~GetData~~	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	~~GetHelpCtx~~	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
~~DataSize~~	~~HandleEvent~~	SetCursor
DisableCommands	Hide	~~SetData~~
DragView	HideCursor	~~SetState~~
Draw	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TCluster
EnableMask
Sel
Strings
Value
Init
Load
Done
ButtonState
DataSize
DrawBox
DrawMultiBox
GetData
GetHelpCtx
GetPalette
HandleEvent
Mark
MovedTo
MultiMark
Press
SetButtonState
SetData
SetState
Store

A cluster is a group of controls that all respond in the same way. *TCluster* is an abstract object type that provides the behavior common to check boxes and radio buttons.

While buttons generate commands and input lines are used to edit strings, clusters enable the user to toggle bits in the *Value* field, which is of type *Longint*. The two standard descendants of *TCluster* use different algorithms when changing *Value*: *TCheckBoxes* simply toggles a bit, while *TRadioButtons* toggles the enabled one and clears the previously selected bit. Both inherit almost all of their behavior from *TCluster*.

Fields

EnableMask

```
EnableMask: Longint;
```

EnableMask contains the enabled state of the first 32 items in a cluster, with each bit corresponding to a cluster item. The lowest order bit controls the first item in the cluster. If the *EnableMask* bit is set, the item is enabled. Clearing the bit disables the corresponding cluster item. Disabled cluster items cannot be pressed. By default, the cluster constructor sets *EnableMask* to $FFFFFFFF, meaning that all items are enabled.

Sel

```
Sel: Integer;                                             Read only
```

The currently selected item of the cluster.

Strings

```
Strings: TStringCollection;                               Read only
```

The list of items in the cluster.

Value

```
Value: Longint;                                           Read only
```

Current value of the control. The actual meaning of this field is determined by the methods developed in the object types derived from *TCluster*.

Methods

Init

```
constructor Init(var Bounds: TRect; AStrings: PSItem);
```

Clears the *Value* and *Sel* fields. The *AStrings* parameter is usually a series of nested calls to the global function *NewSItem*, allowing you to create an entire cluster of radio buttons or check boxes in one constructor call:

```
var Control: PView;
   ⋮
R.Assign(30, 5, 52, 7);
Control := New(PRadioButtons, Init(R,
  NewSItem('~F~orward',
  NewSItem('~B~ackward', nil))));
   ⋮
```

To add additional radio buttons or check boxes to a cluster, just copy the first call to *NewSItem* and replace the title with the desired text. Then add an additional closing parenthesis for each new line you added and the statement will compile without syntax errors.

See also: *TSItem* type

Load

```
constructor Load(var S: TStream);
```

Creates a *TCluster* object by first calling the *Load* constructor inherited from *TView*, then setting the *Value* and *Sel* fields with *S.Read* calls. Finally loads the *Strings* field for the cluster from *S* with *Strings.Load(S)*. Use in conjunction with *TCluster.Store* to save and retrieve *TCluster* objects on a stream.

See also: *TCluster.Store, TView.Load*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the cluster's string memory allocation then destroys the view by calling the *Done* destructor inherited from *TView*.

See also: *TView.Done*

ButtonState

```
function ButtonState(Item: Integer): Boolean;
```

Returns the enabled state of the *Item*th button in the cluster. A *True* value indicates the button is enabled; *False* indicates disabled. Cluster objects call *ButtonState* in their *Draw* and *HandleEvent* methods to ensure that disabled items look different and that users can't interact with disabled items.

See also: *TCluster.EnableMask*

DataSize

```
function DataSize: Word; virtual;
```

Override: Seldom

Returns the size of *Value*. If you derive new types from *TCluster* that change *Value* or add other fields, you need to override *DataSize* to return the size of any data transferred by *GetData* and *SetData*.

See also: *TCluster.GetData, TCluster.SetData*

DrawBox

```
procedure DrawBox(const Icon: String; Marker: Char);
```

Called by the *Draw* methods of descendant types to draw the box in front of the string for each item in the cluster. *Icon* is a 5-character string (`' [ ] '` for check boxes, `' ( ) '` for radio buttons). *Marker* is the character to use to indicate the box has been marked (`'X'` for check boxes, `'•'` for radio buttons).

See also: *TCheckBoxes.Draw, TRadioButtons.Draw*

DrawMultiBox

```
procedure DrawMultiBox(const Icon, Marker: String);
```

Multistate check boxes call *DrawMultiBox* instead of *DrawBox*, passing a string of characters instead of a single character for the marker. The characters in *Marker* correspond to the possible states of the button.

See also: *TCluster.DrawBox, TCluster.MultiMark*

GetData `procedure GetData(var Rec); virtual;`

Override: Seldom Writes the *Value* field to *Rec*. If you derive new object types from *TCluster* that change the *Value* field, you need to override *GetData* in order to work with *DataSize* and *SetData*.

See also: *TCluster.DataSize, TCluster.SetData*

GetHelpCtx `function GetHelpCtx: Word; virtual;`

Override: Seldom Returns the value of *Sel* added to *HelpCtx*. This enables you to have separate help contexts for each item in the cluster. Reserve a range of help contexts equal to *HelpCtx* plus the number of cluster items minus one.

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes Returns a pointer to the default palette, *CCluster*.

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Seldom Calls the *HandleEvent* method inherited from *TView*, then handles all mouse and keyboard events appropriate to this cluster. The user selects individual items within the cluster by mouse click or cursor movement keys (including *Spacebar*). Redraws the cluster to show the newly selected controls.

See also: *TView.HandleEvent*

Mark `function Mark(Item: Integer): Boolean; virtual;`

Override: Always The default *TCluster.Mark* returns *False*. Any new object types derived from *TCluster* must override *Mark* to return *True* if the *Item*th control in the cluster is marked, otherwise, *False*. *Draw* calls *Mark* to determine which items are marked so it can draw the proper box for each item.

MovedTo `procedure MovedTo(Item: Integer); virtual;`

Override: Seldom Moves the selection bar to the *Item*th control of the cluster. *HandleEvent* calls *MovedTo* in response to mouse click or arrow key events.

MultiMark `function MultiMark(Item: Integer): Byte; virtual;`

Returns the mark state of the *Item*th button in a multistate cluster. In regular clusters, each button has only two states, so *Mark* returns a Boolean value. But multistate clusters need to provide more information, so multistate clusters call *MultiMark* instead of *Mark*.

See also: *TCluster.Mark*

Press `procedure Press(Item: Integer); virtual;`

Override: Always

HandleEvent calls *Press* when the user "presses" the *Item*'th control in the cluster by clicking the mouse or pressing Spacebar. *Press* is an abstract method that you must override whenever you derive a new type from *TCluster*.

See also: *TCheckBoxes.Press, TRadioButtons.Press*

SetButtonState

```
procedure SetButtonState(AMask: Longint; Enable: Boolean);
```

Sets or clears the bits in *EnableMask* corresponding to the bits set in *AMask*. If *Enable* is *True*, any bits set in *AMask* are enabled in *EnableMask*; if *Enable* is *False*, the bits are cleared. If disabling individual buttons produces a complete cluster of disabled buttons, *SetButtonState* makes the cluster unselectable.

SetData

```
procedure SetData(var Rec); virtual;
```

Override: Seldom

Reads the *Value* field from the given record and calls *DrawView* to update the cluster. If you derive new types from *TCluster* that change *Value* or add other fields, you must override *SetData* to work with *DataSize* and *GetData*.

See also: *TCluster.DataSize, TCluster.GetData, TView.DrawView*

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Override: Seldom

Calls the *SetState* method inherited from *TView* to set or clear the state bits passed in *AState*, then calls *DrawView* to update the cluster if *AState* is *sfSelected*.

See also: *TView.SetState, TView.DrawView*

Store

```
procedure Store(var S: TStream);
```

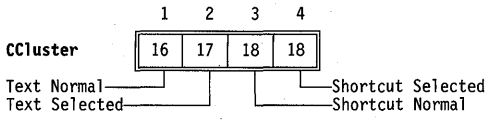
Stores the cluster object on the stream *S* by first calling the *Store* method inherited from *TView*, then writing *Value* and *Sel* to *S*, then storing the cluster's *Strings* field by using its *Store* method. Use in conjunction with *TCluster.Load* to save and retrieve *TCluster* objects on a stream.

See also: *TCluster.Load, TStream.Write*

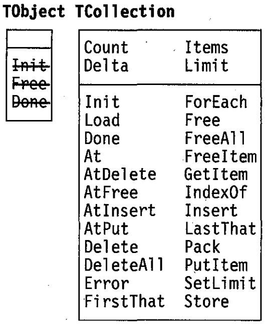
T

Palette

TCluster objects use *CCluster*, the default palette for all cluster objects, to map onto entries 16 through 18 of the standard dialog box palette.



TCollection Objects



TCollection is the base type for implementing any collection of items, including other objects. *TCollection* is a more general concept than the traditional array, set, or list. Collection objects size themselves dynamically at run time and offer a base type for many specialized types such as *TSortedCollection*, *TStringCollection*, and *TResourceCollection*. In addition to methods for adding and deleting items, *TCollection* offers several *iterator* routines that call a procedure or function for each item in the collection.

TCollection assumes that the items in the collection are derived directly or indirectly from *TObject*, so it can call the item's *Done* destructor to dispose of items. If you want to use a collections of items that don't descend from *TObject*, be sure to override the *FreeItem* method to dispose of the item properly. The string collection, for example, implements a collection of dynamic Pascal strings.

Fields

Count

`Count: Integer;` **Read only**

The current number of items in the collection, up to *MaxCollectionSize*. Note that collections index their items based at 0, meaning that *Count* is often higher by 1 than the index of the last item.

See also: *MaxCollectionSize* variable

Delta

`Delta: Integer;` **Read only**

The number of items by which to increase the *Items* list whenever it becomes full. If *Delta* is zero, the collection cannot grow beyond the size set by *Limit*.

☞ Increasing the size of a collection is fairly costly in terms of performance. To minimize the number of times it has to occur, try to set the initial *Limit* to an amount that will encompass all the items you might want to collect, and set *Delta* to a figure that will allow a reasonable amount of expansion.

See also: *Limit*, *TCollection.Init*

Items `Items: PItemList;` **Read only**

A pointer to an array of item pointers.

See also: *TItemList* type

Limit `Limit: Integer;` **Read only**

The currently allocated size (in elements) of the *Items* list.

See also: *Delta*, *TCollection.Init*

Methods

Init `constructor Init(ALimit, ADelta: Integer);`

Constructs a collection object, setting *Limit* to *ALimit* and *Delta* to *ADelta*. The collection allocates enough space to handle *ALimit* items, but the collection can grow in increments of *ADelta* until memory runs out or the number of items reaches *MaxCollectionSize*.

See also: *TCollection.Limit*, *TCollection.Delta*

Load `constructor Load(var S: TStream);`

Constructs and loads a collection from the stream *S*. *Load* calls *GetItem* for each item in the collection.

See also: *TCollection.GetItem*

Done `destructor Done; virtual;`

Override: Often

Deletes and disposes of all items in the collection by calling *FreeAll* and setting *Limit* to 0.

See also: *TCollection.FreeAll*

At `function At(Index: Integer): Pointer;`

Returns a pointer to the item at the position *Index* in the collection. *At* lets you treat a collection as a zero-based indexed array. If *Index* is less than zero or greater than or equal to *Count*, *At* calls *Error* with an argument of *coIndexError*, then returns **nil**.

See also: *TCollection.IndexOf*

AtDelete

```
procedure AtDelete(Index: Integer);
```

Deletes the item at the *Index*'th position from the collection and moves the following items up by one position. Decrements *Count* by 1, but does not reduce the memory allocated to the collection. If *Index* is less than zero or greater than or equal to *Count*, *AtDelete* calls *Error* with an argument of *coIndexError*.

AtDelete does *not* dispose of the deleted item. Use *AtFree* if you need to both delete and dispose of items.

See also: *TCollection.FreeItem, TCollection.Free, TCollection.Delete*

AtFree

```
procedure TCollection.AtFree(Index: Integer);
```

Deletes and disposes of the item at the given *Index*. Equivalent to

```
Item := At(Index);                                    { get pointer to the item }
AtDelete(Index);                               { remove pointer from collection }
FreeItem(Item);                                           { dispose of the item }
```

AtInsert

```
procedure AtInsert(Index: Integer; Item: Pointer);
```

Inserts *Item* at the *Index*'th position and moves any following items down by one position. If *Index* is less than zero or greater than *Count*, *AtInsert* calls *Error* with an argument of *coIndexError* and does not insert *Item*.

If *Count* is equal to *Limit* before inserting the new item, the collection calls *SetLimit* to increase the memory allocated to the collection. If *SetLimit* fails to expand the collection, *AtInsert* calls *Error* with an argument of *coOverflow* and does not insert *Item*.

See also: *TCollection.At, TCollection.AtPut, TCollection.SetLimit*

AtPut

```
procedure AtPut(Index: Integer; Item: Pointer);
```

Replaces the item at position *Index* with *Item*. If *Index* is less than zero or greater than or equal to *Count*, *AtPut* calls *Error* with an argument of *coIndexError*.

See also: *TCollection.At, TCollection.AtInsert*

Delete

```
procedure Delete(Item: Pointer);
```

Deletes the item given by *Item* from the collection. Equivalent to *AtDelete(IndexOf(Item)). Delete* does not dispose of *Item*. If you need to delete and dispose of an item, call *Free*.

See also: *TCollection.AtDelete, TCollection.DeleteAll*

DeleteAll

```
procedure DeleteAll;
```

Deletes all items from the collection by setting *Count* to zero. *DeleteAll* does not dispose of the items in the collection.

See also: *TCollection.Delete, TCollection.AtDelete*

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Override: Sometimes

Called by various other collection object methods when they encounter errors. By default, this method produces a run-time error of (212 – *Code*), where *Code* is one of the *coXXXX* constants, indicating the nature of the error. You can override *Error* to notify the user of details of the error or recover from the error without terminating the program.

See also: *coXXXX* collection constants

FirstThat

```
function FirstThat(Test: Pointer): Pointer;
```

FirstThat applies a Boolean function, given by the function pointer *Test*, to each item in the collection until *Test* returns *True*. Returns a pointer to the item for which *Test* returned *True*, or **nil** if *Test* returned *False* for all items. *Test* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean* value. For example

```
function Matches(Item: Pointer): Boolean; far;
```

☞ The *Test* function *cannot* be a global function.

Assuming that *List* is a *TCollection*, the statement

```
P := List.FirstThat(@Matches);
```

corresponds to

```
I := 0;
while (I < List.Count) and not Matches(List.At(I)) do Inc(I);
if I < List.Count then P := List.At(I) else P := nil;
```

See also: *TCollection.LastThat, TCollection.ForEach*

ForEach

```
procedure ForEach(Action: Pointer);
```

ForEach applies an action, given by the procedure pointer *Action*, to each item in the collection. *Action* must point to a **far** local procedure taking one *Pointer* parameter. For example

```
procedure PrintItem(Item: Pointer); far;
```

☞ The *Action* procedure *cannot* be a global procedure.

Assuming that *List* is a *TCollection*, the statement

```
List.ForEach(@PrintItem);
```

corresponds to

```
for I := 0 to List.Count - 1 do PrintItem(List.At(I));
```

See also: *TCollection.FirstThat, TCollection.LastThat*

Free

```
procedure Free(Item: Pointer);
```

Deletes *Item* from the collection and disposes of *Item*. Equivalent to

```
Delete(Item);                                { remove pointer from collection }
FreeItem(Item);                                              { dispose of Item }
```

See also: *TCollection.FreeItem, TCollection.Delete*

FreeAll

```
procedure FreeAll;
```

Deletes and disposes of all items in the collection. To remove all items from the collection without disposing of them, call *DeleteAll*.

See also: *TCollection.DeleteAll*

FreeItem

Override: Sometimes

```
procedure FreeItem(Item: Pointer); virtual;
```

The *FreeItem* method must dispose the given *Item*. By default, *FreeItem* assumes that *Item* points to a descendant of *TObject*, and thus calls *Item*'s *Done* destructor:

```
if Item <> nil then Dispose(PObject(Item), Done);
```

Descendant collection objects that don't use descendants of *TObject* as their items, such as string collections, must override *FreeItem* to dispose of the given *Item*.

☞ *FreeItem* is called by *Free* and *FreeAll*, but it should never be called directly.

See also: *TCollection.Free, TCollection.FreeAll*

GetItem

Override: Sometimes

```
function TCollection.GetItem(var S: TStream): Pointer; virtual;
```

Reads an item from the stream *S*. By default, *GetItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Get* to load the item:

```
GetItem := S.Get;
```

Descendant collection objects that don't use descendants of *TObject* as their items, such as string collections, need to override *GetItem* to read the appropriate item from the stream and return a pointer to it.

Load calls *GetItem* to read each item in the collection. This method can be overridden but should not be called directly.

See also: *TStream.Get, TCollection.Load, TCollection.Store*

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Override: Never

Returns the index of *Item*. *IndexOf* is the converse operation to *At*. If *Item* is not in the collection, *IndexOf* returns –1.

See also: *TCollection.At*

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Override: Never

Inserts *Item* into the collection, and adjusts other indexes if necessary. By default, *Insert* adds *Item* to the end of the collection by calling *AtInsert(Count, Item)*. Descendant collection types, such as sorted collections, might insert items at other points.

See also: *TCollection.AtInsert*

LastThat

```
function LastThat(Test: Pointer): Pointer;
```

LastThat applies a Boolean function, given by the function pointer *Test*, to each item in the collection in reverse order until *Test* returns *True*. Returns a pointer to the item for which *Test* returned *True*, or **nil** if *Test* returned *False* for all items. *Test* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean*. For example

```
function Matches(Item: Pointer): Boolean; far;
```

☞ The *Test* function *cannot* be a global function.

Assuming that *List* is a *TCollection*, the statement

```
P := List.LastThat(@Matches);
```

corresponds to

```
I := List.Count - 1;
while (I >= 0) and not Matches(List.At(I)) do Dec(I);
if I >= 0 then P := List.At(I) else P := nil;
```

See also: *TCollection.FirstThat, TCollection.ForEach*

Pack

```
procedure Pack;
```

Deletes all **nil** pointers in the collection.

See also: *TCollection.Delete*

PutItem

```
procedure PutItem(var S: TStream; Item: Pointer); virtual;
```

Override: Sometimes

Writes *Item* to the stream *S*. By default, *PutItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Put* to store the item:

```
S.Put(Item);
```

Descendant collection types that don't use descendants of *TObject* as their items, such as string collections, must override *PutItem* to write *Item* to the stream.

Store calls *PutItem* for each item in the collection. This method can be overridden but should not be called directly.

See also: *TCollection.GetItem, TCollection.Store, TCollection.Load*

SetLimit

```
procedure SetLimit(ALimit: Integer); virtual;
```

Override: Seldom

Expands or shrinks the collection by changing the memory allocated for items to handle *ALimit* items. If *ALimit* is less than *Count*, it is set to *Count*, and if *ALimit* is greater than *MaxCollectionSize*, it is set to *MaxCollectionSize*. Then, if *ALimit* is different from the current *Limit*, *SelLimit* allocates a new *Items* array that holds *ALimit* elements, copies the old *Items* into the new array, and disposes of the old array.

See also: *TCollection.Limit, TCollection.Count, MaxCollectionSize* variable

Store

```
procedure Store(var S: TStream);
```

Stores the collection and all its items on the stream *S*. *Store* calls *PutItem* for each item in the collection.

See also: *TCollection.PutItem*

TColorDialog ColorSel

The color dialog box is a specialized dialog box titled 'Colors' which enables users to change palette colors throughout an application while viewing the selected color combinations in the dialog box.

TColorDialog uses a number of specialized views, including *TColorItem, TColorGroup, TColorSelector,* and *TColorDisplay.* For a complete explanation of how to use the color dialog box, see Chapter 14.

Fields

BakLabel

```
BakLabel: PLabel;
```

Points to the label for the background color selector.

BakSel

```
BakSel: PColorSelector;
```

Points to the background color selector for the dialog box.

Display

```
Display: PColorDisplay;
```

Points to the color display object for the dialog box. The color display shows text in the currently selected colors.

ForLabel

```
ForLabel: PLabel;
```

Points to the label for the foreground color selector.

ForSel

```
ForSel: PColorSelector;
```

Points to the foreground color selector for the dialog box.

GroupIndex

```
GroupIndex: Byte;
```

Indicates which group in the color group list was most recently focused.

Groups

```
Groups: PColorGroupList;
```

Points to the color group list for the dialog box. The color group list shows all the groups of items for which the user can select colors.

MonoLabel

```
MonoLabel: PLabel;
```

Points to the label for the monochrome attribute selector.

MonoSel

```
MonoSel: PMonoSelector;
```

Points to the selector for monochrome attributes.

Pal

```
Pal: TPalette;
```

Holds a copy of the palette being modified.

Methods

Init

```
constructor Init(APalette: TPalette; AGroups: PColorGroup);
```

Creates a 62-column, 19-line dialog box with the title 'Colors' by calling the *Init* constructor inherited from *TDialog*, adding *ofCentered* to the *Options* flags. Sets *Pal* to *APalette*. Creates and inserts a color group list linked to *AGroups* and a color item list linked to *AGroups^.Items*, with their associated scroll bars and labels. Creates and inserts color selectors for foreground and background colors, assigning them to *ForSel* and *BakSel*, and creates and inserts labels for the selectors, assigning them to *ForLabel* and *BakLabel*. Creates and inserts a hidden monochrome selector and its label. Creates and inserts Ok and Cancel buttons and gives the focus to the color group list.

See also: *TDialog.Init*

Load

```
constructor Load(var S: TStream);
```

Creates and loads a color dialog box from the stream *S* by first calling the *Load* constructor inherited from *TDialog*, then reading subview pointers for the subviews introduced by *TColorDialog*, and finally reading the palette.

See also: *TDialog.Load*

DataSize

```
function DataSize: Word; virtual;
```

Returns the size of a palette, which is the amount of data passed to or from a color dialog box by *SetData* or *GetData*.

See also: *TColorDialog.GetData*, *TColorDialog.SetData*

GetData

```
procedure GetData(var Rec); virtual;
```

Calls *GetIndexes* to copy the selected items in each group into *ColorIndexes*, then copies *DataSize* bytes from *Rec* into *Pal*, by typecasting *Rec* into type *TPalette*.

See also: *TColorDialog.DataSize*

GetIndexes

```
procedure GetIndexes(var Colors: PColorIndex);
```

Sets the color indexes in *Colors* to the indexed colors in each group in *Groups*. *TColorDialog.GetData* uses *GetIndexes* to set the indexes in *ColorIndexes* to the indexes in each group in *Groups*. By storing *ColorIndexes* on a stream, you can then restore the state of the dialog box using *LoadIndexes* and *SetData*.

See also: *ColorIndexes* variable

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Calls the *HandleEvent* method inherited from *TDialog* to deal with standard dialog behavior, then responds to broadcasts of *cmNewColorIndex* commands by setting the new color in the dialog box's color display.

See also: *TDialog.HandleEvent*, *TColorDisplay.SetColor*

SetData `procedure SetData(var Rec); virtual;`

Copies *DataSize* bytes from *Rec* to *Pal*, specifically by typecasting *Rec* to type *TPalette*. If *ShowMarkers* is *True*, displays the monochrome selector and hides the color selectors.

See also: *TColorDialog.DataSize*, *ShowMarkers* variable

SetIndexes `procedure SetIndexes(var Colors: PColorIndex);`

Sets the indexes in each color group in *Groups* to the corresponding index from *Colors*. *TColorDialog.SetData* calls *SetIndexes* to set the color group indexes from *ColorIndexes*, restoring the selected items from the last time *ColorIndexes* was set from *Groups*.

See also: *TColorDialog.SetData*

Store `procedure Store(var S: TStream);`

Writes the color dialog box to the stream *S* by first calling the *Store* method inherited from *TDialog*, then writing pointers for the subviews introduced by *TColorDialog*, and finally writing the palette stored in *Pal*.

See also: *TDialog.Store*

TColorDisplay object — ColorSel

TColorDisplay is a simple view that shows a given text string in the color selected in its dialog box's color selectors. Color selection dialog boxes use a color display view to show the user what selected color combinations look like.

Details of *TColorDisplay*'s fields and methods are in the online Help.

TColorGroup type — ColorSel

Declaration

```
TColorGroup = record
  Name:  PString;
  Index: Byte;
  Items: PColorItem;
  Next:  PColorGroup;
end;
```

Function A color group defines a named group of related items for which a user can select colors. *Name* holds the name of the group, *Index* holds the ordinal position of the color in the color list, and *Items* points to the first item in a linked list of color items. *Next* points to the next item in a linked list of color groups.

A color dialog box contains a group list box that uses as its list a linked list of *TColorGroup* records.

Use the *ColorGroup* function to create and initialize color group records.

See also *ColorGroup* function

TColorGroupList object — ColorSel

A color group list is a specialized list box object that provides a scrollable list of named color groups for selection in a color selection dialog box. *TColorGroupList* behaves like a regular list box, but its list is a linked list of *TColorGroup* records.

Details of *TColorGroupList*'s fields and methods are in the online Help.

TColorIndex type — ColorSel

Declaration

```
TColorIndex = record
  GroupIndex: Byte;
  ColorSize: Byte;
  ColorIndex: array[0..255] of Byte;
end;
```

Function Color selection dialog boxes use *TColorIndex* records to save the ordinal position of the focused items in the color group list and the color item list, enabling the dialog box to restore its previous state when loaded. You never need to use this type directly. It's used by the *LoadIndexes* and *StoreIndexes* procedures.

See also *LoadIndexes* procedure, *StoreIndexes* procedure

TColorItem type — ColorSel

Declaration

```
TColorItem = record
  Name: PString;
  Index: Byte;
  Next: PColorItem;
end;
```

Function A color item defines a named item in a group for which a user can select colors. *Name* holds the name of the color item, and *Index* holds the index of the application color palette entry that defines the color of the item. *Next* points to the next item in a linked list of color items.

A color dialog box contains an item list box that builds its list from a linked list of *TColorItem* records.

Use the *ColorItem* function to create and initialize new *TColorItem* records.

TColorItemList object ColorSel

A color item list is a specialized descendant of *TListViewer* that provides a list of the items in a color group for which a user can select colors. The list for a color item list comes from the *Items* field of a *TColorGroup* record. Color Selection dialog boxes use a color item list to enable the user to pick groups of color items for color selection.

Details of *TColorItemList*'s fields and methods are in the online Help.

TColorSel type ColorSel

Declaration

```
TColorSel = (csBackground, csForeground);
```

Function Color selector objects use the *TColorSel* enumerated type to specify what kind of selector it is, background or foreground.

See also *TColorSelector.SelType*

TColorSelector object ColorSel

Color selector objects display the colors available for a given view. There are two variations, one for background colors and one for foreground colors. Color selection dialog boxes use one of each kind to show the available color choices as well as the currently-selected colors.

Details of *TColorSelector*'s fields and methods are in the online Help.

TCommandSet type Views

Declaration

```
TCommandSet = set of Byte;
```

Function *TCommandSet* is useful for holding arbitrary sets of up to 256 commands. It allows for simple testing whether a given command meets certain criteria in event handling routines and lets you establish command masks.

For example, *TView's* methods *EnableCommands, DisableCommands, GetCommands,* and *SetCommands* all take arguments of type *TCommandSet*. You can declare and initialize command sets using the Pascal set syntax:

```
CurCommandSet: TCommandSet = [0..255] - [cmZoom, cmClose, cmResize, cmNext];
```

See also *cmXXXX, TView.DisableCommands, TView.EnableCommands, TViewGetCommands, TView.SetCommands.*

TDesktop App

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
~~Awaken~~
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
~~EndModal~~
EventAvail
~~Execute~~
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
~~GetHelpCtx~~
GetPalette
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
SizeLimits
~~Store~~
TopView
~~Valid~~
WriteBuf
WriteChar
WriteLine
WriteStr

TGroup

Buffer
Current
Last
Phase

~~Init~~
Load
Done
Awaken
ChangeBounds
DataSize
Delete
Draw
EndModal
EventError
ExecView
Execute
First
FirstThat
FocusNext
ForEach
GetData
GetHelpCtx
GetSubViewPtr
~~HandleEvent~~
Insert
InsertBefore
Lock
PutSubViewPtr
Redraw
SelectNext
SetData
SetState
Store
Unlock
Valid

TDeskTop

Background
TileColumnsFirst

Init
Cascade
HandleEvent
InitBackground
Tile
TileError

The desktop is a simple group that owns the background view upon which the application's windows and other views appear. *TDesktop* represents the desktop area of the screen between the top menu bar and bottom status line.

☞ The desktop object has one new field in version 2.0, allowing you to specify default tiling behavior.

Fields

Background

```
Background: PBackground
```

Points to the desktop's background object.

TileColumnsFirst

```
TileColumnsFirst: Boolean;
```

TileColumnsFirst controls whether tiling windows on the desktop favors windows stacked vertically or horizontally. By default, *TileColumnsFirst* is *False*, maintaining the behavior of version 1.0, which favors stacking windows vertically. Setting the field *True* will favor horizontal tiling, so for example tiling two windows places them side-by-side, rather than one above the other.

See also: *TDesktop.Tile*

Methods

Init

```
constructor Init(var Bounds: TRect);
```

Creates a desktop group with size *Bounds* by calling the *Init* constructor inherited from *TGroup*. Sets *GrowMode* to *gfGrowHiX* + *gfGrowHiY*. Calls *InitBackground* to construct a background view, and if *Background* is non-**nil**, inserts it.

See also: *TDesktop.InitBackground, TGroup.Init, TGroup.Insert*

Cascade

```
procedure Cascade(var R: TRect);
```

Redisplays all tileable windows owned by the desktop in cascaded format. The first tileable window in Z-order (the window "in back") is zoomed to fill the desktop, and each succeeding window fills a region beginning one line lower and one space farther to the right than the one before. The active window appears "on top," as the smallest window.

If the desktop is unable to cascade the windows, it leaves them in place and calls *TileError*

See also: *ofTileable, TDesktop.Tile, TDesktop.TileError*

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Seldom

Calls the *HandleEvent* method inherited from *TGroup*, then takes care of the commands *cmNext* (usually the hot key *F6*) and *cmPrevious* by cycling through the windows (starting with the currently selected view) owned by the desktop.

See also: *TGroup.HandleEvent*, *cmXXXX* command constants

InitBackground `procedure InitBackground; virtual;`

Override: Sometimes

Constructs a background view for the desktop and assigns it to *Background*. *TDesktop.Init* calls this method, then inserts *Background* into the desktop. Descendant objects can change the background type by overriding this method and assigning a different background object to *Background*.

See also: *TDesktop.Init*

Tile `procedure Tile(var R: TRect);`

Redisplays all *ofTileable* views owned by the desktop in tiled format. If the desktop cannot arrange the windows, it leaves them in place and calls *TileError*.

See also: *TDesktop.Cascade*, *ofTileable*, *TDesktop.TileError*

TileError `procedure TileError; virtual;`

Override: Sometimes

TileError is called if an error occurs during *Tile* or *Cascade*. By default, *TileError* does nothing. You might want to override *TileError* to notify the user that the application is unable to rearrange the windows.

See also: *TDesktop.Tile*, *TDesktop.Cascade*

TDialog

Dialogs

TObject	TView	TGroup	TWindow	TDialog
	Cursor DragMode EventMask GrowMode HelpCtx Next Options Origin Owner Size State	Buffer Current Last Phase	Flags Frame Number Palette Title ZoomRect	
~~Init~~ Free ~~Done~~	~~Init~~ ~~Load~~ ~~Done~~ ~~Awaken~~ BlockCursor CalcBounds ~~ChangeBounds~~ ClearEvent CommandEnabled ~~DataSize~~ DisableCommands DragView ~~Draw~~ DrawView EnableCommands ~~EndModal~~ EventAvail ~~Execute~~ Exposed Focus GetBounds GetClipRect GetColor GetCommands ~~GetData~~ GetEvent GetExtent ~~GetHelpCtx~~ ~~GetPalette~~ GetPeerViewPtr GetState GrowTo ~~HandleEvent~~ Hide HideCursor KeyEvent Locate MakeFirst MakeGlobal MakeLocal MouseEvent MouseInView MoveTo NextView NormalCursor Prev PrevView PutEvent PutInFrontOf PutPeerViewPtr Select SetBounds SetCommands SetCmdState SetCursor ~~SetData~~ ~~SetState~~ Show ShowCursor ~~SizeLimits~~ ~~Store~~ TopView ~~Valid~~ WriteBuf WriteChar WriteLine WriteStr	~~Init~~ ~~Load~~ ~~Done~~ Awaken ChangeBounds DataSize Delete Draw EndModal EventError ExecView Execute First FirstThat FocusNext ForEach GetData GetHelpCtx GetSubViewPtr ~~HandleEvent~~ Insert InsertBefore Lock PutSubViewPtr Redraw SelectNext SetData ~~SetState~~ ~~Store~~ Unlock ~~Valid~~	~~Init~~ ~~Load~~ Done Close ~~GetPalette~~ GetTitle ~~HandleEvent~~ InitFrame SetState SizeLimits StandardScrollBar Store Zoom	Init Load GetPalette HandleEvent Valid

TDialog is a specialialized descendant of *TWindow*, specifically designed for modal use and for holding controls. Dialog box objects differ from windows by default in the following ways:

- *GrowMode* is zero; that is, dialog boxes are not growable.
- Flag masks *wfMove* and *wfClose* are set; that is, dialog boxes are moveable and closable (a close icon is provided).
- The *TDialog* event handler calls *TWindow.HandleEvent*, but also handles the special cases of *Esc* and *Enter* key responses. *Esc* generates a *cmCancel* command, while *Enter* generates the *cmDefault* command.
- Dialog boxes have no window numbers.
- The *TDialog.Valid* method returns *True* on *cmCancel*, otherwise it calls its *TGroup.Valid*.

T

☞ In version 2.0, dialog boxes now support blue and cyan palettes in addition to the default gray palette. Previous versions of *TDialog* ignored the *Palette* field. Dialog box objects can now specify a palette by assigning *dpXXXX* constants to *Palette*.

Methods

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr);
```

Creates a dialog box with the given size and title by calling the *Init* constructor inherited from *TWindow*, passing *Bounds*, *ATitle*, and *wnNoNumber*. Sets *GrowMode* to 0, and *Flags* to *wfMove* + *wfClose*. This means that, by default, dialog boxes can move and close (via the close icon) but cannot grow (resize).

☞ *TDialog* does not define its own destructor, but uses *Close* and *Done* inherited via *TWindow*, *TGroup*, and *TView*.

See also: *TWindow.Init*

Load

```
constructor Load(var S: TStream);
```

Reads a dialog box object from the stream *S* by first calling the *Load* constructor inherited from *TWindow*, then updating the palette information as needed. *Load* checks the *Options* flags of the loaded dialog box, and if the *ovVersion* bits are *ofVersion10*, *Load* sets the *Palette* field to *dpGrayDialog* and updates *Options* to include *ofVersion20*.

See also: *TWindow.Load*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Sometimes

Handles most events by calling the *HandleEvent* method inherited from TWindow, then handles *Enter* and *Esc* key events specially. In particular, *Esc* generates a *cmCancel* command, and the *Enter* key broadcasts a *cmDefault* command.

HandleEvent also handles *cmOK*, *cmCancel*, *cmYes*, and *cmNo* command events by ending the modal state of the dialog box.

See also: *TWindow.HandleEvent*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Seldom

Returns a pointer to the palette given by the palette index in the *Palette* field. Table 19.39 shows the palettes returned for the different values of *Palette*.

Table 19.39 Dialog box palettes returned based on Palette

Palette field	Palette returned
dpBlueDialog	*CBlueDialog*
dpCyanDialog	*CCyanDialog*
dpGrayDialog	*CGrayDialog*

In version 1.0, *GetPalette* always returned a pointer to the default palette, *CDialog*. For backward compatibility, *CDialog* is still available. The default palette in version 2.0, *CGrayDialog,* is identical to *CDialog*.

See also: *TWindow.Palette*

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

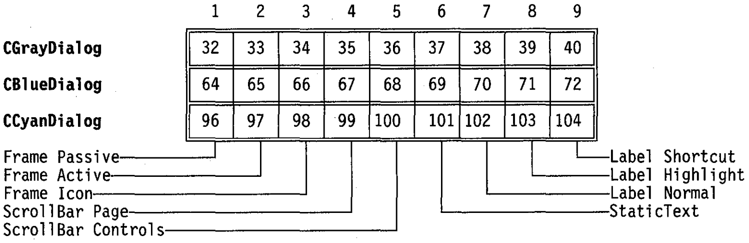
Override: Seldom

Returns *True* if the command given is *cmCancel* or if the *Valid* method inherited from *TWindow* returns *True.*

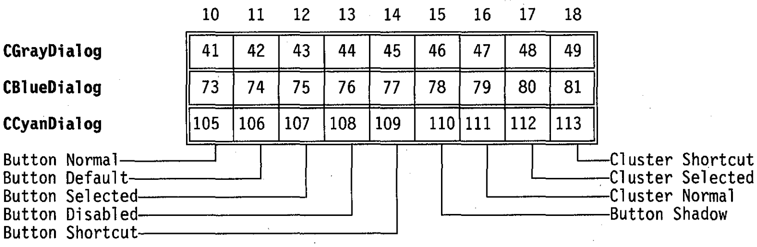
See also: *TGroup.Valid*

Palette

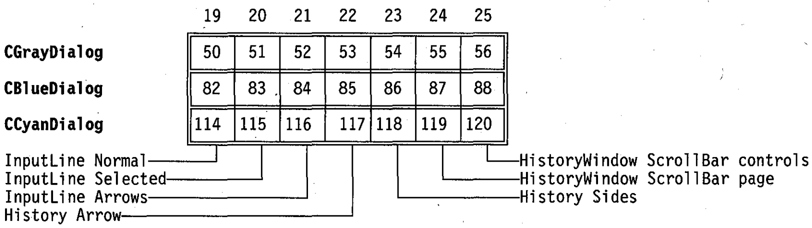
Dialog box objects use different palettes, depending on the value of the *Palette* field. Note that the *CDialog* palette used by all dialog boxes in version 1.0 is identical to the default dialog box palette, *CGrayDialog,* in version 2.0.



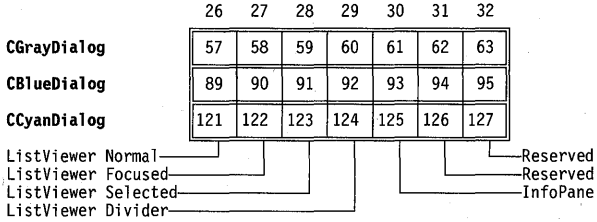
	1	2	3	4	5	6	7	8	9
CGrayDialog	32	33	34	35	36	37	38	39	40
CBlueDialog	64	65	66	67	68	69	70	71	72
CCyanDialog	96	97	98	99	100	101	102	103	104



	10	11	12	13	14	15	16	17	18
CGrayDialog	41	42	43	44	45	46	47	48	49
CBlueDialog	73	74	75	76	77	78	79	80	81
CCyanDialog	105	106	107	108	109	110	111	112	113



	19	20	21	22	23	24	25
CGrayDialog	50	51	52	53	54	55	56
CBlueDialog	82	83	84	85	86	87	88
CCyanDialog	114	115	116	117	118	119	120



	26	27	28	29	30	31	32
CGrayDialog	57	58	59	60	61	62	63
CBlueDialog	89	90	91	92	93	94	95
CCyanDialog	121	122	123	124	125	126	127

See also: *GetPalette* method for each object type

TDirCollection object — StdDlg

TDirCollection is a collection of *TDirEntry* records used by *TDirListBox.*

Details of *TDirCollection*'s fields and methods are in the online Help.

TDirEntry type — StdDlg

Declaration

```
TDirEntry = record
  DisplayText: PString;
  Directory: PString;
end;
```

Function *TDirEntry* is a simple record type holding directory path strings and descriptions. These records are used in *TDirCollection* objects to hold directory information for the change-directory dialog box.

See also *TDirCollection* object

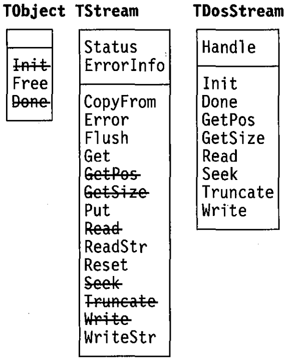
TDirListBox object — StdDlg

TDirListBox is a specialized kind of list box that displays a tree of directories stored in a *TDirCollection* object, for use in the *TChDirDialog*.

By default, the directories appear in a single column with a vertical scroll bar.

Details of *TDirListBox*'s fields and methods are in the online Help.

TDosStream Objects



TDosStream is a specialized stream derivative implementing unbuffered DOS file streams. The constructor lets you create or open a DOS file by specifying its name and access mode: *stCreate*, *stOpenRead*, *stOpenWrite*, or *stOpen*. The one additional field of *TDosStream* is *Handle*, the traditional DOS file handle used to access an open file.

Most applications will use the buffered derivative of *TDosStream*, *TBufStream*, rather than an unbuffered DOS stream. *TDosStream* overrides all the abstract methods of *TStream* except for *TStream.Flush*.

Field

Handle

```
Handle: Word                                                    Read only
```

Handle is the DOS file handle used to access an open file stream.

Methods

Init

```
constructor Init(FileName: FNameStr; Mode: Word);
```

Creates a DOS file stream with the given *FileName* and access mode. If successful, the *Handle* field is set with the DOS file handle. If opening the file fails, *Init* calls *Error* with an argument of *stInitError*.

The *Mode* argument must be one of the values *stCreate, stOpenRead, stOpenWrite*, or *stOpen*. These constant values are explained in this chapter under *"stXXXX* constants."

Done

```
destructor Done; virtual;
```

Override: Never

Closes and disposes of the DOS file stream.

See also: *TDosStream.Init*

GetPos

```
function GetPos: Longint; virtual;
```

Override: Never

Returns the stream's current position. The first position in the stream is 0.

See also: *TDosStream.Seek*

GetSize

```
function GetSize: Longint; virtual;
```

Override: Never

Returns the size in bytes of the stream.

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Override: Never

Reads *Count* bytes from the stream, starting at the current position, into the *Buf* buffer.

See also: *TDosStream.Write, stReadError*

Seek

```
procedure Seek(Pos: Longint); virtual;
```

Override: Never

Sets the current position to *Pos* bytes from the beginning of the stream. The first position in the stream is 0.

See also: *TDosStream.GetPos, TDosStream.GetSize*

Truncate

```
procedure Truncate; virtual;
```

Override: Never

Deletes all data on the stream from the current position to the end.

See also: *TDosStream.GetPos, TDosStream.Seek*

Write

```
procedure Write(var Buf; Count: Word); virtual;
```

Writes *Count* bytes from the *Buf* buffer to the stream, starting at the current position.

See also: *TDosStream.Read, stWriteError*

TDrawBuffer type — Views

Declaration

```
TDrawBuffer = array[0..MaxViewWidth-1] of Word;
```

Function The *TDrawBuffer* type is used to declare buffers for *Draw* methods. Typically, data and attributes are stored and formatted line by line in a *TDrawBuffer*, then written to the screen:

```
var
  B: TDrawBuffer;
begin
  MoveChar(B, ' ', GetColor(1), Size.X);          { fill buffer with spaces }
  WriteLine(0, 0, Size.X, Size.Y, B);              { write buffer to screen }
end;
```

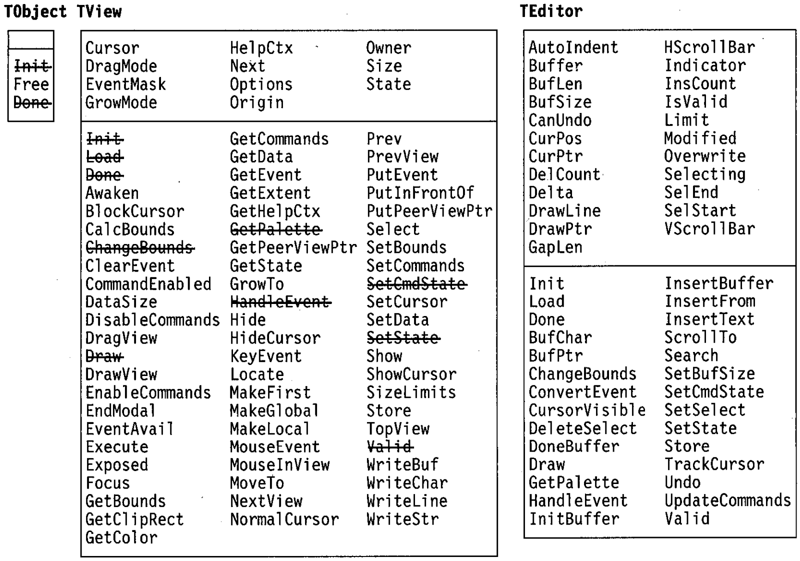
See also *TView.Draw, MoveBuf, MoveChar, MoveCStr, MoveStr*

TEditBuffer type Editors

Declaration `TEditBuffer = array[0..65519] of Char;`

TEditBuffer defines an array of characters for editing. *TEditor* and *TMemo* objects use *TEditBuffer* arrays to hold their edit buffers.

TEditor object Editors



TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
Awaken
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
DataSize
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
GetData
GetEvent
GetExtent
GetHelpCtx
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
~~SetCmdState~~
SetCursor
SetData
~~SetState~~
Show
ShowCursor
SizeLimits
Store
TopView
~~Valid~~
WriteBuf
WriteChar
WriteLine
WriteStr

TEditor

AutoIndent
Buffer
BufLen
BufSize
CanUndo
CurPos
CurPtr
DelCount
Delta
DrawLine
DrawPtr
GapLen
HScrollBar
Indicator
InsCount
IsValid
Limit
Modified
Overwrite
Selecting
SelEnd
SelStart
VScrollBar

Init
Load
Done
BufChar
BufPtr
ChangeBounds
ConvertEvent
CursorVisible
DeleteSelect
DoneBuffer
Draw
GetPalette
HandleEvent
InitBuffer
InsertBuffer
InsertFrom
InsertText
ScrollTo
Search
SetBufSize
SetCmdState
SetSelect
SetState
Store
TrackCursor
Undo
UpdateCommands
Valid

TEditor implements a simple, fast 64K editor view for use in Turbo Vision applications. It provides mouse support, undo, clipboard cut, copy, and paste, automatic modes for indenting and overwriting, key binding, and search and replace. You can use editor views for editing files and for multiple-line memo fields in dialog boxes or forms.

Several other objects such as *TMemo* and *TFileEditor* provide immediately useful editor objects, but they all derive their basic functions from *TEditor*.

Use of editor objects is described fully in Chapter 15, "Editor and text views."

Fields

AutoIndent

```
AutoIndent: Boolean;
```

If *AutoIndent* is *True*, the editor automatically indents typed lines to the column where the preceding line starts; otherwise new lines start at the leftmost column.

Buffer

```
Buffer: PEditBuffer;
```

Points to the buffer where the editor object holds the text currently being edited. The buffer can hold up to 64K characters.

See also: *TEditBuffer* type

BufLen

```
BufLen: Word;
```

BufLen holds the numbers of characters between the start of the buffer and the current cursor position.

BufSize

```
BufSize: Word;
```

BufSize is the size in bytes of the text buffer.

CanUndo

```
CanUndo: Boolean;
```

CanUndo indicates whether the editor supports undo. By default, *TEditor.Init* sets *CanUndo* to *True*, indicating that the editor can undo changes.

CurPos

```
CurPos: TPoint;
```

CurPos is the line/column position of the cursor within the file. *Cursor.X* gives the current column and *Cursor.Y* gives the current line.

CurPtr

```
CurPtr: Word;
```

CurPtr is the position of the cursor in the edit buffer.

DelCount

```
DelCount: Word;
```

Number of characters in the end of the gap that were deleted from the text. *DelCount* is used to undo the deletions.

Delta `Delta: TPoint;`

Delta is the top line and leftmost column shown in the view. *Delta.X* is the leftmost visible column and *Delta.Y* is the topmost visible line.

DrawLine `DrawLine: Integer;`

DrawLine is the column position on the screen where inserted characters are drawn. The *Draw* method uses *DrawLine* to optimize what parts of the view it redraws.

DrawPtr `DrawPtr: Word;`

DrawPtr is the buffer position of the cursor, used by *Draw*.

GapLen `GapLen: Word;`

GapLen is the size of the "gap" between the text before the cursor and the text after the cursor. The gap is explained in Chapter 15.

HScrollBar `HScrollBar: PScrollBar;`

Points to the horizontal scroll bar object associated with the editor. A **nil** indicates there is no such scroll bar.

Indicator `Indicator: PIndicator;`

Points to the indicator object associated with the editor. An indicator object shows the line and column currently being edited.

See also: *TIndicator* object

InsCount `InsCount: Word;`

Number of characters inserted into the text since the last cursor movement. *InsCount* is used to undo the insertions.

IsValid `IsValid: Boolean;`

Holds *True* if the view is valid. *IsValid* is used by the *Valid* method.

See also: *TEditor.Valid*

Limit `Limit: TPoint;`

Limit contains the maximum width and length of the text. *Limit.X* gives the length of the longest line, while *Limit.Y* gives the number of lines in the file.

Modified `Modified: Boolean;`

Modified contains *True* if the edit buffer has changed.

Overwrite

```
Overwrite: Boolean;
```

If *Overwrite* is *True*, typed characters replace existing characters in the buffer; otherwise, the editor inserts typed characters.

Selecting

```
Selecting: Boolean;
```

Selecting is *True* if the user is selecting a block, such as after marking the start of the block, but before marking the end. At all other times, *Selecting* is *False*.

SelEnd

```
SelEnd: Word;
```

SelEnd is the position in the buffer of the end of selected text.

SelStart

```
SelStart: Word;
```

SelStart is the position in the buffer of the start of selected text.

VScrollBar

```
VScrollBar: PScrollBar;
```

Points to the vertical scroll bar object associated with the editor. A **nil** indicates there is no such scroll bar.

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar;
AIndicator: PIndicator; ABufSize: Word);
```

Creates a view with the boundaries specified in *Bounds* by calling the *Init* constructor inherited from *TView*. Sets *GrowMode* to *gfGrowHiX* + *gfGrowHiY*, *Options* to *Options* **or** *ofSelectable*, and *EventMask* to *evMouseDown* + *evKeyDown* + *evCommand* + *evBroadcast*. Shows the cursor in the editor, and assigns the fields *HScrollBar*, *VScrollBar*, *Indicator*, and *BufSize* to the values passed in the parameters. Sets *CanUndo* to *True*.

Allocates an edit buffer by calling *InitBuffer*. If the allocation fails, *Init* calls *EditorDialog* to display an "Out of memory" warning and sets the buffer size to zero. Calls *SetBufLen(0)* to initialize the buffer.

See also: *TView.Init*, *TEditor.InitBuffer*, *TEditor.SetBufLen*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an editor object from the stream *S* by first calling the *Load* constructor inherited from *TView*, then reading the fields introduced by *TEditor*. Allocates and initializes the buffer in the same manner as *TEditor.Init*.

See also: *TView.Load, TEditor.InitBuffer, TEditor.SetBufLen*

Done

```
destructor Done; virtual;
```

Deletes the edit buffer by calling *DoneBuffer*, then disposes of the editor object by calling the *Done* destructor inherited from *TView*.

See also: *TEditor.DoneBuffer, TView.Done*

BufChar

```
function BufChar(P: Word): Char;
```

Returns the *P*th character in the buffer.

BufPtr

```
function BufPtr(P: Word): Word;
```

Returns the buffer position of the *P*th character in the buffer, taking into account that the gap might be behind that character.

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Changes the boundaries of the editor view to *Bounds*, then adjusts *Delta* to make sure the text is still visible and redraws the view if necessary. As with the *TView* method it overrides, *TEditor.ChangeBounds* is called by other methods, but should not be called directly.

ConvertEvent

```
procedure ConvertEvent(var Event: TEvent); virtual;
```

Used by *HandleEvent* to handle key binding and basic editing operations. If you want to change or extend the default key bindings, you should override *ConvertEvent*.

See also: *TEditor.HandleEvent*

CursorVisible

```
function CursorVisible: Boolean;
```

Returns *True* if the cursor is visible within the view.

DeleteSelect

```
procedure DeleteSelect;
```

Deletes the selected text, if any. For example, after *ClipCut* copies selected text to the clipboard, it deletes the text from the buffer with *DeleteSelect*.

DoneBuffer

```
procedure DoneBuffer; virtual;
```

Deallocates the memory assigned to the edit buffer and sets *Buffer* to **nil**.

Draw

```
procedure Draw; virtual;
```

Draws the portion of the editor text that is currently in view. That is, it draws the lines that are within the boundaries of the view, taking into account the value of *Delta*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Returns a pointer to *CEditor*, the default editor view palette.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles events for the editor view by first calling the *HandleEvent* method inherited from *TView*, then calling *ConvertEvent* to remap keystrokes to commands, and then processing specific editor behavior.

Editor specific events handled include

- Mouse: Selection of text
- Key: Character insert/overwrite
- Command: cursor movement, selection, editing, clipboard stuff
- Broadcast: scroll bar changes

InitBuffer

```
procedure InitBuffer; virtual;
```

Calls *MemAlloc* to allocate *BufSize* bytes of memory from the heap to an edit buffer, then assigns it to *Buffer*.

InsertBuffer

```
function InsertBuffer(var P: PEditBuffer; Offset, Length: Word;
  AllowUndo, SelectText: Boolean): Boolean;
```

This is a low-level text insertion routine used by *InsertFrom* and *InsertText*; you will rarely, if ever, call it directly.

InsertBuffer inserts *Length* bytes of text from *P* (starting with *P[Offset]*) into the text buffer at *CurPtr*, deleting any selected text. If *AllowUndo* is *True*, *InsertBuffer* records information that will enable the user to undo the insertion. If *SelectText* is *True*, the inserted text will appear as a selected block once inserted.

Returns *True* if the insertion succeeds. If the insertion fails (because insertion would exceed the buffer size), *InsertBuffer* calls *EditorDialog* to show an "Out of memory" warning, then returns *False*.

See also: *TEditor.InsertFrom*, *TEditor.InsertText*

InsertFrom

```
function InsertFrom(Editor: PEditor): Boolean; virtual;
```

Inserts the selected text from *Editor* into the editor buffer by calling *InsertBuffer*.

See also: *TEditor.InsertBuffer*

InsertText

```
function InsertText(Text: Pointer; Length: Word;
  SelectText: Boolean): Boolean;
```

Copies *Length* bytes from *Text* into the editor buffer, selecting the inserted text if *SelectText* is *True*.

ScrollTo

```
procedure ScrollTo(X, Y: Integer);
```

Moves column *X* and line *Y* to the upper left corner of the edit view and redraws the view as needed.

Search

```
function Search(const FindStr: String; Opts: Word): Boolean;
```

Searches the editor buffer starting at *CurPtr* for the text contained in *FindStr*. *Opts* contains zero for a default search, *efCaseSensitive* for a case-sensitive search, or *efWholeWordsOnly* to match whole words only.

Returns *True* and selects the matching text if a match occurs; otherwise, returns *False*.

SetBufSize

```
function SetBufSize(NewSize: Word): Boolean; virtual;
```

Returns *True* if the buffer size can be changed to *NewSize*. By default, *SetBufSize* returns *True* if *NewSize* is less than or equal to *BufSize*. *SetBufSize* doesn't actually change the buffer size; it only indicates whether such a change can work. The actual change in buffer size should be done by *SetBufferSize*.

See also: *SetBufferSize* function

SetSelect

```
procedure SetSelect(NewStart, NewEnd: Word; CurStart: Boolean);
```

Sets the text between positions *NewStart* and *NewEnd* to be selected and redraws the view if needed. If *CurStart* is *True*, *SetSelect* places the cursor at the beginning of the selected block, otherwise it places the cursor at the end of the block.

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Calls the *SetState* method inherited from *TView* to actually set state flags, then hides or shows the scroll bar and indicator views associated with the editor, showing them if the editor is active, hiding them if it's inactive. After updating the associated views, *SetState* calls *UpdateCommands* to enable or disable commands based on whether the editor is active. If you want to enable or disable additional commands, you should override *UpdateCommands* rather than *SetState*.

See also: *TView.SetState, TEditor.UpdateCommands*

Store

```
procedure Store(var S: TStream);
```

Writes the editor object to the stream *S* by first calling the *Store* method inherited from *TView*, then writing the fields introduced by *TEditor*.

See also: *TView.Store*

TrackCursor

```
procedure TrackCursor(Center: Boolean);
```

Forces the view to scroll so it includes the cursor position. If *Center* is *True*, the line including the cursor moves to the middle of the view.

Undo `procedure Undo;`

Undoes the changes since the last cursor movement, restoring the edit buffer to the state it had at the last cursor movement.

UpdateCommands `procedure UpdateCommands; virtual;`

Updates commands based on the current state of the editor. *cmUndo* is enabled only if edits occurred since the last cursor movement. Sets the cut, copy, and paste commands to states appropriate to whether the editor is a clipboard and whether there is selected text. Enables *cmClear* if there is selected text. Enables all the search and replace commands.

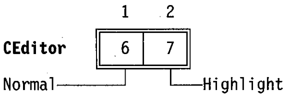
Be sure to call the inherited *TEditor.UpdateCommands* method if descendant objects override *UpdateCommands*.

Valid `function Valid(Command: Word): Boolean; virtual;`

Returns whether the editor view is valid for the command passed in *Command*. By default, *Valid* ignores the *Command* parameter and returns the value of the *IsValid* field. *IsValid* is *False* only if the constructor was unable to allocate an edit buffer.

Palette

Editor objects use the default palette *CEditor* to map onto the 6th and 7th entries in the standard window palette.



TEditorDialog type — Editors

Declaration `TEditorDialog = function(Dialog: Integer; Info: Pointer): Word;`

Function *TEditorDialog* is a procedural type used by *TEditor* objects to display various dialog boxes. Because dialog boxes are application dependent, editor objects don't display their own dialog boxes directly. Instead, they call the *EditorDialog* function, which displays the appropriate dialog box based on the value passed in the *Dialog* parameter.

The *Dialog* parameter should be one of the *edXXXX* constants. *Info* can point to any additional data the dialog box function might need.

Dialog box functions need to provide valid behavior for all values of *Dialog*. The *StdEditorDialog* function provides usable responses for all legal values of *Dialog*. Table 19.40 summarizes the values for *Info*, the expected message, and returns values for each value of *Dialog*.

Table 19.40 TEditorDialog parameter values, messages, and return values

Dialog constant	Values	Description
edOutOfMemory	*Info*	**nil**
	Message	Inform user that application ran out of memory
	Return	Ignored
edReadError	*Info*	*PString* pointing to file name
	Message	Inform user of file read error
	Return	Ignored
edWriteError	*Info*	*PString* pointing to file name
	Message	Inform user of file write error
	Return	Ignored
edCreateError	*Info*	*PString* pointing to file name
	Message	Inform user that program couldn't create file
	Return	Ignored
edSaveModify	*Info*	*PString* pointing to file name
	Message	Ask user whether to save changes before closing file
	Return	*cmYes* to save changes, *cmNo* to not save changes, *cmCancel* to not close file
edSaveUntitled	*Info*	*PString* pointing to file name
	Message	Ask user whether to save untitled file
	Return	*cmYes* to save file, *cmNo* to not save file, *cmCancel* to not close file
edSaveAs	*Info*	*PString* pointing to buffer to hold file name
	Message	Prompt user for file name
	Return	*cmCancel* to not save file; anything else to save file with the name in the buffer pointed to by *Info*
edFind	*Info*	Points to a record of type *TFindDialogRec*
	Message	Prompt user for search text and options
	Return	*cmCancel* if user chooses not to search; otherwise, fill the record pointed to by *Info*
edSearchFailed	*Info*	**nil**
	Message	Tell the user the text wasn't found
	Return	Ignored
edReplace	*Info*	Points to a record of type *TReplaceDialogRec*
	Message	Prompt user for search text, replacement text, and options
	Return	*cmCancel* if user chooses not to search; otherwise, fill the record pointed to by *Info*
edReplacePrompt	*Info*	An object of type *TPoint* with the global coordinates of the start of the located text
	Message	Ask the user if text should be replaced

Table 19.40: TEditorDialog parameter values, messages, and return values (continued)

Return	*cmYes* to replace text and continue search; *cmNo* to not replace text, but continue search; *cmCancel* to not replace text and terminate search

See also *edXXXX* constants, *EditorDialog* variable, *DefEditorDialog* function, *StdEditorDialog* function

TEditWindow object Editors

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
~~Awaken~~
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
~~EndModal~~
EventAvail
~~Execute~~
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
~~GetHelpCtx~~
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
~~SizeLimits~~
~~Store~~
TopView
~~Valid~~
WriteBuf
WriteChar
WriteLine
WriteStr

TGroup

Buffer
Current
Last
Phase

~~Init~~
~~Load~~
~~Done~~
Awaken
ChangeBounds
DataSize
Delete
Draw
EndModal
EventError
ExecView
Execute
First
FirstThat
FocusNext
ForEach
GetData
GetHelpCtx
GetSubViewPtr
~~HandleEvent~~
Insert
InsertBefore
Lock
PutSubViewPtr
Redraw
SelectNext
SetData
~~SetState~~
~~Store~~
Unlock
Valid

TWindow

Flags
Frame
Number
Palette
Title
ZoomRect

~~Init~~
~~Load~~
Done
~~Close~~
GetPalette
~~GetTitle~~
~~HandleEvent~~
InitFrame
SetState
SizeLimits
StandardScrollBar
~~Store~~
Zoom

TEditWindow

Editor

Init
Load
Close
GetTitle
HandleEvent
Store

An editor window is a window specifically designed to hold an editor object, specifically either a file editor or the clipboard. Editor windows change their titles to show the name of the file being edited, and automatically create scroll bars and an indicator for the editor. If you don't pass a file name to the editor window, the file is untitled.

Field

Editor

```
Editor: PFileEditor;
```

Points to the editor object associated with the editor window.

Methods

Init

```
constructor Init(var Bounds: TRect; FileName: FNameStr; ANumber: Integer);
```

Constructs an editor window object by first calling the *Init* constructor inherited from *TWindow* to create a window with the boundaries specified in *Bounds*, no title, and the window number passed in *ANumber*, then constructing and inserting horizontal and vertical scroll bars and an indicator object. Finally, *Init* constructs a file editor object, passing it the boundaries of the area inside the window frame, the scroll bars, the indicator, and the file name passed in *FileName*.

See also: *TWindow.Init, TFileEditor.Init*

Load

```
constructor Load(var S: TStream);
```

Creates and loads an editor window from the stream *S* by first calling the *Load* constructor inherited from *TWindow*, then reading the editor field introduced by *TEditWindow*.

See also: *TWindow.Load*

Close

```
procedure Close; virtual;
```

Calls the *Close* method inherited from *TWindow* unless the editor is a clipboard, in which case it calls *Hide* to hide the clipboard editor.

See also: *TWindow.Close*

GetTitle

```
function GetTitle(MaxSize: Integer): TTitleStr; virtual;
```

Returns the name of the file in the editor or 'Clipboard' if the editor is a clipboard.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles events for the editor window by calling the *HandleEvent* method inherited from *TWindow*, then handles the *cmUpdateTitle* broadcast event by redrawing the window's frame to change its title. *cmUpdateTitle* broadcasts occur when the name of the file being edited changes.

See also: *TWindow.HandleEvent*

Store `procedure Store(var S: TStream);`

Writes the editor window object to the stream *S* by first calling the *Store* method inherited from *TWindow*, then writes the editor to the stream using *PutSubViewPtr*.

See also: *TWindow.Store*, *TGroup.PutSubViewPtr*

TEmsStream Objects

TObject	TStream	TEmsStream
	Status ErrorInfo	Handle PageCount Size Position
~~Init~~ Free ~~Done~~	CopyFrom Error Flush Get ~~GetPos~~ ~~GetSize~~ Put ~~Read~~ ReadStr Reset ~~Seek~~ ~~Truncate~~ ~~Write~~ WriteStr	Init Done GetPos GetSize Read Seek Truncate Write

TEmsStream is a specialized stream derivative that implements streams in EMS memory. The additional fields provide an EMS handle, a page count, stream size, and current position. *TEmsStream* overrides the six abstract methods of *TStream* as well as providing a specialized constructor and destructor.

☞ When debugging a program using EMS streams, the IDE cannot recover EMS memory allocated by your program if your program terminates prematurely or if you forget to call the *Done* destructor for an EMS stream. Only the *Done* method (or rebooting) can release the EMS pages owned by the stream.

Fields

Handle `Handle: Word;` **Read only**

The EMS handle for the stream.

PageCount `PageCount: Word;` **Read only**

The number of allocated pages for the stream, with 16K per page.

Position `Position: Longint;` **Read only**

The current position within the stream. The first position is 0.

Size `Size: Longint;` **Read only**

The size of the stream in bytes.

Methods

Init `constructor Init(MinSize, MaxSize: Longint);`

Creates an EMS stream with the given minimum size in bytes. Calls the *Init* constructor inherited from *TStream*, then sets *Handle*, *Size* and *PageCount*. Calls *Error* with an argument of *stInitError* if initialization fails.

EMS drivers earlier than version 4.0 don't support resizeable expanded memory blocks. With a pre-4.0 driver, an EMS stream cannot expand beyond its initial size once allocated. To properly support both older and newer EMS drivers, an EMS stream's *Init* constructor takes two parameters which specify the minimum and maximum size of the initial EMS memory block allocation. *Init* always allocates at least *MinSize* bytes.

- If the EMS driver version number is greater than or equal to 4.0, *Init* allocates only *MinSize* bytes of EMS, and then expands the block as required by subsequent calls to *TEmsStream.Write,* ignoring *MaxSize.*
- If the driver version number is less than 4.0, *Init* allocates as much expanded memory as available, up to *MaxSize* bytes, and an error occurs if subsequent calls to *TEmsStream.Write* attempt to expand the stream beyond the allocated size.

Done `destructor Done; virtual;`

Override: Never Disposes of the EMS stream and releases EMS pages used.

See also: *TEmsStream.Init*

GetPos `function GetPos: Longint; virtual;`

Override: Never Returns the stream's current position. The first position is 0.

See also: *TEmsStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never Returns the size of the stream in bytes.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never Reads *Count* bytes from the stream, starting at the current position, into the *Buf* buffer.

See also: *TEmsStream.Write, stReadError*

Seek

`procedure Seek(Pos: Longint); virtual;`

Override: Never

Sets the current position to *Pos* bytes from the start of the stream. The first position is 0.

See also: *TEmsStream.GetPos, TEmsStream.GetSize*

Truncate

`procedure Truncate; virtual;`

Override: Never

Deletes all data on the stream from the current position to the end. Sets the current position to the new end of the stream.

See also: *TEmsStream.GetPos, TEmsStream.Seek*

Write

`procedure Write(var Buf; Count: Word); virtual;`

Override: Never

Writes *Count* bytes from the *Buf* buffer to the stream, starting at the current position.

See also: *TStream.Read, TEmsStream.GetPos, TEmsStream.Seek*

TEvent type — Drivers

Declaration

```
TEvent = record
  What: Word;
  case Word of
    evNothing: ();
    evMouse: (
      Buttons: Byte;
      Double: Boolean;
      Where: TPoint);
    evKeyDown: (
      case Integer of
        0: (KeyCode: Word);
        1: (CharCode: Char;
            ScanCode: Byte));
    evMessage: (
      Command: Word;
      case Word of
        0: (InfoPtr: Pointer);
        1: (InfoLong: Longint);
        2: (InfoWord: Word);
        3: (InfoInt: Integer);
        4: (InfoByte: Byte);
        5: (InfoChar: Char));
end;
```

Function The *TEvent* variant record type plays a fundamental role in Turbo Vision's event handling strategy. Both outside events, such as mouse and keyboard events, and command events generated by inter-communicating views, are stored and transmitted as *TEvent* records.

See also *evXXXX, HandleEvent* methods, *GetKeyEvent, GetMouseEvent*

TFileCollection object StdDlg

TFileCollection is a sorted collection of *TSearchRec* records. File dialog boxes use *TFileCollection* objects to provide alphabetically sorted file lists.

Details of *TFileCollection*'s fields and methods are in the online Help.

TFileDialog object StdDlg

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
~~Awaken~~
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
~~EndModal~~
EventAvail
~~Execute~~
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
~~GetHelpCtx~~
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
~~SizeLimits~~
~~Store~~
TopView
~~Valid~~
WriteBuf
WriteChar
WriteLine
WriteStr

TGroup

Buffer
Current
Last
Phase

~~Init~~
~~Load~~
~~Done~~
Awaken
ChangeBounds
DataSize
Delete
Draw
EndModal
EventError
ExecView
Execute
First
FirstThat
FocusNext
ForEach
~~GetData~~
GetHelpCtx
GetSubViewPtr
~~HandleEvent~~
Insert
InsertBefore
Lock
PutSubViewPtr
Redraw
SelectNext
~~SetData~~
~~SetState~~
~~Store~~
Unlock
~~Valid~~

TWindow

Flags
Frame
Number
Palette
Title
ZoomRect

~~Init~~
~~Load~~
~~Done~~
Close
~~GetPalette~~
GetTitle
~~HandleEvent~~
InitFrame
SetState
SizeLimits
StandardScrollBar
~~Store~~
Zoom

TDialog

~~Init~~
~~Load~~
GetPalette
~~HandleEvent~~
~~Valid~~

TFileDialog

FileName
FileList
WildCard
Directory

Init
Load
Done
GetData
GetFileName
HandleEvent
SetData
Store
Valid

TFileDialog is a standard file name input dialog box.

Fields

Directory

```
Directory: PString;
```

Directory points to a string containing the current directory name.

FileList

```
FileList: PFileList;
```

FileList points to the file list object in the dialog box.

See also: *TFileList* object

FileName

```
FileName: PFileInputLine;
```

FileName points to the file input line object in the dialog box.

See also: *TFileInputLine* object

WildCard

```
WildCard: TWildStr;
```

WildCard contains the current drive, path, and file name.

Methods

Init

```
constructor Init(AWildCard: TWildStr; const ATitle, InputName: String;
  AOptions: Word; HistoryId: Byte);
```

Constructs a file dialog box with the title given by *ATitle* by calling the *Init* constructor inherited from *TDialog*. Initializes the *WildCard* field to the value of *AWildCard*. Creates a file input line object and assigns it to the *FileName* field, setting the initial value of *FileName* to *WildCard*. Creates a label object using the string passed in *InputName* and associates it with *FileName*. Also creates a history list object with the ID passed in *HistoryID* and associates it with *FileName*.

Creates a file list object with an associated label, 'Files', and a vertical scroll bar.

Depending on the values passed in the bitmapped parameter *AOptions*, *Init* constructs and inserts buttons for Ok, Open, Replace, Clear, and Help. There is always a Cancel button. If *AOptions* includes *fdNoLoadDir*, the dialog box does not load the current directory contents into the file list; otherwise, it reads the current directory and builds the list. Use *fdNoLoadDir* when you want to store the dialog on a stream so you don't write an entire directory listing to the stream along with the dialog box.

A file information pane object is constructed and inserted at the bottom of the dialog box.

See also: *TDialog.Init*, *fdXXXX* constants

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a *TFileDialog* object from the stream *S* by first calling the *Load* constructor inherited from *TDialog* and then reading the fields introduced by *TFileDialog* and reading the current directory information.

See also: *TDialog.Load*

Done

```
destructor Done; virtual;
```

Disposes of the file dialog box object by first disposing of the *Directory* string, then calling the *Done* destructor inherited from *TDialog*.

See also: *TDialog.Done*

GetData

```
procedure GetData(var Rec); virtual;
```

Reads a string from *Rec*, casts it into type *PathStr*, and expands it to a full path name by calling *GetFileName*.

See also: *TFileDialog.GetFileName*

GetFileName

```
procedure GetFileName(var S: PathStr);
```

Expands the name of the currently selected file into a fully qualified path name, including drive, path, and file name and stores it in *S*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles most events by calling the *HandleEvent* method inherited from *TDialog*, then handles the commands *cmFileOpen*, *cmFileReplace*, and *cmFileClear* by calling *EndModal* with the command constant as its parameter, thus returning the command to the view that executed the file dialog box.

See also: *TDialog.HandleEvent*, *TGroup.EndModal*

SetData

```
procedure SetData(var Rec); virtual;
```

Calls the *SetData* method inherited from *TDialog* to ensure that all subviews get a chance to read data from *Rec*, then if the remaining data in *Rec* is a file name, checks the validity of the file name by calling *Valid* and making *FileName* the selected subview.

See also: *TDialog.SetData*

Store

```
procedure Store(var S: TStream);
```

Writes the file dialog box object to the stream *S* by first calling the *Store* method inherited from *TDialog,* then writing the fields introduced by *TFileDialog.*

See also: *TDialog.Store*

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

Returns *True* if *Command* is *cmValid,* indicating successful construction of the object. For all other values of *Command, Valid* first calls the *Valid* function inherited from *TDialog.* If *TDialog.Valid* returns *True, Valid* tests the current *FileName* string to ensure that it's a valid file name. If the file name is valid, *Valid* returns *True;* otherwise it calls up an "Invalid file name" message box and returns *False.*

See also: *TDialog.Valid*

TFileEditor Editors

TObject

~~Init~~
Free
~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
~~ChangeBounds~~	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	~~SetCmdState~~
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	~~SetState~~
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	Store
EventAvail	MakeLocal	TopView
Execute	MouseEvent	~~Valid~~
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TEditor

AutoIndent	HScrollBar
Buffer	Indicator
BufLen	InsCount
BufSize	IsValid
CanUndo	Limit
CurPos	Modified
CurPtr	Overwrite
DelCount	Selecting
Delta	SelEnd
DrawLine	SelStart
DrawPtr	VScrollBar
GapLen	

~~Init~~	InsertBuffer
~~Load~~	InsertFrom
Done	InsertText
BufChar	ScrollTo
BufPtr	Search
ChangeBounds	~~SetBufSize~~
ConvertEvent	SetCmdState
CursorVisible	SetSelect
DeleteSelect	SetState
~~DoneBuffer~~	~~Store~~
Draw	TrackCursor
GetPalette	Undo
~~HandleEvent~~	~~UpdateCommands~~
~~InitBuffer~~	~~Valid~~

TFileEditor

FileName

Init
Load
DoneBuffer
HandleEvent
InitBuffer
LoadFile
Save
SaveAs
SaveFile
SetBufSize
Store
UpdateCommands
Valid

A file editor object is a specialized descendant of *TEditor,* designed to edit the contents of a text file.

Field

```
FileName: FNameStr;
```

FileName is the name of the file being edited.

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar;
  AIndicator: PIndicator; AFileName: FNameStr);
```

Constructs a file editor object by first calling the *Init* constructor inherited from *TEditor*, passing *Bounds, AHScrollBar, AVScrollBar,* and *AIndicator,* with a buffer size of zero, then expanding *AFileName* and loading the file by calling *LoadFile.*

See also: *TEditor.Init, TFileEditor.LoadFile*

Load

```
constructor Load(var S: TStream);
```

Creates and loads a file editor object from the stream *S* by first calling the *Load* constructor inherited from *TEditor,* then reading the file name. If the file name is valid, *Load* then loads the file into the buffer by calling *LoadFile.*

See also: *TEditor.Load*

DoneBuffer

```
procedure DoneBuffer; virtual;
```

If the buffer is non-**nil**, *DoneBuffer* calls *DisposeBuffer* to dispose of the buffer.

See also: *DisposeBuffer* procedure

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles events for the file editor by first calling the *HandleEvent* method inherited from *TEditor,* then handles command events to save the file being edited.

See also: *TEditor.HandleEvent*

InitBuffer

```
procedure InitBuffer; virtual;
```

Allocates an edit buffer on the heap for the file text by calling *NewBuffer.*

See also: *NewBuffer* procedure

LoadFile

```
function LoadFile: Boolean;
```

Returns *True* if the file does not exist (meaning the user edits a new file) or the file loaded successfully, otherwise returns *False.* Reads the text of the file specified by *FileName* into the edit buffer, setting the buffer length to the size of the file read.

Save

```
function Save: Boolean;
```

Saves the contents of the editor buffer to the disk by calling *SaveAs* if the file has no name, or *SaveFile* if the file has a name already.

See also: *TFileEditor.SaveAs*, *TFileEditor.SaveFile*

SaveAs

```
function SaveAs: Boolean;
```

Calls *EditorDialog* to invoke a dialog box to get a file name for the edited text. If the user does not cancel the dialog box, *SaveAs* changes the title of the edit window to reflect the new name of the file, then calls *SaveFile* to save the buffer. *SaveAs* returns the value returned from *SaveFile*.

See also: *EditorDialog* procedure, *TFileEditor.SaveFile*

SaveFile

```
function SaveFile: Boolean;
```

If *EditorFlags* contains the *efBackupFiles* bit, *SaveFile* renames the original file to its original name with an extension of .BAK. Writes the contents of the edit buffer to the file specified by *FileName*, and sets the *Modified* flag to *False*. Returns *True* if the file save succeeds, otherwise returns *False* after displaying an appropriate dialog box explaining the failure.

SetBufSize

```
function SetBufSize(NewSize: Word): Boolean; virtual;
```

Increases or decreases the size of the edit buffer in 4K increments, adjusting *GapLen* as necessary.

Store

```
procedure Store(var S: TStream);
```

Writes the file editor object to the stream *S* by first calling the *Store* method inherited from *TEditor*, then writing the file name and selected text offsets.

UpdateCommands

```
procedure UpdateCommands; virtual;
```

Calls the *UpdateCommands* method inherited from *TEditor*, then enables the *cmSave* and *cmSaveAs* commands that apply only to the file editor.

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

If *Command* is *cmValid*, returns the value of *IsValid*, which is only *False* if the file editor could not create its buffer or read its file. Otherwise, *Valid* checks the *Modified* field to see if altered text needs to be saved before closing. If *Modified* is *True*, *Valid* brings up a dialog box to give the user the chance to save changes. If the user cancels the dialog box, *Valid* returns

False, leaving the editor open; otherwise the buffer is either saved or lost, depending on the user choice, and *Valid* returns *True*.

TFileInfoPane StdDlg

TFileInfoPane represents a file information pane, a view that displays the information about the currently selected file in the file list of a *TFileDialog*.

Details of *TFileInfoPane*'s fields and methods are in the online Help.

TFileInputLine StdDlg

TFileInputLine is a special input line used by *TFileDialog* that updates its contents in response to a *cmFileFocused* command from a *TFileList*. File input lines allow editing of file names that include optional paths and wildcards.

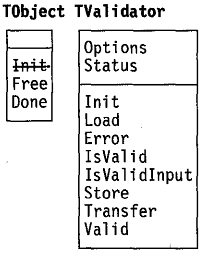
Details of *TFileInfoPane*'s fields and methods are in the online Help.

TFileList StdDlg

TFileList is a sorted list box that assumes it contains a *TFileCollection* as its collection. When a file name becomes selected, the file list object broadcasts a *cmFileFocused* message, which informs *TFileInputLine* and *TFileInfoPane* objects that they need to update their displays to reflect the new selection. By default, the file list is a two-column list box with an optional horizontal scroll bar below it.

Details of *TFileList*'s fields and methods are in the online Help.

TFilterValidator Validate



Filter validator objects check an input field as the user types into it. The validator holds a set of allowed characters. If the user types one of the legal characters, the filter validator indicates that the character is valid. If the user types any other character, the validator indicates that the input is invalid.

Field

ValidChars

```
ValidChars: TCharSet;
```

Contains the set of all characters the user can type. For example, to allow only numeric digits, set *ValidChars* to ['0'..'9']. *ValidChars* is set by the *AValidChars* parameter passed to the *Init* constructor.

Methods

Init

```
constructor Init(AValidChars: TCharSet);
```

Constructs a filter validator object by first calling the *Init* constructor inherited from *TValidator*, then setting *ValidChars* to *AValidChars*.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a filter validator object from the stream *S* by first calling the *Load* constructor inherited from *TValidator*, then reading the set of valid characters into *ValidChars*.

See also: *TValidator.Load*

Error

```
procedure Error; virtual;
```

Displays a message box indicating that the text string contains an invalid character.

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Returns *True* if all characters in *S* are in the set of allowed characters, *ValidChar*; otherwise returns *False*.

IsValidInput

```
function IsValidInput(var S: string; SuppressFill: Boolean): Boolean;
  virtual;
```

Checks each character in the string *S* to make sure it is in the set of allowed characters, *ValidChar*. Returns *True* if all characters in *S* are valid; otherwise, returns *False*.

Store

```
procedure Store(var S: TStream);
```

Stores the filter validator object on the stream *S* by writing *ValidChars*.

TFindDialogRec type Editors

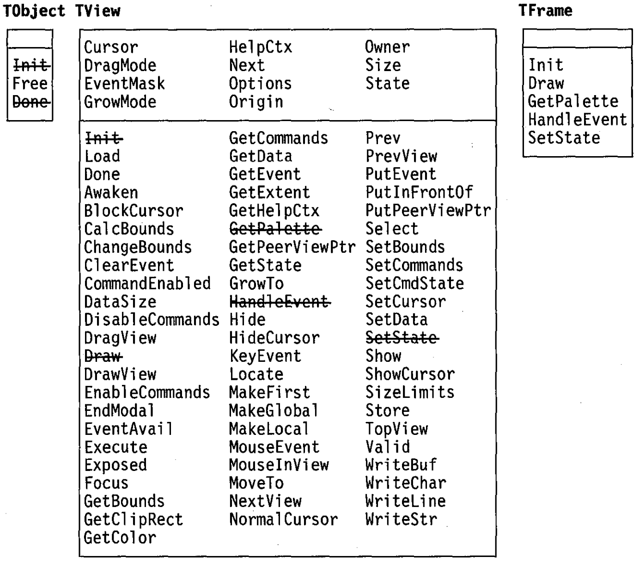
Declaration

```
TFindDialogRec = record
  Find: String[80];
  Options: Word;
end;
```

Function Find text dialog boxes, invoked by *EditorDialog* when passed *edFind,* take a pointer to a *TFindDialogRec* as their second parameter. *Find* holds the default string to search for. *Options* holds some combination of the *efXXXX* editor flag constants, specifying how the search operation should work.

See also *TEditorDialog* type

TFrame Views



TObject **TView** **TFrame**

~~Init~~
Free
~~Done~~

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
Load
Done
Awaken
BlockCursor
CalcBounds
ChangeBounds
ClearEvent
CommandEnabled
DataSize
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
GetData
GetEvent
GetExtent
GetHelpCtx
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
SetData
~~SetState~~
Show
ShowCursor
SizeLimits
Store
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

Init
Draw
GetPalette
HandleEvent
SetState

TFrame provides the distinctive frames around windows and dialog boxes. Users will probably never need to deal with frame objects directly, as they are added to window objects by default.

Methods

Init

```
constructor Init(var Bounds: TRect);
```

Calls the *Init* constructor inherited from *TView*, then sets *GrowMode* to *gfGrowHiX* + *gfGrowHiY* and sets *EventMask* to *EventMask* **or** *evBroadcast*, so *TFrame* objects default to handling broadcast events.

See also: *TView.Init*

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the frame with color attributes and icons appropriate to the current *State* flags: active, inactive, being dragged. Adds zoom, close and resize icons depending on the owner window's *Flags*. Adds the title, if any, from the owner window's *Title* field. Active windows are drawn with a double-lined frame and any icons, inactive windows with a single-lined frame and no icons.

See also: *sfXXXX* state flag constants, *wfXXXX* window flag constants

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Seldom

Returns a pointer to the default frame palette, *CFrame*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Seldom

Handles most events by calling the *HandleEvent* method inherited from *TView*, then handles mouse events specially. If the mouse is clicked on the close icon, *TFrame* generates a *cmClose* event. Clicking the zoom icon or double-clicking the top line of the frame generates a *cmZoom* event. Dragging the top line of the frame moves the window, and dragging the resize icon moves the lower-right corner of the view and therefore changes its size.

See also: *TView.HandleEvent*

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

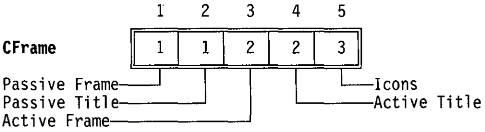
Override: Seldom

Calls *TView.SetState*, then if the new state is *sfActive* or *sfDragging*, calls *DrawView* to redraw the view.

See also: *TView.SetState*

Palette

Frame objects use the default palette, *CFrame*, to map onto the first three entries in the standard window palette.



TGroup

Views

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
~~Awaken~~
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
~~EndModal~~
EventAvail
~~Execute~~
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
~~GetHelpCtx~~
GetPalette
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
SizeLimits
~~Store~~
TopView
~~Valid~~
WriteBuf
WriteChar
WriteLine
WriteStr

TGroup

Buffer
Current
Last
Phase

Init
Load
Done
Awaken
ChangeBounds
DataSize
Delete
Draw
EndModal
EventError
ExecView
Execute
First
FirstThat
FocusNext
ForEach
GetData
GetHelpCtx
GetSubViewPtr
HandleEvent
Insert
InsertBefore
Lock
PutSubViewPtr
Redraw
SelectNext
SetData
SetState
Store
Unlock
Valid

TGroup objects and their derivatives (which we call groups for short) provide the central driving power to Turbo Vision. A group is a special breed of view. In addition to all the fields and methods derived from *TView*, a group has additional fields and methods (including many overrides) allowing it to control a dynamically linked list of views (including other groups) as though they were a single object. We often talk about the subviews of a group even when these subviews are often groups in their own right.

Although a group has a rectangular boundary from its *TView* ancestry, a group is only visible through the displays of its subviews. A group draws itself via the *Draw* methods of its subviews. A group owns its subviews, and together they must be capable of drawing (filling) the group's entire rectangular *Bounds*. During the life of an application, subviews are created, inserted into groups, and displayed as a result of user activity and events generated by the application itself. The subviews can just as easily be hidden, deleted from the group, or disposed of by user actions (such as closing a window or quitting a dialog box).

Three derived object types of *TGroup*, namely *TWindow*, *TDesktop*, and *TApplication* (via *TProgram*) illustrate the group and subgroup concept. The application typically owns a desktop object, a status line object, and a menu view object. *TDesktop* is a *TGroup* derivative, so it, in turn, can own *TWindow* objects, which in turn own *TFrame* objects, *TScrollBar* objects, and so on.

TGroup objects delegate both drawing and event handling to their subviews, as explained in Chapter 8, "Views" and Chapter 9, "Event-driven programming."

TGroup overrides many of the basic *TView* methods in a natural way. For example, storing and loading groups on streams can be achieved with single calls to *TGroup.Store* and *TGroup.Load*, which in turn iteratively store and load the group's subviews.

You'll rarely construct an instance of *TGroup* itself; rather you'll usually use one or more of *TGroup*'s derived object types: *TApplication*, *TDesktop*, and *TWindow*.

Fields

Buffer

`Buffer: PVideoBuf;` **Read only**

Points to a buffer used to cache the group's screen image, or **nil** if the group has no cache buffer. Cache buffers are created and destroyed automatically, unless the *ofBuffered* flag is cleared in the group's *Options* field.

See also: *TGroup.Draw*, *TGroup.Lock*, *TGroup.Unlock*

Current

`Current: PView;` **Read only**

Points to the subview that is currently selected, or **nil** if no subview is selected.

See also: *sfSelected, TView.Select*

Last `Last: PView` **Read only**

Points to the last subview in the group (the one furthest from the top in Z-order). The *Next* field of the last subview points to the first subview, whose *Next* field points to the next subview, and so on, forming a circular list.

Phase `Phase: (phFocused, phPreProcess, phPostProcess);` **Read only**

The current phase of processing for a focused event. Subviews that have the *ofPreProcess* and/or *ofPostProcess* flags set can examine *Owner^.Phase* to determine whether a call to their *HandleEvent* is happening in the *phPreProcess, phFocused,* or *phPostProcess* phase.

See also: *ofPreProcess, ofPostProcess, TGroup.HandleEvent*

Methods

Init `constructor Init(var Bounds: TRect);`

Constructs a group object with the given bounds by calling the *Init* instructor inherited from *TView,* then sets *ofSelectable* and *ofBuffered* in *Options,* and sets *EventMask* to $FFFF.

See also: *TView.Init*

Load `constructor Load(var S: TStream);`

Loads an entire group from a stream by first calling the *Load* constructor inherited from *TView,* then using *S.Get* to read each subview. After loading all subviews, the group makes a pass over the subviews to fix up all pointers read using *GetPeerViewPtr.*

If an object type derived from *TGroup* contains fields that point to subviews, it should use *GetSubViewPtr* within its *Load* to read these fields.

If the owner is **nil**, calls *Awaken* after all subviews are loaded.

See also: *TView.Load, TGroup.Store, TGroup.GetSubViewPtr, TStream.Get*

Done `destructor Done; virtual;`

Override: Often

Hides the group using *Hide,* disposes of each subview in the group, and finally calls the *Done* destructor inherited from *TView.*

See also: *TView.Done*

Awaken

```
procedure Awaken; virtual;
```

Calls the *Awaken* methods of each of the group's subviews in Z-order.

See also: *TView.Awaken*

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Override: Never

Changes the group's bounds to *Bounds* and then calls *CalcBounds* followed by *ChangeBounds* for each subview in the group.

See also: *TView.CalcBounds, TView.ChangeBounds*

DataSize

```
function DataSize: Word; virtual;
```

Override: Seldom

Returns the total size of the group's data record by calling and accumulating *DataSize* for each subview.

See also: *TView.DataSize*

Delete

```
procedure Delete(P: PView);
```

Deletes the subview *P* from the group and redraws the other subviews as required. Sets *P*'s *Owner* and *Next* fields to **nil**. *Delete* does not dispose of *P*, however.

See also: *TGroup.Insert*

Draw

```
procedure Draw; virtual;
```

Override: Never

If a cache buffer exists, then the buffer is written to the screen using *WriteBuf*. Otherwise, calls *Redraw* to draw all the group's subviews.

See also: *TGroup.Buffer, TGroup.Redraw*

EndModal

```
procedure EndModal(Command: Word); virtual;
```

Override: Never

If the group is the current modal view, it terminates its modal state, passing *Command* to *ExecView* (which made this view modal in the first place), which then returns *Command* as its result. If this group is *not* the current modal view, it calls the *EndModal* method inherited from *TView*.

See also: *TGroup.ExecView, TGroup.Execute*

EventError

```
procedure EventError(var Event: TEvent); virtual;
```

Override: Sometimes

Execute calls *EventError* whenever the event-handling loop encounters an event it can't handle. The default action is: If the group's *Owner* is not **nil,** *EventError* calls the owner's *EventError*. Normally this chains back to *TApplication*'s *EventError*. You can override *EventError* to trigger appropriate action.

See also: *TGroup.Execute, TGroup.ExecView, sfModal*

ExecView

```
function ExecView(P: PView): Word;
```

ExecView is the modal counterpart of the modeless *Insert* and *Delete* methods. Unlike *Insert,* after inserting a view into the group, *ExecView* waits for the view to execute, then removes the view, and finally returns the result of the execution. *ExecView* is used in a number of places throughout Turbo Vision, most notably to implement *TApplication.Run* and *TProgram.ExecuteDialog.*

ExecView saves the current context (the selected view, the modal view, and the command set), makes *P* modal by calling *P^.SetState(sfModal, True),* inserts *P* into the group (if it isn't already inserted), and calls *P^.Execute.* When *P^.Execute* returns, the group is restored to its previous state, and the result of *P^.Execute* is returned as the result of the *ExecView* call. If *P* is **nil**, *ExecView* returns *cmCancel.*

See also: *TGroup.Execute, sfModal.*

Execute

```
function Execute: Word; virtual;
```

Override: Seldom

Execute is a group's main event loop. It repeatedly gets events using *GetEvent* and handles them using *HandleEvent.* The event loop is terminated by the group or some subview through a call to *EndModal.* Before returning, however, *Execute* calls *Valid* to verify that the modal state can indeed be terminated.

The actual implementation of *TGroup.Execute* is shown below. Note that *EndState* is a **private** field in *TGroup* which gets set by a call to *EndModal.*

```
function TGroup.Execute: Word;
var E: TEvent;
begin
  repeat
    EndState := 0;
    repeat
      GetEvent(E);
      HandleEvent(E);
      if E.What <> evNothing then EventError(E);
    until EndState <> 0;
  until Valid(EndState);
  Execute := EndState;
end;
```

See also: *TGroup.GetEvent, TGroup.HandleEvent, TGroup.EndModal, TGroup.Valid*

First

```
function First: PView;
```

Returns a pointer to the group's first subview (the one closest to the top in Z-order), or **nil** if the group has no subviews.

See also: *TGroup.Last*

FirstThat `function FirstThat(P: Pointer): PView;`

FirstThat applies a Boolean function, given by the function pointer *P*, to each subview in Z-order until *P* returns *True*. The result is a pointer to the subview for which *P* returned *True*, or **nil** if *P* returned *False* for all subviews. *P* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean* value. For example:

```
function MyTestFunc(P: PView): Boolean; far;
```

The *SubViewAt* method shown below returns a pointer to the first subview that contains a given point.

```
function TMyGroup.SubViewAt(Where: TPoint): PView;

  function ContainsPoint(P: PView): Boolean; far;
  var
    Bounds: TRect;
  begin
    P^.GetBounds(Bounds);
    ContainsPoint := (P^.State and sfVisible <> 0) and
      Bounds.Contains(Where);
  end;

begin
  SubViewAt := FirstThat(@ContainsPoint);
end;
```

See also: *TGroup.ForEach*

FocusNext `function FocusNext(Forwards: Boolean): Boolean;`

If *Forwards* is *True*, *FocusNext* gives the input focus to the next selectable subview (one with its *ofSelectable* bit set) in the group's Z-order. If *Forwards* is *False*, the method focuses the previous selectable subview. Returns *True* if successful; otherwise, returns *False*.

If the view's *ofValidate* bit is set, it calls *Valid(cmReleaseFocus)* to determine whether it's allowed to release focus. If *Valid* returns *False*, the view keeps the focus and *FocusNext* returns *False*.

See also: *TView.Focus*

ForEach `procedure ForEach(P: Pointer);`

ForEach applies an action, given by the procedure pointer *P*, to each subview in the group in Z-order. *P* must point to a **far** local procedure taking one *Pointer* parameter, for example:

```
procedure MyActionProc(P: PView); far;
```

The *MoveSubViews* method shown below moves all subviews in a group by a given *Delta* value. Notice the use of *Lock* and *Unlock* to limit the number of redraw operations performed, thus eliminating any unpleasant flicker.

```
procedure TMyGroup.MoveSubViews(Delta: TPoint);

  procedure DoMoveView(P: PView); far;
  begin
    P^.MoveTo(P^.Origin.X + Delta.X, P^.Origin.Y + Delta.Y);
  end;

begin
  Lock;
  ForEach(@DoMoveView);
  Unlock;
end;
```

See also: *TGroup.FirstThat*

GetData

```
procedure GetData(var Rec); virtual;
```

Override: Seldom

Calls *GetData* for each subview in reverse Z-order, incrementing the location given by *Rec* by the *DataSize* of each subview.

See also: *TView.GetData, TGroup.SetData*

GetHelpCtx

```
function GetHelpCtx: Word; virtual;
```

Override: Seldom

Returns the help context of the current focused view by calling the selected subview's *GetHelpCtx* method. If no subview specifies a help context, *GetHelpCtx* returns the value of its own *HelpCtx* field.

GetSubViewPtr

```
procedure GetSubViewPtr(var S: TStream; var P);
```

Loads a subview pointer *P* from the stream *S*. *GetSubViewPtr* should only be used inside a *Load* constructor to read pointer values that were written by a call to *PutSubViewPtr* from a *Store* method.

See also: *TView.PutSubViewPtr, TGroup.Load, TGroup.Store*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Often

A group handles events by passing them on to the *HandleEvent* methods of one or more of its subviews. The actual routing, however, depends on the event class.

- For focused events (by default *evKeyDown* and *evCommand;* see *FocusedEvents* variable), event handling happens in three phases:
  - **Pre-process**. The group sets its *Phase* field to *phPreProcess* and passes the event to the *HandleEvent* methods of any subviews that have the *ofPreProcess* flag set.
  - **Process**. The group sets *Phase* to *phFocused* and passes the event to the *HandleEvent* method of the currently selected subview.
  - **Post-process**. The group sets *Phase* to *phPostProcess* and passes the event to the *HandleEvent* methods of any subviews that have the *ofPostProcess* flag set.
- For positional events (by default *evMouse,* see *PositionalEvents* variable), the group passes the event to the *HandleEvent* method of the first subview (in Z-order) whose bounding rectangle contains the point in *Event.Where.*
- For broadcast events (events that aren't focused or positional), the group passes the event to the *HandleEvent* method of each subview in the group in Z-order.

If a subview's *EventMask* field masks out an event class, *TGroup.HandleEvent* will *never* send events of that class to the subview. For example, the default *EventMask* of *TView* disables *evMouseUp, evMouseMove,* and *evMouseAuto,* so *TGroup.HandleEvent* will never send such events to a standard *TView.*

See also: *FocusedEvents, PositionalEvents, evXXXX* event constants, *TView.EventMask, HandleEvent* methods

Insert

```
procedure Insert(P: PView);
```

Inserts the view *P* into the group's subview list. The new subview appears in front of all other subviews. If the subview has its *ofCenterX* or *ofCenterY* flags set, it is centered accordingly in the group. If the subview has its *sfVisible* flag set, it will be shown in the group; otherwise, it remains invisible until specifically shown. If the subview has the *ofSelectable* flag set, it becomes the group's currently selected subview.

See also: *TGroup.Delete, TGroup.ExecView, TGroup.Delete*

InsertBefore

```
procedure InsertBefore(P, Target: PView);
```

Inserts the view *P* into the group's subview in front of the view given by *Target.* If *Target* is **nil**, the view is placed behind all other subviews in the group.

See also: *TGroup.Insert, TGroup.Delete*

Lock

```
procedure Lock;
```

Locks the group, delaying any screen writes by subviews until the group is unlocked. *Lock* has no effect unless the group has a cache buffer (see *ofBuffered* and *TGroup.Buffer*). *Lock* works by incrementing a lock count, which is decremented correspondingly by *Unlock*. When a call to *Unlock* decrements the count to zero, the entire group is written to the screen using the image constructed in the cache buffer.

By "sandwiching" draw-intensive operations between calls to *Lock* and *Unlock*, you can reduce or eliminate unpleasant screen flicker. For example, the *TDesktop.Tile* and *TDesktop.Cascade* methods use *Lock* and *Unlock* to reduce flicker while rearranging windows.

☞ *Lock* and *Unlock* calls *must* be balanced, otherwise a group might end up in a permanently locked state, causing it to not redraw itself properly when so requested.

See also: *TGroup.Unlock*

PutSubViewPtr

```
procedure PutSubViewPtr(var S: TStream; P: PView);
```

Stores a subview pointer *P* on the stream *S*. You should only use *PutSubViewPtr* inside a *Store* method to write pointer values that can later be read by a call to *GetSubViewPtr* from a *Load* constructor.

See also: *TGroup.GetSubViewPtr*, *TGroup.Store*, *TGroup.Load*

Redraw

```
procedure Redraw;
```

Redraws the group's subviews in Z-order. *TGroup.Redraw* differs from *TGroup.Draw* in that *Redraw* will never draw from the cache buffer.

See also: *TGroup.Draw*

SelectNext

```
procedure SelectNext(Forwards: Boolean);
```

If *Forwards* is *True*, *SelectNext* selects (makes current) the next selectable subview (one with its *ofSelectable* bit set) in the group's Z-order. If *Forwards* is *False*, the method selects the previous selectable subview.

☞ *SelectNext* ignores validation and always selects the next subview. If you need to validate views on focus change, call *FocusNext* instead of *SelectNext*.

See also: *ofXXXX* option flag constants, *TView.FocusNext*

SetData

```
procedure SetData(var Rec); virtual;
```

Override: Seldom

Calls *SetData* for each subview in reverse Z-order, incrementing the location given by *Rec* by the *DataSize* of each subview.

See also: *TGroup.GetData, TView.SetData*

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Override: Seldom

First calls the *SetState* method inherited from *TView*, then updates the subviews as follows:

- If *AState* is *sfActive*, *sfExposed*, or *sfDragging*, calls each subview's *SetState* to update the subview correspondingly.
- If *AState* is *sfFocused*, calls the currently selected subview's *SetState* to set its *sfFocused* flag.

See also: *TView.SetState*

Store

```
procedure Store(var S: TStream);
```

Stores an entire group on a stream by first calling the *Store* method inherited from *TView*, then using *TStream.Put* to write each subview.

If an object type derived from *TGroup* contains fields that point to subviews, it should use *PutSubViewPtr* within its *Store* to write these fields.

See also: *TView.Store, TGroup.PutSubViewPtr, TGroup.Load*

Unlock

```
procedure Unlock;
```

Unlocks the group by decrementing its lock count. If the lock count becomes zero, then the entire group is written to the screen using the image constructed in the cache buffer.

See also: *TGroup.Lock*

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

Calls the *Valid* method of each subview in Z-order and returns *True* if every subview's *Valid* returns *True*; otherwise, returns *False*. *TGroup.Valid* is used at the end of the event handling loop in *TGroup.Execute* to confirm that termination is allowed. A modal state cannot terminate until all *Valid* calls return *True*. A subview can return *False* if it wants to retain control.

See also: *TView.Valid, TGroup.Execute*

THistory Dialogs

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
Done
Awaken
BlockCursor
CalcBounds
ChangeBounds
ClearEvent
CommandEnabled
DataSize
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
GetData
GetEvent
GetExtent
GetHelpCtx
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
SetData
SetState
Show
ShowCursor
SizeLimits
~~Store~~
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

THistory

Link
HistoryId

Init
Load
Draw
GetPalette
HandleEvent
InitHistoryWindow
RecordHistory
Store

A *THistory* object implements a pick-list of previous entries, actions, or choices from which the user can select a "rerun." *THistory* objects are linked to an input line object and to a history list. History list information is stored in a block of memory on the heap. When the block fills up, the oldest history items are deleted as new ones are added.

THistory itself shows up as an icon (↓) next to an input line. When the user clicks the history icon, Turbo Vision opens up a history window (see *THistoryWindow*) with a history viewer (see *THistoryViewer*) containing a list of previous entries for that list.

Different input lines can share the same history list by using the same ID number.

T

Fields

HistoryID

`HistoryID: Word;` **Read only**

Each history list has a unique ID number, assigned by the programmer. Different history objects in different windows may share a history list by using the same history ID.

Link

```
Link: PInputLine;                                                  Read only
```

A pointer to the linked *TInputLine* object.

Methods

Init

```
constructor Init(var Bounds: TRect; ALink: PInputLine; AHistoryId: Word);
```

Creates a history view of the given size by calling the *Init* constructor inherited from *TView,* then setting the *Link* and *HistoryId* fields to *ALink* and *AHistoryId*. Sets *Options* to *ofPostProcess* and *EventMask* to *evBroadcast*.

See also: *TView.Init*

Load

```
constructor Load(var S: TStream);
```

Creates and initializes a history object from the stream *S* by calling the *Load* constructor inherited from *TView* and reading *Link* and *HistoryId* from *S*.

See also: *TView.Load*

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the history icon (↓) in the default palette.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default palette, *CHistory*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles most events by calling the *HandleEvent* method inherited from *TView,* then responds to two special events:

- If the user clicks the history list icon or presses ↓ while in the associated input line, this history view constructs a history window. By default, the window is one space larger than the linked input line, and six lines taller, but clipped to fit inside the owner dialog box. *HandleEvent* passes that bounding rectangle to *InitHistoryWindow* to actually construct the history window.
- If the linked input line loses the input focus, or the history icon gets an explicit *cmRecordHistory* command, *HandleEvent* calls *RecordHistory* to record the current contents of the input line in the history block.

See also: *TView.HandleEvent, THistory.InitHistoryWindow, THistory.RecordHistory*

InitHistoryWindow

```
function InitHistoryWindow(var Bounds: TRect): PHistoryWindow; virtual;
```

Constructs a history window object with the bounding rectangle passed in *Bounds* and the history ID in *HistoryID*, returning a pointer to the newly constructed window. Also sets the help context for the history window to the linked input line's help context. *THistory*'s event handler calls *InitHistoryWindow* in response to mouse clicks the history icon or certain keystrokes in the linked input line.

See also: *THistoryWindow.Init*, *THistory.HandleEvent*

RecordHistory

```
procedure RecordHistory(const S: String); virtual;
```

Adds the string *S* to the history list associated with the view, identified by *HistoryID*.

See also: *HistoryAdd* procedure

Store

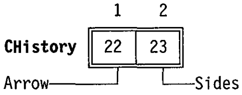
```
procedure Store(var S: TStream);
```

Saves a history object on the stream *S* by calling the *Store* method inherited from *TView*, then writing *Link* and *HistoryId* to *S*.

See also: *TView.Store*

Palette

History icons use the default palette, *CHistory*, to map onto the 22nd and 23rd entries in the standard dialog box palette.



THistoryViewer — Dialogs

THistoryViewer is a straightforward descendant of *TListViewer* used by the history list system. The history viewer appears inside the history window set up by clicking the history icon. For details on how *THistory*, *THistoryWindow*, and *THistoryViewer* cooperate, see the entry for *THistory* in this chapter.

Details of *THistoryViewer*'s field and methods are in the online Help.

THistoryWindow — Dialogs

THistoryWindow is a specialized descendant of *TWindow* used for holding a history list viewer when the user clicks the history icon next to an input line. By default, the window has no title and no number. The history window's frame has a close icon so the window can be closed, but cannot be resized or zoomed.

For details on the use of history lists and their associated objects, see the entry for *THistory* in this chapter.

Details of *THistoryWindow*'s field and methods are in the online Help.

TIndicator Editors

TObject	TView			TIndicator
	Cursor DragMode EventMask GrowMode	HelpCtx Next Options Origin	Owner Size State	Location Modified
~~Init~~ Free ~~Done~~	~~Init~~ Load Done Awaken BlockCursor CalcBounds ChangeBounds ClearEvent CommandEnabled DataSize DisableCommands DragView ~~Draw~~ DrawView EnableCommands EndModal EventAvail Execute Exposed Focus GetBounds GetClipRect GetColor	GetCommands GetData GetEvent GetExtent GetHelpCtx ~~GetPalette~~ GetPeerViewPtr GetState GrowTo HandleEvent Hide HideCursor KeyEvent Locate MakeFirst MakeGlobal MakeLocal MouseEvent MouseInView MoveTo NextView NormalCursor	Prev PrevView PutEvent PutInFrontOf PutPeerViewPtr Select SetBounds SetCommands SetCmdState SetCursor SetData ~~SetState~~ Show ShowCursor SizeLimits Store TopView Valid WriteBuf WriteChar WriteLine WriteStr	Init Draw GetPalette SetState SetValue

An indicator object implements a line and column counter in the lower left corner of an editor window. Editor window objects create indicators by default, and associate them with editor objects. Indicators can also work with editors outside the context of an editor window, however.

Fields

Location `Location: TPoint;`

Location holds the current column and line position to display. Editor objects update *Location* automatically.

Modified `Modified: Boolean;`

Modified is *True* if the text in the associated editor has changed. *Draw* check *Modified* and shows a special character to alert the user of the status of the edit buffer.

Methods

Init

```
constructor Init(var Bounds: TRect);
```

Constructs an indicator with the boundaries specified in *Bounds* by calling the *Init* constructor inherited from *TView*, then anchors the view to the bottom left corner of the owner window by setting *GrowMode* to *gfGrowLoY* + *gfGrowHiY*.

See also: *TView.Init*

Draw

```
procedure Draw; virtual;
```

Draws the indicator in the form line:column, followed by a ☼ if *Modified* is *True*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Returns a pointer *CIndicator*, the default indicator palette.

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Calls the *SetState* method inherited from *TView* to handle normal state-setting, then redraws the indicator if the *sfDragging* flag is set, meaning that the indicator needs to redraw itself using the frame's dragging color, rather than the normal color.

See also: *TView.SetState*

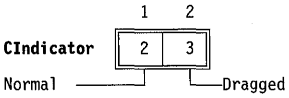
SetValue

```
procedure SetValue(ALocation: TPoint; AModified: Boolean);
```

Sets *Location* to *ALocation* and *Modified* to *AModified* and redraws the indicator. Editor objects call this method to keep the indicator's values current.

Palette

Indicator objects use the default palette *CIndicator* to map onto the second and third entries in the standard application palette. These are the same colors used by window frames.



TInputLine Dialogs

TObject

~~Init~~ Free ~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	~~GetData~~	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
~~DataSize~~	~~HandleEvent~~	SetCursor
DisableCommands	Hide	~~SetData~~
DragView	HideCursor	~~SetState~~
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	~~Valid~~
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TInputLine

Data MaxLen CurPos FirstPos SelStart SelEnd Validator
Init Load Done DataSize Draw GetData GetPalette HandleEvent SelectAll SetData SetState SetValidator Store Valid

A *TInputLine* object provides a basic input line string editor. It handles keyboard input and mouse clicks and drags for block marking and a variety of line editing functions (see *TInputLine.HandleEvent*). The selected text is deleted and then replaced by the first text input. If *MaxLen* is greater than the X dimension (*Size.X*), horizontal scrolling is supported and indicated by left and right arrows.

The *GetData* and *SetData* methods are available for writing and reading data strings (referenced via the *Data* pointer field) into the given record. *TInputLine.SetState* simplifies the redrawing of the view with appropriate colors when the state changes from or to *sfActive* and *sfSelected*.

Input lines frequently have labels, history lists, and perhaps validators associated with them.

You can modify the basic input line to handle data types other than strings. To do so, you'll generally add additional fields to hold the data, and then override the *Init*, *Load*, *Store*, *Valid*, *DataSize*, *GetData*, and *SetData* methods.

Fields

CurPos

`CurPos: Integer;` **Read/write**

Index to insertion point (that is, to the current cursor position).

See also: *TInputLine.SelectAll*

Data `Data: PString;` **Read/write**

Pointer to the string containing the edited information.

FirstPos `FirstPos: Integer;` **Read/write**

Index to the first displayed character.

See also: *TInputLine.SelectAll*

MaxLen `MaxLen: Integer;` **Read only**

Maximum length allowed for the string, excluding the length byte.

See also: *TInputLine.DataSize*

SelEnd `SelEnd: Integer;` **Read only**

Index to the end of the selection area (that is, to the last character block marked).

See also: *TInputLine.SelectAll*

SelStart `SelStart: Integer;` **Read only**

Index to the beginning of the selection area (that is, to the first character block marked).

See also: *TInputLine.SelectAll*

Validator `Validator: PValidator;`

Points to the data validator object associated with the input line, or **nil** if the input line has no validator. You should use the *SetValidator* method to assign a value to *Validator*, rather than assigning the value directly.

See also: *TInputLine.SetValidator*

Methods

Init `constructor Init(var Bounds: TRect; AMaxLen: Integer);`

Constructs an input line control with the given argument values by calling the *Init* constructor inherited from *TInputLine*. Sets *State* to *sfCursorVis*, *Options* to (*ofSelectable* + *ofFirstClick*), and *MaxLen* to *AMaxLen*. Allocates *AMaxlen* + 1 bytes of memory and sets *Data* to point at this allocation.

See also: *TView.Init, TView.sfCursorVis, TView.ofSelectable, TView.ofFirstClick*

Load

```
constructor Load(var S: TStream);
```

Constructs and initializes an input line object from the stream *S* by first calling the *Load* constructor inherited from *TView* to load the view, then reading the integer fields off the stream using *S.Read*. Allocates *MaxLen* + 1 bytes for *Data*, and then reads the string-length byte and data from *S* using *S.Read*. Use *Load* in conjunction with *Store* to save and retrieve input line objects on streams.

Override this method if you define descendants that contain additional fields.

See also: *TView.Load, TInputLine.Store, TStream.Read*

Done

Override: Seldom

```
destructor Done; virtual;
```

Disposes of the memory allocated to *Data*, then calls the *Done* destructor inherited from *TView* to dispose of the input line object.

See also: *TView.Done*

DataSize

Override: Sometimes

```
function DataSize: Word; virtual;
```

Returns the size of the record for *GetData* and *SetData* calls. By default, it returns *MaxLen* + 1. Override this method if you define descendants to handle other data types.

See also: *TInputLine.GetData, TInputLine.SetData*

Draw

Override: Seldom

```
procedure Draw; virtual;
```

Draws the input line and its data. The view has different colors depending on whether it has the focus, with arrows drawn if the text string exceeds the size of the view (in either direction). Selected (block marked) characters are drawn with the appropriate palette.

GetData

Override: Sometimes

```
procedure GetData(var Rec); virtual;
```

Returns the value of the input line string. By default, *GetData* writes *DataSize* bytes from the string *Data^* to *Rec*. You can override *GetData* if you define descendants to handle non-string data types. For example, a numeric input line could convert the string value to a number and copy the number into *Rec*. If you override *GetData*, you must also override *SetData* to read the same data returned by *GetData*, and override *DataSize* to return the size of the data passed.

See also: *TInputLine.DataSize, TInputLine.SetData*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default palette, *CInputLine.*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Sometimes

Calls the *HandleEvent* method inherited from *TView,* then handles all mouse and keyboard events if the input line is selected. This method implements the standard editing capability of the box.

Editing features include: block marking with mouse click and drag; block deletion; insert or overwrite control with automatic cursor shape change; automatic and manual scrolling as required (depending on relative sizes of *Data* string and *Size.X*); manual horizontal scrolling via mouse clicks on the arrow icons; manual cursor movement by arrow, *Home,* and *End* keys (and their standard *Ctrl* key equivalents); character and block deletion with *Del* and *Ctrl+G.* The view is redrawn as required and the *TInputLine* fields are adjusted appropriately.

See also: *sfCursorIns, TView.HandleEvent, TInputLine.SelectAll*

SelectAll

```
procedure SelectAll(Enable: Boolean);
```

Sets *CurPos, FirstPos,* and *SelStart* to 0. If *Enable* is *True, SelEnd* is set to *Length(Data^),* selecting the whole input line; if *Enable* is *False, SelEnd* is set to 0, deselecting the whole line. Finally, redraws the view by calling *DrawView.*

See also: *TView.DrawView*

SetData

```
procedure SetData(var Rec); virtual;
```

Override: Sometimes

By default, reads *DataSize* bytes from *Rec* into *Data^* and calls *SelectAll(True)* to display the newly set text as selected. Override this method if you define descendants to handle non-string data types, using this method to convert your data type to a string for editing by *TInputLine.*

See also: *TInputLine.DataSize, TInputLine.GetData, TView.DrawView*

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Override: Seldom

Called when the input line needs redrawing following a change of *State.* Calls the *SetState* method inherited from *TView* to set or clear the bit(s) passed in *AState* in the input line's *State* field. Then if *AState* is *sfSelected* or if *AState* is *sfActive* and the input line is *sfSelected,* calls *SelectAll(Enable).*

See also: *TView.SetState, TView.DrawView*

SetValidator

```
procedure SetValidator(AValid: PValidator);
```

If the input line already has an associated validator, *SetValidator* disposes of the existing validator by calling its *Free* method. Sets *Validator* to *AValid*. You should pass **nil** to dispose of an associated validator without assigning a new one.

Store `procedure Store(var S: TStream);`

Stores the view on the stream *S* by first calling the *Store* method inherited from *TView*, then writing the five integer fields and the *Data* string with *S.Write* calls. Use in conjunction with *Load* to save and restore entire input line objects. Override this method if you define descendants that contain additional fields.

See also: *TView.Store, TInputLine.Load, TStream.Write*

Valid `function Valid(Command: Word): Boolean; virtual;`

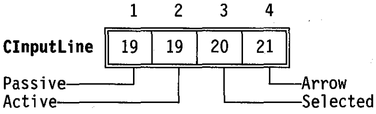
If the input line has no associated validator object or *Command* is *cmCancel*, *Valid* returns the value returned from a call to the *Valid* method inherited from *TView*.

If the input line has a validator, it checks the validator to determine its return value. If *Command* is *cmValid*, *Valid* returns *True* if the validator's *Status* is *vsOK*, otherwise, it returns *False*. If *Command* is anything other than *cmValid* or *cmCancel*, *Valid* passes *Data^* to the validator's *Valid* method. If the validator's *Valid* returns *False*, the input line calls *Select* to take the input focus and returns *False*.

See also: *TView.Valid, TValidator.Valid*

Palette

Input lines use the default palette, *CInputLine*, to map onto the 19th through 21st entries in the standard dialog palette.



TItemList type — Objects

Declaration `TItemList = array[0..MaxCollectionSize - 1] of Pointer;`

Function An array of generic pointers used internally by *TCollection* objects.

TLabel

Dialogs

TObject
~~Init~~ Free ~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	SetState
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TStaticText
Text
~~Init~~ ~~Load~~ Done ~~Draw~~ ~~GetPalette~~ GetText ~~Store~~

TLabel
Light Link
Init Load Draw GetPalette HandleEvent Store

A *TLabel* object is a piece of text in a view that can be selected (highlighted) by mouse click, cursor keys, or *Alt*+letter shortcut. The label is usually "attached" via a *PView* pointer to some other control view such as an input line, cluster, or list viewer to guide the user. Selecting (or "pressing") the label will select the attached control. Conversely, the label is highlighted when the linked control is selected.

Fields

Light

`Light: Boolean;` **Read only**

If *True*, the label and its linked control has been selected and will be highlighted.

Link

`Link: PView;` **Read only**

Pointer to the control associated with this label.

Methods

Init

```
constructor Init(var Bounds: TRect; const AText: String; ALink: PView);
```

Creates a label object of the given size and text by calling the *Init* constructor inherited from *TStaticText,* then sets *Link* to *ALink* for the associated control. Sets *Options* to *ofPreProcess* and *ofPostProcess* and *EventMask* to *evBroadcast. AText* can designate a shortcut letter for the label by surrounding the letter with tildes ('~').

You should never construct a label object with a **nil** link. Use static text objects for unlinked labels.

See also: *TStaticText.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a label object from the stream *S* by first calling the *Load* constructor inherited from *TStaticText,* then calling *GetPeerViewPtr(S, Link)* to reestablish the link to the associated control.

See also: *TLabel.Store*

Draw

```
procedure Draw; virtual;
```

Override: Never

Draws the view with the appropriate colors from the default palette.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default palette, *CLabel.*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Never

Handles most events by calling the *HandleEvent* method inherited from *TStaticText.* If the label receives an *evMouseDown* event or shortcut key event, the linked control is selected. This method also responds to *cmReceivedFocus* and *cmReleasedFocus* broadcast events from the linked control in order to adjust the value of the *Light* field and redraw the label as necessary.

See also: *TView.HandleEvent, cmXXXX* command constants

Store

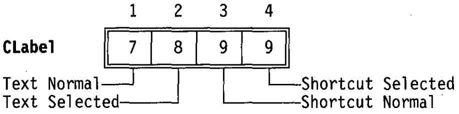
```
procedure Store(var S: TStream);
```

Stores the view on the stream *S* by first calling the *Store* method inherited from *TStaticText,* then recording the link to the associated control with *PutPeerViewPtr.*

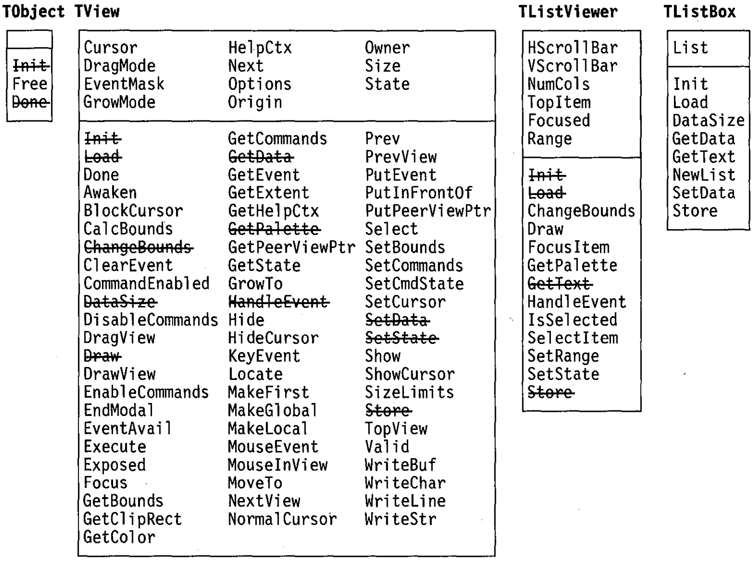
See also: *TLabel.Load*

Palette

Labels use the default palette, *CLabel*, to map onto the 7th, 8th and 9th entries in the standard dialog palette.



TListBox Dialogs



TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
Done
Awaken
BlockCursor
CalcBounds
~~ChangeBounds~~
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
GetHelpCtx
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
SizeLimits
~~Store~~
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

TListViewer

HScrollBar
VScrollBar
NumCols
TopItem
Focused
Range

~~Init~~
~~Load~~
ChangeBounds
Draw
FocusItem
GetPalette
~~GetText~~
HandleEvent
IsSelected
SelectItem
SetRange
SetState
~~Store~~

TListBox

List

Init
Load
DataSize
GetData
GetText
NewList
SetData
Store

TListBox is derived from *TListViewer* to help you set up the most commonly used list boxes, namely those displaying collections of strings. List box objects represent displayed lists of strings in one or more columns with an optional vertical scroll bar. The horizontal scroll bars of *TListViewer* are not supported. The inherited *TListViewer* methods let you highlight and select items by mouse and keyboard cursor actions. *TListBox* does not override *HandleEvent* or *Draw*.

TListBox has an additional field called *List* not found in *TListViewer*. *List* points to a collection object that holds the items to be listed and selected. Inserting data into the collection is your responsibility, as are the actions to be performed when an item is selected.

TListBox inherits its *Done* method from *TView,* so it is also your responsibility to dispose of the contents of *List* when you are finished with it. A call to *NewList will* dispose of the old list, so calling *NewList(**nil**)* and then disposing the list box will free everything.

Field

List

```
List: PCollection;                                              Read only
```

List points at the collection of items to scroll through. Typically, this might be a collection of *PStrings* representing the item texts.

Methods

Init

```
constructor Init(var Bounds: TRect; ANumCols: Word; AScrollBar: PScrollBar);
```

Constructs a list box control with the given size, number of columns, and the vertical scroll bar passed in *AScrollBar* by calling the *Init* constructor inherited from *TListViewer* with a **nil** horizontal scroll bar parameter.

Sets *List* to **nil** (empty list) and *Range* to 0. Your application must provide a suitable collection holding the strings (or other objects) to be listed. Set *List* to point to this collection using *NewList*.

See also: *TListViewer.Init, TListBox.NewList*

Load

```
constructor Load(var S: TStream);
```

Constructs a list box object and loads it with values from the stream *S* by first calling the *Load* constructor inherited from *TListViewer* then reading *List* from *S* with *S.Get*.

See also: *TListViewer.Load, TListBox.Store, TStream.Get*

DataSize

```
function DataSize: Word; virtual;
```

Override: Sometimes

Returns the size of the data read and written to the records passed to *GetData* and *SetData*. By default, *TListBox.DataSize* returns the size of a pointer plus the size of a word (for *List* and the selected item).

See also: *TListBox.GetData, TListBox.SetData*

GetData

```
procedure GetData(var Rec); virtual;
```

Override: Sometimes

Writes *TListBox* object data to the target record. By default, this method writes the current *List* and *Focused* fields to *Rec*.

See also: *TListBox.DataSize, TListBox.SetData*

GetText

```
function GetText(Item: Integer; MaxLen: Integer): String; virtual;
```

Override: Sometimes

Returns the *Item*th string from the list box object. By default, *GetText* returns the string obtained from the *Item*'th item in the string collection using *PString(List^.At(Item))^*. If *List* contains non-string objects, you will need to override this method. If *List* is **nil**, *GetText* returns an empty string.

See also: *TCollection.At*

NewList

```
procedure NewList(AList: PCollection); virtual;
```

Override: Seldom

If *AList* is non-**nil**, a new list given by *AList* replaces the current *List*. Sets *Range* to the *Count* field of the new *TCollection*, and focuses the first item by calling *FocusItem(0)*. Finally, the new list is displayed with a *DrawView* call. If the previous *List* field is non-**nil**, *NewList* disposes of it before the assigning the new list.

See also: *TListBox.SetData, TListViewer.SetRange, TListViewer.FocusItem, TView.DrawView*

SetData

```
procedure SetData(var Rec); virtual;
```

Override: Sometimes

Replaces the current list with *List* and *Focused* values read from the given *Rec* record. *SetData* calls *NewList* so that the new list is displayed with the correct focused item. As with *GetData* and *DataSize*, you might need to override this method for your own applications.

See also: *TListBox.DataSize, TListBox.GetData, TListBox.NewList*

Store

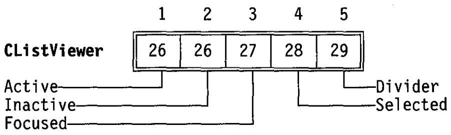
```
procedure Store(var S: TStream);
```

Writes the list box to the stream *S* by first calling the *Store* method inherited from *TListViewer* and then writing the collection onto the stream by calling *S.Put(List)*.

See also: *TListBox.Load, TListViewer.Store, TStream.Put*

Palette

List boxes use the default palette, *CListViewer*, to map onto the 26th through 29th entries in the standard application palette.



TListViewer Views

TObject	TView			TListViewer
	Cursor	HelpCtx	Owner	HScrollBar
~~Init~~	DragMode	Next	Size	VScrollBar
Free	EventMask	Options	State	NumCols
~~Done~~	GrowMode	Origin		TopItem
				Focused
	~~Init~~	GetCommands	Prev	Range
	~~Load~~	GetData	PrevView	
	Done	GetEvent	PutEvent	Init
	Awaken	GetExtent	PutInFrontOf	Load
	BlockCursor	GetHelpCtx	PutPeerViewPtr	ChangeBounds
	CalcBounds	~~GetPalette~~	Select	Draw
	~~ChangeBounds~~	GetPeerViewPtr	SetBounds	FocusItem
	ClearEvent	GetState	SetCommands	GetPalette
	CommandEnabled	GrowTo	SetCmdState	GetText
	DataSize	~~HandleEvent~~	SetCursor	HandleEvent
	DisableCommands	Hide	SetData	IsSelected
	DragView	HideCursor	~~SetState~~	SelectItem
	~~Draw~~	KeyEvent	Show	SetRange
	DrawView	Locate	ShowCursor	SetState
	EnableCommands	MakeFirst	SizeLimits	Store
	EndModal	MakeGlobal	~~Store~~	
	EventAvail	MakeLocal	TopView	
	Execute	MouseEvent	Valid	
	Exposed	MouseInView	WriteBuf	
	Focus	MoveTo	WriteChar	
	GetBounds	NextView	WriteLine	
	GetClipRect	NormalCursor	WriteStr	
	GetColor			

The *TListViewer* object type is a base type from which to derive list viewers of various kinds, such as *TListBox*. *TListViewer's* basic fields and methods offer the following functionality:

- A view for displaying linked lists of items (but no list)
- Control over one or two scroll bars
- Basic scrolling of lists in two dimensions
- Loading and storing the view and its scroll bars from and to a *TStream*
- Ability to mouse or key select (highlight) items on list
- *Draw* method that copes with resizing and scrolling

TListViewer has an abstract *GetText* method, so you need to supply the mechanism for creating and manipulating the text of the items to be displayed.

TListViewer has no list storage mechanism of its own. Use it to display scrollable lists of arrays, linked lists, or similar data structures. You can also use its descendants, such as *TListBox*, which associates a collection with a list viewer.

Fields

Focused

`Focused: Integer;` **Read only**

The item number of the focused item. Items are numbered from 0 to *Range* – 1. *Init* sets *Focused* to 0, the first item. *Focused* changes with mouse clicks or *Spacebar* selection.

See also: *Range*

HScrollBar

`HScrollBar: PScrollBar;` **Read only**

Pointer to the horizontal scroll bar associated with this view. If **nil**, the view does not have such a scroll bar.

NumCols

`NumCols: Integer;` **Read only**

The number of columns in the list viewer.

Range

`Range: Integer;` **Read only**

The current total number of items in the list. Items are numbered from 0 to *Range* – 1.

See also: *TListViewer.SetRange*

TopItem

`TopItem: Integer;` **Read/write**

The item number of the top visible item. Items are numbered from 0 to *Range* – 1. This number depends on the number of columns, the size of the view, and the value of *Range*.

See also: *Range*

VScrollBar

`VScrollBar: PScrollBar;` **Read only**

Pointer to the vertical scroll bar associated with this view. If **nil**, the view does not have such a scroll bar.

Methods

Init

```
constructor Init(var Bounds: TRect; ANumCols: Integer; AHScrollBar, AVScrollBar:
PScrollBar);
```

Constructs and initializes a list viewer object with the given size by first calling the *Init* constructor inherited from *TView*. Sets *NumCols* to *ANumCols*. Sets *Options* to (*ofFirstClick* + *ofSelectable*) so that mouse clicks that select the list viewer are also passed to *HandleEvent*. Sets *EventMask* to *evBroadcast*, *Range* and *Focused* to 0. Sets *VScrollBar* and *HScrollBar* to the vertical and/or horizontal scroll bars passed in *AVScrollBar* and *AHScrollBar*.

If you provide valid scroll bars, *Init* adjusts their *PgStep* and *ArStep* fields according to the *TListViewer* size and number of columns. For a single-column *TListViewer*, for example, the default vertical *PgStep* is *Size.Y* – 1, and the default vertical *ArStep* is 1.

See also: *TView.Init, TScrollBar.SetStep*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a list viewer object from the stream *S* by first calling the *Load* constructor inherited from *TView*, then reading the scroll bars using calls to *GetPeerViewPtr*, and finally reading the integer fields using *S.Read*.

See also: *TView.Load, TListViewer.Store*

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Override: Never

Changes the size of the list viewer object by calling the *ChangeBounds* method inherited from *TView*. If the viewer has a horizontal scroll bar, *ChangeBounds* adjusts *PgStep* as needed.

See also: *TView.ChangeBounds, TScrollBar.ChangeStep*

Draw

```
procedure Draw; virtual;
```

Override: Never

Draws the list viewer object with the default palette, calling *GetText* for each visible item. Takes into account the focused and selected items and whether the view is *sfActive*.

See also: *TListViewer.GetText*

FocusItem

```
procedure FocusItem(Item: Integer); virtual;
```

Override: Never

Makes the given item focused by setting *Focused* to *Item*. *FocusItem* also sets the *Value* field of the vertical scroll bar (if any) to *Item* and adjusts the *TopItem* field.

See also: *TListViewer.IsSelected, TScrollBar.SetValue*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default palette, *CListViewer*.

GetText

```
function GetText(Item: Integer; MaxLen: Integer): String; virtual;
```

Override: Always

This is an abstract method. Derived types must supply a mechanism for returning a string for the item *Item,* not exceeding *MaxLen* characters.

See also: *TListViewer.Draw*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Seldom

Handles most events by calling the *HandleEvent* method inherited from *TView*. Mouse clicks and "auto" movements over the list change the focused item. The user can select items with double mouse clicks. Keyboard events are handled: *Spacebar* selects the currently focused item; the arrow keys, *PgUp, PgDn, Ctrl+PgDn, Ctrl+PgUp, Home,* and *End* are tracked to set the focused item. Finally, broadcast events from the scroll bars are handled by changing the focused item and redrawing the view as required.

See also: *TView.HandleEvent, TListViewer.FocusItem*

IsSelected

```
function IsSelected(Item: Integer): Boolean; virtual;
```

Override: Sometimes

Returns *True* if the given *Item* is focused, that is, if *Item* = *Focused*. Can be overridden to provide a multiple-selection list viewer.

See also: *TListViewer.FocusItem*

SelectItem

```
procedure SelectItem(Item: Integer); virtual;
```

Override: Sometimes

Selects the *Item*'th item in the list and notifies its peers. The default *SelectItem* method sends a *cmListItemSelected* broadcast to its owner as follows:

```
Message(Owner, evBroadcast, cmListItemSelected, @Self);
```

See also: *TListViewer.FocusItem*

SetRange

```
procedure SetRange(ARange: Integer);
```

Sets *Range* to *ARange*. If the list viewer has a vertical scroll bar, its parameters are adjusted as needed. If the currently focused item falls outside the new *Range, Focused* is set to 0.

See also: *TListViewer.Range, TScrollBar.SetParams*

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Override: Seldom

Calls the *SetState* method inherited from *TView* to change the list viewer object's state if *Enable* is *True*. Depending on the *AState* argument, this can result in displaying or hiding the view. Additionally, if *AState* is *sfSelected* and *sfActive,* the scroll bars are redrawn; if *AState* is *sfSelected* but not *sfActive,* the scroll bars are hidden.

See also: *TView.SetState, TScrollBar.Show, TScrollBar.Hide*

Store

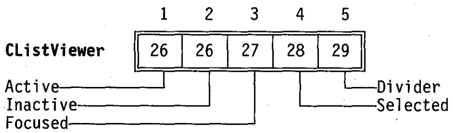
```
procedure Store(var S: TStream);
```

Writes the list viewer object to the stream *S* by first calling the *Store* method inherited from *TView,* then writing the scroll bar objects (if any) by calling *PutPeerViewPtr,* and finally saving the integer fields using *S.Write.*

See also: *TView.Store, TListViewer.Load*

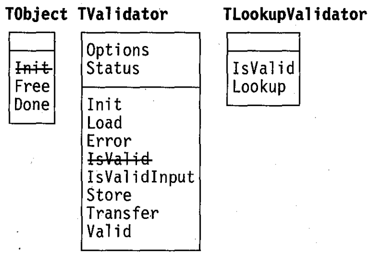
Palette

List viewers use the default palette, *CListViewer,* to map onto the 26th through 29th entries in the standard application palette.



TLookupValidator

Validate



A lookup validator compares the string typed by a user with a list of acceptable values. *TLookupValidator* is an abstract validator type from which you can derive useful lookup validators. You will never create an instance of *TLookupValidator.* When you create a lookup validator type, you need to specify a list of valid items and override the *Lookup* method to return *True* only if the user input matches an item in that list. One example of a working descendant of *TLookupValidator* is *TStringLookupValidator.*

Methods

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Calls *Lookup* to find the string *S* in the list of valid input items. Returns *True* if *Lookup* returns *True,* meaning *Lookup* found *S* in its list; otherwise, returns *False.*

See also: *TLookupValidator.Lookup*

Lookup

```
function Lookup(const S: string): Boolean; virtual;
```

Searches for the string *S* in the list of valid entries and returns *True* if it finds *S*; otherwise, returns *False*. *TLookupValidator*'s *Lookup* is an abstract method that always returns *False*. Descendant lookup validator types must override *Lookup* to perform a search based on the actual list of acceptable items.

TMemo object

Editors

TObject

~~Init~~
Free
~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	~~GetData~~	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
~~ChangeBounds~~	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	~~SetCmdState~~
~~DataSize~~	~~HandleEvent~~	SetCursor
DisableCommands	Hide	~~SetData~~
DragView	HideCursor	~~SetState~~
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	Store
EventAvail	MakeLocal	TopView
Execute	MouseEvent	~~Valid~~
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TEditor

AutoIndent	HScrollBar
Buffer	Indicator
BufLen	InsCount
BufSize	IsValid
CanUndo	Limit
CurPos	Modified
CurPtr	Overwrite
DelCount	Selecting
Delta	SelEnd
DrawLine	SelStart
DrawPtr	VScrollBar
GapLen	

Init	InsertBuffer
~~Load~~	InsertFrom
Done	InsertText
BufChar	ScrollTo
BufPtr	Search
ChangeBounds	SetBufSize
ConvertEvent	SetCmdState
CursorVisible	SetSelect
DeleteSelect	SetState
DoneBuffer	~~Store~~
Draw	TrackCursor
~~GetPalette~~	Undo
~~HandleEvent~~	UpdateCommands
InitBuffer	Valid

TMemo

Load
DataSize
GetData
GetPalette
HandleEvent
SetData
Store

The memo object is a specialized descendant of the standard editor object designed to work like a control inside a dialog box or form. It supports the *Tab* key and the *GetData/SetData* mechanism and has a palette similar to that of an edit object.

Methods

Load

```
constructor Load(var S: TStream);
```

Reads a memo object from the stream *S* by first calling the *Load* constructor inherited from *TEditor*, then reading the length of the text buffer and the text associated with the editor.

See also: *TEditor.Load*

DataSize

```
function DataSize: Word; virtual;
```

Returns the size of the data transferred by *GetData* and *SetData*. By default, that amount is the length of the buffer plus the size of the length word.

See also: *TMemo.GetData, TMemo.SetData*

GetData

```
procedure GetData(var Rec); virtual;
```

Copies *DataSize* bytes data from the editor's text buffer to *Rec*. *GetData* treats *Rec* as a *TMemoData* record, setting the *Length* field to *BufLen*, then copying the text from the text buffer to the *Buffer* field. If the text does not fill the entire buffer, *Rec* is padded with null characters. *GetData* enables your application to read the text from a memo field in a dialog box or data form.

See also: *TMemo.DataSize, TMemo.SetData*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Returns a pointer to *CMemo*, the default memo palette.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Calls the *HandleEvent* method inherited from *TEditor* if the event is not a keystroke or not a *Tab* character. This ensures that the dialog box or window that owns the memo view gets to handle *Tab*.

See also: *TEditor.HandleEvent*

SetData

```
procedure SetData(var Rec); virtual;
```

Copies *DataSize* bytes of information from *Rec* to initialize the data buffer. *SetData* treats *Rec* as a *TMemoData* record, using the *Length* field to set the memo's buffer size and copying the characters in the *Buffer* field to the end of the edit buffer.

Store

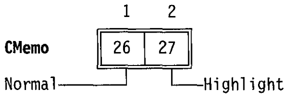
```
procedure Store(var S: TStream);
```

Writes the memo object to the stream *S* by first calling the *Store* method inherited from *TEditor*, then writing the length of the edit buffer and the text from the buffer.

See also: *TEditor.Store*

Palette

Memo objects use the default palette *CMemo* to map onto the 26th and 27th entries in the standard dialog box palette.



TMemoData type — Editors

Declaration

```
TMemoData = record
  Length: Word;
  Buffer: TEditBuffer;
end;
```

Function *TMemo* objects use *TMemoData* records in their *GetData* and *SetData* methods to read or write the length of their text buffers and the actual text of the buffer.

TMenu type — Menus

Declaration

```
TMenu = record
  Items: PMenuItem;
  Default: PMenuItem;
end;
```

Function The *TMenu* type represents one level of a menu tree. The *Items* field points to a list of *TMenuItem* records, and the *Default* field points to the default item within that list (the one to select by default when bringing up this menu). A *TMenuView* object (of which *TMenuBar* and *TMenuBox* are descandants) has a *Menu* field that points to a *TMenu*. *TMenu* records are created and destroyed using the *NewMenu* and *DisposeMenu* routines.

See also *TMenuView, TMenuItem, NewMenu, DisposeMenu, TMenuView.Menu* field

TMenuBar Menus

TObject

~~Init~~
Free
~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	SetState
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
~~Execute~~	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TMenuView

ParentMenu
Menu
Current

~~Init~~
Load
Execute
FindItem
~~GetItemRect~~
GetHelpCtx
GetPalette
HandleEvent
HotKey
NewSubView
Store

TMenuBar

Init
Done
Draw
GetItemRect

TMenuBar objects represent the horizontal menu bars from which menu selections can be made by

- Direct clicking
- *F10* selection and shortcut keys
- Selection (highlighting) and pressing *Enter*
- Hot keys

The main menu selections are displayed in the top menu bar. This is represented by an object of type *TMenuBar* usually owned by your *TApplication* object. Submenus are displayed in objects of type *TMenuBox*. Both *TMenuBar* and *TMenuBox* are descendants of the abstract type *TMenuView* (a child of *TView*).

For most Turbo Vision applications, you will not be involved directly with menu objects. Once you override the application's *InitMenuBar* method to set up a menu structure using nested *New*, *NewSubMenu*, *NewItem* and *NewLine* calls, the default menu behavior handles the creation and management of menu views.

Methods

Init

```
constructor Init(var Bounds: TRect; AMenu: PMenu);
```

Constructs a menu bar with the given *Bounds* by calling the *Init* constructor inherited from *TMenuView*. Sets *GrowMode* to *gfGrowHiX*. Sets *Options* to *ofPreProcess* to allow hot keys to operate. Sets *Menu* to *AMenu*, providing the menu items.

See also: *TMenuView.Init*, *gfXXXX* grow mode flags, *ofXXXX* option flags, *TMenuView.Menu*

Done

```
destructor Done; virtual;
```

Disposes of the menu object by first calling the *Done* destructor inherited from *TMenuView*, then calling *DisposeMenu* to dispose of the lists of menu items.

See also: *TMenuView.Done*, *DisposeMenu* procedure

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the menu bar with the default palette. The *Name* and *Disabled* fields of each *TMenuItem* record in the linked list are read to give the menu legends in the correct colors. The *Current* (selected) item is highlighted.

GetItemRect

```
procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
```

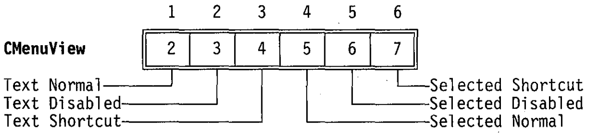
Override: Never

Overrides the abstract method in *TMenuView*. Returns the rectangle occupied by the given menu item in *R*. *HandleEvent* uses *GetItemRect* to determine if a mouse click occurred on a given menu item.

See also: *TMenuView.GetItemRect*

Palette

Menu bars, like all menu views, use the default palette *CMenuView* to map onto the 2nd through 7th entries in the standard application palette.



TMenuBox

Menus

TObject
~~Init~~ Free ~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	SetState
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
~~Execute~~	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TMenuView
ParentMenu Menu Current
~~Init~~ Load Execute FindItem ~~GetItemRect~~ GetHelpCtx GetPalette HandleEvent HotKey NewSubView Store

TMenuBox
Init Draw GetItemRect

TMenuBox objects represent vertical menu boxes. These can contain arbitrary lists of selectable actions, including submenu items. As with menu bars, color coding is used to indicate disabled items. Menu boxes can be instantiated as submenus of the menu bar or other menu boxes, or can be used alone as pop-up menus.

Methods

Init

```
constructor Init(var Bounds: TRect; AMenu: PMenu; AParentMenu: PMenuView);
```

Init adjusts the *Bounds* parameter to accommodate the width and length of the items in *AMenu*, then creates a menu box by calling the *Init* constructor inherited from *TMenuView.*

Sets the *ofPreProcess* bit in *Options* so that hot keys will operate. Sets *State* to include *sfShadow*. Sets *Menu* to *AMenu*, which provides the menu selections, and *ParentMenu* to *AParentMenu.*

See also: *TMenuView.Init*, *sfXXXX* state flags, *ofXXXX* option flags, *TMenuView.Menu*, *TMenuView.ParentMenu*

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the framed menu box and menu items in the default colors.

GetItemRect

```
procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
```

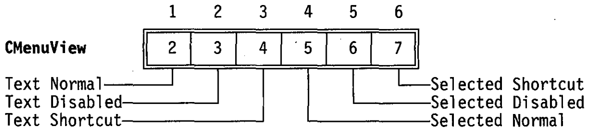
Override: Seldom

Overrides the abstract method in *TMenuView*. Returns the rectangle occupied by the given menu item. *HandleEvent* calls *GetItemRect* to determine if a mouse click occurred on a given menu item.

See also: *TMenuView.GetItemRect*

Palette

Menu boxes, like all menu views, use the default palette *CMenuView* to map onto the 2nd through 7th entries in the standard application palette.



TMenuItem type

Menus

Declaration

```
TMenuItem = record
  Next: PMenuItem;
  Name: PString;
  Command: Word;
  Disabled: Boolean;
  KeyCode: Word;
  HelpCtx: Word;
  case Integer of
    0: (Param: PString);
    1: (SubMenu: PMenu);
  end;
end;
```

Function

The *TMenuItem* type represents a menu item, which can be either a normal item, a submenu, or a divider line. *Next* points to the next *TMenuItem* in a list of menu items, or is **nil** if this is the last item. *Name* points to a string containing the menu item name, or is **nil** if the menu item is a divider line. *Command* contains the command event (see *cmXXXX* constants) to be generated when the menu item is selected, or zero if the menu item represents a submenu. *Disabled* is *True* if the menu item is disabled, *False* otherwise. *KeyCode* contains the scan code of the hot key associated with the menu item, or zero if the menu item has no hot key. *HelpCtx* contains the menu item's help context number (a value of *hcNoContext* indicates that the menu item has no help context). If the menu item is a normal

T

item, *Param* contains a pointer to a parameter string (displayed to the right of the item in a *TMenuBox*), or is **nil** if the item has no parameter string. If the menu item is a submenu (*Command* = 0), *SubMenu* points to the submenu structure.

TMenuItem records are created using the *NewItem*, *NewLine*, and *NewSubMenu* functions.

See also *TMenu, TMenuView, NewItem, NewLine, NewSubMenu*

TMenuStr type — Menus

Declaration `TMenuStr = string[31];`

Function A string type used by *NewItem* and *NewSubMenu*. The maximum menu item title is 31 characters.

See also *NewItem, NewSubMenu*

TMenuView — Menus

TObject

~~Init~~ Free ~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	SetState
Draw	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
~~Execute~~	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TMenuView

ParentMenu Menu Current
Init Load Execute FindItem GetItemRect GetHelpCtx GetPalette HandleEvent HotKey NewSubView Store

TMenuView provides an abstract menu type from which menu bars and menu boxes (either pull-down or popup) are derived. You will probably never construct an instance of *TMenuView* itself.

Fields

Current

`Current: PMenuItem;` **Read only**

A pointer to the currently selected menu item.

Menu

`Menu: PMenu;` **Read only**

Points to the *TMenu* record for this menu, which holds a linked list of menu items. The *Menu* pointer allows access to all the fields of the menu items in this menu view.

See also: *TMenuView.FindItem, TMenuView.GetItemRect, TMenu* type

ParentMenu

`ParentMenu: PMenuView;` **Read only**

Points to the menu view that owns this menu. Note that *TMenuView* is not a group. Ownership here is a much simpler concept than *TGroup* ownership, allowing menu nesting: the selection of submenus and the return back to the "parent" menu. Selections from menu bars, for example, usually result in a submenu being "pulled down." The menu bar in that case is the parent menu of the menu box.

See also: *TMenuBox.Init*

Methods

Init

```
constructor Init(var Bounds: TRect);
```

Constructs a menu view of size *Bounds* by calling the *Init* constructor inherited from *TView*. Sets *EventMask* to *evBroadcast*. This method is not intended to be used for constructing instances of *TMenuView;* rather it should be called by descendant types, such as *TMenuBar* and *TMenuBox*.

See also: *TView.Init, evBroadcast, TMenuBar.Init, TMenuBox.Init*

Load

```
constructor TMenuView.Load(var S: TStream);
```

Creates a menu view object and loads it from the stream *S* by first calling the *Load* constructor inherited from *TView* and then reading the items in the menu list.

See also: *TView.Load, TMenuView.Store*

Execute

```
function Execute: Word; virtual;
```

Override: Never

Executes a menu view until the user selects a menu item or cancels the process. Returns the command assigned to the selected menu item, or zero if the menu was canceled. This method should *never* be called except by *ExecView*.

See also: *TGroup.ExecView*

FindItem

```
function FindItem(Ch: Char): PMenuItem;
```

Returns a pointer to the menu item that has *Ch* as its shortcut key (the highlighted character). Returns **nil** if no such menu item is found or if the menu item is disabled. Note that *Ch* is case-insensitive.

GetItemRect

```
procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
```

Override: Always

This method returns the rectangle occupied by the given menu item in *R*. It is used to determine if a mouse click has occurred on a given menu selection. Descendants of *TMenuView must* override this method in order to respond to mouse events.

See also: *TMenuBar.GetItemRect, TMenuBox.GetItemRect*

GetHelpCtx

```
function GetHelpCtx: Word; virtual;
```

Override: Sometimes

By default, *GetHelpCtx* returns the help context of the current menu item. If this is *hcNoContext*, the parent menu's current context is checked. If there is no parent menu, *GetHelpCtx* returns *hcNoContext*.

See also: *hcXXXX* help context constants

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default *CMenuBar* palette.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Never

Called whenever a menu event needs to be handled. Determines which menu item has been selected with the mouse or keyboard (including hot keys) and generates the linked command by calling *PutEvent*. Also responds to *cmCommandSetChanged* by updating the active items if necessary.

See also: *TView.HandleEvent, TView.PutEvent*

HotKey

```
function HotKey(KeyCode: Word): PMenuItem;
```

Returns a pointer to the menu item associated with the hot key given by *KeyCode*. Returns **nil** if no such menu item exists, or if the item is disabled. Hot keys are usually function keys or *Alt+* key combinations, determined by arguments in *NewItem* and *NewSubMenu* calls during *InitMenuBar*. *HandleEvent* uses *HotKey* to determine whether a keystroke event selects an item in the menu.

NewSubView

```
function NewSubView(var Bounds: TRect; AMenu: PMenu;
  AParentMenu: PMenuView): PMenuView; virtual;
```

Constructs a new menu box with the given *Bounds, AMenu,* and *AParentMenu,* and returns a pointer to it.

Store

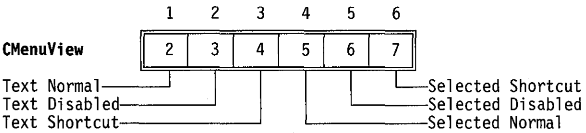
```
procedure Store(var S: TStream);
```

Writes the menu view object (and any of its submenus) to the stream *S* by first calling the *Store* method inherited from *TView,* then writing each menu item to the stream.

See also: *TMenuView.Load*

Palette

All menu views use the default palette *CMenuView* to map onto the 2nd through 7th entries in the standard application palette.



TMonoSelector object ColorSel

A monochrome selector view enables a user to select monochrome video attributes for displayed items, much as one would select colors for those same items on a color display. The possible attributes are normal, highlighted, underlined, or inverse video. Although a monochrome selector looks like a set of radio buttons, it actually descends directly from *TCluster*.

Details about *TMonoSelector* are included in the online Help.

TMultiCheckBoxes Dialogs

TObject	TView			TCluster	TMultiCheckBoxes
	Cursor	HelpCtx	Owner	EnableMask	Flags
~~Init~~	DragMode	Next	Size	Sel	SelRange
Free	EventMask	Options	State	Strings	States
~~Done~~	GrowMode	Origin		Value	
	~~Init~~	GetCommands	Prev	~~Init~~	Init
	~~Load~~	~~GetData~~	PrevView	~~Load~~	Load
	~~Done~~	GetEvent	PutEvent	~~Done~~	Done
	Awaken	GetExtent	PutInFrontOf	ButtonState	DataSize
	BlockCursor	~~GetHelpCtx~~	PutPeerViewPtr	~~DataSize~~	Draw
	CalcBounds	~~GetPalette~~	Select	DrawBox	GetData
	ChangeBounds	GetPeerViewPtr	SetBounds	DrawMultiBox	MultiMark
	ClearEvent	GetState	SetCommands	~~GetData~~	Press
	CommandEnabled	GrowTo	SetCmdState	GetHelpCtx	SetData
	~~DataSize~~	~~HandleEvent~~	SetCursor	GetPalette	Store
	DisableCommands	Hide	~~SetData~~	HandleEvent	
	DragView	HideCursor	~~SetState~~	Mark	
	~~Draw~~	KeyEvent	Show	MovedTo	
	DrawView	Locate	ShowCursor	~~MultiMark~~	
	EnableCommands	MakeFirst	SizeLimits	~~Press~~	
	EndModal	MakeGlobal	~~Store~~	SetButtonState	
	EventAvail	MakeLocal	TopView	~~SetData~~	
	Execute	MouseEvent	Valid	SetState	
	Exposed	MouseInView	WriteBuf	~~Store~~	
	Focus	MoveTo	WriteChar		
	GetBounds	NextView	WriteLine		
	GetClipRect	NormalCursor	WriteStr		
	GetColor				

Fields

Flags

```
Flags: Word;
```

Flags is a bitmapped field that holds one of the *cfXXXX* constants.

See also: *cfXXXX* constants

SelRange

```
SelRange: Byte;
```

SelRange is the actual number of states a check box in the cluster can assume.

States

```
States: PString;
```

States points to a string holding characters to represent each of the check box's possible states.

Methods

Init

```
constructor Init(var Bounds: TRect; AStrings: PSItem;
  ASelRange: Byte; AFlags: Word; const AStates: String);
```

Constructs a cluster of multistate check boxes by first calling the *Init* constructor inherited from *TCluster*, then setting the *SelRange*, and *Flags* fields to the values passed in *ASelRange* and *AFlags*, respectively, and allocating a dynamic copy of *AStates* and assigning it to *States*.

ASelRange indicates the number of states each check box can have. *AFlags* is one of the *cfXXXX* constants, indicating how many bits in *Value* represent each check box. *AStates* has a character to display for each possible state.

See also: *TCluster.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs a cluster of multistate check boxes and loads it from the stream *S* by first calling the *Load* constructor inherited from *TCluster*, then reading the values of the fields introduced by *TMultiCheckBoxes*.

See also: *TCluster.Load*

Done

```
destructor Done; virtual;
```

Disposes of the multistate check boxes object by first deallocating the dynamic string *States*, then calling the *Done* destructor inherited from *TCluster*.

DataSize

```
function DataSize: Word; virtual;
```

Returns the size of the data transferred by *GetData* and *SetData*, which is *SizeOf(Longint)*.

See also: *TMultiCheckBoxes.GetData, TMultiCheckBoxes.SetData*

Draw

```
procedure Draw; virtual;
```

Draws the cluster of multistate check boxes by drawing each check box in turn, using the same box as a regular check box, but using the characters in *States* to represent the state of each box instead of the standard blank and 'X'.

GetData

```
procedure GetData(var Rec); virtual;
```

Typecasts *Rec* into a *Longint* and copies *Value* into it.

MultiMark

```
function MultiMark(Item: Integer): Byte; virtual;
```

Returns the state of the *Item*th check box in the cluster.

Press

```
procedure Press(Item: Integer); virtual;
```

Changes the state of the *Item*th check box in the cluster. Unlike regular check boxes that simply toggle on and off, multistate check boxes cycle through all the states available to them.

SetData

```
procedure SetData(var Rec); virtual;
```

Typecasts *Rec* into a *Longint*, and copies its value into *Value*, then calls *DrawView* to redraw the checkboxes to reflect their new states.

Store

```
procedure Store(var S: TStream);
```

Writes the cluster of multistate check boxes to the stream *S* by first calling the *Store* method inherited from *TCluster*, then writing the fields introduced by *TMultiCheckBoxes*.

See also: *TCluster.Store*

TNode type — Outline

Declaration

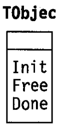
```
TNode = record
  Next: PNode;
  Text: PString;
  ChildList: PNode;
  Expanded: Boolean;
end;
```

Function Outline objects use records of type *TNode* to hold the lists of linked strings that form the outline. Each node record holds the text for that item in the outline in its *Text* field. *ChildList* points to the first in a list of subordinate nodes, or holds **nil** if there are no items subordinate to the node. *Next* points to the next node at the same outline level as the current node. *Expanded* is *True* if the outline view shows the subordinate views listed in *ChildList* or *False* if the subordinate nodes are hidden.

When creating your outline list, allocate new nodes using the *NewNode* function, and dispose of the nodes with *DisposeNode*.

See also *DisposeNode* procedure, *NewNode* function

TObject — Objects



TObject is the starting point of Turbo Vision's object hierarchy. As the base object, it has no ancestor but many descendants. Apart from *TPoint* and

TRect, in fact, all of Turbo Vision's standard objects are ultimately derived from *TObject*. Any object that uses Turbo Vision's stream facilities *must* trace its ancestry back to *TObject*.

Methods

Init `constructor Init;`

Allocates space on the heap for the object and fills it with zeros. Called by all derived objects' constructors.

☞ *TObject.Init* will zero all fields in descendants, so you should always call *TObject.Init before* initializing any fields in the derived objects' constructors.

Done `destructor Done; virtual;`

Performs the necessary cleanup and disposal for dynamic objects.

Free `procedure Free;`

Disposes of the object and calls the *Done* destructor.

TOutline Outline

TObject
~~Init~~ Free ~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
Load	GetData	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	~~SetState~~
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	Store
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TOutlineViewer
Foc
~~Init~~ ~~Adjust~~ CreateGraph Draw ExpandAll FirstThat Focused ForEach GetChild GetGraph ~~GetNumChildren~~ GetNode GetPalette ~~GetRoot~~ ~~GetText~~ HandleEvent ~~HasChildren~~ ~~IsExpanded~~ IsSelected Selected SetState Update

TOutline
Root
Init Done Adjust GetChild GetNumChildren GetRoot GetText HasChildren IsExpanded

T

TOutline implements a simple but useful type of outline viewer. It assumes that the outline is a linked list of records of type *TNode*, so each node consists of a text string, a pointer to any child nodes, and a pointer to the next node at the same level.

Field

Root

```
Root: PNode;
```

Points to the root node of the outline tree.

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar,
  AVScrollBar: PScrollBar; ARoot: PNode);
```

Constructs an outline view by passing *Bounds*, *AHScrollBar*, and *AVScrollBar* to the *Init* constructor inherited from *TOutlineViewer*. Sets *Root* to *ARoot*, then calls *Update* to set the scrolling limits of the view based on the data in the outline.

See also: *TScroller.Init*

Done

```
destructor Done; virtual;
```

Disposes of the outline view by first disposing of the root node, which recursively disposes of all child nodes, then calling the *Done* destructor inherited from *TScroller*.

See also: *TScroller.Done*

Adjust

```
procedure Adjust(Node: Pointer; Expand: Boolean); virtual;
```

Sets the *Expanded* field of *Node* to the value passed in *Expand*. If *Expand* is *True*, this causes the child nodes linked to *Node* to be displayed. If *Expand* is *False*, *Node*'s child nodes are hidden.

GetRoot

```
function GetRoot: Pointer; virtual;
```

Returns *Root*, which points to the top of the list of nodes for the outline.

GetNumChildren

```
function GetNumChildren(Node: Pointer): Integer; virtual;
```

Returns the number of nodes in *Node*'s *ChildList*, or zero if *ChildList* is **nil**.

GetChild

```
function GetChild(Node: Pointer; I: Integer): Pointer; virtual;
```

Returns a pointer to the *I*th child in *Node*'s *ChildList*.

GetText

```
function GetText(Node: Pointer): String; virtual;
```

Returns the string pointed to by *Node*'s *Text* field.

HasChildren

```
function HasChildren(Node: Pointer): Boolean; virtual;
```

Returns *True* if *Node*'s *ChildList* is non-**nil**; otherwise returns *False*.

IsExpanded

```
function IsExpanded(Node: Pointer): Boolean; virtual;
```

Returns the value of *Node*'s *Expanded* field.

TOutlineViewer — Outline

TObject

~~Init~~ Free ~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
Load	GetData	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	~~SetState~~
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	Store
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TOutlineViewer

Foc

Init
Adjust
CreateGraph
Draw
ExpandAll
FirstThat
Focused
ForEach
GetChild
GetGraph
GetNumChildren
GetNode
GetPalette
GetRoot
GetText
HandleEvent
HasChildren
IsExpanded
IsSelected
Selected
SetState
Update

The outline viewer object type provides the abstract methods for displaying, expanding, and iterating the items in an outline. *TOutlineViewer*, however, makes no assumptions about the actual items in the outline. Descendants of *TOutlineViewer*, such as *TOutline*, display outlines of specific kinds of items.

Field

Foc

```
Foc: Integer;
```

Indicates the item number of the focused node in the outline.

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar);
```

Constructs an outline viewer object by first calling the *Init* constructor inherited from *TScroller*, passing *Bounds*, *AHScrollBar*, and *AVScrollBar*. Sets *GrowMode* to *gfGrowHiX* + *gfGrowHiY* and sets *Foc* to zero.

See also: *TScroller.Init*

Adjust

```
procedure Adjust(Node: Pointer; Expand: Boolean); virtual;
```

Adjust is an abstract method that descendant outline viewer types must override to show the child nodes of *Node* if *Expand* is *True*, or hide them if *Expand* is *False*. Called when the user expands or collapses *Node*. If *HasChildren* returns *False* for *Node*, *Adjust* will never be called for that node.

See also: *TOutlineViewer.HasChildren*

CreateGraph

```
function CreateGraph(Level: Integer; Lines: Longint; Flags: Word;
  LevWidth, EndWidth: Integer; const Chars: String): String;
```

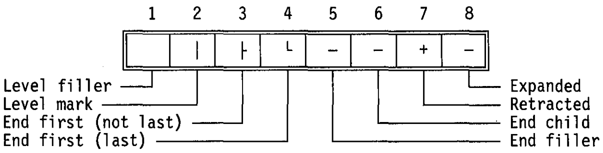
Draws a graph string suitable for returning from *GetGraph*. *Level* indicates the outline level. *Lines* is the set of bits decribing the levels which have a "continuation" mark (usually a vertical line). For example, if bit 3 is set, level 3 is continued beyond this level.

Flags gives extra information about how to draw the end of the graph (see the *ovXXXX* constants). *LevWidth* is how many characters to indent for each level. *EndWidth* is the length of the end characters.

The graphic is divided into two parts: the level marks, and the end or node graphic. The level marks consist of the Level Mark character separated by Level Filler. What marks are present is determined by *Lines*.

The end graphic is constructed by placing one of the End First characters followed by *EndWidth* – 4 End Filler characters, followed by the End Child character, followed by the Retract/Expand character. If *EndWidth* equals 2, End First and Retract/Expand are used. If *EndWidth* equals 1, only the Retract/Expand character is used. Which characters are selected is determined by *Flags*.

The layout for the characters in the *Chars* string is:



Character	Typical	Description
Level filler	' '	Used between level markers.
Level Mark	'|'	Used to mark the levels currently active.
End First (not last child)	'├'	Used as the first character of the end part of a node graphic if the node is not the last child of the parent.
End First (last child)	'└'	Used as the first character of the end part of a node graphic if the node is the last child of the parent.
End Filler	'–'	Used as filler for the end part of a node graphic.
End Child	'–'	If *EndWidth* > *LevWidth*, this character will be placed on top of the markers for next level. If used it is typically a T.
Retracted	'+'	Displayed as the last character of the end node if the level has children and they are not expanded.
Expanded	'–'	Displayed as the last character of the end node if the level has children and they are expanded.

For example, *GetGraph* calls *CreateGraph* with the following parameters:

```
CreateGraph(Level, Lines, Flags, 3, 3, ' '#179#195#192#196#196'+'#196);
```

To use double lines instead of single lines use:

```
CreateGraph(Level, Lines, Flags, 3, 3, ' '#186#204#200#205#205'+'#205);
```

To have the children line drop off prior to the text instead of underneath, use the following call:

```
CreateGraph(Level, Lines, Flags, 2, 4, ' '#179#195#192#196#194'+'#196);
```

Draw

```
procedure Draw; virtual;
```

Called to draw the outline view. Essentially, *Draw* calls *GetGraph* to get the graphical part of the outline, then appends the string returned from *GetText*.

The line containing the focused node in the outline displays in a distinct color. Nodes whose child nodes are not displayed are highlighted.

See also: *TOutlineViewer.GetGraph, TOutlineViewer.GetText*

ExpandAll

```
procedure ExpandAll(Node: Pointer);
```

If *Node* has child nodes, *ExpandAll* recursively expands *Node* by calling *Adjust* with the *Expand* parameter *True*, then expands all its child nodes by calling *ExpandAll* for each of them.

See also: *TOutlineViewer.Adjust*

FirstThat

```
function FirstThat(Test: Pointer): Pointer;
```

FirstThat iterates over the nodes of the outline, calling the function pointed to by *Test* until *Test* returns *True*. *Test* must point to a **far** local function with the following syntax:

```
function MyIter(Cur: Pointer; Level, Position: Integer;
  Lines: LongInt; Flags: Word): Boolean; far;
```

The parameters are as follows:

Cur A pointer to the node being checked.

Level The level of the node (how many nodes are above it), zero-based. This can be used in a call to either *GetGraph* or *CreateGraph*.

Position The display order position of the node in the list. This can be used in a call to *Focused* or *Selected*. If in range, *Position – Delta.Y* is location the node is displayed on the view.

Lines Bits indicating the active levels. This can be used in a call to *GetGraph* or *CreateGraph*. It dictates which horizontal lines need to be drawn.

Flags Various flags for drawing (see *ovXXXX* flags). Can be used in a call to *GetGraph* or *CreateGraph*.

Focused

```
procedure Focused(I: Integer); virtual;
```

Called whenever a node receives focus. The *I* parameter indicates the position of the newly focused node in the outline. By default, *Focused* just sets *Foc* to *I*.

ForEach

```
function ForEach(Action: Pointer): Pointer;
```

Iterates over all the nodes. *Action* points to a **far** local procedure that *ForEach* calls for each node in the outline. The syntax for the iterator procedure is as follows:

```
procedure MyIter(Cur: Pointer; Level, Position: Integer;
  Lines: LongInt; Flags: Word); far;
```

The parameters are as follows:

Cur A pointer to the node being checked.

Level The level of the node (how many nodes are above it), zero-based. This can be used in a call to either *GetGraph* or *CreateGraph*.

Position The display order position of the node in the list. This can be used in a call to *Focused* or *Selected*. If in range, *Position – Delta.Y* is location the node is displayed on the view.

Lines Bits indicating the active levels. This can be used in a call to *GetGraph* or *CreateGraph*. It dictates which horizontal lines need to be drawn.

Flags Various flags for drawing (see *ovXXXX* flags). Can be used in a call to *GetGraph* or *CreateGraph*.

GetChild

```
function GetChild(Node: Pointer; I: Integer): Pointer; virtual;
```

GetChild is an abstract method that descendant outline viewer types must override to return a pointer to the *I*th child of the given *Node*.

If *HasChildren* returns *False*, indicating that *Node* has no child nodes, *GetChild* will not be called for that node. You can safely assume that when an outline viewer calls *GetChild*, the given node has at least *I* child nodes.

See also: *TOutlineViewer.HasChildren*

GetGraph

```
function GetGraph(Level: Integer; Lines: Longint; Flags: Word): String;
```

Returns a string of graphics characters to display to the left of the text returned by *GetText*. By default, *GetGraph* calls *CreateGraph* with the default character values. You only need to override *GetGraph* if you want to change the appearance of the outline.

For example, instead of calling *CreateGraph* to show the hierarchy, you might return a string of characters to merely indent the text by a given amount for each level.

GetNumChildren

```
function GetNumChildren(Node: Pointer): Integer; virtual;
```

GetNumChildren is an abstract method that descendant outline viewer types must override to return the number of child nodes in *Node*. If *HasChildren* returns *False* for *Node*, *GetNumChildren* will never be called.

See also: *TOutlineViewer.HasChildren*

GetNode

```
function GetNode(I: Integer): Pointer;
```

Returns a pointer to the *I*th node in the outline; that is, the node shown *I* lines from the top of the complete outline.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Returns a pointer to the default outline palette, *COutlineViewer*.

GetRoot

```
function GetRoot: Pointer; virtual;
```

GetRoot is an abstract method that descendant outline viewer types must override to return a pointer to the root node of the outline.

GetText

```
function GetText(Node: Pointer): String; virtual;
```

GetText is an abstract method that descendant outline viewer types must override to return the text of *Node*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles most events for the outline viewer by calling the *HandleEvent* method inherited from *TScroller*, then handles certain mouse and keyboard events.

HasChildren

```
function HasChildren(Node: Pointer): Boolean; virtual;
```

HasChildren is an abstract method that descendant outline viewers must override to return *True* if the given *Node* has child nodes and *False* if *Node* has no child nodes.

If *HasChildren* returns *False* for a particular node, the following functions are never called for that node: *Adjust, ExpandAll, GetChild, GetNumChildren,* and *IsExpanded.*

Those methods can assume that if they are called, there are child nodes for them to act on.

IsExpanded

```
function IsExpanded(Node: Pointer): Boolean; virtual;
```

IsExpanded is an abstract method that descendant outline viewer types must override to return *True* if *Node*'s child nodes should be displayed. If *HasChildren* returns *False* for *Node, IsExpanded* will never be called for that node.

IsSelected

```
function IsSelected(I: Integer): Boolean; virtual;
```

Returns *True* if *Node* is selected. By default, *TOutlineViewer* assumes a single-selection outline, so it returns *True* if *Node* is *Focused*. You can override *IsSelected* to handle multiple selections.

Selected

```
procedure Selected(I: Integer); virtual;
```

Called whenever a node is selected by the user, either by keyboard control or by the mouse. The *I* parameter indicates the position in the outline of the newly selected node.

By default, *Selected* does nothing; descendant types can override *Selected* to perform some action in response to selection.

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Sets or clears the *AState* state flags for the view by calling the *SetState* method inherited from *TScroller*. If the new state includes a focus change, *SetState* calls *DrawView* to redraw the outline.

See also: *TScroller.SetState*

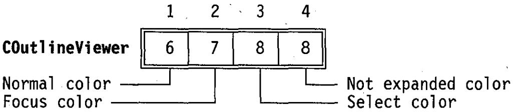
Update `procedure Update;`

Updates the limits of the outline viewer. The limit in the vertical direction is number of nodes in the outline. The limit in the horizontal direction is the length of the longest displayed line.

Your program should call *Update* whenever the data in the outline changes. *TOutlineViewer* assumes that the outline is empty, so if the outline becomes non-empty during initialization, you must explicitly call *Update*. Also, if during the operation of the outline viewer the data being displayed change, you must call *Update* and the inherited *DrawView*.

Palette

Outline viewer objects use the default palette *COutlineViewer* to map onto the 6th through 8th entries in the standard window palette.



TPalette type

Views

Declaration `TPalette = String;`

Function A string type used to declare Turbo Vision palettes.

See also *GetPalette* methods

TParamText

Dialogs

TObject
~~Init~~ Free ~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
~~DataSize~~	HandleEvent	SetCursor
DisableCommands	Hide	~~SetData~~
DragView	HideCursor	SetState
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TStaticText
Text
~~Init~~ ~~Load~~ Done Draw GetPalette ~~GetText~~ ~~Store~~

TParamText
ParamCount ParamList
Init Load DataSize GetText SetData Store

TParamText is a derivative of *TStaticText* that uses parameterized text strings for formatted output, using the *FormatStr* procedure.

Fields

ParamCount

```
ParamCount: Integer;
```

ParamCount indicates the number of parameters contained in *ParamList*.

See also: *TParamText.ParamList*

ParamList

```
ParamList: Pointer;
```

ParamList is an untyped pointer to an array or record of pointers or *Longint* values to be used as formatted parameters for a text string.

Methods

Init

```
constructor Init(var Bounds: TRect; const AText: String; AParamCount: Integer);
```

Constructs a static text object by calling the *Init* constructor inherited from *TStaticText* with the given *Bounds* and a text string, *AText,* that may contain format specifiers in the form `%[-][nnn]X`, which will be replaced by the parameters passed at run time. The parameter count, passed in *AParamCount*, is assigned to the *ParamCount* field.

Format specifiers are described in detail in the entry for the *FormatStr* procedure.

See also: *TStaticText.Init*, *FormatStr* procedure

Load

```
constructor Load(var S: TStream);
```

Constructs a *TParamText* object and loads its value from the stream *S* by first calling the *Load* constructor inherited from *TStaticText* and then reading the *ParamCount* field from the stream.

See also: *TStaticText.Load*

DataSize

```
function DataSize: Word; virtual;
```

Returns the size of the data required by the object's parameters, that is, *ParamCount * SizeOf(Longint)*.

GetText

```
procedure GetText(var S: String); virtual;
```

Produces a formatted text string in *S*, produced by merging the parameters contained in *ParamList* into the text string in *Text*, using *FormatStr(S, Text^, ParamList^)*.

See also: *FormatStr* procedure

SetData

```
procedure SetData(var Rec); virtual;
```

The view reads *DataSize* bytes into *ParamList* from *Rec*.

See also: *TView.SetData*

Store

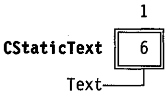
```
procedure Store(var S: TStream);
```

Stores the object on the stream *S* by first calling the *Store* method inherited from *TStaticText* and then writing the *ParamCount* field to the stream.

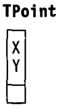
See also: *TStaticText.Store*

Palette

TParamText objects use the default palette *CStaticText* to map onto the sixth entry in the standard dialog palette.



TPoint Objects



TPoint is a simple object representing a point on the screen.

Fields

X

```
X: Integer
```

X is the screen column of the point.

Y

```
Y: Integer
```

Y is the screen row of the point.

TPicResult type Validate

Declaration

```
TPicResult = (prComplete, prIncomplete, prEmpty, prError, prSyntax, prAmbiguous,
prIncompNoFill);
```

Function *TPicResult* is the result type returned by the *Picture* method of *TPXPictureValidator*.

See also *TPXPictureValidator.Picture*

TProgram App

TObject

- ~~Init~~
- Free
- ~~Done~~

TView

- Cursor
- DragMode
- EventMask
- GrowMode
- HelpCtx
- Next
- Options
- Origin
- Owner
- Size
- State

- ~~Init~~
- ~~Load~~
- ~~Done~~
- ~~Awaken~~
- BlockCursor
- CalcBounds
- ~~ChangeBounds~~
- ClearEvent
- CommandEnabled
- ~~DataSize~~
- DisableCommands
- DragView
- ~~Draw~~
- DrawView
- EnableCommands
- ~~EndModal~~
- EventAvail
- ~~Execute~~
- Exposed
- Focus
- GetBounds
- GetClipRect
- GetColor
- GetCommands
- ~~GetData~~
- ~~GetEvent~~
- GetExtent
- ~~GetHelpCtx~~
- ~~GetPalette~~
- GetPeerViewPtr
- GetState
- GrowTo
- ~~HandleEvent~~
- Hide
- HideCursor
- KeyEvent
- Locate
- MakeFirst
- MakeGlobal
- MakeLocal
- MouseEvent
- MouseInView
- MoveTo
- NextView
- NormalCursor
- Prev
- PrevView
- PutEvent
- PutInFrontOf
- PutPeerViewPtr
- Select
- SetBounds
- SetCommands
- SetCmdState
- SetCursor
- ~~SetData~~
- ~~SetState~~
- Show
- ShowCursor
- SizeLimits
- ~~Store~~
- TopView
- ~~Valid~~
- WriteBuf
- WriteChar
- WriteLine
- WriteStr

TGroup

- Buffer
- Current
- Last
- Phase

- ~~Init~~
- Load
- ~~Done~~
- Awaken
- ChangeBounds
- DataSize
- Delete
- Draw
- EndModal
- EventError
- ExecView
- Execute
- First
- FirstThat
- FocusNext
- ForEach
- GetData
- GetHelpCtx
- GetSubViewPtr
- ~~HandleEvent~~
- Insert
- InsertBefore
- Lock
- PutSubViewPtr
- Redraw
- SelectNext
- SetData
- SetState
- Store
- Unlock
- Valid

TProgram

- Init
- Done
- CanMoveFocus
- ExecuteDialog
- GetEvent
- GetPalette
- HandleEvent
- Idle
- InitDeskTop
- InitMenuBar
- InitScreen
- InitStatusLine
- InsertWindow
- OutOfMemory
- PutEvent
- Run
- SetScreenMode
- ValidView

TProgram provides the basic template for all standard Turbo Vision applications. All such programs must be derived from *TProgram* or its descendant, *TApplication*. *TApplication* differs from *TProgram* only in its default constructor and destructor methods. Both object types are provided for added flexibility when designing nonstandard applications. For most Turbo Vision work, your program will be derived from *TApplication*.

TProgram is a *TGroup* descendant because it needs to contain your desktop, status line, and menu bar views.

☞ The base application object *TProgram* has three new methods in version 2.0. *CanMoveFocus* is used internally by the application to determine whether it can activate a window when the user is in a validating window. The other two methods are the safe methods of inserting windows and executing dialog boxes on the desktop.

Methods

Init

```
constructor Init;
```

Override: Sometimes

Sets the *Application* global variable to *@Self*; calls *InitScreen* to initialize screen mode dependent variables; calls the *Init* constructor inherited from *TGroup*, passing a *Bounds* rectangle covering the full screen; sets *State* to *sfVisible* + *sfSelected* + *sfFocused* + *sfModal* + *sfExposed*; sets *Options* to 0; sets *Buffer* to the address of the screen buffer given by *ScreenBuffer*; and finally calls *InitDesktop*, *InitStatusLine*, and *InitMenuBar*, and inserts the resulting views into the *TProgram* group.

See also: *TGroup.Init*, *TProgram.InitDesktop*, *TProgram.InitStatusLine*, *TProgram.InitMenuBar*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes the *Desktop*, *MenuBar*, and *StatusLine* objects, and sets the *Application* global variable to **nil**, then calls the *Done* destructor inherited from *TGroup*.

See also: *TGroup.Done*

CanMoveFocus

```
function CanMoveFocus: Boolean;
```

CanMoveFocus returns *True* if the desktop can safely change its selected window. The method determines whether such a change is possible by having the desktop call its active windows' *Valid* method with the command *cmReleasedFocus*.

If the window holds invalid data that would possibly not get validated if the user moved the focus out of the active window, its *Valid* should return *False*, causing *CanMoveFocus* to also return *False*, thus preventing the window from losing the focus.

ExecuteDialog

```
function ExecuteDialog(P: PDialog; Data: Pointer): Word;
```

Calls *ValidView* to ensure that *P* is a valid dialog box, then executes *P* in the desktop. When the user closes the dialog box, *ExecuteDialog* disposes of the dialog box and returns the command that ended the modal state, as returned by *ExecView*. If *ValidView* returns **nil**, meaning the dialog box was not valid, *ExecuteDialog* returns *cmCancel*.

If *Data* is not **nil**, *ExecuteDialog* automatically handles setting and reading the dialog box's controls, using *Data^* as the data record. *ExecuteDialog* initially calls *P^.SetData*, to set the controls. If the user does not cancel the dialog box, *ExecuteDialog* calls *P^.GetData* to read the new values of the dialog box's controls before disposing of the dialog box object.

☞ You should call your application object's *ExecuteDialog* method rather than calling *Desktop^.ExecView* directly. *ExecuteDialog* is a much more convenient way to handle setting and reading of control values and also has built-in validity checks.

See also: *TProgram.ValidView, TGroup.ExecView*

GetEvent

```
procedure GetEvent(var Event: TEvent); virtual;
```

Override: Seldom

The default *TView.GetEvent* simply calls its owner's *GetEvent*, and since a *TProgram* (or *TApplication*) object is the ultimate owner of every view, every *GetEvent* call will end up in *TProgram.GetEvent* (unless some view along the way has overridden *GetEvent*).

TProgram.GetEvent first checks if *PutEvent* has generated a pending event; if so, *GetEvent* returns that event. If there is no pending event, *GetEvent* calls *GetMouseEvent*; if that returns *evNothing*, it then calls *GetKeyEvent*. If both return *evNothing*, indicating that no user input is available, *GetEvent* calls *Idle* to allow "background" tasks to be performed while the application waits for user input. Before returning, *GetEvent* passes any *evKeyDown* and *evMouseDown* events to the *StatusLine* for it to map into associated *evCommand* hot key events.

See also: *TProgram.PutEvent, GetMouseEvent, GetKeyEvent*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the palette given by the palette index in the *AppPalette* global variable. *TProgram* supports three palettes, *apColor*, *apBlackWhite*, and *apMonochrome*. The *AppPalette* variable is initialized by *TProgram.InitScreen*.

See also: *TProgram.InitScreen, AppPalette, apXXXX* constants

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Always

Handles most events by calling the *HandleEvent* method inherited from *TGroup*. Handles *Alt+1* through *Alt+9* key events by generating an *evBroadcast* event with a *Command* value of *cmSelectWindowNum* and an *InfoInt* value of 1..9. *TWindow.HandleEvent* reacts to such broadcasts by selecting the window if it has the given number.

Handles an *evCommand* event with a *Command* value of *cmQuit* by calling *EndModal(cmQuit)*, which in effect terminates the application.

Your application object will nearly always override *HandleEvent* to introduce handling of commands specific to your application.

See also: *TGroup.HandleEvent*

Idle

```
procedure Idle; virtual;
```

Override: Sometimes

GetEvent calls *Idle* whenever the event queue is empty, allowing the application to perform background tasks while waiting for user input.

The default *TProgram.Idle* calls *StatusLine^.Update* to allow the status line to update itself according to the current help context. Then, if the command set has changed since the last call to *TProgram.Idle*, a broadcast event with a *Command* value of *cmCommandSetChanged* is generated to allow views that depend on the command set to enable or disable themselves.

☞ If you override *Idle*, always make sure to call the inherited *Idle*. Also, make sure that any tasks performed by your *Idle* don't suspend the application for any noticeable length of time, since this would block user input and give an unresponsive feel to the application.

InitDesktop

```
procedure InitDesktop; virtual;
```

Override: Seldom

Constructs a desktop object for the application and stores a pointer to it in the *Desktop* global variable. *TProgram.Init* calls *InitDesktop*, so you should never call it directly. You can override *InitDesktop* to construct a user-defined descendant of *TDesktop* instead of the default *TDesktop*.

See also: *TProgram.Init*, *TDesktop*, *TWindow.Init*

InitMenuBar

```
procedure InitMenuBar; virtual;
```

Override: Always

Constructs a menu bar object for the application and stores a pointer to it in the *MenuBar* global variable. *TProgram.Init* calls *InitMenuBar*, so you should never call it directly. Your applications will nearly always override *InitMenuBar* to provide a user-defined menu bar instead of the default empty *TMenuBar*.

See also: *TProgram.Init*, *TMenuBar*, *TWindow.Init*

InitScreen

```
procedure InitScreen; virtual;
```

Override: Sometimes

TProgram.Init and *TProgram.SetScreenMode* call *InitScreen* every time the screen mode is initialized or changed. This is the method that actually performs the updating and adjustment of screenmode-dependent variables for shadow size, markers and application palette.

See also: *TProgram.Init*, *TProgram.SetScreenMode*

InitStatusLine

Override: Always

```
procedure InitStatusLine; virtual;
```

Constructs a status line object for the application and stores a pointer to it in the *StatusLine* global variable. *TProgram.Init* calls *InitStatusLine* so you should never call it directly. Your applications will usually override *InitStatusLine* to construct a user-defined status line instead of the default *TStatusLine.*

See also: *TProgram.Init, TStatusLine*

InsertWindow

```
function InsertWindow(P: PWindow): PWindow;
```

Calls *ValidView* to ensure that *P* is a valid window, and if it is, calls *CanMoveFocus* to see if inserting the window would cause a validation problem in the active window. If *CanMoveFocus* returns *True, InsertWindow* inserts *P* into the desktop and returns *P*. If *CanMoveFocus* returns *False, InsertWindow* disposes of *P* and returns **nil**.

☞ You should call your application object's *InsertWindow* method rather than calling *Desktop^.Insert* directly. Not only does *InsertWindow* automatically check the validity of window objects, it uses *CanMoveFocus* to protect the validation of data in the active window.

See also: *TProgram.CanMoveFocus, TGroup.Insert*

OutOfMemory

Override: Often

```
procedure OutOfMemory; virtual;
```

ValidView calls *OutOfMemory* whenever it detects that *LowMemory* is *True*. *OutOfMemory* should alert the user to the fact that there is not enough memory to complete an operation. For example, using the *MessageBox* routine in the *MsgBox* unit:

```
procedure TMyApp.OutOfMemory;
begin
  MessageBox('Not enough memory to complete operation.',
    nil, mfError + mfOKButton);
end;
```

See also: *TProgram.ValidView, LowMemory* variable

PutEvent

Override: Seldom

```
procedure PutEvent(var Event: TEvent); virtual;
```

The default *TView.PutEvent* simply calls its owner's *PutEvent*, and since a *TProgram* (or *TApplication*) object is the ultimate owner of every view, every *PutEvent* call will end up in *TProgram.PutEvent* (unless some view along the way has overridden *PutEvent*).

TProgram.PutEvent stores a copy of the *Event* record in a buffer, and the next call to *GetEvent* will return that copy.

See also: *TProgram.GetEvent*, *TView.PutEvent*

Run

```
procedure Run; virtual;
```

Override: Seldom

Runs the application by calling the *Execute* method (which *TProgram* inherited from *TGroup*).

See also: *TGroup.Execute*

SetScreenMode

```
procedure SetScreenMode(Mode: Word);
```

Sets the screen mode. *Mode* is one of the constants *smCO80*, *smBW80*, or *smMono*, optionally with *smFont8x8* added to select 43- or 50-line mode on an EGA or VGA. *SetScreenMode* hides the mouse, calls *SetVideoMode* to actually change the screen mode, calls *InitScreen* to initialize any screenmode-dependent variables, assigns *ScreenBuffer* to *TProgram.Buffer*, calls *ChangeBounds* with the new screen rectangle, and finally shows the mouse.

See also: *TProgram.InitScreen*, *SetVideoMode*, *smXXXX* constants

ValidView

```
function TProgram.ValidView(P: PView): PView;
```

Checks the validity of a newly instantiated view, returning *P* if the view is valid, **nil** if not. First, if *P* is **nil**, a value of **nil** is returned. Second, if *LowMemory* is *True* upon the call to *ValidView*, the view given by *P* is disposed, the *OutOfMemory* method is called, and a value of **nil** is returned. Third, if the call *P^.Valid(cmValid)* returns *False*, the view is disposed and a value of **nil** is returned. Otherwise, the view is considered valid, and *P*, the pointer to the view, is returned.

ValidView is often used to validate a new view before inserting it in its owner. Both *InsertWindow* and *ExecuteDialog* call *ValidView*. You can call *ValidView* directly in cases where you don't want to immediately insert or execute a view.

See also: *LowMemory*, *TProgram.OutOfMemory*, *Valid* methods

Palettes

The palette for an application object controls the final color mappings for all views in the application. All other palette mappings eventually result in the selection of an entry in the application's palette, which provides text attributes.

☞ In version 2.0, the standard application palettes have been extended to accommodate blue and cyan dialog boxes in addition to the default gray dialog boxes. The version 1.0 palettes *CColor*, *CBlackWhite* and

CMonochrome are still included in *App* for compatibility with existing programs that have extended the default palettes.

The version 2.0 palettes *CAppColor, CAppBlackWhite,* and *CAppMonochrome* are identical to the version 1.0 palettes, but the entries from 64 to 127 are new.

The first entry is used by *TBackground* for the background color. Entries 2 through 7 are used by both menu views and status lines.

	1	2	3	4	5	6	7
CAppColor	$71	$70	$78	$74	$20	$28	$24
CAppBlackWhite	$70	$70	$78	$7F	$07	$07	$0F
CAppMonochrome	$70	$07	$07	$0F	$70	$70	$70

Background
Normal Text
Disabled Text
Shortcut text
Shortcut selection
Disabled selection
Normal selection

Entries 8 through 15 are used by blue windows.

	8	9	10	11	12	13	14	15
CAppColor	$17	$1F	$1A	$31	$31	$1E	$71	$00
CAppBlackWhite	$07	$0F	$07	$70	$70	$07	$70	$00
CAppMonochrome	$07	$0F	$07	$70	$70	$07	$70	$00

Frame Passive
Frame Active
Frame Icon
ScrollBar Page
Reserved
Scroller Selected Text
Scroller Normal Text
ScrollBar Reserved

Entries 16 through 23 are used by cyan windows.

	16	17	18	19	20	21	22	23
CAppColor	$37	$3F	$3A	$13	$13	$3E	$21	$00
CAppBlackWhite	$07	$0F	$07	$70	$70	$07	$70	$00
CAppMonochrome	$07	$0F	$07	$70	$70	$07	$70	$00

Frame Passive
Frame Active
Frame Icon
ScrollBar Page
Reserved
Scroller Selected Text
Scroller Normal Text
ScrollBar Reserved

Entries 24 through 31 are used by gray windows.

	24	25	26	27	28	29	30	31
CAppColor	$70	$7F	$7A	$13	$13	$70	$7F	$00
CAppBlackWhite	$70	$7F	$7F	$70	$07	$70	$07	$00
CAppMonochrome	$70	$70	$70	$07	$07	$70	$07	$00

Frame Passive
Frame Active
Frame Icon
ScrollBar Page
Reserved
Scroller Selected Text
Scroller Normal Text
ScrollBar Reserved

Entries 32 through 63 are used by gray dialog box objects. See *TDialog* for individual entries.

	32	33	34	35	36	37	38	39	40
CAppColor	$70	$7F	$7A	$13	$13	$70	$70	$7F	$7E
CAppBlackWhite	$70	$7F	$7F	$70	$07	$70	$70	$7F	$7F
CAppMonochrome	$70	$70	$70	$07	$07	$70	$70	$70	$7F

```
Frame Passive
Frame Active
Frame Icon
ScrollBar Page
ScrollBar Controls
```

```
Label Shortcut
Label Highlight
Label Normal
StaticText
```

	41	42	43	44	45	46	47	48	49
CAppColor	$20	$2B	$2F	$78	$2E	$70	$30	$3F	$3E
CAppBlackWhite	$07	$0F	$0F	$78	$0F	$78	$07	$0F	$0F
CAppMonochrome	$07	$07	$0F	$70	$0F	$70	$07	$0F	$0F

```
Button Normal
Button Default
Button Selected
Button Disabled
Button Shortcut
```

```
Cluster Shortcut
Cluster Selected
Cluster Normal
Button Shadow
```

	50	51	52	53	54	55	56
CAppColor	$1F	$2F	$1A	$20	$72	$31	$31
CAppBlackWhite	$0F	$70	$0F	$07	$70	$70	$70
CAppMonochrome	$07	$70	$07	$07	$70	$07	$07

```
InputLine Normal
InputLine Selected
InputLine Arrows
History Arrow
```

```
HistoryWindow ScrollBar controls
HistoryWindow ScrollBar page
History Sides
```

	57	58	59	60	61	62	63
CAppColor	$30	$2F	$3E	$31	$13	$00	$00
CAppBlackWhite	$07	$70	$0F	$07	$07	$00	$00
CAppMonochrome	$07	$70	$0F	$07	$07	$00	$00

```
ListViewer Normal
ListViewer Focused
ListViewer Selected
ListViewer Divider
```

```
Reserved
Cluster disabled
InfoPane
```

Entries 64 through 95 are used by blue dialog box objects. See *TDialog* for individual entries.

	64	65	66	67	68	69	70	71	72
CAppColor	$17	$1F	$1A	$71	$71	$1E	$17	$1F	$1E
CAppBlackWhite	$07	$0F	$0F	$07	$70	$07	$07	$0F	$0F
CAppMonochrome	$70	$70	$70	$07	$07	$70	$70	$70	$0F

Frame Passive
Frame Active
Frame Icon
ScrollBar Page
ScrollBar Controls

Label Shortcut
Label Highlight
Label Normal
StaticText

	73	74	75	76	77	78	79	80	81
CAppColor	$20	$2B	$2F	$78	$2E	$10	$30	$3F	$3E
CAppBlackWhite	$70	$78	$7F	$08	$7F	$08	$70	$7F	$7F
CAppMonochrome	$07	$07	$0F	$70	$0F	$70	$07	$0F	$0F

Button Normal
Button Default
Button Selected
Button Disabled
Button Shortcut

Cluster Shortcut
Cluster Selected
Cluster Normal
Button Shadow

	82	83	84	85	86	87	88
CAppColor	$70	$2F	$7A	$20	$12	$31	$31
CAppBlackWhite	$7F	$0F	$70	$70	$07	$70	$70
CAppMonochrome	$07	$70	$07	$07	$70	$07	$07

InputLine Normal
InputLine Selected
InputLine Arrows
History Arrow

HistoryWindow ScrollBar controls
HistoryWindow ScrollBar page
History Sides

	89	90	91	92	93	94	95
CAppColor	$30	$2F	$3E	$31	$13	$38	$00
CAppBlackWhite	$70	$07	$7F	$70	$07	$78	$00
CAppMonochrome	$07	$70	$0F	$07	$07	$70	$00

ListViewer Normal
ListViewer Focused
ListViewer Selected
ListViewer Divider

Reserved
Cluster disabled
InfoPane

Entries 96 through 127 are used by gray dialog box objects. See *TDialog* for individual entries.

	96	97	98	99	100	101	102	103	104
CAppColor	$37	$3F	$3A	$13	$13	$3E	$30	$3F	$3E
CAppBlackWhite	$70	$7F	$7F	$70	$07	$70	$70	$7F	$7F
CAppMonochrome	$70	$70	$70	$07	$07	$70	$70	$70	$0F

Frame Passive — Label Shortcut
Frame Active — Label Highlight
Frame Icon — Label Normal
ScrollBar Page — StaticText
ScrollBar Controls

	105	106	107	108	109	110	111	112	113
CAppColor	$20	$2B	$2F	$78	$2E	$30	$70	$7F	$7E
CAppBlackWhite	$07	$0F	$0F	$78	$0F	$78	$07	$0F	$0F
CAppMonochrome	$07	$07	$0F	$70	$0F	$70	$07	$0F	$0F

Button Normal — Cluster Shortcut
Button Default — Cluster Selected
Button Selected — Cluster Normal
Button Disabled — Button Shadow
Button Shortcut

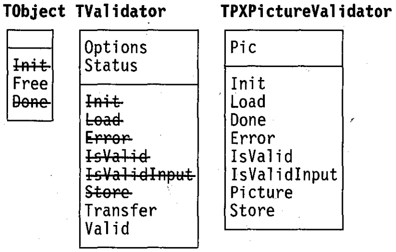
	114	115	116	117	118	119	120
CAppColor	$1F	$2F	$1A	$20	$32	$31	$71
CAppBlackWhite	$0F	$70	$0F	$07	$70	$70	$70
CAppMonochrome	$07	$70	$07	$07	$70	$07	$07

InputLine Normal — HistoryWindow ScrollBar controls
InputLine Selected — HistoryWindow ScrollBar page
InputLine Arrows — History Sides
History Arrow

	121	122	123	124	125	126	127
CAppColor	$70	$2F	$7E	$71	$13	$38	$00
CAppBlackWhite	$07	$70	$0F	$07	$07	$78	$00
CAppMonochrome	$07	$70	$0F	$07	$07	$70	$00

ListViewer Normal — Reserved
ListViewer Focused — Cluster disabled
ListViewer Selected — InfoPane
ListViewer Divider

TPXPictureValidator Validate



Picture validator objects compare user input with a picture of a data format to determine the validity of entered data. The pictures are compatible with the pictures Borland's Paradox relational database uses to control data entry. For a complete description of picture specifiers, see *TPXPictureValidator*'s *Picture* method.

Field

Pic

```
Pic: PString;
```

Points to a string containing the picture that specifies the format for data in the associated input line. The *Init* constructor sets *Pic* to a string passed as one of its parameters.

Methods

Init

```
constructor Init(const APic: string; AutoFill: Boolean);
```

Constructs a picture validator object by first calling the *Init* constructor inherited from *TValidator*, then allocating a copy of *APic* on the heap and setting *Pic* to point to it, then setting the *voFill* bit in *Options* if *AutoFill* is *True*.

See also: *TValidator.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a picture validator object from the stream *S* by first calling the *Load* constructor inherited from *TValidator*, then reading the value for the *Pic* field introduced by *TPXPictureValidator*.

See also: *TValidator.Load*

Done

```
destructor Done; virtual;
```

Disposes of the string pointed to by *Pic,* then disposes of the picture validator object by calling the *Done* destructor inherited from *TValidator.*

Error

```
procedure Error; virtual;
```

Displays a message box indicating an error in the picture format, displaying the string pointed to by *Pic.*

IsValidInput

```
function IsValidInput(var S: string; SuppressFill: Boolean): Boolean;
  virtual;
```

Checks the string passed in *S* against the format picture specified in *Pic* and returns *True* if *Pic* is **nil** or *Picture* does not return *prError* for *S;* otherwise, returns *False.* The *SuppressFill* parameter overrides the value in *voFill* for the duration of the call to *IsValidInput.*

S is a **var** parameter, so *IsValidInput* can modify its value. For example, if *SuppressFill* is *False* and *voFill* is set, the call to *Picture* returns a filled string based on *S,* so the image in the input line automatically reflects the format specified in *Pic.*

See also: *TPXPictureValidator.Picture*

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Compares the string passed in *S* with the format picture specified in *Pic* and returns *True* if *Pic* is **nil** or if *Picture* returns *prComplete* for *S,* indicating that *S* needs no further input to meet the specified format.

See also: *TPXPictureValidator.Picture*

Picture

```
function Picture(var Input: string): TPicResult; virtual;
```

Formats the string passed in *Input* according to the format specified by the picture string pointed to by *Pic.* Returns *prError* if there is an error in the picture string or if *Input* contains data that cannot fit the specified picture. Returns *prComplete* if *Input* can fully satisfy the specifed picture. Returns *prIncomplete* if *Input* contains data that fits the specified picture but not completely.

Table 19.41 shows the characters used in creating format pictures.

Table 19.41
Picture format characters

Type of character	Character	Description
Special	#	Accept only a digit
	?	Accept only a letter (case-insensitive)
	&	Accept only a letter, force to uppercase
	@	Accept any character
	!	Accept any character, force to uppercase
Match	;	Take next character literally

Table 19.41: Picture format characters (continued)

	*	Repetition count
	[]	Option
	{}	Grouping operators
	,	Set of alternatives
All others		Taken literally

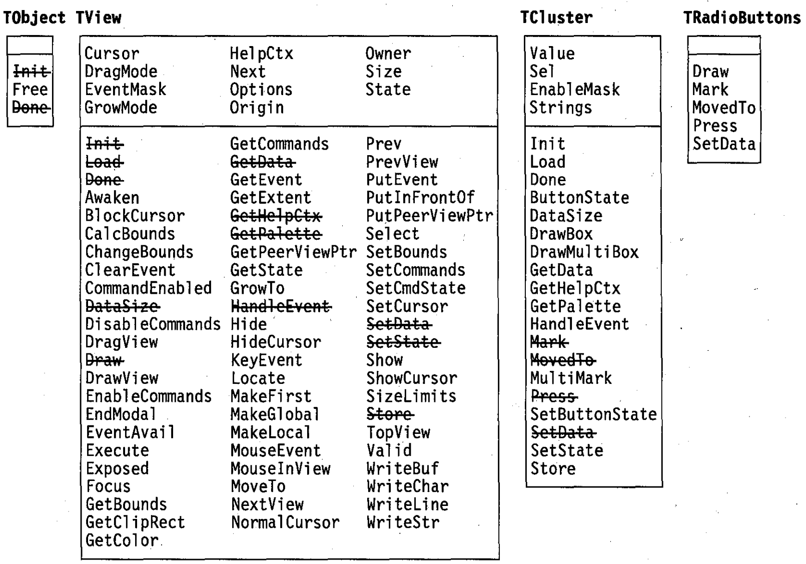
See also: *TPicResult* type

Store

```
procedure Store(var S: TStream);
```

Stores the picture validator object to the stream *S* by first calling the *Store* method inherited from *TValidator,* then writing the string pointed to by *Pic*.

TRadioButtons Dialogs



TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
Awaken
BlockCursor
CalcBounds
ChangeBounds
ClearEvent
CommandEnabled
~~DataSize~~
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
~~GetData~~
GetEvent
GetExtent
~~GetHelpCtx~~
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
~~SetData~~
~~SetState~~
Show
ShowCursor
SizeLimits
~~Store~~
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

TCluster

Value
Sel
EnableMask
Strings

Init
Load
Done
ButtonState
DataSize
DrawBox
DrawMultiBox
GetData
GetHelpCtx
GetPalette
HandleEvent
~~Mark~~
~~MovedTo~~
MultiMark
~~Press~~
SetButtonState
~~SetData~~
SetState
Store

TRadioButtons

Draw
Mark
MovedTo
Press
SetData

TRadioButtons objects are clusters of up to 65,536 controls with the special property that only one control button in the cluster can be selected. Selecting an unselected button will automatically deselect (restore) the previously selected button. Most of the functionality is derived from *TCluster* including *Init, Load,* and *Done*. Radio buttons usually have an associated *TLabel* object.

TRadioButtons interprets the inherited *TCluster.Value* field as the number of the "pressed" button, with the first button in the cluster being number 0.

Methods

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws buttons as " ( ) " surrounded by a box.

Mark

```
function Mark(Item: Integer): Boolean; virtual;
```

Overide: Never

Returns *True* if *Item* = *Value*, that is, if the *Item*'th button represents the current *Value* field (the "pressed" button).

See also: *TCluster.Value, TCluster.Mark*

MovedTo

```
procedure MovedTo(Item: Integer); virtual;
```

Override: Never

Assigns *Item* to *Value.*

See also: *TCluster.MovedTo, TRadioButtons.Mark*

Press

```
procedure Press(Item: Integer); virtual;
```

Override: Never

Assigns *Item* to *Value*. Called when the *Item*'th button is pressed.

SetData

```
procedure SetData(var Rec); virtual;
```

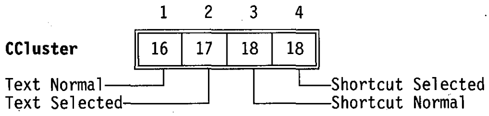
Override: Seldom

Calls the *SetData* method inherited from *TCluster* to set the *Value* field, then sets *Sel* field equal to *Value,* since the selected item *is* the "pressed" button at startup.

See also: *TCluster.SetData*

Palette

TRadioButtons objects use *CCluster*, the default palette for all cluster objects, to map onto the 16th through 18th entries in the standard dialog palette.



TRangeValidator Validate

TObject	TValidator	TFilterValidator	TRangeValidator
	Options Status	ValidChars	Min Max
~~Init~~ Free Done	~~Init~~ ~~Load~~ ~~Error~~ ~~IsValid~~ ~~IsValidInput~~ ~~Store~~ ~~Transfer~~ Valid	~~Init~~ ~~Load~~ ~~Error~~ ~~IsValid~~ IsValidInput ~~Store~~	Init Load Error IsValid Store Transfer

A range validator object determines whether the data typed by a user falls within a designated range of integers.

Fields

Max

```
Max: Longint;
```

Max is the highest valid long integer value for the input line.

Min

```
Min: Longint;
```

Min is the lowest valid long integer value for the input line.

Methods

Init

```
constructor Init(AMin, AMax: Longint);
```

Constructs a range validator object by first calling the *Init* constructor inherited from *TFilterValidator,* passing a set of characters containing the digits '0'..'9' and the characters '+' and '-'. Sets *Min* to *AMin* and *Max* to *AMax,* establishing the range of acceptable long integer values.

See also: *TFilterValidator.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a range validator object from the stream *S* by first calling the *Load* constructor inherited from *TFilterValidator,* then reading the *Min* and *Max* fields introduced by *TRangeValidator.*

See also: *TFilterValidator.Load*

Error

```
procedure Error; virtual;
```

Displays a message box indicating that the entered value did not fall in the specified range.

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Converts the string *S* into an integer number and returns *True* if the result meets all three of these conditions:

- It is a valid integer number.
- Its value is greater than or equal to *Min*.
- It's value is less than or equal to *Max*.

If any of those tests fails, *IsValid* returns *False*.

Store

```
procedure Store(var S: TStream);
```

Stores the range validator object on the stream *S* by first calling the *Store* method inherited from *TFilterValidator*, then writing the *Min* and *Max* fields introduced by *TRangeValidator*.

See also: *TFilterValidator.Store*

Transfer

```
function Transfer(var S: String; Buffer: Pointer; Flag: TVTransfer): Word;
virtual;
```

Incorporates the three functions *DataSize, GetData,* and *SetData* that a range validator can handle for its associated input line. Instead of setting and reading the value of the numeric input line by passing a string representation of the number, *Transfer* can use a *Longint* as its data record, which keeps your application from having to handle the conversion.

S is the input line's string value, and *Buffer* is the data record passed to the input line. Depending on the value of *Flag, Transfer* either sets *S* from the number in *Buffer^* or sets the number at *Buffer* to the value of the string *S*. If *Flag* is *vtSetData, Transfer* sets *S* from *Buffer*. If *Flag* is *vtGetData, Transfer* sets *Buffer* from *S*. If *Flag* is *vtDataSize, Transfer* neither sets nor reads data.

Transfer always returns the size of the data transferred, in this case the size of a *Longint*.

See also: *TVTransfer* type

TRect Objects

TRect

```
A
B

Assign
Contains
Copy
Empty
Equals
Grow
Intersect
Move
Union
```

Fields

A

`A: TPoint`

A is the point defining the top left corner of a rectangle on the screen.

B

`B: TPoint`

B is the point defining the bottom right corner of a rectangle on the screen.

Methods

Assign

```
procedure Assign(XA, YA, XB, YB: Integer);
```

Assigns the parameter values to the rectangle's point fields. *XA* becomes *A.X*, *XB* becomes *B.X*, and so on.

Contains

```
function Contains(P: TPoint): Boolean;
```

Returns *True* if the rectangle contains the point *P*.

Copy

```
procedure Copy(R: TRect);
```

Copy sets all fields equal to those in rectangle *R*.

Empty

```
function Empty: Boolean;
```

Returns *True* if the rectangle is empty, meaning the rectangle contains no character spaces. Essentially, the *A* and *B* fields are equal.

Equals

```
function Equals(R: TRect): Boolean;
```

Returns *True* if *R* is the same as the rectangle.

Grow

```
procedure Grow(ADX, ADY: Integer);
```

Changes the size of the rectangle by subtracting *ADX* from *A.X*, adding *ADX* to *B.X*, subtracting *ADY* from *A.Y*, and adding *ADY* to *B.Y*.

Intersect

```
procedure Intersect(R: TRect);
```

Changes the location and size of the rectangle to the region defined by the intersection of the current location and that of *R*.

Move

```
procedure Move(ADX, ADY: Integer);
```

Moves the rectangle by adding *ADX* to *A.X* and *B.X* and adding *ADY* to *A.Y* and *B.Y*.

Union

```
procedure Union(R: TRect);
```

Changes the rectangle to be the union of itself and the rectangle *R*; that is, to the smallest rectangle containing both the object and *R*.

TReplaceDialogRec type — Editors

Declaration

```
TReplaceDialogRec = record
  Find: String[80];
  Replace: String[80];
  Options: Word;
end;
```

Function Search and replace dialog boxes invoked by *EditorDialog* when passed *edReplace* take a pointer to a *TReplaceDialogRec* as their second parameter. *Find* and *Replace* hold the default string to search for and replace with, respectively. *Options* holds some combination of the *efXXXX* editor flag constants, specifying how the search and replace operation should work.

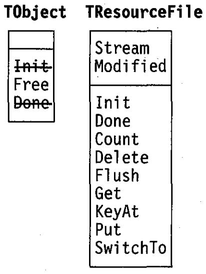
See also: *TEditorDialog* type

TResourceCollection — Objects

TResourceCollection is a descendant of *TStringCollection* used with *TResourceFile* to implement collections of resources. A resource file is a stream that is indexed by key strings. Each resource item, therefore, has an integer *Pos* field and a string *Key* field. The overriding methods of *TResourceCollection* are mainly concerned with handling the extra string element in its items.

TResourceCollection is used internally by *TResourceFile* objects to maintain a resource file's index.

TResourceFile Objects



TResourceFile implements a stream that can be indexed by key strings. When objects are stored in a resource file, using *TResourceFile.Put*, a key string, which identifies the object, is also supplied. The objects can later be retrieved by specifying the key string in a call to *TResourceFile.Get*.

To provide fast and efficient access to the objects stored in a resource file, *TResourceFile* stores the key strings in a sorted string collection (using the *TResourceCollection* type) along with the position and size of the resource data in the resource file.

As is the case with streams, the types of objects written to and read from resource files must have been registered using *RegisterType*.

Fields

Modified

`Modified: Boolean;` **Read/write**

Set *True* if the resource file has been modified.

See also: *TResourceFile.Flush*

Stream

`Stream: PStream;` **Read only**

Pointer to the stream associated with this resource file.

Methods

Init

constructor `Init(AStream: PStream);`

Initializes a resource file using the stream given by *AStream* and sets the *Modified* field to *False*. The stream must have already been initialized. For example:

```
ResFile.Init(New(TBufStream, Init('MYAPP.RES', stOpenRead, 1024)));
```

During initialization, *Init* looks for a resource file header at the current position of the stream. The format of a resource file header is

```
type
  TResFileHeader = record
    Signature: array[1..4] of Char;
    ResFileSize: Longint;
    IndexOffset: Longint;
  end;
```

where *Signature* contains 'FBPR', *ResFileSize* contains the size of the entire resource file excluding the *Signature* and *ResFileSize* fields (i.e. the size of the resource file minus 8 bytes), and *IndexOffset* contains the offset of the index collection from the beginning of the header.

If *Init* does not find a resource file header at the current position of *AStream,* it assumes that a new resource file is being created, and thus constructs an empty index.

If *Init* sees an .EXE file signature at the current position of the stream, it seeks the stream to the end of the .EXE file image, and then looks for a resource file header there. Likewise, *Init* will skip over an overlay file that was appended to the .EXE file (as will *OvrInit* skip over a resource file). This means that you can append both your overlay file and your resource file (in any order) to the end of your application's .EXE file. (This is, in fact, what the IDE's executable file, TURBO.EXE, does.)

See also: *TResourceFile.Done*

Done

```
destructor Done; virtual;
```

Override: Never

Flushes the resource file, using *TResourceFile.Flush*, and then disposes of the index and the stream given by the *Stream* field.

See also: *TResourceFile.Init*, *TResourceFile.Flush*

Count

```
function Count: Integer;
```

Returns the number of resources stored in the resource file.

See also: *TResourceFile.KeyOf*

Delete

```
procedure Delete(Key: String);
```

Deletes the resource indexed by *Key* from the resource file. The space formerly occupied by the deleted resource is not reclaimed. You can reclaim this memory by using *SwitchTo* to create a packed copy of the file on a new stream.

See also: *TResourceFile.SwitchTo*

Flush

```
procedure Flush;
```

If the resource file has been modified (checked using the *Modified* field), *Flush* stores the updated index at the end of the stream and updates the resource header at the beginning of the stream. It then resets *Modified* to *False*.

See also: *TResourceFile.Done, TResourceFile.Modified*

Get

```
function Get(Key: String): PObject;
```

Searches for *Key* in the resource file index. Returns **nil** if the key is not found. Otherwise, seeks the stream to the position given by the index, and calls *Stream^.Get* to create and load the object identified by *Key*. An example:

```
Desktop^.Insert(ValidView(ResFile.Get('EditorWindow')));
```

See also: *TResourceFile.KeyAt, TResourceFile.Put*

KeyAt

```
function KeyAt(I: Integer): String;
```

Returns the string key of the *I*th resource in the calling resource file. The index of the first resource is zero and the index of the last resource is *TResourceFile.Count* minus one. Using *Count* and *KeyAt* you can iterate over all resources in a resource file.

See also: *TResourceFile.Count*

Put

```
procedure Put(Item: PObject; Key: String);
```

Adds the object given by *P* to the resource file with the key string given by *Key*. If the index already contains the *Key*, then the new object replaces the old object. The object is appended to the existing objects in the resource file using *Stream^.Put*.

See also: *TResourceFile.Get*

SwitchTo

```
function SwitchTo(AStream: PStream; Pack: Boolean): PStream;
```

Switches the resource file from the stream it is on to the stream passed in *AStream*, and returns a pointer to the original stream as a result.

If the *Pack* parameter is *True*, the stream will eliminate empty and unused space from the resource file before writing it to the new stream. This is the *only* way to compress resource files. Copying with the *Pack* parameter *False* provides faster copying, but without the compression.

TScrollBar Views

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
Done
Awaken
BlockCursor
CalcBounds
ChangeBounds
ClearEvent
CommandEnabled
DataSize
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
GetData
GetEvent
GetExtent
GetHelpCtx
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
~~HandleEvent~~
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
SetData
SetState
Show
ShowCursor
SizeLimits
~~Store~~
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

TScrollBar

ArStep
Max
Min
PgStep
Value

Init
Load
Draw
GetPalette
HandleEvent
ScrollDraw
ScrollStep
SetParams
SetRange
SetStep
SetValue
Store

Fields

ArStep

`ArStep: Integer;` **Read only**

ArStep is the amount added to or subtracted from the scroll bar's *Value* field when an arrow area is clicked (*sbLeftArrow, sbRightArrow, sbUpArrow,* or *sbDownArrow*) or the equivalent keystroke made. *Init* sets *ArStep* to 1 by default.

See also: *TScrollBar.SetStep, TScrollBar.SetParam, TScrollBar.ScrollStep*

Max

`Max: Integer;` **Read only**

Max represents the maximum value for the *Value* field. *Init* sets *Max* to 0 by default.

See also: *TScrollBar.SetRange, TScrollBar.SetParams*

Min

`Min: Integer;` **Read only**

Min represents the minimum value for the *Value* field. *Init* sets *Min* to 0 by default.

See also: *TScrollBar.SetRange, TScrollBar.SetParams*

PgStep

`PgStep: Integer;` **Read only**

PgStep is the amount added to or subtracted from the scroll bar's *Value* field when a mouse click event occurs in any of the page areas (*sbPageLeft, sbPageRight, sbPageUp*, or *sbPageDown*) or an equivalent keystroke is detected (*Ctrl*←, *Ctrl*→, *PgUp*, or *PgDn*). *Init* sets *PgStep* to 1 by default. You can change *PgStep* using *SetStep, SetParams* or *SetLimit*.

See also: *TScrollBar.SetStep, TScrollBar.SetParams, TScroller.SetLimit, TScrollBar.ScrollStep*

Value

`Value: Integer;` **Read only**

The *Value* field represents the current position of the scroll bar indicator. This specially colored marker moves along the scroll bar strip to indicate the relative position (horizontally or vertically depending on the scroll bar orientation) of the scrollable text being viewed relative to the total text available for scrolling. Many events can directly or indirectly change *Value*, such as mouse-clicking the designated scroll bar parts, resizing the window, or changing the text in the scroller. Similarly, changes in *Value* may need to trigger other events. *TScrollBar.Init* sets *Value* to 0 by default.

See also: *TScrollBar.SetValue, TScrollBar.SetParams, TScrollBar.ScrollDraw, TScrollBar.Init*

Methods

Init

```
constructor Init(var Bounds: TRect);
```

Creates and initializes a scroll bar with the given *Bounds* by calling the *Init* constructor inherited from *TView*. Sets *Value, Max*, and *Min* to 0, *PgStep* and *ArStep* to 1. Sets the shapes of the scroll bar parts to the defaults in *TScrollChars*.

If *Bounds* produces *Size.X = 1*, you get a vertical scroll bar; otherwise, you get a horizontal scroll bar. Vertical scroll bars have the *GrowMode* field set to *gfGrowLoX* + *gfGrowHiX* + *gfGrowHiY*; horizontal scroll bars have the *GrowMode* field set to *gfGrowLoY* + *gfGrowHiX* + *gfGrowHiY*.

Load

```
constructor Load(var S: TStream);
```

Constructs then loads a scroll bar object from the stream *S* by calling the *Load* constructor inherited from *TView* and then reading the five integer fields with *S.Read*.

See also: *TScrollBar.Store*

Draw `procedure Draw; virtual;`

Overide: Never

Draws the scroll bar depending on the current *Bounds*, *Value* and palette.

See also: *TScrollBar.ScrollDraw*, *TScrollBar.Value*

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes

Returns a pointer to *CScrollBar*, the default scroll bar palette.

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Never

Handles scroll bar events by calling the *HandleEvent* method inherited from *TView*, then analyzing *Event.What*. Mouse events are broadcast to the scroll bar's owner (see *Message* function) which must handle the implications of the scroll bar changes (for example, by scrolling text). *HandleEvent* also determines which scroll bar part has received a mouse click (or equivalent keystroke). The *Value* field is adjusted according to the current *ArStep* or *PgStep* values and the scroll bar indicator is redrawn.

See also: *TView.HandleEvent*

ScrollDraw `procedure ScrollDraw; virtual;`

Override: Seldom

ScrollDraw is called whenever the *Value* field changes. By default, *ScrollDraw* sends a *cmScrollBarChanged* broadcast to the scroll bar's owner:

```
Message(Owner, evBroadcast, cmScrollBarChanged, @Self);
```

See also: *TScrollBar.Value*, *Message* function

ScrollStep `function ScrollStep(Part: Integer): Integer; virtual;`

Override: Never

By default, *ScrollStep* returns a positive or negative step value depending on the scroll bar part given by *Part*, and the current values of *ArStep* and *PgStep*. *Part* should be one of the *sbXXXX* scroll bar part constants described in this chapter.

See also: *TScrollBar.SetStep*, *TScrollBar.SetParams*, *sbXXXX* constants

SetParams `procedure SetParams(AValue, AMin, AMax, APgStep, AArStep: Integer);`

SetParams sets the *Value*, *Min*, *Max*, *PgStep*, and *ArStep* fields to the values passed in *AValue*, *AMin*, *AMax*, *APgStep*, and *AArStep*. Some adjustments are made if your arguments conflict. For example, *Min* cannot be set higher than *Max*, so if *AMax* < *AMin*, *Max* is set to *AMin*. *Value* must lie in the closed range *[Min,Max]*, so if *AValue* < *AMin*, *Value* is set to *AMin*; and if *AValue* > *AMax*, *Value* is set to *AMax*. The scroll bar is redrawn by calling *DrawView*. If *Value* is changed, *ScrollDraw* is also called.

See also: *TView.DrawView, TScrollBar.ScrollDraw, TScrollBar.SetRange, TScrollBar.SetValue*

SetRange

```
procedure SetRange(AMin, AMax: Integer);
```

SetRange sets the legal range for the *Value* field by setting *Min* and *Max* to *AMin* and *AMax*. *SetRange* calls *SetParams*, so *DrawView* and *ScrollDraw* will be called if the changes require the scroll bar to be redrawn.

See also: *TScrollBar.SetParams*

SetStep

```
procedure SetStep(APgStep, AArStep: Integer);
```

SetStep sets *PgStep* and *ArStep* to *APgStep* and *AArStep*, respectively. This method calls *SetParams* with the other parameters set to their current values.

See also: *TScrollBar.SetParams, TScrollBar.ScrollStep*

SetValue

```
procedure SetValue(AValue: Integer);
```

Sets *Value* to *AValue* by calling *SetParams* with the other parameters set to their current values. *DrawView* and *ScrollDraw* will be called if this call changes *Value*.

See also: *TScrollBar.SetParams, TView.DrawView, TScrollBar.ScrollDraw, TScroller.ScrollTo*

Store

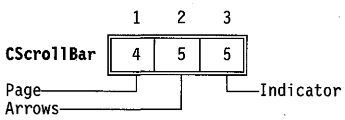
```
procedure Store(var S: TStream);
```

Writes the scroll bar object to the stream *S* by first calling the *Store* method inherited from *TView*, and then writing the five integer fields to the stream using *S.Write*.

See also: *TScrollBar.Load*

Palette

Scroll bar objects use the default palette, *CScrollBar*, to map onto the 4th and 5th entries in the standard application palette.



TScrollChars type   Views

Declaration `TScrollChars = array[0..4] of Char;`

Function An array representing the characters used to draw a *TScrollBar*.

See also *TScrollBar*

TScroller   Views

TObject

~~Init~~ Free ~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
~~ChangeBounds~~	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	~~SetState~~
Draw	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TScroller

HScrollBar VScrollBar Delta Limit
Init Load ChangeBounds GetPalette HandleEvent ScrollDraw ScrollTo SetLimit SetState Store

TScroller provides a scrolling virtual window onto a larger view. That is, a scrolling view lets the user scroll a large view within a clipped boundary. The scroller provides an offset from which the *Draw* method fills the visible region. All methods needed to provide both scroll bar and keyboard scrolling are built into *TScroller*.

The basic scrolling view provides a useful starting point for scrolling views such as text views.

Fields

Delta

`Delta: TPoint;` **Read only**

Delta holds the *X* (horizontal) and *Y* (vertical) components of the scroller's position relative to the virtual view being scrolled. Automatic scrolling is achieved by changing either or both of these components in response, for example, to scroll bar events that change the *Value* field(s). Conversely, manual scrolling changes *Delta*, triggers changes in the scroll bar *Value* fields, and leads to updating of the scroll bar indicators.

See also: *TScroller.ScrollDraw, TScroller.ScrollTo*

HScrollBar

`HScrollBar: PScrollBar;` **Read only**

HScrollBar points to the horizontal scroll bar associated with the scroller. If there is no such scroll bar, *HScrollBar* is **nil**.

Limit

`Limit: TPoint;` **Read only**

Limit.X and *Limit.Y* are the maximum allowed values for *Delta.X* and *Delta.Y*

See also: *TScroller.Delta*

VScrollBar

`VScrollBar: PScrollBar;` **Read only**

VScrollBar points to the vertical scroll bar associated with the scroller. If there is no such scroll bar, *VScrollBar* is **nil**.

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar);
```

Constructs and initializes a scroller object with the given size and scroll bars. Calls the *Init* constructor inherited from *TView* to set the view's size. Sets *Options* to *ofSelectable* and *EventMask* to *evBroadcast*. *AHScrollBar* should be **nil** if you do not want a horizontal scroll bar; similarly, *AVScrollBar* should be **nil** if you do not want a vertical scroll bar.

See also: *TView.Init, TView.Options, TView.EventMask*

Load

```
constructor Load(var S: TStream);
```

Loads the scrolling view from the stream *S* by first calling the *Load* constructor inherited from *TView*, then restoring pointers to the scroll bars using *GetPeerViewPtr*, and finally reading the *Delta* and *Limit* fields using *S.Read*.

See also: *TScroller.Store*

ChangeBounds

Override: Never

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Changes the scroller's size by calling *SetBounds*. If necessary, the scroller and scroll bars are then redrawn by calling *DrawView* and *SetLimit*.

See also: *TView.SetBounds, TView.DrawView, TScroller.SetLimit*

GetPalette

Override: Sometimes

```
function GetPalette: PPalette; virtual;
```

Returns a pointer to *CScroller*, the default scroller palette.

HandleEvent

Override: Seldom

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles most events by calling the *HandleEvent* method inherited from *TView*. Broadcast events with the command *cmScrollBarChanged*, if they come from either *HScrollBar* or *VScrollBar*, result in a call to *ScrollDraw*.

See also: *TView.HandleEvent, TScroller.ScrollDraw*

ScrollDraw

Override: Never

```
procedure ScrollDraw; virtual;
```

Checks to see if *Delta* matches the current positions of the scroll bars. If not, sets *Delta* to the correct value and calls *DrawView* to redraw the scroller.

See also: *TView.DrawView, TScroller.Delta, TScroller.HScrollBar, TScroller.VScrollBar*

ScrollTo

```
procedure ScrollTo(X, Y: Integer);
```

Sets the scroll bars to (X,Y) by calling *HScrollBar^.SetValue(X)* and *VScrollBar^.SetValue(Y)*, and redraws the view by calling *DrawView*.

See also: *TView.DrawView, TScroller.SetValue*

SetLimit

```
procedure SetLimit(X, Y: Integer);
```

Sets *Limit.X* to *X* and *Limit.Y* to *Y*, then calls *HScrollBar^.SetParams* and *VScrollBar^.SetParams* (if these scroll bars exist) to adjust their *Max* field(s). These calls might trigger scroll bar redraws. Finally, calls *DrawView* to redraw the scroller if necessary.

See also: *TScroller.Limit, TScroller.HScrollBar, TScroller.VScrollBar, TScrollBar.SetParams*

SetState

Override: Seldom

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

This method is called whenever the scroller's state changes. Calls the *SetState* method inherited from *TView* to set or clear the state flags in *AState*. If the new state is *sfSelected* and *sfActive*, *SetState* displays the scroll bars, otherwise they are hidden.

T

See also: *TView.SetState*

Store

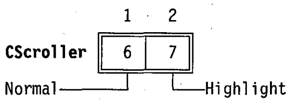
```
procedure Store(var S: TStream);
```

Writes the scroller to the stream *S* by first calling the *Store* method inherited from *TView*, then storing references to the scroll bars using *PutPeerViewPtr*, and finally writing the values of *Delta* and *Limit* using *S.Write*.

See also: *TScroller.Load, TStream.Write*

Palette

Scroller objects use the default palette, *CScroller*, to map onto the 6th and 7th entries in the standard window palette.



TSearchRec type StdDlg

Declaration

```
TSearchRec = record
  Attr: Byte;
  Time: Longint;
  Size: Longint;
  Name: string[12];
end;
```

Function *TSearchRec* records are used in file collection objects to hold information on the files collected. *TSearchRec* is actually a subset of the *SearchRec* type defined in the *Dos* unit, with the 21 bytes of unused information stripped. *Attr* is a bitmapped byte holding file attributes as defined by the *Dos* unit. *Time* is a DOS date-and-time stamp that can be decoded with the *UnpackTime* procedure in the *Dos* unit. *Size* is the size of the file in bytes. *Name* is a string containing the file name.

See also: *Dos* unit in the *Language Guide*

TSItem type Dialogs

Declaration

```
TSItem = record
  Value: PString;
  Next: PSItem;
end;
```

Function The *TSItem* record type provides a singly-linked list of *PStrings*. Such lists can be useful in many Turbo Vision applications where the full flexibility of string collections is not required (see *TCluster.Init*, for example). A utility function *NewSItem* is provided for adding records to a *TSItem* list.

TSortedCollection Objects

TObject	TCollection		TSortedCollection
	Count	Items	Duplicates
	Delta	Limit	
~~Init~~	Init	ForEach	Load
~~Free~~	~~Load~~	Free	Compare
~~Done~~	Done	FreeAll	IndexOf
	At	FreeItem	Insert
	AtDelete	GetItem	KeyOf
	AtFree	~~IndexOf~~	Search
	AtInsert	~~Insert~~	Store
	AtPut	LastThat	
	Delete	Pack	
	DeleteAll	PutItem	
	Error	SetLimit	
	FirstThat	~~Store~~	

TSortedCollection is a specialized derivative of *TCollection* implementing collections sorted by key. Sort order is determined by the virtual method *Compare,* which you override to provide your own definition of element ordering. As you add new items, they are automatically inserted in the order given by *Compare.* You can locate items with the method *Search.* If *Compare* needs additional information, override the virtual method *KeyOf,* which returns a pointer for *Compare.*

TSortedCollection implements sorted collections both with or without duplicate keys. The *Duplicates* field controls whether duplicates are allowed. It defaults to *False,* indicating that duplicate keys are not allowed, but after creating a sorted collection, you can set *Duplicates* to *True* to allow elements with duplicate keys in the collection.

Field

Duplicates

```
Duplicates: Boolean;
```

Determines whether the collection accepts items with duplicate keys. By default, *Duplicates* is *False,* and calling *Insert* for an item whose key matches that of an item already in the collection causes the collection to not insert the new item; the collection keeps only the first item inserted with a given key.

If you set *Duplicates* to *True*, the collection inserts duplicate-key items immediately before the first existing item with the same key.

See also: *TSortedCollection.Insert, TSortedCollection.Search*

Methods

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a sorted collection from the stream *S* by first calling the *Load* constructor inherited from *TCollection,* then reading the *Duplicates* field introduced by *TSortedCollection*.

See also: *TCollection.Load*

Compare

```
function Compare(Key1, Key2: Pointer): Integer; virtual;
```

Override: Always

Compare is an abstract method that must be overridden in all descendant types. *Compare* should compare the two key values, and return a result as follows:

–1	if *Key1* < *Key2*
0	if *Key1* = *Key2*
1	if *Key1* > *Key2*

Key1 and *Key2* are pointer values, as extracted from their corresponding collection items by the *TSortedCollection.KeyOf* method. The *TSortedCollection.Search* method implements a binary search through the collection's items using *Compare* to compare the items.

☞ Make sure *Compare* returns all possible values –1, 0, and 1. Even collections that will never hold duplicate items need to return 0 if *Compare* receives matching keys. If *Compare* never returns 0, *Search* will not function properly.

See also: *TSortedCollection.KeyOf, TSortedCollection.Compare*

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Override: Never

Uses *TSortedCollection.Search* to find the index of the given *Item*. If the item is not in the collection, *IndexOf* returns –1. The actual implementation of *TSortedCollection.IndexOf* is:

```
if Search(KeyOf(Item), I) then IndexOf := I else IndexOf := -1;
```

See also: *TSortedCollection.Search*

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Override: Never

Calls *TSortedCollection.Search* to determine if the item already exists, and if not, where to insert it. If no item with the same key as *Item* is already in the collection, inserts *Item* at the correct index position. If an item with the same key does exist and *Duplicates* is *False*, the collection ignores the duplicate item. If *Duplicates* is *True*, the collection inserts *Item* before the first existing item with the same key.

The actual implementation of *TSortedCollection.Insert* is:

```
if not Search(KeyOf(Item), I) or Duplicates then AtInsert(I, Item);
```

See also: *TSortedCollection.Search*

KeyOf

```
function KeyOf(Item: Pointer): Pointer; virtual;
```

Override: Sometimes

Given an *Item* from the collection, *KeyOf* should return the corresponding key of the item. The default *KeyOf* simply returns *Item*. *KeyOf* is overridden in cases where the key of the item is not the item itself.

See also: *TSortedCollection.IndexOf*

Search

```
function Search(Key: Pointer; var Index: Integer): Boolean; virtual;
```

Override: Seldom

Returns *True* if the item identified by *Key* is found in the sorted collection. If the item is found, *Index* is set to the found index; otherwise, *Index* is set to the index where the item would be placed if inserted. *Search* relies on *Compare* to locate the specified item.

See also: *TSortedCollection.Compare*, *TSortedCollection.Insert*

Store

```
procedure Store(var S: TStream);
```

Writes the sorted collection and all its items to the stream *S* by first calling the *Store* method inherited from *TCollection*, then writing the *Duplicates* field introduced by *TSortedCollection*.

See also: *TCollection.Store*

TSortedListBox object StdDlg

TObject

~~Init~~
Free
~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	~~GetData~~	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
~~ChangeBounds~~	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
~~DataSize~~	~~HandleEvent~~	SetCursor
DisableCommands	Hide	~~SetData~~
DragView	HideCursor	~~SetState~~
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TListViewer

HScrollBar
VScrollBar
NumCols
TopItem
Focused
Range
~~Init~~
~~Load~~
ChangeBounds
Draw
FocusItem
GetPalette
~~GetText~~
~~HandleEvent~~
IsSelected
SelectItem
SetRange
SetState
~~Store~~

TListBox

List
~~Init~~
Load
DataSize
GetData
GetText
~~NewList~~
SetData
Store

TSortedListBox

SearchPos
ShiftState
Init
GetKey
HandleEvent
NewList

TSortedListBox is a *TListBox* that assumes it has a *TSortedCollection* instead of just a *TCollection*. It will perform an incremental search on the contents. It is used as the ancestor of the file list box, *TFileList*.

Fields

SearchPos

```
SearchPos: Word;
```

SearchPos indicates which character position is being checked for incremental searching.

ShiftState

```
ShiftState: Byte;
```

ShiftState holds the current state of the keyboard shift keys for multiple selection purposes.

Methods

Init

```
constructor Init(var Bounds: TRect; ANumCols: Word; AScrollBar: PScrollBar);
```

Constructs a sorted list box by calling the *Init* constructor inherited from *TListBox*, passing the bounding rectangle *Bounds*, number of columns *ANumCols*, and horizontal scroll bar *AScrollBar*. The *ShiftState* field is initialized to zero, and the cursor is shown at the first item.

See also: *TListBox.Init*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Handles normal list box events such as mouse clicks and cursor keys by a call to the *HandleEvent* method inherited from *TListBox*.

Other keyboard events are handled directly to implement incremental searching. That is, if the user presses a character key, the first item name beginning with that character gets focused. If the user presses another character key, focus will move to the first item whose second character matches the pressed character, if such an item exists; otherwise, the focus stays. This process continues until the user either selects an item or moves the focus with arrow keys or the mouse, in which case incremental search reverts to the first character.

The *SearchPos* field tracks which character is currently being matched in the incremental search. Pressing **Backspace** backs up the incremental search one character, to the item selected by the previous character.

GetKey

```
function GetKey(var S: String): Pointer; virtual;
```

Sorted list boxes need a key on which to sort their entries. *GetKey* returns a pointer to the key for the string *S*. By default, *GetKey* returns a pointer to *S*. Depending on the sorting strategy your derived objects use, you will probably want to override *GetKey* to return some other key.

NewList

```
procedure NewList(AList: PCollection); virtual;
```

Replaces the sorted collection currently pointed to by *List* by calling the *NewList* method inherited from *TListBox*, which disposes of *List* and sets *List* to *AList*, which should point to a sorted collection. *NewList* then resets *SearchPos* to zero, so that incremental searches in the new list start with the first character in the item string.

See also: *TListBox.NewList*

TStaticText Dialogs

TObject

~~Init~~
Free
~~Done~~

TView

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

~~Init~~
~~Load~~
~~Done~~
Awaken
BlockCursor
CalcBounds
ChangeBounds
ClearEvent
CommandEnabled
DataSize
DisableCommands
DragView
~~Draw~~
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
GetData
GetEvent
GetExtent
GetHelpCtx
~~GetPalette~~
GetPeerViewPtr
GetState
GrowTo
HandleEvent
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
SetData
SetState
Show
ShowCursor
SizeLimits
~~Store~~
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

TStaticText

Text

Init
Load
Done
Draw
GetPalette
GetText
Store

TStaticText objects represent the simplest possible views: they contain fixed text and they ignore all events passed to them. They are generally used as messages or passive labels. Descendants of *TStaticText* perform more active roles.

Field

Text

`Text: PString;` **Read only**

A pointer to the text string to be displayed in the view.

Methods

Init

```
constructor Init(var Bounds: TRect; const AText: String);
```

Constructs a static text object of the given size by calling the *Init* constructor inherited from *TView,* then sets *Text* to *NewStr(AText).*

See also: *TView.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and initializes a static text object from the stream *S* by first calling the *Load* constructor inherited from *TView,* then reading *Text* with

S.ReadStr. Use in conjunction with *TStaticText.Store* to save and retrieve static text views on a stream.

See also: *TView.Load, TStaticText.Store, TStream.ReadStr*

Done `destructor Done; virtual;`

Override: Seldom

Disposes of the *Text* string then calls the *Done* destructor inherited from *TView* to dispose of the object.

Draw `procedure Draw; virtual;`

Override: Seldom

Draws the text string inside the view, word wrapped if necessary. A *Ctrl+M* in the text indicates the beginning of a new line. A line of text is centered in the view if the line begins with *Ctrl+C*.

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes

Returns a pointer to the default palette, *CStaticText*.

GetText `procedure GetText(var S: String); virtual;`

Override: Sometimes

Returns the string pointed to by *Text* in *S*.

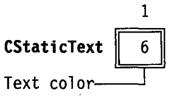
Store `procedure TStaticText.Store(var S: TStream);`

Writes the static text object on the stream *S* by first calling the *Store* method inherited from *TView*, reading *Text* with *S.WriteStr*. Use in conjunction with *Load* to save and retrieve static text views on a stream.

See also: *TStaticText.Load, TView.Store, TStream.WriteStr*

Palette

Static text objects use the default palette, *CStaticText*, to map onto the 6th entry in the standard dialog palette.



TStatusDef type — Menus

Declaration

```
TStatusDef = record
  Next: PStatusDef;
  Min, Max: Word;
  Items: PStatusItem;
end;
```

Function The *TStatusDef* type represents a status line definition. The *Next* field points to the next *TStatusDef* in a list of status lines, or is **nil** if this is the last status definition. *Min* and *Max* define the range of help contexts that correspond to the status line. *Items* points to a list of status line items, or is **nil** if there are no status line items.

A *TStatusLine* object (the actual status line view) has a pointer to a list of *TStatusDef* records, and will always display the first status line for which the current help context is within *Min* and *Max*. A Turbo Vision application automatically updates the status line view by calling *TStatusLine.Update* from *TProgram.Idle*.

TStatusDef records are created using the *NewStatusDef* function.

See also *TStatusLine, TProgram.Idle, NewStatusDef* function

TStatusItem type — Menus

Declaration

```
TStatusItem = record
  Next: PStatusItem;
  Text: PString;
  KeyCode: Word;
  Command: Word;
end;
```

Function The *TStatusItem* type represents a status line item that can be visible or invisible. *Next* points to the next *TStatusItem* within a list of status items, or is **nil** if this is the last item. *Text* points to a string containing the status item legend (such as '**Alt+X** Exit'), or is **nil** if the status item is invisible (in which case the item serves only to define a hot key). *KeyCode* contains the scan code of the hot key associated with the status item, or zero if the status item has no hot key. *Command* contains the command event (see *cmXXXX* constants) to be generated when the status item is selected.

TStatusItem records function not only as definitions of the visual appearance of the status line, but are also used to define hot keys, that is, an automatic mapping of key codes into commands. The *TProgram.GetEvent* method calls *TStatusLine.HandleEvent* for all *evKeyDown* events. *TStatusLine.HandleEvent* scans the current status line for an item containing the given key code, and if one is found, it converts that *evKeyDown* event to an *evCommand* event with the *Command* value given by the *TStatusItem*.

TStatusItem records are created using the *NewStatusKey* function.

See also *TStatusLine, NewStatusKey, TStatusLine.HandleEvent*

TStatusLine Menus

TObject

~~Init~~ Free ~~Done~~

TView

Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	

~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
~~Done~~	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
ChangeBounds	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	SetState
~~Draw~~	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TStatusLine

Items Defs
Init Load Done Draw GetPalette HandleEvent Hint Store Update

The *TStatusLine* object is a specialized view, usually displayed at the bottom of the screen. Typical status line displays are lists of available hot keys, displays of available memory, time of day, current edit modes, and hints for users. The items to be displayed are set up in a linked list by the application object's *InitStatusLine* method, called by the application's constructor. The status line displayed depends on the help context of the currently focused view. Like the menu bar and desktop, the status line is normally owned by a *TApplication* group.

Status line items are records of type *TStatusItem*, which contain fields for a text string to be displayed on the status line, a key code to bind a hot key (typically a function key or an *Alt*+key combination), and a command to be generated if the displayed text is clicked with the mouse or the hot key is pressed.

Status line displays are help context-sensitive. Each status line object contains a linked list of status line *Defs* (of type *TStatusDef*), which define a range of help contexts and a list of status items to be displayed when the current help context is in that range. In addition, *hints* or predefined strings can be displayed according to the current help context.

Fields

Defs

`Defs: PStatusDef;` **Read only**

A pointer to the current linked list of *TStatusDef* records. The list to use is determined by the current help context.

See also: *TStatusDef, TStatusLine.Update, TStatusLine.Hint*

Items

`Items: PStatusItem;` **Read only**

A pointer to the current linked list of *TStatusItem* records.

See also: *TStatusItem*

Methods

Init

```
constructor Init(var Bounds: TRect; ADefs: PStatusDef);
```

Constructs a status line object with the given *Bounds* by calling the *Init* constructor inherited from *TView*. Sets the *ofPreProcess* bit in *Options*, sets *EventMask* to include *evBroadcast*, and sets *GrowMode* to *gfGrowLoY* + *gfGrowHiX* + *gfGrowHiY*. Sets *Defs* to *ADefs*. If *ADefs* is **nil**, sets *Items* to **nil**; otherwise, sets *Items* to *ADefs^.Items*

See also: *TView.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs a status line object and loads it from the stream *S* by first calling the *Load* constructor inherited from *TView* and then reading the *Defs* and *Items* from the stream.

See also: *TView.Load, TStatusLine.Store*

Done

```
destructor Done; virtual;
```

Override: Never

Disposes of all the *Items* and *Defs* in the status line object, then calls the *Done* destructor inherited from *TView* to dispose of the object.

See also: *TView.Done*

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Draws the status line by writing the *Text* string for each status item that has one, then any hints defined for the current help context, following a divider bar.

See also: *TStatusLine.Hint*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the default palette, *CStatusLine*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Seldom

Handles most events sent to the status line by calling the *HandleEvent* method inherited from *TView*, then checking for three kinds of special events.

- Mouse clicks that fall within the rectangle occupied by any status item generate a command event with *Event.What* set to the *Command* in that status item.
- Key events are checked against the *KeyCode* field in each item; a match causes a command event with that item's *Command*.
- Broadcast events with the command *cmCommandSetChanged* cause the status line to redraw itself to reflect any hot keys that might have been enabled or disabled.

See also: *TView.HandleEvent*

Hint

```
function Hint(AHelpCtx: Word): String; virtual;
```

Override: Often

By default, *Hint* returns a null string. You can override it in descendant status line objects to return a context-sensitive hint string for the *AHelpCtx* parameter. A non-null string will be drawn on the status line after a divider bar.

See also: *TStatusLine.Draw*

Store

```
procedure Store(var S: TStream);
```

Writes the status line object to the stream *S* by first calling the *Store* method inherited from *TView*, then writing all the status definitions and their associated lists of items onto the stream. The saved object can be recovered by using *TStatusLine.Load*.

See also: *TView.Store, TStatusLine.Load*

Update

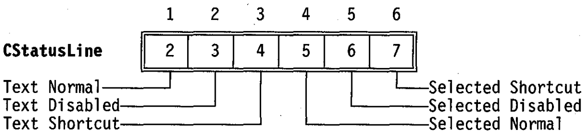
```
procedure Update;
```

Selects the correct *Items* from the lists in *Defs*, depending on the current help context, then calls *DrawView* to redraw the status line if the items have changed.

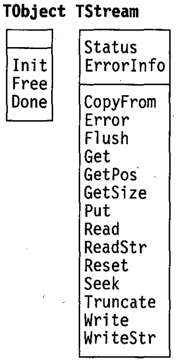
See also: *TStatusLine.Defs*

Palette

Status lines use the default palette *CStatusLine* to map onto the 2nd through 7th entries in the standard application palette.



TStream Objects



TStream is a general abstract object providing polymorphic I/O to and from a storage device. You can create your own descendant stream objects by overriding the virtual methods *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate*, and *Write*. Turbo Vision itself does this to derive *TDosStream* and *TEmsStream*. For buffered stream descendants, you must also override *TStream.Flush*.

Fields

ErrorInfo

`ErrorInfo: Integer` **Read/write**

ErrorInfo contains additional information when *Status* is not *stOk*. For *Status* values of *stError*, *stInitError*, *stReadError*, and *stWriteError*, *ErrorInfo* contains the DOS or EMS error code, if one is available. When *Status* is *stGetError*, *ErrorInfo* contains the object type ID (the *ObjType* field of a *TStreamRec*) of the unregistered object type. When *Status* is *stPutError*, *ErrorInfo* contains the VMT data segment offset (the *VmtLink* field of a *TStreamRec*) of the unregistered object type.

Status `Status: Integer` **Read/write**

Indicates the current status of the stream. The value of *Status* is one of the *stXXXX* constants. If *Status* is not *stOk*, all operations on the stream are suspended until *Reset* is called.

See also: *stXXXX* stream constants

Methods

CopyFrom `procedure CopyFrom(var S: TStream; Count: Longint);`

Copy *Count* bytes to the stream from stream *S*. For example:

```
{Create a copy of entire stream}
NewStream := New(PEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

See also: *TStream.GetSize, TObject.Init*

Error `procedure Error(Code, Info: Integer); virtual;`

Override: Sometimes

Called whenever a stream error occurs. The default *Error* stores *Code* and *Info* in the *Status* and *ErrorInfo* fields and then, if the global variable *StreamError* is not **nil**, calls the procedure pointed to by *StreamError*. Once an error has occurred, all stream operations on the stream are suspended until *Reset* is called.

See also: *TStream.Reset, StreamError* variable

Flush `procedure Flush; virtual;`

Override: Sometimes

An abstract method that must be overridden if your descendant implements a buffer. This method can flush any buffers by clearing the read buffer, by writing the write buffer, or both. The default *TStream.Flush* does nothing.

See also: *TDosStream.Flush*

Get `function Get: PObject;`

Reads an object from the stream and returns a pointer to it. The object must have been previously written to the stream by *Put*. *Get* first reads an object type ID (a word) from the stream. It then finds the corresponding object type by comparing the ID to the *ObjType* field of all registered object types (see the *TStreamRec* type), and finally calls the *Load* constructor of that object type to construct and initialize the object. If the object type ID read from the stream is zero, *Get* returns a **nil** pointer; if the object type ID has not been registered (using *RegisterType*), *Get* calls *TStream.Error* and returns a **nil** pointer; otherwise, *Get* returns a pointer to the newly created object.

See also: *TStream.Put, RegisterType, TStreamRec, Load* methods

GetPos

```
function GetPos: Longint; virtual;
```

Override: Always

Returns the stream's current position. This is an abstract method that must be overridden.

See also: *TStream.Seek*

GetSize

```
function GetSize: Longint; virtual;
```

Override: Always

Returns the total size of the stream. This is an abstract method that must be overridden.

Put

```
procedure Put(P: PObject);
```

Writes the object *P* to the stream. The object can later be read from the stream using *TStream.Get. Put* first finds the type registration record of the object by comparing the object's VMT offset to the *VmtLink* field of all registered object types (see the *TStreamRec* type). It then writes the object type ID (the *ObjType* field of the registration record) to the stream, and finally calls the *Store* method of that object type to write the object.

If *P* is **nil**, *Put* writes a word containing zero to the stream. If the object type of *P* has not been registered (using *RegisterType*), *Put* calls *TStream.Error* and doesn't write anything to the stream.

See also: *TStream.Get, RegisterType, TStreamRec, Store* methods

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Override: Always

This is an abstract method that must be overridden in all descendant types. *Read* should read *Count* bytes from the stream into *Buf* and advance the current position of the stream by *Count* bytes. If an error occurs, *Read* should call *Error*, and fill *Buf* with *Count* bytes of 0.

See also: *TStream.Write, TStream.Error.*

ReadStr

```
function ReadStr: PString;
```

Reads a string from the current position of the calling stream, returning a *PString* pointer. *TStream.ReadStr* calls *GetMem* to allocate (Length + 1) bytes for the string.

See also: *TStream.WriteStr*

Reset `procedure Reset;`

Resets any stream error condition by setting *Status* and *ErrorInfo* to 0. *Reset* enables you to continue stream processing after correcting an error condition.

See also: *TStream.Status, TStream.ErrorInfo, stXXXX* error codes

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Always

This is an abstract method that must be overridden by all descendants. *TStream.Seek* sets the current position to *Pos* bytes from the start of the calling stream. The start of a stream is position 0.

See also: *TStream.GetPos*

Truncate `procedure Truncate; virtual;`

Override: Always

This is an abstract method that must be overridden by all descendants. *Truncate* deletes all data on the calling stream from the current position to the end.

See also: *TStream.GetPos, TStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Always

This is an abstract method that must be overridden in all descendant types. *Write* should write *Count* bytes from *Buf* onto the stream and advance the current position of the stream by *Count* bytes. If an error occurs, *Write* should call *Error*.

See also: *TStream.Read, TStream.Error.*

WriteStr `procedure WriteStr(P: PString);`

Writes the string *P^* to the calling stream, starting at the current position.

See also: *TStream.ReadStr*

TStreamRec type — Objects

T

Declaration

```
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

Function A Turbo Vision object type must have a registered *TStreamRec* before its objects can be loaded or stored on a *TStream* object. The *RegisterTypes* routine registers an object type by setting up a *TStreamRec* record.

The fields in the stream registration record are defined as follows:

Table 19.42
Stream record fields

Field	Contents
ObjType	A unique numerical id for the object type
VmtLink	A link to the object type's virtual method table entry
Load	A pointer to the object type's *Load* constructor
Store	A pointer to the object type's *Store* method
Next	A pointer to the next *TStreamRec*

☞ Turbo Vision reserves object type IDs (*ObjType*) values 0 through 999 for its own use. Programmers can define their own values in the range 1,000 to 65,535.

By convention, a *TStreamRec* for a *Txxxx* object type is called *Rxxxx*. For example, the *TStreamRec* for a *TCalculator* type is called *RCalculator*, as shown in the following code:

```
type
  TCalculator = object(TDialog)
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
      ⋮
  end;

const
  RCalculator: TStreamRec = (
    ObjType: 2099;
    VmtLink: Ofs(TypeOf(TCalculator)^);
    Load: @TCalculator.Load;
    Store: @TCalculator.Store);

begin
  RegisterType(RCalculator);
   ⋮
end;
```

See also *RegisterType*

TStrIndex type Objects

Declaration `TStrIndex = array[0..9999] of TStrIndexRec;`

Function Used internally by *TStringList* and *TStrListMaker*.

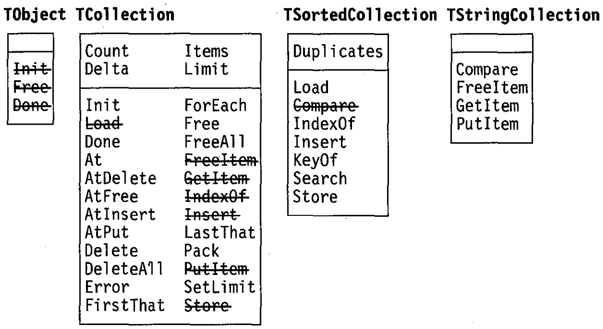
TStrIndexRec type — Object

Declaration

```
TStrIndexRec = record
  Key, Count, Offset: Word;
end;
```

Function Used internally by *TStringList* and *TStrListMaker*.

TStringCollection — Objects



TStringCollection is a simple derivative of *TSortedCollection* implementing a sorted list of ASCII strings. The *Compare* method is overridden to provide ASCII string ordering. You can override *Compare* to allow for other orderings, such as those for non-English character sets.

Methods

Compare

```
function Compare(Key1, Key2: Pointer): Integer; virtual;
```

Override: Sometimes

Compares the strings *Key1*^ and *Key2*^ as follows: return –1 if *Key1* < *Key2*; 0 if *Key1* = *Key2*; and +1 if *Key1* > *Key2*.

See also: *TSortedCollection.Search*

FreeItem

```
procedure FreeItem(Item: Pointer); virtual;
```

Override: Seldom

Removes the string *Item*^ from the sorted collection and disposes of the string.

GetItem

```
function GetItem(var S: TStream): Pointer; virtual;
```

Override: Seldom

By default, reads a string from the stream *S* by calling *S.ReadStr*.

See also: *TStream.ReadStr*

PutItem

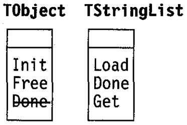
```
procedure PutItem(var S: TStream; Item: Pointer); virtual;
```

Override: Seldom

By default, writes the string *Item^* to the stream *S* by calling *S.WriteStr*.

See also: *TStream.WriteStr*

TStringList — Objects



TStringList provides a mechanism for accessing strings stored on a stream. Each string in a string list is identified by a unique number (its key) between 0 and 65,535. String lists take up less memory than normal string literals, since the strings are stored on a stream instead of in memory. Also, string lists permit easy internationalization, as the strings are not "burned into" the program.

TStringList has methods only for accessing strings; you must use *TStrListMaker* to create string lists.

Note that *TStringList* and *TStrListMaker* have the same object type ID (*ObjType* field in a *TStreamRec*), and that they can therefore not both be registered and used in the same program.

Methods

Load

```
constructor Load(var S: TStream);
```

Loads the string list index from the stream *S* and stores internally a reference to *S* so that *TStringList.Get* can later access the stream when reading strings.

Assuming that *TStringList* has been registered using *RegisterType(RStringList)*, here's how to instantiate string list (created using *TStrListMaker* and *TResourceFile.Put*) from a resource file:

```
ResFile.Init(New(TBufStream, Init('MYAPP.RES', stOpenRead, 1024)));
Strings := PStringList(ResFile.Get('Strings'));
```

See also: *TStrListMaker.Init, TStringList.Get*

Done

```
destructor Done; virtual;
```

Override: Never Deallocates the memory allocated to the string list.

See also: *TStrListMaker.Init, TStringList.Done*

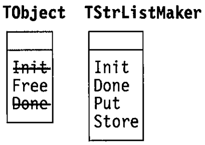
Get `function Get(Key: Word): String;`

Returns the string given by *Key*, or an empty string if there is no string with the given *Key*. An example:

```
P := @FileName;
FormatStr(S, Strings^.Get(sLoadingFile), P);
```

See also: *TStrListMaker.Put*

TStrListMaker Objects



TStrListMaker is a simple object type used to create string lists for use with *TStringList.*

The following code fragment shows how to create and store a string list in a resource file.

```
const
  sInformation  = 100;
  sWarning      = 101;
  sError        = 102;
  sLoadingFile  = 200;
  sSavingFile   = 201;

var
  ResFile: TResourceFile;
  S: TStrListMaker;

begin
  RegisterType(RStrListMaker);

  ResFile.Init(New(TBufStream, Init('MYAPP.RES', stCreate, 1024)));
  S.Init(16384, 256);

  S.Put(sInformation, 'Information');
  S.Put(sWarning, 'Warning');
  S.Put(sError, 'Error');
  S.Put(sLoadingFile, 'Loading file %s.');
```

```
      S.Put(sSavingFile, 'Saving file %s.');

      ResFile.Put(@S, 'Strings');
      S.Done;
      ResFile.Done;
   end;
```

Methods

Init `constructor Init(AStrSize, AIndexSize: Word);`

Creates an in-memory string list of size *AStrSize* with an index of *AIndexSize* elements. A string buffer and an index buffer of the specified size is allocated on the heap.

AStrSize must be large enough to hold all strings to be added to the string list—each string occupies its length plus one bytes.

As strings are added to the string list (using *TStrListMaker.Put*), a string index is built. Strings with contiguous keys (such as *sInformation*, *sWarning*, and *sError* in the example above) are recorded in one index record, up to 16 at a time. *AIndexSize* must be large enough to allow for all index records generated as strings are added. Each index entry occupies 6 bytes.

See also: *TStringList.Load, TStrListMaker.Done*

Done `destructor Done; virtual;`

Frees the memory allocated to the string list maker.

See also: *TStrListMaker.Init*

Put `procedure Put(Key: Word; S: String);`

Add the given string *S* to the string list with the given numerical *Key*.

Store `procedure Store(var S: TStream);`

Stores the string list on the stream *S*.

TSysErrorFunc type Drivers

Declaration `TSysErrorFunc = function(ErrorCode: Integer; Drive: Byte): Integer;`

Function *TSysErrorFunc* defines the type of a system error handler function.

See also *SysErrorFunc, SystemError, InitSysError, DoneSysError*

TStringLookupValidator Validate

TObject	TValidator	TLookupValidator	TStringLookupValidator
	Options Status		Strings
~~Init~~ Free ~~Done~~	~~Init~~ ~~Load~~ ~~Error~~ ~~IsValid~~ IsValidInput ~~Store~~ Transfer Valid	IsValid ~~Lookup~~	Init Load Done Error Lookup NewStringList Store

A string lookup validator object verifies the data in its associated input line by searching through a collection of valid strings. Use string lookup validators when your input line needs to accept only members of a certain set of strings.

Field

Strings

```
Strings: PStringCollection;
```

Points to a string collection containing all the valid strings the user can type. If *Strings* is **nil**, all input will be invalid.

Methods

Init

```
constructor Init(AStrings: PStringCollection);
```

Constructs a string lookup validator object by first calling the *Init* constructor inherited from *TLookupValidator*, then setting *Strings* to *AStrings*.

See also: *TLookupValidator.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a string lookup validator object from the stream *S* by first calling the *Load* constructor inherited from *TLookupValidator*, then reading the string collection *Strings*.

See also: *TLookupValidator.Load*

Done

```
destructor Done; virtual;
```

Disposes of the list of valid strings by calling *NewStringList(**nil**)*, then disposes of the string lookup object by calling the *Done* destructor inherited from *TLookupValidator*.

See also: *TLookupValidator.Done, TStringLookupValidator.NewStringList*

Error

```
procedure Error; virtual;
```

Displays a message box indicating that the typed string does not match an entry in the string list.

Lookup

```
function Lookup(const S: string): Boolean; virtual;
```

Returns *True* if the string passed in *S* matches any of the strings in *Strings*. Uses the *Search* method of the string collection to determine if *S* is present.

See also: *TSortedCollection.Search*

NewStringList

```
procedure NewStringList(AStrings: PStringCollection);
```

Sets the list of valid input strings for the string lookup validator. Disposes of any existing string list, then sets *Strings* to *AStrings*. Passing **nil** in *AStrings* disposes of the existing list without assigning a new one.

Store

```
procedure Store(var S: TStream);
```

Stores the string lookup validator on the stream *S* by first calling the *Store* method inherited from *TValidator* and then writing the string collection held in *Strings*.

TTerminal TextView

TObject	TView			TScroller	TTextDevice	TTerminal
	Cursor DragMode EventMask GrowMode	HelpCtx Next Options Origin	Owner Size State	HScrollBar VScrollBar Delta Limit		Buffer BufSize QueBack QueFront
~~Init~~ Free ~~Done~~	~~Init~~ ~~Load~~ ~~Done~~ Awaken BlockCursor CalcBounds ~~ChangeBounds~~ ClearEvent CommandEnabled DataSize DisableCommands DragView ~~Draw~~ DrawView EnableCommands EndModal EventAvail Execute Exposed Focus GetBounds GetClipRect GetColor	GetCommands GetData GetEvent GetExtent GetHelpCtx ~~GetPalette~~ GetPeerViewPtr GetState GrowTo ~~HandleEvent~~ Hide HideCursor KeyEvent Locate MakeFirst MakeGlobal MakeLocal MouseEvent MouseInView MoveTo NextView NormalCursor	Prev PrevView PutEvent PutInFrontOf PutPeerViewPtr Select SetBounds SetCommands SetCmdState SetCursor SetData ~~SetState~~ Show ShowCursor SizeLimits ~~Store~~ TopView Valid WriteBuf WriteChar WriteLine WriteStr	~~Init~~ Load ChangeBounds GetPalette HandleEvent ScrollDraw ScrollTo SetLimit SetState Store	~~StrRead~~ ~~StrWrite~~	Init Done BufDec BufInc CalcWidth CanInsert Draw NextLine PrevLines QueEmpty StrRead StrWrite

TTerminal implements a "dumb" terminal with buffered string reads and writes. The default is a cyclic buffer of 64K bytes. The terminal view is an example of a text file device driver connected to a scrolling view.

Fields

Buffer

`Buffer: PTerminalBuffer;` **Read only**

Points to the first byte of the terminal's buffer.

BufSize

`BufSize: Word;` **Read only**

The size of the terminal's buffer in bytes.

QueBack

`QueBack: Word;` **Read only**

Offset (in bytes) of the last byte stored in the terminal buffer.

QueFront

`QueFront: Word;` **Read only**

Offset (in bytes) of the first byte stored in the terminal buffer.

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar;
  ABufSize: Word);
```

Constructs a *TTerminal* object with the given *Bounds*, horizontal and vertical scroll bars, and buffer by calling the *Init* constructor inherited from *TTextDevice* with the *Bounds* and scroller arguments, then creating a buffer (pointed to by *Buffer*) with *BufSize* equal to *ABufSize*. Sets *GrowMode* to *gfGrowHiX* + *gfGrowHiY*. *QueFront* and *QueBack* are both initialized to 0, indicating an empty buffer. Shows the cursor at the view's origin, (0,0).

See also: *TScroller.Init*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Deallocates the buffer and calls the *Done* destructor inherited from *TTextDevice* to dispose of the object.

See also: *TScroller.Done, TTextDevice.Done*

BufDec

```
procedure BufDec(var Val: Word);
```

Used to manipulate queue offsets with wrap around: If *Val* is zero, *Val* is set to (*BufSize* – 1); otherwise, *Val* is decremented.

See also: *TTerminal.BufInc*

BufInc

```
procedure BufInc(var Val: Word);
```

Used to manipulate queue offsets with wrap around: Increments *Val* by 1, then if *Val* >= *BufSize, Val* is set to zero.

See also: *TTerminal.BufDec*

CalcWidth

```
function CalcWidth: Integer;
```

Returns the length of the longest line in the text buffer.

CanInsert

```
function CanInsert(Amount: Word): Boolean;
```

Returns *True* if the number of bytes given in *Amount* can be inserted into the terminal buffer without having to discard the top line.

Draw

```
procedure Draw; virtual;
```

Override: Seldom

Called whenever the *TTerminal* scroller needs to be redrawn, for example, when the scroll bars are clicked on, the view is unhidden or resized, the *Delta* values are changed, or when added text forces a scroll.

NextLine

```
function NextLine(Pos:Word): Word;
```

Returns the buffer offset of the start of the line that follows the position given by *Pos*.

See also: *TTerminal.PrevLines*

PrevLines `function PrevLines(Pos:Word; Lines: Word): Word;`

Returns the offset of the start of the line that is *Lines* lines previous to the position given by *Pos*.

See also: *TTerminal.NextLine*

QueEmpty `function QueEmpty: Boolean;`

Returns true if *QueFront* is equal to *QueBack*.

See also: *TTerminal.QueFront, TTerminal.QueBack*

StrRead `function StrRead(var S: TextBuf): Byte; virtual;`

Override: Sometimes

Abstract method returning 0. You must override *StrRead* if you want a descendant type to be able to read strings from the text buffer.

StrWrite `procedure StrWrite(var S: TextBuf; Count: Byte); virtual;`

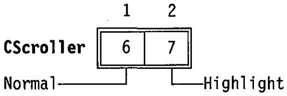
Override: Seldom

Inserts *Count* lines of the text given by *S* into the terminal's buffer. This method handles any scrolling required by the insertion and selectively redraws the view with *DrawView*.

See also: *TView.DrawView*

Palette

Terminal objects use the default palette, *CScroller*, to map onto the 6th and 7th entries in the standard application palette.



TTerminalBuffer type TextView

Declaration `TTerminalBuffer = array[0..65534] of Char;`

Function Used internally by *TTerminal*.

See also *TTerminal*

TTextDevice

TextView

TObject
~~Init~~
Free
~~Done~~

TView		
Cursor	HelpCtx	Owner
DragMode	Next	Size
EventMask	Options	State
GrowMode	Origin	
~~Init~~	GetCommands	Prev
~~Load~~	GetData	PrevView
Done	GetEvent	PutEvent
Awaken	GetExtent	PutInFrontOf
BlockCursor	GetHelpCtx	PutPeerViewPtr
CalcBounds	~~GetPalette~~	Select
~~ChangeBounds~~	GetPeerViewPtr	SetBounds
ClearEvent	GetState	SetCommands
CommandEnabled	GrowTo	SetCmdState
DataSize	~~HandleEvent~~	SetCursor
DisableCommands	Hide	SetData
DragView	HideCursor	~~SetState~~
Draw	KeyEvent	Show
DrawView	Locate	ShowCursor
EnableCommands	MakeFirst	SizeLimits
EndModal	MakeGlobal	~~Store~~
EventAvail	MakeLocal	TopView
Execute	MouseEvent	Valid
Exposed	MouseInView	WriteBuf
Focus	MoveTo	WriteChar
GetBounds	NextView	WriteLine
GetClipRect	NormalCursor	WriteStr
GetColor		

TScroller
HScrollBar
VScrollBar
Delta
Limit
Init
Load
ChangeBounds
GetPalette
HandleEvent
ScrollDraw
ScrollTo
SetLimit
SetState
Store

TTextDevice
StrRead
StrWrite

TTextDevice is a scrollable TTY type text viewer/device driver. Apart from the fields and methods inherited from *TScroller*, *TTextDevice* defines virtual methods for reading and writing strings from and to the device. *TTextDevice* exists solely as a base type for deriving real terminal drivers. *TTextDevice* uses *TScroller's* constructor and destructor.

Methods

StrRead

```
function StrRead(var S: TextBuf): Byte; virtual;
```

Override: Often

Abstract method returning 0 by default. You must override *StrRead* in any descendant type to read a string from a text device into *S*. *StrRead* returns the number of lines read.

StrWrite

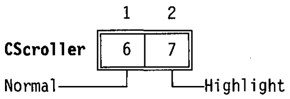
```
procedure StrWrite(var S: TextBuf; Count: Byte); virtual;
```

Override: Always

Abstract method to write a string to the device. It must be overridden by derived types. For example, *TTerminal.StrWrite* inserts *Count* lines of the text given by *S* into the terminal's buffer and redraws the view.

Palette

Text device objects use the default palette *CScroller* to map onto the 6th and 7th entries in the standard application palette.



TTitleStr type

Views

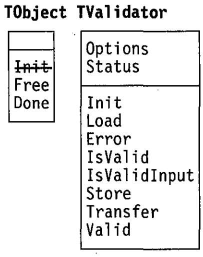
Declaration `TTitleStr = string[80];`

Function This type is used to declare text strings for titled windows.

See also *TWindow.Title*

TValidator

Validate



TValidator defines an abstract data validation object. You will never actually create an instance of *TValidator*, but it provides much of the abstract functions for the other data validation objects.

Fields

Options `Options: Word;`

Options is a bitmapped field used to control options for various descendants of *TValidator*. By default, *TValidator.Init* clears all the bits in *Options*.

See also: *voXXXX* constants

Status `Status: Word;`

Indicates the status of the validator object. If *Status* is *vsOK,* the validator object constructed correctly. Any value other than *vsOK* indicates that an error occurred.

See also: *TInputLine.Valid, ValidatorOK* constant

Methods

Init

```
constructor Init;
```

Constructs an abstract validator object by first calling the *Init* constructor inherited from *TObject,* then setting the *Options* and *Status* fields to zero.

See also: *TObject.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs a validator object by calling the *Init* constructor inherited from *TObject,* then reads the *Options* word from the stream *S.*

See also: *TObject.Init*

Error

```
procedure Error; virtual;
```

Error is an abstract method called by *Valid* when it detects that the user has entered invalid information. By default, *TValidator.Error* does nothing, but descendant types can override *Error* to provide feedback to the user.

IsValidInput

```
function IsValidInput(var S: string; SuppressFill: Boolean): Boolean;
  virtual;
```

If an input line has an associated validator object, it calls *IsValidInput* after processing each keyboard event. This gives validators such as filter validators an opportunity to catch errors before the user fills the entire item or screen.

By default, *TValidator.IsValidInput* returns *True.* Descendant data validators can override *IsValidInput* to validate data as the user types it, returning *True* if *S* holds valid data and *False* otherwise.

S is the current input string. *SuppressFill* determines whether the validator should automatically format the string before validating it. If *SuppressFill* is *True,* validation takes place on the unmodified string *S.* If *SuppressFill* is *False,* the validator should apply any filling or padding before validating data. Of the standard validator objects, only *TPXPictureValidator* checks *SuppressFill.*

Because *S* is a **var** parameter, *IsValidInput* can modify the contents of the input string, such as forcing characters to uppercase or inserting literal characters from a format picture. *IsValidInput* should not, however, delete invalid characters from the string. By returning *False*, *IsValidInput* indicates that the input line should erase the offending characters.

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

By default, *TValidator.IsValid* returns *True*. Descendant validator types can override *IsValid* to validate data for a completed input line. If an input line has an associated validator object, its *Valid* method calls the validator object's *Valid* method, which in turn calls *IsValid* to determine whether the contents of the input line are valid.

See also: *TInputLine.Valid*, *TValidator.Valid*

Store

```
procedure Store(var S: TStream);
```

Writes the validator object to the stream *S* by writing the value of the *Options* field.

Transfer

```
function Transfer(var S: String; Buffer: Pointer; Flag: TVTransfer): Word;
  virtual;
```

Transfer allows a validator to take over setting and reading the values of its associated input line, which is mostly useful for validators that check non-string data, such as numeric values. For example, *TRangeValidator* uses *Transfer* to read and write *Longint*-type values to a data record, rather than transferring an entire string.

By default, input lines with validators give the validator the first chance to respond to *DataSize*, *GetData*, and *SetData* by calling the validator's *Transfer* method. If *Transfer* returns anything other than 0, it indicates to the input line that it has handled the appropriate transfer. The default action of *TValidator.Transfer* is to return 0 always. If you want the validator to transfer data, you need to override its *Transfer* method.

Transfer's first two parameters are the associated input line's text string and the *GetData* or *SetData* data record. Depending on the value of *Flag*, *Transfer* can set *S* from *Buffer* or read the data from *S* into *Buffer*. The return value is always the number of bytes transferred.

If *Flag* is *vtDataSize*, *Transfer* doesn't change either *S* or *Buffer*, but just returns the data size. If *Flag* is *vtSetData*, *Transfer* reads the appropriate number of bytes from *Buffer*, converts them into the proper string form, and sets them into *S*, returning the number of bytes read. If *Flag* is *vtGetData*, *Transfer* converts *S* into the appropriate data type and writes the value into *Buffer*, returning the number of bytes written.

See also: *TInputLine.DataSize*, *TInputLine.GetData*, *TInputLine.SetData*

Valid `function Valid(const S: string): Boolean;`

Returns *True* if *IsValid(S)* returns *True*. Otherwise, calls *Error* and returns *False*. A validator's *Valid* method is called by the *Valid* method of its associated input line.

Input lines with associated validators call the validator's *Valid* method under two conditions: either the input line has its *ofValidate* option set, in which case it calls *Valid* when it loses focus, or the dialog box that contains the input line calls *Valid* for all its controls, usually because the user requested to close the dialog box or accept an entry screen.

See also: *TInputLine.Valid*, *TValidator.Error*, *TValidator.IsValid*

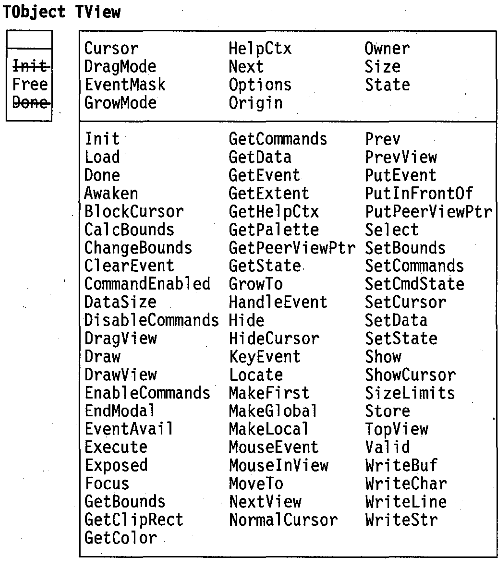
TVideoBuf type Views

Declaration `TVideoBuf = array[0..3999] of Word;`

Function This type is used to declare video buffers.

See also *TGroup.Buffer*

TView Views



TObject TView

~~Init~~
Free
~~Done~~

Cursor
DragMode
EventMask
GrowMode
HelpCtx
Next
Options
Origin
Owner
Size
State

Init
Load
Done
Awaken
BlockCursor
CalcBounds
ChangeBounds
ClearEvent
CommandEnabled
DataSize
DisableCommands
DragView
Draw
DrawView
EnableCommands
EndModal
EventAvail
Execute
Exposed
Focus
GetBounds
GetClipRect
GetColor
GetCommands
GetData
GetEvent
GetExtent
GetHelpCtx
GetPalette
GetPeerViewPtr
GetState
GrowTo
HandleEvent
Hide
HideCursor
KeyEvent
Locate
MakeFirst
MakeGlobal
MakeLocal
MouseEvent
MouseInView
MoveTo
NextView
NormalCursor
Prev
PrevView
PutEvent
PutInFrontOf
PutPeerViewPtr
Select
SetBounds
SetCommands
SetCmdState
SetCursor
SetData
SetState
Show
ShowCursor
SizeLimits
Store
TopView
Valid
WriteBuf
WriteChar
WriteLine
WriteStr

The *TView* object type exists to provide basic fields and methods for its descendants. You'll probably never need to construct an instance of *TView* itself, but most of the common behavior of visible elements in Turbo Vision applications comes from *TView*.

Fields

Cursor

`Cursor: TPoint;` **Read only**

The location of the hardware cursor within the view. The cursor is visible only if the view is focused (*sfFocused*) and the cursor turned on (*sfCursorVis*). The shape of the cursor is either underline or block (determined by *sfCursorIns*).

See also: *SetCursor, ShowCursor, HideCursor, NormalCursor, BlockCursor*

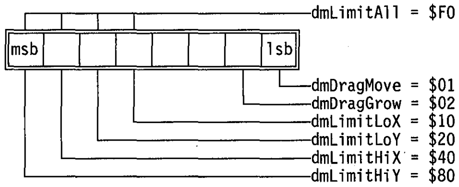
DragMode

`DragMode: Byte;` **Read/write**

Determines how the view should behave when mouse-dragged.

The *DragMode* bits are defined as follows:

Figure 19.12 DragMode bit mapping



The *DragMode* masks are defined in this chapter under "*dmXXXX* DragMode constants."

See also: *TView.DragView*

EventMask

`EventMask: Word;` **Read/write**

EventMask is a bit mask that determines which event classes will be recognized by the view. The default *EventMask* enables *evMouseDown*, *evKeyDown*, and *evCommand*. Assigning $FFFF to *EventMask* causes the view to react to all event classes; conversely, a value of zero causes the view to not react to any events. For detailed descriptions of event classes, see "*evXXXX* event constants" in this chapter.

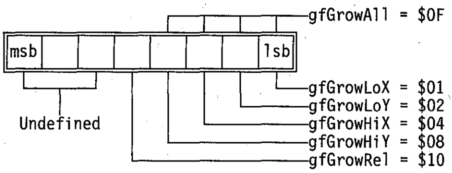
See also: *HandleEvent* methods

GrowMode

`GrowMode: Byte;` **Read/write**

Determines how the view will grow when its owner view is resized. *GrowMode* is assigned one or more of the following *GrowMode* masks:

Figure 19.13 GrowMode bit mapping



Example: `GrowMode := gfGrowLoX or gfGrowLoY;`

See also: *gfXXXX* grow mode constants

HelpCtx

`HelpCtx: Word;` **Read/write**

The help context of the view. When the view has the focus, this field will represent the help context of the application unless the context number is *hcNoContext*, in which case the context defaults to the help context of its owner.

See also: *TView.GetHelpCtx*.

Next

`Next: PView;` **Read only**

Pointer to the owner view's next subview in Z-order. If this is the last subview, *Next* points to *Owner*'s first subview.

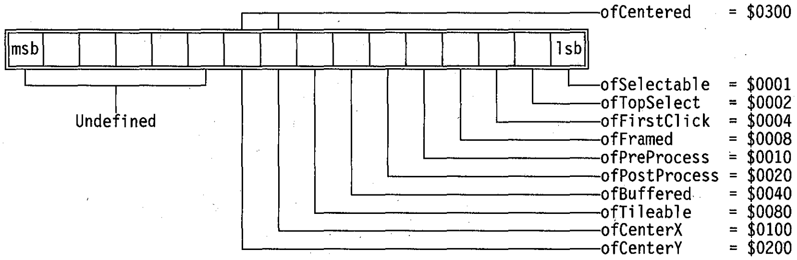
Options

`Options: Word;` **Read/write**

The *Options* word flags determine various behaviors of the view.

The *Options* bits are defined as follows:

Figure 19.14 Options bit flags



For detailed descriptions of the option flags, see "*ofXXXX* option flag constants" in this chapter.

Origin

`Origin: TPoint;` **Read only**

The (X, Y) coordinates, relative to the owner's *Origin*, of the top left corner of the view.

See also: *MoveTo, Locate*

Owner

```
Owner: PGroup;                                        Read only
```

Owner points to the group object that owns this view. If **nil**, the view has no owner. The view is displayed within its owner and will be clipped by the owner's bounding rectangle.

Size

```
Size: TPoint;                                         Read only
```

The size of the view.

See also: *GrowTo, Locate*

State

```
State: Word;                                          Read only
```

The state of the view is represented by bits in the *State* field. Many *TView* methods test and/or alter the *State* field by calling *SetState*. *GetState(AState)* returns *True* if the view's *State* is *AState*. The *State* bits are represented mnemonically by *sfXXXX* constants, described in this chapter under "*sfXXXX* state flag constants."

Methods

Init

```
constructor Init(var Bounds: TRect);
```

Override: Often

Constructs a view object with the given *Bounds* rectangle. *Init* calls the *Init* constructor inherited from *TObject*, then sets the fields of the new *TView* to the following values:

Owner	**nil**
Next	**nil**
Origin	(Bounds.A.X, Bounds.A.Y)
Size	(Bounds.B.X – Bounds.A.X, Bounds.B.Y – Bounds.A.Y)
Cursor	(0, 0)
GrowMode	0
DragMode	*dmLimitLoY*
HelpCtx	*hcNoContext*
State	*sfVisible*
Options	0
EventMask	*evMouseDown* + *evKeyDown* + *evCommand*

☞ *TObject.Init* will zero all fields in *TView* descendants. Always call the inherited *Init* constructor *before* initializing any fields.

See also: *TObject.Init*

Load

```
constructor Load(var S: TStream);
```

Override: Often

Constructs a view object and loads it from the stream *S*. The size of the data read from the stream must correspond exactly to the size of the data written to the stream by the view's *Store* method. If the view contains pointers to peer views, *Load* should use *GetPeerViewPtr* to read these pointers. An overridden *Load* constructor should always call its inherited *Load* constructor.

The default *TView.Load* sets the *Owner* and *Next* fields to **nil**, and reads the remaining fields from the stream. *Owner* and *Next* are set by the view's owner after all subviews have loaded.

See also: *TView.Store, TStream.Get, TStream.Put*

Done

```
destructor Done; virtual;
```

Override: Often

Hides the view and then, if it has an owner, deletes it from the group.

Awaken

```
procedure Awaken; virtual;
```

The default *TView.Awaken* does nothing. When a group is loaded from a stream, the last thing the *Load* constructor does is call the *Awaken* methods of all subviews, giving those views a chance to initialize themselves once all subviews have loaded. This guarantees that all pointers read with *GetPeerViewPtr* are valid.

If you create objects that depend on other subviews to initialize themselves after being read from a stream, you can override *Awaken* to perform that initialization.

See also: *TView.GetPeerViewPtr*

BlockCursor

```
procedure BlockCursor;
```

Override: Never

Sets *sfCursorIns* to change the cursor to a solid block. The cursor will be visible only if *sfCursorVis* is also set (and the view is visible).

See also: *sfCursorIns, sfCursorVis, TView.NormalCursor, TView.ShowCursor, TView.HideCursor*

CalcBounds

```
procedure CalcBounds(var Bounds: TRect; Delta: TPoint); virtual;
```

Override: Seldom

When a view's owner changes size, the owner calls *CalcBounds* and *ChangeBounds* for all its subviews. *CalcBounds* must calculate the new bounds of the view given that its owner's size has changed by *Delta*, and return the new bounds in *Bounds*.

TView.CalcBounds calculates the new bounds using the flags specified in *GrowMode*.

See also: *TView.GetBounds, TView.ChangeBounds, gfXXXX* grow mode constants

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Override: Seldom

ChangeBounds must change the view's bounds (*Origin* and *Size* fields) to the rectangle given by the *Bounds* parameter. Having changed the bounds, *ChangeBounds* must then redraw the view. *ChangeBounds* is called by various *TView* methods but should never be called directly.

TView.ChangeBounds first calls *SetBounds(Bounds)* and then calls *DrawView*.

See also: *TView.Locate, TView.MoveTo, TView.GrowTo*

ClearEvent

```
procedure ClearEvent(var Event: TEvent);
```

Standard method used in *HandleEvent* to signal that the view has successfully handled the event. Sets *Event.What* to *evNothing* and *Event.InfoPtr* to *@Self*.

See also: *HandleEvent* methods

CommandEnabled

```
function CommandEnabled(Command: Word): Boolean;
```

Returns *True* if *Command* is currently enabled; otherwise, it returns *False*. Note that when you change a modal state, you can disable and enable commands as needed; when you return to the previous modal state, however, the original command set will be restored.

See also: *TView.DisableCommand, TView.EnableCommand, TView.SetCommands*.

DataSize

```
function DataSize: Word; virtual;
```

Override: Seldom

DataSize must return the size of the data read from and written to data records by *SetData* and *GetData*. The data record mechanism is typically used only in views that implement controls for dialog boxes.

TView.DataSize returns 0 to indicate that no data is transferred.

See also: *TView.GetData, TView.SetData*

DisableCommands

```
procedure DisableCommands(Commands: TCommandSet);
```

Disables the commands specified in the *Commands* parameter.

See also: *TView.CommandEnabled, TView.EnableCommands, TView.SetCommands*.

DragView

```
procedure DragView(Event: TEvent; Mode: Byte; var Limits: TRect;
  MinSize, MaxSize: TPoint);
```

Drags the view using the dragging mode given by *dmXXXX* flags in *Mode*. *Limits* specifies the rectangle (in the owner's coordinate system) within which the view can be moved, and *Min* and *Max* specifies the minimum and maximum sizes the view can shrink or grow to. The event leading to the dragging operation is needed in *Event* to distinguish mouse dragging from use of the cursor keys.

See also: *TView.DragMode*, *dmXXXX* drag mode constants

Draw `procedure Draw; virtual;`

Override: Always

Called whenever the view must draw (display) itself. *Draw* must cover the entire area of the view. This method must be overridden appropriately for each descendant.

In general, you shouldn't call *Draw* directly, since it is more efficient to use *DrawView*, which draws only views that are exposed, that is, if any part of the view is visible on the screen. If required, *Draw* can call *GetClipRect* to obtain the rectangle that needs redrawing, and then only draw that area. For complicated views, this can improve performance noticeably.

See also: *TView.DrawView*

DrawView `procedure DrawView;`

Calls *Draw* if *TView.Exposed* returns *True*, indicating that the view is exposed (see *sfExposed*). You should call *DrawView* rather than *Draw* whenever you need to redraw a view after making a change that affects its visual appearance.

See also: *TView.Draw*, *TGroup.ReDraw*, *TView.Exposed*

EnableCommands `procedure EnableCommands(Commands: TCommandSet);`

Enables all the commands in the *Commands* parameter.

See also: *TView.DisableCommands*, *TView.GetCommands*, *TView.CommandEnabled*, *TView.SetCommands*.

EndModal `procedure EndModal(Command: Word); virtual;`

Override: Never

Terminates the current modal state and returns *Command* as the result of the *ExecView* function call that created the modal state.

See also: *TGroup.ExecView*, *TGroup.Execute*, *TGroup.EndModal*

EventAvail `function EventAvail: Boolean;`

Returns *True* if an event is available for *GetEvent*.

See also: *TView.MouseEvent*, *TView.KeyEvent*, *TView.GetEvent*

Execute `function Execute: Word; virtual;`

Override: Seldom

Execute is called from *TGroup.ExecView* whenever a view becomes modal. If a view is to allow modal execution, it must override *Execute* to provide an event loop. The result of *Execute* becomes the value returned from *ExecView*.

TView.ExecView simply returns *cmCancel*.

See also: *sfModal, TGroup.Execute, TGroup.ExecView.*

Exposed

```
function Exposed: Boolean;
```

Returns *True* if any part of the view is visible on the screen.

See also: *sfExposed, TView.DrawView*

Focus

```
function Focus: Boolean;
```

Selects and focuses the view, returning *True* if the view's owner returns *True* from *Focus*, and if the view is neither selected nor modal, or if the view has no owner. Otherwise, returns *False*.

The difference between *Focus* and *Select* is that *Focus* can fail. That is, another view might not give up the focus, usually because it holds invalid data that must be corrected before giving up the focus.

See also: *TView.Select*

GetBounds

```
procedure GetBounds(var Bounds: TRect);
```

Returns, in *Bounds*, the bounding rectangle of the view in its owners coordinate system. *Bounds.A* is set to *Origin*, and *Bounds.B* is set to the sum of *Origin* and *Size*.

See also: *TView.Origin, TView.Size, TView.CalcBounds, TView.ChangeBounds, TView.SetBounds, TView.GetExtent*

GetClipRect

```
procedure GetClipRect(var Clip: TRect);
```

Returns, in *Clip*, the minimum rectangle that needs redrawing during a call to *Draw*. For complicated views, *Draw* can use *GetClipRect* to improve performance noticeably.

See also: *TView.Draw*

GetColor

```
function GetColor(Color: Word): Word;
```

Maps the palette indexes in the low and high bytes of *Color* into physical character attributes by tracing through the palette of the view and the palettes of all its owners.

See also: *TView.GetPalette.*

GetCommands

```
procedure GetCommands(var Commands: TCommandSet);
```

Sets *Commands* to the current command set.

See also: *TView.CommandsEnabled, TView.EnableCommands, TView.DisableCommands, TView.SetCommands.*

GetData

```
procedure GetData(var Rec); virtual;
```

Override: Seldom

GetData must copy *DataSize* bytes from the view to the data record given by *Rec*. The data record mechanism is typically used only in views that implement controls for dialog boxes.

The default *TView.GetData* does nothing.

See also: *TView.DataSize, TView.SetData*

GetEvent

```
procedure GetEvent(var Event: TEvent); virtual;
```

Override: Seldom

Sets *Event* to the next available event. Returns *evNothing* if no event is available. By default, it calls the view's owner's *GetEvent*.

See also: *TView.EventAvail, TProgram.Idle, TView.HandleEvent, TView.PutEvent*

GetExtent

```
procedure GetExtent(var Extent: TRect);
```

Sets *Extent* to the extent rectangle of the view. *Extent.A* is set to (0, 0), and *Extent.B* is set to *Size.*

See also: *TView.Origin, TView.Size, TView.CalcBounds, TView.ChangeBounds, TView.SetBounds, TView.GetBounds*

GetHelpCtx

```
function GetHelpCtx: Word; virtual;
```

Override: Seldom

GetHelpCtx must return the view's help context.

The default *TView.GetHelpCtx* returns the value in the *HelpCtx* field, or returns *hcDragging* if the view is being dragged (see *sfDragging*).

See also: *HelpCtx*

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Always

GetPalette must return a pointer to the view's palette, or **nil** if the view has no palette. *GetPalette* is called by *GetColor, WriteChar,* and *WriteStr* when converting palette indexes to physical character attributes. A return value of **nil** causes no color translation to be performed by this view. *GetPalette* is almost always overridden in descendant object types.

The default *TView.GetPalette* returns **nil**.

See also: *TView.GetColor, TView.WriteXXX*

GetPeerViewPtr

```
procedure GetPeerViewPtr(var S: TStream; var P);
```

Loads a peer view pointer *P* from the stream *S*. A *peer view* is a view with the same owner as this view—a *TScroller*, for example, contains two peer view pointers, *HScrollBar* and *VScrollBar*, that point to the scroll bars associated with the scroller. *GetPeerViewPtr* should only be used inside a *Load* constructor to read pointer values that were written by a call to *PutPeerViewPtr* from a *Store* method.

☞ The value loaded into *P* does not become valid until the view's owner completes it's *Load* operation; therefore, dereferencing a peer view pointer within a *Load* constructor does not produce the correct value. Peer view pointers can be dereferenced in *Awaken* methods, which are called by the group *Load* after all subviews exist.

See also: *TView.PutPeerViewPtr, TGroup.Load, TGroup.Store, TView.Awaken*

GetState

```
function GetState(AState: Word): Boolean;
```

Returns *True* if the bit(s) given in *AState* is (are) set in the field *State*.

See also: *State, TView.SetState*

GrowTo

```
procedure GrowTo(X, Y: Integer);
```

Grows or shrinks the view to the given size by calling *Locate*.

See also: *TView.Origin, TView.Size, TView.Locate, TView.MoveTo*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Always

HandleEvent is the central method through which all Turbo Vision event handling is implemented. The *What* field of the *Event* parameter contains the event class (*evXXXX*), and the remaining *Event* fields further describe the event. To indicate that it has handled an event, *HandleEvent* should call *ClearEvent*. *HandleEvent* is almost always overridden in descendant object types.

HandleEvent handles *evMouseDown* events as follows: If the view is not selected (*sfSelected*) and not disabled (*sfDisabled*) and if the view is selectable (*ofSelectable*), then the view selects itself by calling *Select*. No other events are handled by *TView.HandleEvent*.

See also: *TView.ClearEvent*

Hide

```
procedure Hide;
```

Hides the view by calling *SetState* to clear the *sfVisible* flag in *State*.

See also: *sfVisible, TView.SetState, TView.Show*

HideCursor

```
procedure HideCursor;
```

Hides the cursor by clearing the *sfCursorVis* bit in *State*.

See also: *sfCursorVis, TView.ShowCursor*

KeyEvent `procedure KeyEvent(var Event: TEvent);`

Returns, in *Event*, the next *evKeyDown* event. It waits, ignoring all other events, until a keyboard event becomes available.

See also: *TView.GetEvent, TView.EventAvail*

Locate `procedure Locate(var Bounds: TRect);`

Changes the bounds of the view to *Bounds* and redraws the view in its new location. *Locate* calls *SizeLimits* to verify that *Bounds* is valid, and then calls *ChangeBounds* to change the bounds and redraw the view.

See also: *TView.GrowTo, TView.MoveTo, TView.ChangeBounds*

MakeFirst `procedure MakeFirst;`

Moves the view to the top of its owner's subview list. A call to *MakeFirst* corresponds to *PutInFrontOf(Owner^.First)*.

See also: *TView.PutInFrontOf*

MakeGlobal `procedure MakeGlobal(Source: TPoint; var Dest: TPoint);`

Converts the *Source* point coordinates from local (view) to global (screen) and returns the result in *Dest*. *Source* and *Dest* can be the same variable.

See also: *TView.MakeLocal*

MakeLocal `procedure MakeLocal(Source: TPoint; var Dest: TPoint);`

Converts the *Source* point coordinates from global (screen) to local (view) and returns the result in *Dest*. Useful for converting the *Event.Where* field of an *evMouse* event from global coordinates to local coordinates, for example *MakeLocal(Event.Where, MouseLoc)*.

See also: *TView.MakeGlobal, TView.MouseInView*

MouseEvent `function MouseEvent(var Event: TEvent; Mask: Word): Boolean;`

Returns the next mouse event in the *Event* argument. Returns *True* if the returned event is in the *Mask* argument, and *False* if an *evMouseUp* event occurs. This method lets you track a mouse while its button is down, such as in block-marking operations for text editors.

Here's an extract of a *HandleEvent* routine that tracks the mouse with the view's cursor:

```
procedure TMyView.HandleEvent(var Event: TEvent);
begin
  TView.HandleEvent(Event);
  case Event.What of
    evMouseDown:
      begin
        repeat
          MakeLocal(Event.Where, Mouse);
          SetCursor(Mouse.X, Mouse.Y);
        until not MouseEvent(Event, evMouseMove);
       ClearEvent(Event);
      end;
       :
  end;
end;
```

See also: *TView.EventMask, TView.KeyEvent, TView.GetEvent.*

MouseInView

`function MouseInView(Mouse: TPoint): Boolean;`

Returns true if the *Mouse* argument (given in *global* coordinates) is within the calling view.

See also: *TView.MakeLocal*

MoveTo

`procedure MoveTo(X, Y: Integer);`

Moves the *Origin* to the point (X,Y) relative to the owner's view without changing the view's *Size.*

See also: *Origin, Size, TView.Locate, TView.GrowTo*

NextView

`function NextView: PView;`

Returns a pointer to the next subview in the owner's subview list. Returns **nil** if the view is the last subview in its owner's list.

See also: *TView.PrevView, TView.Prev, TView.Next*

NormalCursor

`procedure NormalCursor;`

Clears the *sfCursorIns* bit in *State,* thereby making the cursor into an underline. If *sfCursorVis* is set, the new cursor will be displayed.

See also: *sfCursorIns, sfCursorVis, TView.HideCursor, TView.BlockCursor, TView.HideCursor*

Prev

`function Prev: PView;`

Returns a pointer to the previous subview in the owner's subview list. If the calling view is the first one in its owner's list, *Prev* returns the last view in the list. Note that *Prev* treats the list as circular, whereas *PrevView* treats the list linearly.

See also: *TView.NextView, TView.PrevView, TView.Next*

PrevView

```
function PrevView: PView;
```

Returns a pointer to the previous subview in the owner's subview list, or **nil** if the view is the first subview in its owner's list. Note that *Prev* treats the list as circular, whereas *PrevView* treats the list linearly.

See also: *TView.NextView, TView.Prev*

PutEvent

```
procedure PutEvent(var Event: TEvent); virtual;
```

Override: Seldom

Puts *Event* into the event queue, causing it to be the next event returned by *GetEvent*. Only one event can be pushed onto the event queue in this fashion. Often used by views to generate command events, for example:

```
Event.What := evCommand;
Event.Command := cmSaveAll;
Event.InfoPtr := nil;
PutEvent(Event);
```

The default *TView.PutEvent* calls the view's owner's *PutEvent*.

See also: *TView.EventAvail, TView.GetEvent, TView.HandleEvent*

PutInFrontOf

```
procedure PutInFrontOf(Target: PView);
```

Move the calling view in front of *Target* in the owner's subview list. The call

```
TView.PutInFrontOf(Owner^.First);
```

is equivalent to *TView.MakeFirst*. This method works by changing pointers in the subview list. Depending on the position of the other views and their visibility states, *PutInFrontOf* may obscure (clip) underlying views. If the view is selectable (see *ofSelectable*) and is put in front of all other subviews, the view becomes selected.

See also: *TView.MakeFirst*

PutPeerViewPtr

```
procedure PutPeerViewPtr(var S: TStream; P: PView);
```

Stores a peer view pointer *P* on the stream *S*. A peer view is a view with the same owner as this view. *PutPeerViewPtr* should only be used inside a *Store* method to write pointer values that can later be read by a call to *GetPeerViewPtr* from a *Load* constructor.

See also: *TView.PutPeerViewPtr, TGroup.Load, TGroup.Store*

Select

```
procedure Select;
```

Selects the view (see *sfSelected*). If the view's owner is focused then the view also becomes focused (see *sfFocused*). If the view has the *ofTopSelect* flag set in its *Options* field then the view is moved to the top of its owner's subview list (by calling *MakeFirst*).

See also: *sfSelected, sfFocused, ofTopSelect, TView.MakeFirst*

SetBounds

```
procedure SetBounds(var Bounds: TRect);
```

Sets the bounding rectangle of the view to *Bounds*. Sets *Origin* to *Bounds.A*, and *Size* to the difference between *Bounds.B* and *Bounds.A*. *SetBounds* is intended to be called only from within an overridden *ChangeBounds* method—you should never call *SetBounds* directly.

See also: *TView.Origin, TView.Size, TView.CalcBounds, TView.ChangeBounds, TView.GetBounds, TView.GetExtent*

SetCmdState

```
procedure SetCmdState(Commands: TCommandSet; Enable: Boolean);
```

Enables *Commands* if *Enable* is *True* or disables *Commands* if *Enable* is *False*. *SetCmdState* is a shortcut to using *EnableCommands* or *DisableCommands*.

See also: *TView.DisableCommands, TView.EnableCommands*

SetCommands

```
procedure SetCommands(Commands: TCommandSet);
```

Changes the current command set to the given *Commands* argument.

See also: *TView.EnableCommands, TView.DisableCommands*

SetCursor

```
procedure SetCursor(X, Y: Integer);
```

Moves the hardware cursor to the point (X,Y) using view-relative (local) coordinates. (0,0) is the top left corner.

See also: *TView.MakeLocal, TView.HideCursor, TView.ShowCursor*

SetData

```
procedure SetData(var Rec); virtual;
```

Override: Seldom

GetData must copy *DataSize* bytes from the data record given by *Rec* to the view. The data record mechanism is typically used only in views that implement controls for dialog boxes.

The default *TView.SetData* does nothing.

See also: *TView.DataSize, TView.GetData*

SetState

```
procedure SetState(AState: Word; Enable: Boolean); virtual;
```

Override: Sometimes

Sets or clears bits in the *State* field. *AState* specifies the state flags to modify (see *sfXXXX*), and the *Enable* parameter specifies whether to turn the flag off (*False*) or on (*True*). *SetState* then carries out any appropriate action to reflect the new state, such as redrawing views that become exposed when the view is hidden (*sfVisible*), or reprogramming the hardware when the cursor shape is changed (*sfCursorVis* and *sfCursorIns*).

If a view overrides *SetState*, it should *always* call its inherited *SetState* method first, to ensure the specified bits get set or cleared.

SetState is sometimes overridden to trigger additional actions based on state flags. The *TFrame* type, for example, overrides *SetState* to redraw itself whenever a window becomes selected or is dragged:

```
procedure TFrame.SetState(AState: Word; Enable: Boolean);
begin
  inherited SetState(AState, Enable);
  if AState and (sfActive + sfDragging) <> 0 then DrawView;
end;
```

Another common reason to override *SetState* is to enable or disable commands that are handled by a particular view:

```
procedure TMyView.SetState(AState: Word; Enable: Boolean);
const
  MyCommands = [cmCut, cmCopy, cmPaste, cmClear];
begin
  inherited SetState(AState, Enable);
  if AState = sfFocused then
    if Enable then
      EnableCommands(MyCommands) else
      DisableCommands(MyCommands);
end;
```

See also: *TView.GetState*, *TView.State*, *sfXXXX* state flag constants

Show `procedure Show;`

Shows the view by calling *SetState* to set the *sfVisible* flag in *State*.

See also: *TView.SetState*

ShowCursor `procedure ShowCursor;`

Turns on the hardware cursor by setting *sfCursorVis*. Note that the cursor is invisible by default.

See also: *sfCursorVis*, *TView.HideCursor*

SizeLimits `procedure SizeLimits(var Min, Max: TPoint); virtual;`

Override: Sometimes

Sets *Min* and *Max* to the minimum and maximum values that the *Size* field can assume. *Locate* won't allow the view to be larger than these limits.

The default *SizeLimits* returns (0, 0) in *Min* and *Owner^.Size* in *Max*.

See also: *TView.Size*

Store

```
procedure Store(var S: TStream);
```

Override: Often

Writes the view to the stream *S*. The size of the data written to the stream must correspond exactly to the size of the data read from the stream by the view's *Load* constructor. If the view contains peer view pointers, *Store* should use *PutPeerViewPtr* to write these pointers. An overridden *Store* method should always call its parent's *Store* method.

The default *TView.Store* writes all fields but *Owner* and *Next* to the stream.

See also: *TView.Load, TStream.Get, TStream.Put*

TopView

```
function TopView: PView;
```

Returns a pointer to the current modal view.

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

Override: Sometimes

This method is used to check the validity of a view after it has been constructed (using *Init* or *Load*) or when a modal state ends (due to a call to *EndModal*).

A *Command* parameter value of *cmValid* (zero) indicates that the view should check the result of its construction: *Valid(cmValid)* should return *True* if the view was successfully constructed and is now ready to be used, *False* otherwise.

Any other (nonzero) *Command* parameter value indicates that the current modal state (such as a modal dialog box) is about to end with a resulting value of *Command*. In this case, *Valid* should check the validity of the view. The most common validation command is *cmClose*, indicating that the window is about to close.

If the view's *ofValidate* flag is set, *Valid* is called with the command *cmReleaseFocus* before the view loses the input focus. If *Valid* returns *False*, the view will not release the focus.

Valid should alert the user in case the view is invalid, for example, by using the *MessageBox* routine in the *MsgBox* unit to show an error message.

The default *TView.Valid* simply returns *True*.

See also: *TGroup.Valid, TDialog.Valid, TProgram.ValidView*

WriteBuf

```
procedure WriteBuf(X, Y, W, H: Integer; var Buf);
```

Writes the buffer *Buf* to the screen starting at the coordinates (*X*,*Y*), and filling the region of width *W* and height *H*. Should only be used in *Draw* methods. The *Buf* parameter is typically of type *TDrawBuffer*, but it can be any array of words, each word containing a character in the low byte and an attribute in the high byte.

See also: *TView.Draw*, *TDrawBuffer* type

WriteChar

```
procedure WriteChar(X, Y: Integer; Ch: Char; Color: Byte; Count: Integer);
```

Beginning at the point (*X*,*Y*), writes *Count* copies of the character *Ch* in the color determined by the *Color*'th entry in the view's palette. Should only be used in *Draw* methods.

See also: *TView.Draw*

WriteLine

```
procedure TView.WriteLine(X, Y, W, H: Integer; var Buf);
```

Writes the line contained in the buffer *Buf* to the screen, beginning at the point (*X*,*Y*), and within the rectangle defined by the width *W* and the height *H*. If *H* is greater than 1, the line is repeated *H* times. Should only be used in *Draw* methods. The *Buf* parameter is typically of type *TDrawBuffer*, but it can be any array of words, each word containing a character in the low byte and an attribute in the high byte.

See also: *TView.Draw*

WriteStr

```
procedure TView.WriteStr(X, Y: Integer; Str: String; Color: Byte);
```

Writes the string *Str* with the color attributes of the *Color*'th entry in the view's palette, beginning at the point (*X*,*Y*). Should only be used in *Draw* methods.

See also: *TView.Draw*

TVTransfer type   Validate

Declaration

```
TVTransfer = (vtDataSize, vtSetData, vtGetData);
```

Function Validator objects use parameters of type *TVTransfer* in their *Transfer* methods to control data transfer when setting or reading the value of the associated input line.

See also *TValidator.Transfer*

TWildStr type StdDlg

Declaration

```
TWildStr = PathStr;
```

Function *TWildStr* is identical to the *PathDir* type defined in the *Dos* unit. It is used in standard dialog box types to pass wildcard file name templates.

TWindow Views

TObject	TView		TGroup	TWindow
	Cursor	Options	Buffer	Flags
	DragMode	Origin	Current	Frame
	EventMask	Owner	Last	Number
	GrowMode	Size	Phase	Palette
	HelpCtx	State		Title
	Next			ZoomRect

~~Init~~	~~Init~~	HideCursor	~~Init~~	Init
Free	~~Load~~	KeyEvent	~~Load~~	Load
~~Done~~	~~Done~~	Locate	~~Done~~	Done
	~~Awaken~~	MakeFirst	Awaken	Close
	BlockCursor	MakeGlobal	ChangeBounds	GetPalette
	CalcBounds	MakeLocal	DataSize	GetTitle
	~~ChangeBounds~~	MouseEvent	Delete	HandleEvent
	ClearEvent	MouseInView	Draw	InitFrame
	CommandEnabled	MoveTo	EndModal	SetState
	~~DataSize~~	NextView	EventError	SizeLimits
	DisableCommands	NormalCursor	ExecView	StandardScrollBar
	DragView	Prev	Execute	Store
	~~Draw~~	PrevView	First	Zoom
	DrawView	PutEvent	FirstThat	
	EnableCommands	PutInFrontOf	FocusNext	
	~~EndModal~~	PutPeerViewPtr	ForEach	
	EventAvail	Select	GetData	
	~~Execute~~	SetBounds	GetHelpCtx	
	Exposed	SetCommands	GetSubViewPtr	
	Focus	SetCmdState	~~HandleEvent~~	
	GetBounds	SetCursor	Insert	
	GetClipRect	~~SetData~~	InsertBefore	
	GetColor	~~SetState~~	Lock	
	GetCommands	Show	PutSubViewPtr	
	~~GetData~~	ShowCursor	Redraw	
	GetEvent	~~SizeLimits~~	SelectNext	
	GetExtent	~~Store~~	SetData	
	~~GetHelpCtx~~	TopView	~~SetState~~	
	~~GetPalette~~	~~Valid~~	~~Store~~	
	GetPeerViewPtr	WriteBuf	Unlock	
	GetState	WriteChar	Valid	
	GrowTo	WriteLine		
	~~HandleEvent~~	WriteStr		
	Hide			

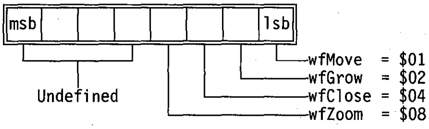
A *TWindow* object is a specialized group that typically owns a *TFrame* object, an interior *TScroller* object, and one or two *TScrollBar* objects. These attached subviews provide the "visibility" to the *TWindow* object. The *TFrame* object provides the familiar border, a place for an optional title and number, and functional icons (close, zoom, drag). *TWindow* objects have the "built-in" capability of moving and growing via mouse drag or cursor keystrokes. They can be zoomed and closed via mouse clicks in the appropriate icon regions. They also "know" how to work with scroll bars and scrollers. Numbered windows from 1–9 can be selected with the *Alt*+n keys (n = 1 to 9).

Fields

Flags

`Flags: Byte;` **Read/write**

The *Flags* field contains combinations of the following bits:



For definitions of the window flags, see "*wfXXXX* window flag constants" in this chapter.

Frame

`Frame: PFrame;` **Read only**

Frame is a pointer to the window's associated *TFrame* object.

See also: *TWindow.InitFrame*

Number

`Number: Integer;` **Read/write**

The number assigned to this window. If *Number* is between 1 and 9, the number appears in the frame title, and the window can be selected with the *Alt+n keys (n = 1 to 9).*

Palette

`Palette: Integer;` **Read/write**

Specifies which palette the window is to use: *wpBlueWindow*, *wpCyanWindow*, or *wpGrayWindow*. The default palette is *wpBlueWindow*.

See also: *TWindow.GetPalette*, *wpXXXX* constants

Title

`Title: PString;` **Read/write**

A character string giving the title that appears on the frame.

ZoomRect

`ZoomRect: TRect;` **Read only**

The normal, unzoomed boundary of the window.

Methods

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr; ANumber: Integer);
```

Constructs a window view with the boundaries passed in *Bounds* by calling the *Init* constructor inherited from *TGroup*. Sets *State* to include *sfShadow*. Sets *Options* to (*ofSelectable* + *ofTopSelect*). Sets *GrowMode* to *gfGrowAll* + *gfGrowRel*. Sets *Flags* to (*wfMove* + *wfGrow* + *wfClose* + *wfZoom*). Sets *Title* to *NewStr(ATitle)*, *Number* field to *ANumber*. Calls *InitFrame*, and if the *Frame* field is non-**nil**, inserts it in this window's group. Finally, sets *ZoomRect* to *Bounds*.

See also: *TFrame.InitFrame*

Load

```
constructor Load(var S: TStream);
```

Constructs a window view and loads it from the stream *S* by first calling the *Load* constructor inherited from *TGroup*, then reading the additional fields introduced by *TWindow*.

See also: *TGroup.Load*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the window and any subviews.

Close

```
procedure Close; virtual;
```

Override: Seldom

Calls the window's *Valid* method with a *Command* value of *cmClose* and if *Valid* returns *True*, closes the window by calling its *Done* method.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Override: Sometimes

Returns a pointer to the palette given by the palette index in the *Palette* field. Table 19.43 shows the palettes returned for the different values of *Palette*.

Table 19.43 Window palettes returned based on Palette

Palette field	Palette returned
wpBlueWindow	*CBlueWindow*
wpCyanWindow	*CCyanWindow*
wpGrayWindow	*CGrayWindow*

See also: *TWindow.Palette*

GetTitle

```
function GetTitle(MaxSize: Integer): TTitleStr; virtual;
```

Override: Seldom

GetTitle should return the window's title string. If the title string is longer than *MaxSize* characters, *GetTitle* should attempt to shorten it; otherwise, it will be truncated by dropping any text beyond the *MaxSize*'th character.

TFrame.Draw calls *Owner^.GetTitle* to obtain the title string to display in the frame.

The default *TWindow.GetTitle* returns the string *Title^*, or an empty string if *Title* is **nil**.

See also: *TWindow.Title, TFrame.Draw*

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Often Handles most events by first calling the *HandleEvent* method inherited from *TGroup*, then handles events specific to windows as follows:

- Command events, if *Flags* permits that operation:
  - *cmResize* (move or resize the window using *DragView*)
  - *cmClose* (close the window using *Close*)
  - *cmZoom* (zoom the window using *Zoom*)
- Keyboard events with a *KeyCode* value of *kbTab* or *kbShiftTab* select the next or previous selectable subview.
- Broadcast events with a *Command* value of *cmSelectWindowNum* select the window if the *Event.InfoInt* field is equal to *Number*.

See also: *TGroup.HandleEvent*, *wfXXXX* constants

InitFrame `procedure InitFrame; virtual;`

Override: Seldom Constructs a frame object for the window and stores a pointer to the frame in the window's *Frame* field. *InitFrame* is called by *Init* but should never be called directly. You can override *InitFrame* to construct a user-defined descendant of *TFrame* instead of the standard frame.

See also: *TWindow.Init*

SetState `procedure SetState(AState: Word; Enable: Boolean); virtual;`

Override: Seldom First calls the *SetState* method inherited from *TGroup*. Then, if *AState* is equal to *sfSelected*, activates or deactivates the window and all its subviews by calling *SetState(sfActive, Enable)*, and calls *EnableCommands* or *DisableCommands* for *cmNext*, *cmPrev*, *cmResize*, *cmClose*, and *cmZoom*.

See also: *TGroup.SetState, EnableCommands, DisableCommands*

SizeLimits `procedure SizeLimits(var Min,Max: TPoint); virtual;`

Override: Seldom First calls the *SizeLimits* method inherited from *TGroup*, then sets *Min* to *MinWinSize*.

See also: *TView.SizeLimits*, *MinWinSize* variable

StandardScrollBar `function StandardScrollBar(AOptions: Word): PScrollBar;`

Constructs, inserts, and returns a pointer to a "standard" scroll bar for the window. "Standard" means the scroll bar fits onto the frame of the window without covering corners or the resize icon.

AOptions can be either *sbHorizontal* to produce a horizontal scroll bar along the bottom of the window or *sbVertical* to produce a vertical scroll bar along the right side of the window. You can combine either with *sbHandleKeyboard* to allow the scroll bar to respond to arrows and page keys from the keyboard in addition to mouse clicks.

See also: *sbXXXX* scroll bar constants.

Store

```
procedure Store(var S: TStream);
```

Writes the window to the stream *S* by first calling the *Store* method inherited from *TGroup*, then writing the additional fields introduced by *TWindow*.

See also: *TGroup.Store*

Zoom

```
procedure TWindow.Zoom; virtual;
```

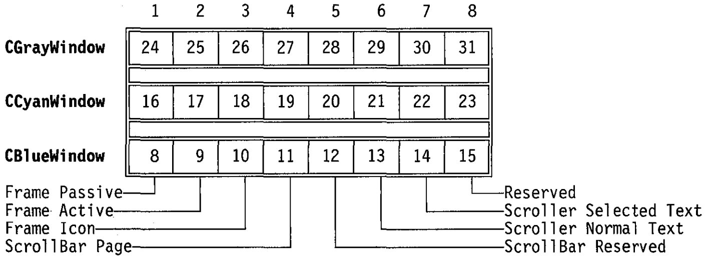
Override: Seldom

Zooms the window. This method is usually called in response to a *cmZoom* command (triggered by a click on the zoom icon). *Zoom* takes into account the relative sizes of the calling window and its owner, and the value of *ZoomRect*.

See also: *cmZoom*, *ZoomRect*

Palette

Window objects use the default palettes *CBlueWindow* (for text windows), *CCyanWindow* (for messages), and *CGrayWindow* (for dialog boxes).



	1	2	3	4	5	6	7	8
CGrayWindow	24	25	26	27	28	29	30	31
CCyanWindow	16	17	18	19	20	21	22	23
CBlueWindow	8	9	10	11	12	13	14	15

TWordArray type

Objects

Declaration

```
TWordArray = array[0..16383] of Word;
```

Function A word array type for general use.

vmtHeaderSize constant Objects

Declaration `vmtHeaderSize = 8;`

Function Used internally by streams, collections, and views as an offset.

voXXXX constants Validate

Function Constants beginning with *vo* represent the bits in the bitmapped *Options* word in validator objects.

Values The validator *Options* bits are defined as follows:

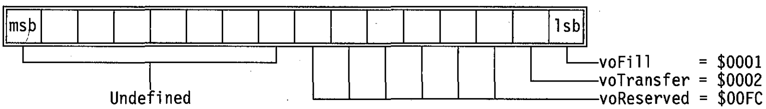


Figure 19.15 Validator option flags

Table 19.44 Validator option flags

Constant	Value	Meaning
voFill	$0001	Used by picture validators to indicate whether to fill in literal characters as the user types.
voTransfer	$0002	The validator handles data transfer for the input line. Currently only used by range validators.
voReserved	$00FC	The bits in this mask are reserved by Borland.

vsXXXX constants Validate

Function Input line objects use *vsOK* to check that their associated validator objects were constructed properly. When called with a command parameter of *cmValid*, an input line object's *Valid* method checks its validator's *Status* field. If *Status* is *vsOK*, the input line's *Valid* returns *True*, indicating that the validator object is ready to use.

The only value defined for *Status* other than *vsOK* is *vsSyntax*, used by *TPXPictureValidator* to indicate that it could not interpret the picture string passed to it. If you create your own validator objects, you can define error codes and pass them in the *Status* field.

Values The *Validate* unit defines two constants used by validator objects to report their status:

Table 19.45 Validator status constants

Constant	Value	Meaning
vsOK	0	Validator constructed properly
vsSyntax	1	Error in the syntax of a picture validator's picture

See also *TValidator.Status*

wfXXXX constants — Views

Function These mnemonics define bits in the *Flags* field of *TWindow* objects. If the bits are set, the window has the corresponding attribute: The window can move, grow, close, or zoom.

Values The window flags are defined as follows:

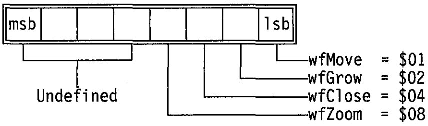


Table 19.46 Window flag constants

Constant	Value	Meaning
wfMove	$01	Window can be moved.
wfGrow	$02	Window can be resized and has a grow icon in the lower-right corner.
wfClose	$04	Window frame has a close icon that can be mouse-clicked to close the window.
wfZoom	$08	Window frame has a zoom icon that can be mouse-clicked to zoom the window.

If a particular bit is set (=1), the corresponding property is enabled; otherwise, if clear (=0), that property is disabled.

See also *TWindows.Flags*

WindowColorItems function — ColorSel

Declaration

```
function WindowColorItems(Palette: Word; const Next: PColorItem): PColorItem;
```

Function Returns a linked list of *TColorItem* records for standard window objects. For programs that allow the user to change window colors with the color selection dialog box, *WindowColorItems* simplifies the process of setting up the color items.

wnNoNumber constant — Views

Declaration

```
wnNoNumber = 0;
```

Function If a window object's *Number* field holds this constant, it indicates that the window is not numbered and cannot be selected via the *Alt*+number key. If the *Number* field is between 1 and 9, the window frame displays the number, and *Alt*+number selection is available.

See also *TWindow.Number*

WordChars variable — Editors

Declaration

```
WordChars: set of Char = ['0'..'9', 'A'..'Z', '_', 'a'..'z'];
```

Function Editor objects use *WordChars* to determine whether a character is part of a word. Such functions as cursor movements and searching by whole words need to know where words start and end.

WordRec type — Objects

Declaration

```
WordRec = record
  Lo, Hi: Byte;
end;
```

Function A utility record allowing access to the *Lo* and *Hi* bytes of a word.

See also *LongRec*

wpXXXX constants — Views

Function These constants define the three standard color mapping assignments for windows. By default, a window object has a *Palette* of *wpBlueWindow*. The default for dialog box objects is *wpGrayWindow*.

Values Turbo Vision defines three standard window palettes:

Table 19.47
Standard window palettes

Constant	Value	Meaning
wpBlueWindow	0	Window text is yellow on blue.
wpCyanWindow	1	Window text is blue on cyan.
wpGrayWindow	2	Window text is black on gray.

See also *TWindow.Palette, TWindow.GetPalette*

I N D E X

TURBO VISION™

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-8400. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom ▪ Part #11MN-TPL01-70 ▪ BOR 4685