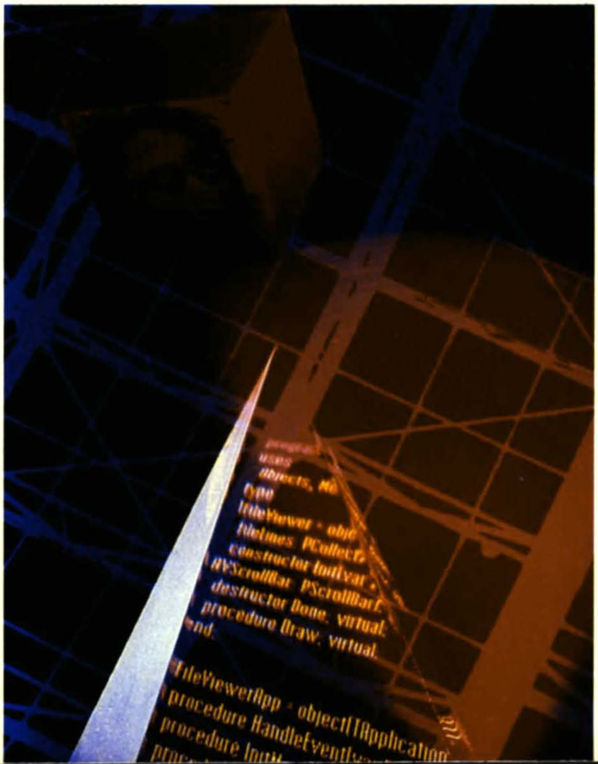
TURBO PASCAL®

6.0

TURBO VISION GUIDE



BORLAND

Turbo Pascal®

Version 6.0

Turbo Vision Guide

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

PRINTED IN THE USA.
R2 10 9 8 7 6 5 4 3 2 1

C O N T E N T S

Chapter 13 Object reference 205

T A B L E S

F I G U R E S

INTRODUCTION

This volume contains complete documentation for Turbo Vision, a whole new way of looking at application development. We describe 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.

Why Turbo Vision?

After creating a number of programs with windows, dialogs, 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.

Does it work? You bet! We used Turbo Vision to write the new integrated development environment for Turbo Pascal in a fraction of the time it would have taken to write it from scratch. Now you can use these same tools to write your own applications.

With Turbo Vision and object-oriented programming, you don't have to reinvent the wheel—you can inherit ours!

If you write character-based applications that need a high-performance, flexible, and consistent interactive user interface, Turbo Vision is for you.

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 library, including:

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

Using Turbo Vision, all your applications can have this state-of-the-art look and feel, with very little effort on your part.

What you need to know

You need to be pretty comfortable with object-oriented programming in order to use Turbo Vision. Applications written in Turbo Vision make extensive use of object-oriented techniques, including inheritance and polymorphism. These topics are covered in Chapter 4, "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. You may want to review the extended syntax of the *New* function, which allows the inclusion of a constructor as a parameter. Most instances of Turbo Vision objects are created that way.

What's in this book?

Because Turbo Vision is new, and because it uses some techniques that might be unfamiliar to many programmers, we have included a lot of explanatory material and a complete reference section.

This manual is divided into three parts:

- Part 1 introduces you to the basic principles behind Turbo Vision and provides a tutorial that walks you through the process of writing Turbo Vision applications.
- Part 2 gives greater detail on all the essential elements of Turbo Vision, including explanations of the members of the Turbo Vision object hierarchy and suggestions for how to write better applications.
- Part 3 is a complete reference lookup for all the objects and other elements included in the Turbo Vision units.

P A R T

1

Learning Turbo Vision

C H A P T E R

1

Inheriting the wheel

How much of your last application was meat, and how much was bones?

The meat of an application is the part that solves the problem the application was written to address: Calculations, database manipulations, and so on. The bones, on the other hand, are the parts that hold the whole thing together: Menus, editable fields, error reporting, mouse handlers, and so on. If your applications are like most, you spend as much or more time writing the bones as you do the meat. And while this sort of program infrastructure can in general be applied to *any* application, out of habit most programmers just keep writing new field editors, menu managers, event handlers, and so on, with only minor differences, for each new project they begin.

You've been warned often enough to avoid reinventing the same old wheel. So here's your chance to stop reinventing the wheel—and start *inheriting* it.

The framework of a windowing application

Turbo Vision is the framework of an event-driven, windowing application. There's no meat as delivered—just a strong, flexible skeleton. You flesh the skeleton out by using the extensibility feature of Turbo Pascal object-oriented programming. Turbo Vision provides you with a skeleton application object, *TApplication*, and you create a descendant object of *TApplication*—call it *MyApplication*, perhaps—to act as your application. Then you add to *MyApplication* what it needs to get your job done.

At the very highest level, that's all there is to it. The entire **begin..end** block of your application program looks like this:

```
begin
  MyApplication.Init;                    { Set the application up,... }
  MyApplication.Run;                                  { ...run it,... }
  MyApplication.Done;     { ...and then put it away when you're done! }
end.
```

A new Vision of application development

You've probably used procedure/function libraries before, and at first glance Turbo Vision sounds a lot like traditional libraries. After all, libraries can be purchased to provide menus, windows, mouse bindings, and so on. But beneath that superficial resemblance is a radical difference, one that is worth understanding to avoid running up against some very high and very hard conceptual walls.

The first thing to do is remind yourself that you're now in object country. In traditional structured programming, when a tool such as a menu manager doesn't quite suit your needs, you modify the tool's source code until it does. Going in and changing the tool's source code is a bold step that is difficult to reverse, unless you somehow take note of *exactly* what the code originally looked like. Furthermore, changing proven source code (especially source code written by somebody else) is a fine way to introduce obnoxious new bugs into a proven subsystem, bugs that could propagate far beyond your area of original concern.

With Turbo Vision, you *never* have to modify the actual 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 disjoint 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.

These two recommendations are the foundation of the Turbo Vision development philosophy: *Use object-oriented techniques fully*, and *embrace the entirety of Turbo Vision on its own terms*. This means playing by Turbo Vision's "rules" and using its component object types as they were intended to be used. 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. To get the most benefit from it, let Turbo Vision do the work.

The elements of a Turbo Vision application

Before we look at how a Turbo Vision application works, let's take a look at "what's in the box"—what tools Turbo Vision gives you to build your applications with.

Naming of parts

A Turbo Vision application is a cooperating society of *views*, *events*, and *mute objects*.

Views

Views are covered in detail in Chapter 4.

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 dialog boxes 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

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 up by

Events are explained in detail in Chapter 5.

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.

Mute objects

Mute objects are any other objects in the program that are *not* views. They are "mute" because they do not speak to the screen themselves. They perform calculations, communicate with peripherals, and generally do the work of the application. When a mute object 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 mute objects from writing to the display with Turbo 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.

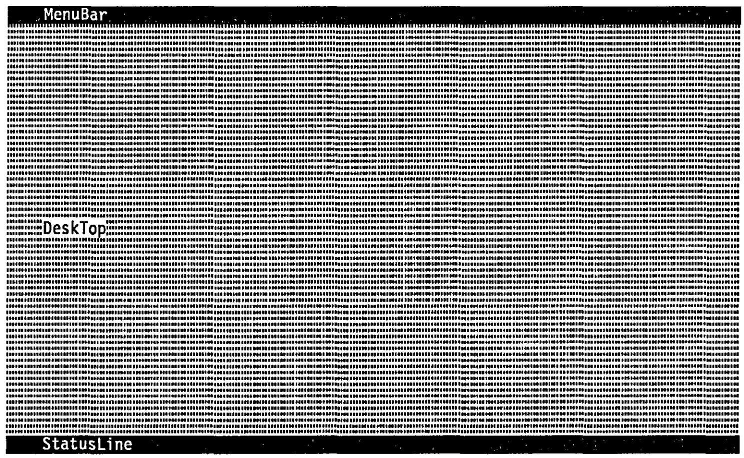
A common "look and feel"

Because Turbo Vision was designed to take 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.

All these items are described in Chapter 4, "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 on the words with the mouse pointer or by pressing *hot keys*.

Figure 1.1
Turbo Vision objects onscreen



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.

Dialog boxes often contain *buttons*, which are highlighted words that can be selected by clicking on them (or by *Tab*bing to the button and pressing *Spacebar*). The displayed words appear to move "downward" in response to the click (as a physical push-button would) and can be set to transmit a command to the application.

"Hello, World!" Turbo Vision style

The traditional way to demonstrate how to use any new language or user interface toolkit is to present a "Hello, world" program written with the tools in question. This program usually consists of only enough code to display the string "Hello, World" on the screen, and to return control to DOS.

Turbo Vision gives us a different way to say "Hello, World!"

The classic "Hello, World" program is *not* interactive (it "talks" but it doesn't "listen") and Turbo Vision is above all *a tool for producing interactive programs.*

The "Hello, World" code is given in the file HELLO.PAS on your distribution disks.

The simplest Turbo Vision application is much more involved than a *Writeln* sandwiched between **begin** and **end.** Compared to the classic "Hello, World" program, Turbo Vision's HELLO.PAS does a fair number of things, including

- clearing the desktop to a halftone pattern
- displaying a menu bar and a status line at the top and bottom of the screen
- establishing a handler for keystrokes and mouse events
- building a menu object "behind the scenes" and connecting it to the menu bar
- building a dialog box, also "behind the scenes"
- connecting the dialog box to the menu
- waiting for *you* to take some action, through the mouse or keyboard

Nowhere in this list is there anything about displaying text to the screen. Some text has been prepared, but it's all in the background, waiting to be called up on command. That's something to keep in mind while you're learning Turbo Vision: The essence of programming with Turbo Vision is designing a custom view and teaching it what to do when it receives commands. Turbo Vision—the framework—worries about getting your view the proper commands. You only have to worry about what to do when the keystroke, mouse click, or menu command finds its way to your view's code.

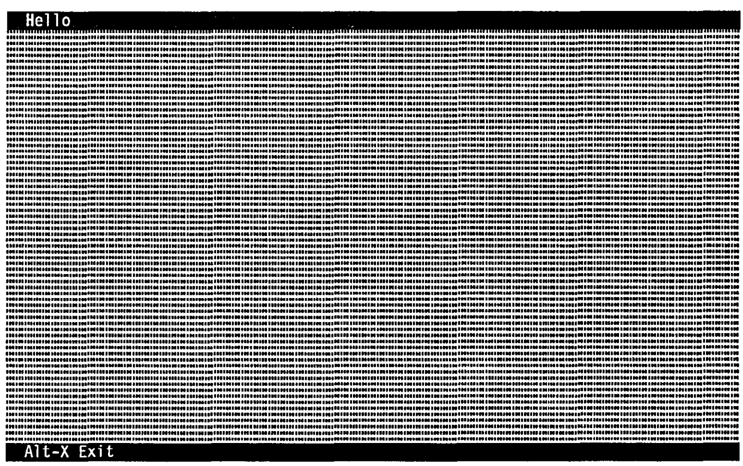
The meat of your program is the code that performs some meaningful work in response to commands entered by the user—and this "meaty" code is contained in the view objects you create.

Running HELLO.PAS

Before we dissect HELLO.PAS in detail, it would be a good idea to load the program, compile it, and follow through its execution.

When run, *Hello* clears the screen, and creates a desktop like that shown in Figure 1.2. No windows are open, and only one item appears in the menu bar at the top of the screen: the command **Hello**. Notice that the "**H**" in **Hello** is set off in a different color from the "ello", and that the status bar contains a message: `Alt-X Exit`.

Figure 1.2
The HELLO.PAS startup screen



This is a good time to point out two general rules for programming in *any* user environment: *Never put the user at a loss as to what to do next,* and *always give the user a way forward and a way back.* Before doing anything at all, the user of *Hello* has two clear choices: Either select the menu item **Hello** or press *Alt-X* to leave the program entirely.

Pulling down a menu

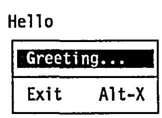
With that in mind, select **H**ello in the menu bar. There are actually three ways to do this:

- Move the mouse pointer over **H**ello and click the left button.
- Press *F10* to take the cursor to the menu bar, where **H**ello becomes highlighted. Then press *Enter* to select **H**ello.
- Press *Alt-H*, where H is the highlighted character in the menu command **H**ello.

In all three cases, a pull-down menu appears beneath the item **H**ello. This should feel familiar to you, as a Turbo Pascal programmer. It's the same way the Turbo Pascal IDE operates. You'll find that Turbo Vision uses all of the conventions of the Turbo Pascal integrated environment. After all, the IDE is a Turbo Vision application!

The menu that appears is shown in Figure 1.3. There are only two items in the menu, separated by a single line into two separate *panes*. *Hello* is so simple that there is only one menu item in each pane, but in fact there may be any number of items in a pane, subject to the limitations of the screen.

Figure 1.3
The HELLO.PAS Hi menu



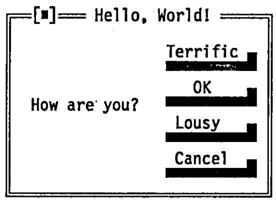
You can select a menu item either from the keyboard or with the mouse. The arrow keys move the *highlight bar* up and down the menu. Selecting a highlighted item from the keyboard is done by pressing *Enter* when the desired item is under the highlight bar. More interesting is selection by mouse: You "grasp" the highlight bar by pressing the left mouse button down while the mouse pointer is on the highlight bar *and holding the button down*. As long as you hold the button down, you can move the bar up and down the list of menu items within the menu. You select one of the menu items by *letting go* of the mouse button when the highlight bar is over the menu item that you wish to select.

A dialog box

An ellipsis (...) after a menu item is used to indicate that the item invokes a dialog box.

However you select it, the **G**reeting item in the menu brings up a rectangular window called a *dialog box,* as shown in Figure 1.4. The dialog box appears in the center of the screen, but you can move it around the screen by moving the mouse pointer to the top line of the dialog box, pressing the left mouse button, and moving the mouse *while you hold the button down*. As soon as you let the button up, the dialog box will stop where it is and remain there.

Figure 1.4
The Hello World! dialog box



The dialog box has a *title,* "Hello, World!", and a *close icon* at its upper left corner. The close icon, when clicked by the mouse, closes the dialog box and make it disappear. Inside the dialog box is a short text string: "How are you?" This is an example of *static text,* which is text that can be read but which contains no interactive power. In other words, static text is used to label things, but nothing happens if you click on it.

Buttons

The four rectangles on the right side of the box are the most interesting parts of the "Hello, World!" dialog box. These are called *buttons,* and are examples of *controls*. They are called controls because they resemble the controls on electronic instruments. Each button has a label, which indicates what happens when that button is pushed.

You push a button by clicking on it with the mouse, or by making the button the *default* (described later in this section) and then pressing *Enter*. Try pressing one of the buttons with the mouse (holding down the mouse button while the pointer is on the button) and see what happens: The body of the button moves one position to the right, and its shadow vanishes. The illusion is that of a rectangular button being pressed "downward" toward the screen. When you release the mouse button, the action specified by the button takes place.

Monochrome systems indicate the default button with "» «" characters.

Notice that the title inside the Cancel button is colored differently than the others. The difference in color indicates that the Cancel button is currently the default control within the dialog box. If you press *Enter* while Cancel is the default, you are in effect pressing the Cancel button.

The default control within a dialog box can be changed by pressing the *Tab* key. Try *Tab*bing around in the "Hello, World!" dialog box. The distinctive default colors move from one button to the next with each press of the *Tab* key. This allows the user to press a button without using a mouse, by moving the default to the chosen button with the *Tab* key, and then pressing *Enter* or *Spacebar* to perform the actual "press of the button."

Getting out

Pressing any of the buttons in *Hello* "puts away" the dialog box and leaves you with an empty desktop. You can pull down the **H**ello menu again, and bring up the dialog box again, any number of times. To exit the program, you can either select the E**x**it item in the **H**ello menu, or simply press the E**x**it shortcut, *Alt-X*. Note that this shortcut is presented both inside the **H**ello menu and in the status line at the bottom of the screen.

☞ This is good practice: *Always make it easy for the user to exit the program.* Frustrated users who can't find the door are quite likely to reboot the system, preventing your application from closing files or otherwise cleaning house before shutting down.

Inside HELLO.PAS

That's what *Hello* does if you run it. Now, how does it make all this happen? Much—in fact, most—of the code comprising *Hello* is inherited from predefined objects provided in Turbo Vision. So much is inherited that when the program runs, *how* it works may first seem a bit of a mystery. Tracing execution with the integrated debugger will not show you the whole picture, since Turbo Vision is provided as compiled units. Fortunately, if you take the time to understand what is going on, the exact *how* won't be necesssary.

To understand a Turbo Vision application, start by reminding yourself that *a Turbo Vision application is a society of objects working together*. Find the major objects and understand how they work together. Then see how the lesser objects support the major objects.

Be sure you read and understand the object definitions before you read the method implementations. It's important that you first understand what an object contains and how it relates to the other objects in the system.

The application object

The cornerstone object of any application is the *TApplication* object. Actually, you never create an instance of object type *TApplication*. *TApplication* is an abstract object type—just bones, no meat. It doesn't *do* anything. You use *TApplication* by creating a descendant object type of *TApplication* that contains the meat of the program you're writing.

In *Hello*, that descendant object type is *THelloApp*:

```
PHelloApp = ^THelloApp;
THelloApp = object(TApplication)
  procedure GreetingBox;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure InitMenuBar; virtual;
  procedure InitStatusLine; virtual;
end;
```

As shown here, it's a good idea to define a pointer type to every object type that you define, since most serious work with objects operates through pointer references. Polymorphism works primarily through pointer references.

THelloApp contains much more than just these four methods, of course; a descendant object inherits *everything* from its ancestor. In defining *THelloApp*, you define how the new object *differs* from its ancestor, *TApplication*. Everything that you do not redefine is inherited unchanged from *TApplication*.

If you think about it, the four method definitions in *THelloApp* pin down the "big picture" of your entire application:

- How the application functions is dictated by what events it responds to, and how it responds to them. You must define a *HandleEvent* method to fulfill this all-important requirement. A *HandleEvent* method is defined in *TApplication* to deal with generic events that occur within any application, but you must provide one that handles events specific to your own application.

- The *InitMenuBar* method sets up the menus behind the menu bar for your application. *TApplication* has a menu bar but no menus; if you want menus (and it would be a poor application indeed without them!) you simply define a method to define the menus. You might wonder why *InitMenuBar*'s code isn't part of *THelloApp*'s constructor. It could be, but a more advanced application might wish to choose among several possible menus for its initial menu display. Best to leave that outside of the constructor, and allow the constructor to set up only those things that are *always* done the same way *every* time the application is run.
- The *InitStatusLine* method sets up the status line text at the bottom of the screen. This text typically displays messages about the current state of the application, shows the available hot keys, or notifies the user of some action to be taken.
- The *GreetingBox* method brings up the dialog box in response to the menu item **G**reeting. *GreetingBox* is called from within the *HandleEvent* method, in response to the event triggered by the selection of the **G**reeting menu item. In more advanced applications, you would have separate methods to respond to each of the menu items defined in the initial menu.

In short, *THelloApp*'s methods provide what all main-program objects must provide: a means to set the application up, an "engine" (the event handler) to respond to events, and methods to embody the responses to particular events. These three things are, by and large, what you must add to *TApplication* when you create descendant object types of *TApplication*.

The dialog box object

The only other major object used in *Hello* is a dialog box object. Because the dialog box doesn't have to do anything special, *Hello* uses an instance of the *TDialog* object. There is no need to derive a special object from *TDialog*.

TDialog itself contains no interactive elements. It is nothing more than a frame (albeit a clever frame); you provide whatever fields or controls are to interact with the user.

THelloApp.GreetingBox builds on *TDialog* by inserting four buttons which are also Turbo Vision views. (Remember that *all* program elements that display *anything* to the screen *must* be Turbo Vision views!) This is typical when using dialog boxes. Usually you just insert the controls you want to have in the dialog box. Everything else that a dialog box must have (including an event handler) is built into *TDialog*.

Flow of execution and debugging

Because Turbo Vision applications are event-driven, the code is structured somewhat differently than conventional programs. Specifically, event-driven programs separate the control structures that read and evaluate user input (and other events) from the procedures and functions that act on that input.

Conventional programs typically contain many blocks of code, each of which involves getting some input, deciding which code gets that input, calling the appropriate routine(s) to process the input, then doing the same thing again. In addition, the code that finishes processing the input must then know where to give control for the next round of input.

Event-driven programs, on the other hand, have a central event-dispatching mechanism, so the bulk of your program does not have to worry about fetching input and deciding what to do with it. Your routines simply wait for the central dispatcher to hand them input to process. This has important implications for debugging your programs: You will probably want to rethink your debugging strategies, setting breakpoints in event-handling routines to check the dispatching of messages, and setting breakpoints in your event-responding code to check that it functions properly.

For more hints and tips on debugging Turbo Vision applications, see Chapter 10, "Hints and tips."

HELLO's main program

At the very highest level, the main program portion of all Turbo Vision applications look pretty much like HELLO:

```
var
  HelloWorld: THelloApp;
begin
  HelloWorld.Init;
  HelloWorld.Run;
  HelloWorld.Done;
end.
```

Each of these three methods deserves some explanation.

The Init method

The first of the three statements (*HelloWorld.Init*) is the necessary constructor call. All objects containing virtual methods must be constructed (through a call to their constructor) before any other method of the object is called. As a convention, all Turbo Vision constructors are named *Init*. This is a very good convention for you to follow in your own code as well.

HelloWorld.Init sets up the main program object for use. It clears the screen, provides initial values for certain important variables, builds the halftone desktop, and lays out the status line and the menu bar. It calls the constructors of a great many other objects, some of which you never see because all these calls happen "offstage."

It's interesting to use the integrated debugger to step over the *HelloWorld.Init* call via *F8*, and then press *Alt-F5* to inspect the display. The desktop, menu bar, and status line will all be laid out and complete, ready for the main program to use. Setting up a main program object via its constructor is pretty straightforward.

The Run method

Nearly all of the mystery in a Turbo Vision application is in the main program's *Run* method. The mystery starts when you look in the definition of *THelloApp* to find the *Run* method definition. It's not there—because *Run* is inherited intact from *THelloApp*'s parent object type, *TApplication*.

Run is where your application will probably spend the bulk of its time. It consists primarily of a **repeat..until** loop, shown here in pseudo-code format:

```
repeat
  Get an event;
  Handle the event;
until Quit;
```

For more detail on how events are handled, refer to Chapter 5.

Again, this is not the exact code, but a conceptual summary of what *Run* does with all the details removed. In essence, a Turbo Vision application loops through two tasks: Getting an event (where an event is essentially "something to do"), and servicing that event. Eventually, one of the events resolves to some sort of quit command, and the loop terminates.

The Done method

The *Done* destructor is really quite simple: It disposes of the objects owned by the application—the menu bar, the status line, and the desktop—and shuts down Turbo Vision's error handler and drivers. In general, your application's *Done* method should undo anything special that the *Init* constructor set up, then call *TApplication.Done*, which handles all the standard elements. If you override *TApplication.Init*, you will probably have to override *TApplication.Done*.

Summary

In this chapter you've had just a taste of what Turbo Vision is all about. You have seen objects interacting in an event-driven framework and gotten some idea of the kinds of tools that Turbo Vision provides.

At this point you may feel confident enough to try modifying the HELLO.PAS program to do some other things. Feel free to do so. One of the nicest features of Turbo Vision is the freedom it gives you to change your programs with very little effort.

The next chapter will take you through the steps of building a Turbo Vision program of your own from the skeleton we provide.

C H A P T E R

2

Writing Turbo Vision applications

Now that you've seen what a Turbo Vision application looks like, inside and out, you're probably itching to write one yourself. In this chapter, you'll do just that, starting with an extremely simple framework and adding small fragments of code at each step so you can see what each of them does.

You probably have a lot of questions at this point. How exactly do views work? What can I do with them? How can I customize them for my applications? If Turbo Vision were a traditional run-time library, most likely you would dig into the source code to get the answers.

But Turbo Vision is already a working application. The best way to answer your questions about Turbo Vision is to actually try out views. As you'll see, you can initialize them with a minimum of code.

Your first Turbo Vision application

A Turbo Vision application always begins by instantiating an object descended from *TApplication*. In the following example, you will create a descendant of *TApplication* called *TMyApp*, and in it, begin to override *TApplication* methods. This new object is then instantiated as *MyApp*.

☞ In the rest of this chapter, we will refer often to *MyApp*. By that we mean your application, an instance of an object descended from

There is normally only one TApplication object in a program.

TApplication. When you write your own Turbo Vision applications, you will probably call them something else, something indicative of the function of each application. We use *MyApp*, because it is shorter than saying "the instance of the object you derived from *TApplication*."

*Several stages of the example code are on your distribution disks. The file names are indicated next to the code examples, and they correspond to the names declared in the **program** statement.*

Beginning with the following code example, you're going to be building an example program. Rather than giving the entire program listing each time, we've only included the added or changed parts in the text. If you follow along and make all the indicated changes, you should get a good feel for what it takes to add each increment of functionality. We also strongly recommend that you try modifying the examples.

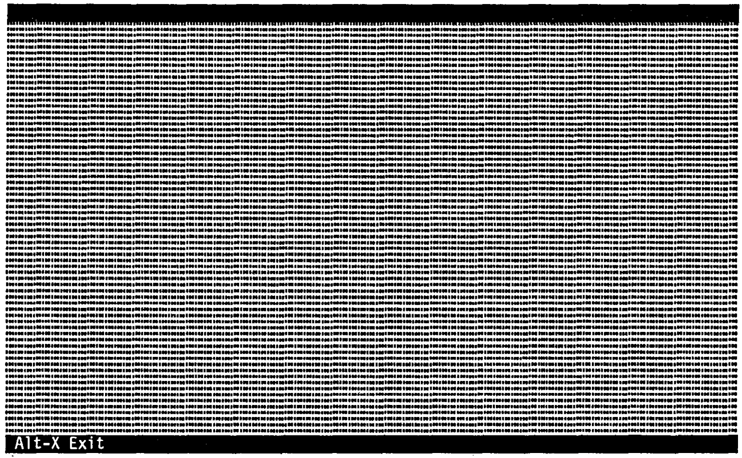
The main block of *TVGUID01* (and of every Turbo Vision application) looks like this:

This program is in TVGUID01.PAS, which is included with the demo programs on your distribution disks.

```
program TFirst;

uses App;                         { application objects are in APP.TPU }

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

var
  MyApp: TMyApp;                 { you need an instance of your new type }

begin
  MyApp.Init;                                            { set it up }
  MyApp.Run;                                 { interact with the user }
  MyApp.Done;                                     { clean up afterward }
end.
```

Note that you haven't added any new functionality to *TMyApp* (yet). Normally, you would never declare a whole new object type with no new fields or methods. You would simply declare the variable *MyApp* as an instance of the *TApplication* type. Since you'll be adding to it later, as you will when writing Turbo Vision applications, you've set up *TMyApp* for flexible expansion. For now, it will behave as a "plain vanilla" *TApplication*. The default behavior of a *TApplication* produces a screen like that in Figure 2.1.

Figure 2.1
Default TApplication screen



This working program does only one thing: It responds to *Alt-X* to terminate the program. To get it to do more, you need to add to the default behavior by adding commands to the status line and/or the menu bar. In the next section, you'll do both.

The desktop, menu bar, and status line

Objects used:
TView
TMenuView
TMenuBar
TMenuBox
TStatusLine
TGroup
TDeskTop

TFirst's desktop, menu bar, and status line are created by the *TApplication* methods *InitDeskTop*, *InitMenuBar*, and *InitStatusLine*. These three methods are called by *TApplication.Init*, so you never need to call them directly. Instead, your application's *Init* method will call *TApplication.Init* in its first line. For example:

```
procedure TMyApp.Init;
begin
  TApplication.Init;                  { call ancestor's method first }
  { initialization code specific to your application goes here }
end;
```

Objects and their units are cross-referenced in Chapter 12.

Note that you'll need to add some Turbo Vision units to the **uses** line in the program. In order to use menus and the status bar and the standard key definitions, you'll need to use *Objects*, *Menus*, and *Drivers* in addition to *App*.

If your program doesn't need to do any special initialization, you simply use the inherited *Init* method. Because the *Init* and *InitDeskTop*, *InitMenuBar*, and *InitStatusLine* methods are virtual, calling the inherited *Init* calls the proper *InitStatusLine* and

InitMenuBar methods. You'll see an example of this in TVGUID02.PAS.

InitDeskTop, *InitMenuBar*, and *InitStatusLine* give values to the global variables *DeskTop*, *MenuBar*, and *StatusLine*, respectively. Let's look at each of these in turn.

The desktop

The desktop is an extremely important object, but it needs little attention from you. You should never need to override the inherited initialization method. Let *TApplication.InitDeskTop* handle it. *DeskTop* is owned by *MyApp*, and whenever *MyApp* instantiates a new view in response to the user clicking on a menu selection, it should attach the new view to *DeskTop*. Beyond this, the desktop knows how to manage views by itself.

The status line

Hot keys are single keystrokes that act like menu or status line items.

TApplication.InitStatusLine instantiates a *TStatusLine* view called *StatusLine* to define and display hot key definitions. *StatusLine* is displayed starting at the left edge of the screen, and any part of the bottom screen line not needed for status line items is free for other views. *StatusLine* binds hot keys to commands, and the items themselves can also be clicked on with the left mouse button.

TVGUID02.PAS creates a working status line by overriding *TApplication.InitStatusLine* like this:

This is TVGUID02.PAS

```
procedure TMyApp.InitStatusLine;
var R: TRect;     { this will hold the boundaries of the status line }
begin
  GetExtent(R);         { set R to the coordinates of the full screen }
  R.A.Y := R.B.Y - 1;              { move top to 1 line above bottom }
  StatusLine := New(PStatusLine, Init(R,         { create status line }
    NewStatusDef(0, $FFFF,              { set range of help contexts }
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,     { define item }
      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,     { another }
      nil)),                                         { no more keys }
    nil)                                             { no more defs }
  ));
end;
```

☞ Don't forget to add `procedure InitStatusLine; virtual;` to the declaration of *TMyApp*.

Turbo Vision commands are constants. Their identifiers start with "cm."

The initialization is a sequence of nested calls to standard Turbo Vision functions *NewStatusDef*, *NewStatusKey*, and *NewStatusBar* (described in detail in Chapter 14). *TVGUID02* defines a status line to be displayed for a range of *help contexts* from 0 through $FFFF and in it binds the standard Turbo Vision command *cmQuit* to the *Alt-X* keystroke, and the standard command *cmClose* to the *Alt-F3* key.

You may note that, unlike *TMyApp.Init*, the *InitStatusLine* method does not call the method it overrides, *TApplication.InitStatusLine*. The reason is simple: Both routines set up status lines that cover the same range of help contexts, and assign them to the same variable. There is nothing in *TApplication.InitStatusLine* that would help *TMyApp.InitStatusLine* do its job more easily, and in fact, you would waste time and memory by calling it.

The last string displayed on the command line by this initialization is 'Alt-F3 Close.' The part of the string enclosed by tildes (~) will be highlighted on the screen. The user will be able to click with the left mouse button anywhere within the string to activate the command.

When you run *TVGUID02*, you'll notice that the *Alt-F3* status item is not highlighted, and clicking on it has no effect. This is because the *cmClose* command is *disabled* by default, and items that generate disabled commands are also disabled. Once you open a window, *cmClose* and the status item will be activated.

Your status line work is over once you've initialized *StatusLine*, because you are using only predefined commands (*cmQuit* and *cmClose*). *StatusLine* can handle the user's input without any further attention from you.

Creating new commands

Note that *cmQuit* and *cmClose*, the commands you bound to the status line items, are standard Turbo Vision commands, so you don't have to define them. In order to use customized commands, you simply declare your commands as constant values. For example, you can define a new command for opening a new window:

Turbo Vision reserves some constants for its own commands. See "Defining commands" in Chapter 5.

```
const
  cmNewWin = 199;
```

Next you can bind that command to a hot key and a status line item:

```
StatusLine := New(PStatusLine, Init(R,
  NewStatusDef(0, $FFFF,
    NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
    NewStatusKey('~F4~ New', kbF4, cmNewWin,      { bind new command }
    NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
    nil))),
  nil)
));
```

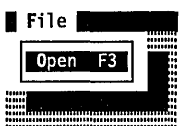
The status line's initialization syntax is a good introduction to menu initialization, which is somewhat more complex.

The menu bar

The Turbo Vision menu bar variable *MenuBar* is initialized with nested calls to the standard Turbo Vision functions *NewMenu*, *NewSubMenu*, *NewItem*, and *NewLine*.

Once you've initialized a menu, your work is finished. The menu bar knows how to handle the user's input without your help.

Initialize a simple menu bar, one menu containing one selection, like this:

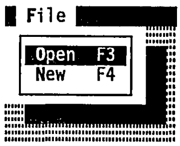



```
const
  cmFileOpen = 200;                          { define a new command }
procedure TMyApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);                         { get area of the application }
  R.B.Y := R.A.Y + 1;                   { set bottom 1 line below top }
  MenuBar := New(PMenuBar, Init(R, NewMenu(  { create bar with menu }
    NewSubMenu('~F~ile', hcNoContext, NewMenu(         { define menu }
      NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,  { item }
      nil)),                                       { no more items }
    nil)                                        { no more submenus }
  )));                                             { end of the bar }
end;
```

The single menu produced by this code is called 'File,' and the single menu selection is called 'Open.' The tildes (~) make *F* the shortcut letter in 'File,' and *O* the shortcut letter in 'Open,' and the *F3* key is bound as a hot key for 'Open.'

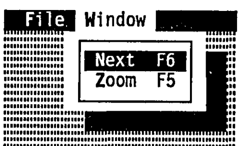
All Turbo Vision views can have a help context number associated with them. The number makes it easy for you to implement context-sensitive help throughout your application. By default, views have a context of *hcNoContext*, which is a special context that doesn't change the current context. Help context numbers appear in the initialization of the menu bar because the nested structure of this object makes it difficult to add numbers later. When you're ready to add help context to the menu bar, you can substitute your own values for *hcNoContext* in the *Init* code.

To add a second item to the 'File' menu, you simply nest another *NewItem* function, like this:




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

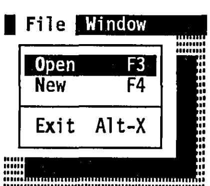
To add a second menu, you nest another *NewSubMenu* function call, like this:




```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
    NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
    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))),
  nil))) {closing parens for menus}
)));
```

You just bound two more standard Turbo Vision commands, *cmNext* and *cmZoom*, to menu items and hot keys.

To add a horizontal line between menu selections, insert a call to *NewLine* between the *NewItem* calls, like this:



This is TVGUID03.PAS

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

You may notice that the version of TVGUID03.PAS supplied on your disk also adds a status key to the status line, binding the *F10* key to the *cmMenu* command. *cmMenu* is a standard Turbo Vision command that helps non-mouse users make use of the menu bar. In this case, the *F10* keystroke causes the menu bar to be activated, allowing menus and menu items to be selected using cursor keys.

You may also notice that the status item has a null string as its text, so nothing appears on the screen for it. Although it might be nice to alert users that *F10* will activate the menus, it is rather pointless to have an item to click on that performs that action. Clicking directly on the menu bar makes much more sense.

A note on structure

At this point, a number of commands are available, but most of them are disabled, and the *cmNewWin* and *cmFileOpen* commands don't yet perform any actions.

If your initial reaction is one of disappointment, it shouldn't be—you've accomplished a lot! In fact, what you've just discovered is one of the big advantages of event-driven programming: You separate the function of *getting* your input from the function of *responding* to that input.

With traditional programming techniques, you would need to go back into the code you've just written and start adding code to open windows and such. But you don't have to do that: You've got a solid engine that knows how to generate commands. All you need to do is write a few routines that respond to those commands. And that's just what you'll do in the next section.

The Turbo Vision application framework takes you one step beyond traditional modular programming. Not only do you break your code up into functional, reusable blocks, but those blocks can be smaller, more independent, and more interchangeable.

Your program now has several different ways to generate a command (*cmNewWin*) to open a window: a status line item, a menu item, and a hot key. In a moment, you'll see how easy it is to tell your application to open a window when that command shows up. The most important thing is that the application doesn't care *how* the command was generated, and neither will the window. All that functionality is independent.

If, later on, you decide you want to change the binding of the command—move the menu selection, remap the hot keys, whatever—you don't have to worry or even *think* about how it will affect your other code. That's what event-driven programming buys you: It separates your user interface design from your program workings, and as you'll see, it also allows different parts of your program to function just as independently.

Opening a window

Objects used:
TRect
TView
TWindow
TGroup
TScroller
TScrollBar

If you're a typical programmer, you may have jumped directly to this section as soon as you opened the book. After all, what's more central to writing a windowed application than making a window?

It's true that if Turbo Vision were a collection of traditional library routines, then jumping right to this section and trying to get right to work might be a good idea. You could very well get a good sense of the library's overall quality and organization.

But Turbo Vision isn't a traditional library. If you've read the preceding chapters, you already know that. In order to program in Turbo Vision, there are some things you need to do before it makes sense to create a window. You need to understand just what a Turbo Vision window is (it's an object!), and how it is different from windows you might have used before. When you've done this, you will be further along in your first application than you'd ever imagine.

So, if you've jumped into the cookbook at this point, you need to go back to the preceding sections and lay a little groundwork. It will be well worth it.

Standard window equipment

A Turbo Vision window is an object, and built into it is the ability to respond to much of the user's input without you having to write a line of code. A Turbo Vision window already knows how to open, resize, move, and close. But you don't write on a Turbo Vision window. A Turbo Vision window is a container that holds and manages other objects: It is these objects that represent themselves on the screen, not the window itself. The window manages the views, and your application's unique functionality is in the views that the window owns and manages. The views you create retain great flexibility about where and how they will appear.

So how do you combine the standard window tools with the things you want to put in the window? Over and over again, remind yourself that you've got a strong framework to build on—and use it! Start with a standard window, then add the features you want. As you go through the next few examples you'll see how easy it is to flesh out the skeleton Turbo Vision provides.

The following code initializes a window and attaches it to the desktop. Remember to add the new methods to the declaration of your *TMyApp* type. Note that again you are defining a new type (*TDemoWindow*) without adding any fields or methods to its ancestor type. As before, you're doing that just to provide a simple platform you can build on easily. You'll add new methods as you go.

This is TVGUID04.PAS

Note that we always declare a pointer type for each new object type.

```
uses Views;

const
  WinCount: Integer = 0;                    { initialize window counter }

type
  PDemoWindow = ^TDemoWindow;
  TDemoWindow = object(TWindow)              { define a new window type }
  end;

procedure TMyApp.NewWindow;
var
  Window: PDemoWindow;
  R: TRect;
begin
  Inc(WinCount);
  R.Assign(0, 0, 26, 7);              { set initial size and position }
  R.Move(Random(53), Random(16));       { randomly move around screen }
  Window := New(PDemoWindow, Init(R, 'Demo Window', WinCount));
  DeskTop^.Insert(Window);                  { put window into desktop }
end;

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);     { basically, act like ancestor }
  if Event.What = evCommand then
  begin
    case Event.Command of         { but respond to additional commands }
      cmNewWin: NewWindow;        { define action for cmNewWin command }
    else
      Exit;
    end;
    ClearEvent(Event);                   { clear event after handling }
  end;
end;
```

To use this window in your program, you first need to bind the command *cmNewWin* to a menu option or status line hot key, as you did earlier. When the user invokes *cmNewWin*, Turbo Vision dispatches the command to *TMyApp.HandleEvent*, which responds by calling *TMyApp.NewWindow*.

Window initialization

You need to give a Turbo Vision window three parameters for it to initialize itself: its size and position on the screen, a title, and a window number.

The TRect object is described in detail in Chapter 4, "Views."

The first parameter, determining the window's size and position, is a *TRect*, Turbo Vision's rectangle object. *TRect* is a very simple object. Its *Assign* method gives it a size and position, based on its top-left corner and its bottom-right corner. There are several other ways to assign or change the values of a *TRect* object. Consult Chapter 14, "Global reference," for complete descriptions.

In *TVGUID04*, *R* is created at the origin of *DeskTop*, then moved a random distance into the desktop. "Normal" programs probably won't do that kind of random movement, but for this exercise you want to be able to open a lot of windows and not have them all be in the same place.

The second initialization parameter is a string, which is displayed as the window's title.

The last initialization parameter is stored in the window's *Number* field. If *Number* is between 1 and 9, it will be displayed on the window frame, and the user can select a numbered window by pressing *Alt-1* through *Alt-9*.

If you don't need to assign a number to a window, just pass it the Turbo Vision constant *wnNoNumber*.

The Insert method

Inserting a window into *DeskTop* automatically makes the window appear. The *Insert* method is used to give a view control over another view. When you execute the instruction

```
DeskTop^.Insert(Window);
```

you are inserting *Window* into the desktop. You may insert any number of views into a *group* object like the desktop. The group you insert a view into is called the *owner* view, and the view you insert into it is called a *subview*. Note that a subview may itself be a group, and may have its own subviews. For instance, when you insert a window into the desktop, the window is a subview, but the window may itself own a frame, scroll bars, or other subviews.

All these relationships among views are explained in Chapter 4.

This process of establishing links between view objects creates a *view tree*, so named because the multiple linkages of views and subviews branch out from the central view, the application, much as limbs branch out from the trunk of a tree.

Closing a window

Clicking the close icon on a window generates the same *cmClose* command you bound to the *Alt-F3* keystroke and a status line item. By default, opening a window (with *F4* or the **File | Open** menu choice) automatically enables the *cmClose* command and the views that generate it (as well as other window-related commands like *cmZoom* and *cmNext*).

You don't have to write any new code to close the window. When the user clicks on the window's close icon, Turbo Vision does the rest. By default, a window responds to the *cmClose* command by calling its *Done* destructor:

```
Dispose(MyWindow, Done);
```

As part of the window's *Done* method, it calls the *Done* methods of all its subviews. If you've allocated any additional memory yourself in the window's constructor, you need to make sure that you deallocate it in the window's *Done* method.

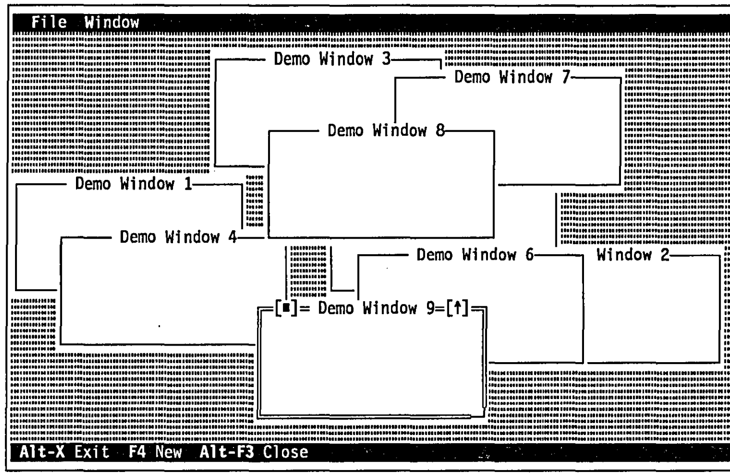
Window behavior

Take some time to play with the program you've written. It has a great deal of capability already. It knows how to open, close, select, move, resize, and zoom multiple windows on the desktop. Not bad for fewer than 100 lines of code!

After *TMyApp* initializes the window, it inserts it into the desktop. As you recall, *DeskTop* is a group, which means that its purpose is to own and manage subviews, like your window. If you compile and run the code, you'll notice that you can resize, move, and close the new window. Your mouse input is being turned into a series of events and routed from the desktop to the new window, which knows how to handle them.

If you keep invoking *cmNewWin*, more windows will appear on the desktop, each with a unique number. These windows can be resized, selected, and moved over one another. Figure 2.2 shows the desktop as it appears with several windows open.

Figure 2.2
TVGUID04 with multiple windows open



A *TWindow* is a group that initially owns one view, a *TFrame*. The user clicks on the frame's icons to move, resize, or close the window. The frame displays the title that it receives during the window's initialization, and it draws the window's background, just as *TBackGround* does for the desktop. All this happens, as you've seen, without you writing any code.

Look through any window

If you were dealing with a traditional window here, the next step would be to write something in it. But a *TWindow* isn't a blank slate to be written on: It's a Turbo Vision group, a *TGroup* object, with no screen representation at all beyond its frame view. To put something "in" a window, you need to take an additional step, a step that puts tremendous power in your hands.

To make something appear in the window, you create a view that knows how to draw itself and insert it into the window. This view is called an *interior*.

This first interior will entirely fill the window, but you'll find it easy later to reduce its size and make room for other views. A window can own multiple interiors, and any number of other useful views—input lines, labels, buttons, or check boxes. You'll also see how easy it is to place scroll bar views on a window's frame.

You can tile or overlap the subviews within a group—how the views interact is up to you. *TDeskTop* has a method, *Tile*, that can tile subviews after they are initialized, but that method is for the desktop alone to use.

The interior you'll create next is a simple descendant of *TView*. Any *TView* can have a frame that operates like a traditional static window frame. A *TView*'s frame, which can't be clicked on, is outside the clipping region of any writing that takes place inside the view. It's just a line around the view.

If your *TView* interior fills its entire owner window, it doesn't matter if it has a frame—the window's frame covers the interior's frame. If the interior is smaller than the window, the interior frame is visible. Multiple interiors within a window can then be delineated by frames, as you'll see in a later example.

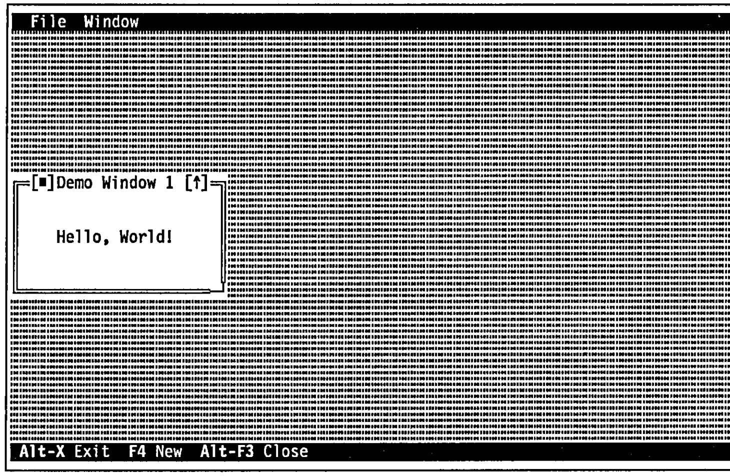
The following code writes "Hello, World!" in the demonstration window, and the results are shown in Figure 2.3.

This makes TVGUID05.PAS

```
PInterior = ^TInterior;
TInterior = object(TView)
  constructor Init(var Bounds: TRect);
  procedure Draw; virtual;
end;

constructor TInterior.Init(var Bounds: TRect);
begin
  TView.Init(Bounds);
  GrowMode := gfGrowHiX + gfGrowHiY;      { make size follow window's }
end;

procedure TInterior.Draw;
begin
  TView.Draw;
  WriteStr(4, 2, 'Hello, World!', 1);
end;

constructor TDemoWindow.Init(Bounds: TRect; WinTitle: TString;
  WindowNo: Integer);
var
  Interior: PInterior;
  S: string[3];
begin
  Str(WindowNo, S);                   { put window number into title }
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  GetClipRect(Bounds);
  Bounds.Grow(-1,-1);         { make interior fit inside window frame }
  Interior := New(PInterior, Init(Bounds));
```

```
    Insert(Interior);                       { add interior to window }
  end;
```

Figure 2.3
TVGUID05 with open window



What do you see?

All Turbo Vision views know how to draw themselves. A view's drawing takes place within the method *Draw*. If you create a descendant view with a new screen representation, you need to override its ancestor's *Draw* method and teach the new object how to represent itself on the screen. *TInterior* is a descendant of *TView*, and it needs a new *Draw* method.

Notice that the new *TInterior.Draw* first calls the *Draw* of its ancestor, *TView*, which in this case just clears the rectangle of the view. Normally you would not do this: Your interior view's *Draw* method should take care of its entire region, making the *TView.Draw* call redundant.

If you really have something to put into a window's interior, you won't want to call the inherited *Draw* method anyway. Calling *TView.Draw* will tend to cause flickering, because parts of the interior are being drawn more than once.

As an exercise, you might try recompiling TVGUID05.PAS with the call to *TView.Draw* commented out. Then move and resize the window. This should make quite clear why a view needs to take responsibility for covering its entire region!

☞ Turbo Vision calls a view's *Draw* method whenever the user opens, closes, moves, or resizes views. If you need to ask a view to redraw itself, call *DrawView* instead of *Draw*. *DrawView* draws the view only if it is exposed. This is important: You override *Draw*, but never call it directly; you call *DrawView*, but you never override it!

A better way to Write

While you can make Turbo Pascal's *Write* procedure work in Turbo Vision, it is the wrong tool for the job. First, if you simply write something, there's no way you can keep a window or other view from eventually coming along and obliterating it. Second, you need to write to the current view's local coordinates, and clip to the view's boundary. Third, there is the question of what color to use when writing.

Turbo Vision's *WriteStr* not only knows how to write with local coordinates and how to be clipped by the view's boundaries, but also how to use the view's color palette. The *WriteStr* procedure takes x- and y-coordinates, the string to be written, and a color index as parameters.

Similar to *WriteStr* is *WriteChar*, defined as

```
WriteChar(X, Y, Ch, Color, Count)
```

Like *WriteStr*, *WriteChar* positions its output at x- and y-coordinates within the view, and writes *Count* copies of the character *Ch* in the color indicated by the *Color*'th entry in the view's palette.

Each of these *Write* methods should only be called from within a view's *Draw* method. That's the only place you need to write anything in Turbo Vision.

A simple file viewer

In this section you'll add some new functionality to your window and put something real in the interior. You'll add methods to read a text file from disk and display it in the interior.

Warning! This program will display some "garbage" characters. Don't worry—we did that on purpose!

This is TVGUID06.PAS.

```
const
  MaxLines = 100;          { This is an arbitrary number of lines }
var
  LineCount: Integer;
  Lines: array[0..MaxLines - 1] of PString;

PInterior = ^TInterior;
```

```
TInterior = object(TView)
  constructor Init(var Bounds: TRect);
  procedure Draw; virtual;
end;

procedure TInterior.Draw;                        { this will look ugly! }
var
  Y: Integer;
begin
  for Y := 0 to Size.Y - 1 do                     { simple line counter }
  begin
    WriteStr(0, Y, Lines[Y]^, $01);                  { write each line }
  end;
end;

procedure ReadFile;
var
  F: Text;
  S: String;
begin
  LineCount := 0;
  Assign(F, FileToRead);
  Reset(F);
  while not Eof(F) and (LineCount < MaxLines) do
  begin
    ReadLn(F, S);
    Lines[LineCount] := NewStr(S);
    Inc(LineCount);
  end;
  Close(F);
end;

procedure DoneFile;
var
  I: Integer;
begin
  for I := 0 to LineCount - 1 do
    if Lines[I] <> nil then DisposeStr(Lines[i]);
end;
```

Reading a text file

Your application needs to call *ReadFile* to load the text file into the array *Lines,* and *DoneFile* after executing to deallocate the space used by *Lines.*

In *ReadFile,* the Turbo Vision global type *PString* is a string pointer. Turbo Vision also supplies a function called *NewStr* that stores a string on the heap and returns a pointer to it. Even though *NewStr* returns a pointer, don't use *Dispose* to get rid of it.

Always use the companion procedure *DisposeStr* to deallocate the string.

Buffered drawing

You will notice that when you run this program, there are "garbage" characters displayed on the screen where there should be empty lines. That's a result of the incomplete *Draw* method. It violates the principle that a view's *Draw* method needs to cover the entire area for which the view is responsible.

Also, the text array *Lines* is not really in the proper form to be displayed in a view. Text typically consists of variable length strings, many of which will be of zero length. Because the *Draw* method needs to cover the entire area of the interior, the text lines need to be padded to the width of the view.

The draw buffer

To take care of this, create a new *Draw* that assembles each line in a buffer before writing it in the window. *TDrawBuffer* is a global type:

MaxViewWidth is 132 characters.

```
TDrawBuffer = array[0..MaxViewWidth-1] of Word;
```

TDrawBuffer holds alternating attribute and character bytes.

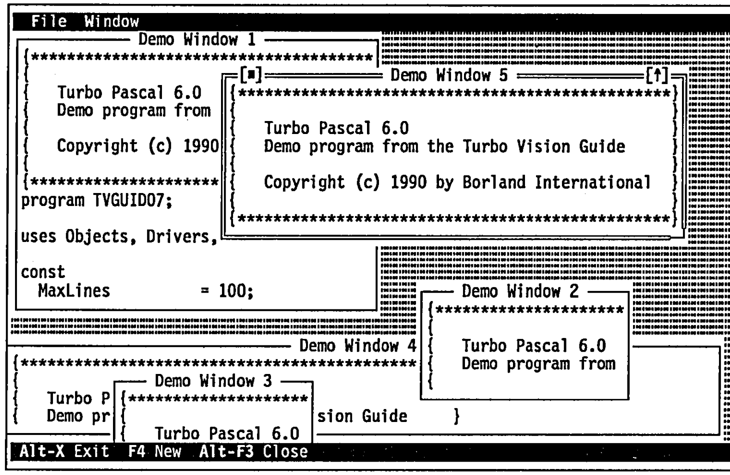
The new *TInterior.Draw* looks like this:

This is TVGUID07.PAS

```
procedure TInterior.Draw;                          { corrected Draw method }
var
  Color: Byte;
  Y: Integer;
  B: TDrawBuffer;
begin
  Color := GetColor(1);
  for Y := 0 to Size.Y - 1 do
  begin
    MoveChar(B, ' ', Color, Size.X);          { fill line with spaces }
    if (Y < LineCount) and (Lines[Y] <> nil) then
      MoveStr(B, Copy(Lines[Y]^, 1, Size.X), Color);  { copy in text }
    WriteLine(0, Y, Size.X, 1, B);                     { write the line }
  end;
end;
```

Figure 2.4 shows *TVGUID07* with several windows open.

Figure 2.4
Multiple file views



Draw first uses a *MoveChar* call to move *Size.X* number of spaces (the width of your interior) of the proper color into a *TDrawBuffer*. Now every line it writes will be padded with spaces to the width of the interior. Next, *Draw* uses *MoveStr* to copy a text line into the *TDrawBuffer*, then displays the entire buffer with a *WriteLine* call.

Moving text into a buffer

Turbo Vision supplies four global procedures for moving text into a *TDrawBuffer*: *MoveStr*, which you just looked at, and *MoveChar*, *MoveCStr*, and *MoveBuf*, which move characters, control strings (strings with tildes for menus and status items), and other buffers, respectively, into a buffer. All these procedures are explained in detail in Chapter 14, "Global reference."

Writing buffer contents

Turbo Vision provides two different procedures for writing the contents of a buffer to a view. One, *WriteLine(X, Y, W, H, Buf)*, was shown in *TVGUID07*.

In *TInterior.Draw*, *WriteLine* writes *TDrawBuffer* on one line. If the fourth parameter, *H* (for height), is greater than 1, *WriteLine* repeats the buffer on subsequent lines. Thus, if *Buf* holds "Hello, World!", `WriteLine(0,0,13,4,Buf)` will write

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

The other procedure, *WriteBuf(X, Y, W, H, Buf)*, will also write a rectangular area of the screen. *W* and *H* refer to the width and height of the buffer. If *Buf* holds "ABCDEFGHIJKLMNOP", *WriteBuf(0,0,4,4,Buf)* will write

```
ABCD
EFGH
IJKL
MNOP
```

Unlike their non-buffered counterparts, *WriteStr* and *WriteChar*, you'll notice that you don't specify the color palette entry to use when writing a draw buffer. This is because colors are specified when the text is moved into the buffer, meaning that text with differing attributes may appear in the same buffer.

Both *WriteLine* and *WriteBuf* are explained in detail in Chapter 14, "Global reference."

Knowing how much to write

Note that *TInterior.Draw* draws just enough of the file to fill the interior. Otherwise, *Draw* would spend much of its time writing parts of the file that would just end up being clipped by the boundaries of *TInterior*.

If a view requires a lot of time to draw itself, you can first call *GetClipRect*. *GetClipRect* returns the rectangle that is exposed within the owner, so you only need to draw the part of the view that is exposed. For example, if you have a complex dialog box with a number of controls in it, and you move it most of the way off the screen so you can look at something behind it, calling *GetClipRect* before drawing would save having to redraw the parts of the dialog box that are temporarily off the screen.

Scrolling up and down

Obviously, a file viewer isn't much use if you can only look at the first few lines of the file. So next you'll change the interior to a scrolling view, and give it scroll bars, so that *TInterior* becomes a scrollable window on the textfile. You'll also change *TDemoWindow*, giving it a *MakeInterior* method to separate that function from the mechanics of opening the window.

This is TVGUID08.PAS

```
type
  PInterior = ^TInterior;
```

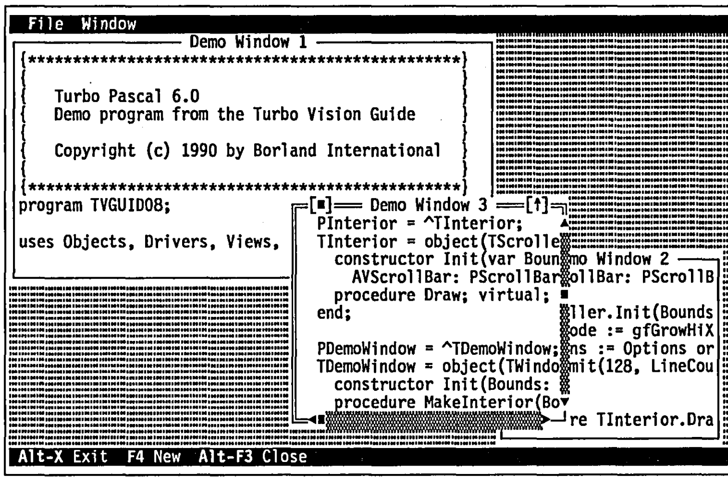
☞

Note that you have changed the ancestor of TInterior!

```
  TInterior = object(TScroller)
    constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar:
      PScrollBar);
    procedure Draw; virtual;
  end;
  PDemoWindow = ^TDemoWindow;
  TDemoWindow = object(TWindow)
    constructor Init(Bounds: TRect; WinTitle: String; WindowNo:
      Word);
    procedure MakeInterior(Bounds: TRect);
  end;

constructor TInterior.Init(var Bounds: TRect; AHScrollBar,
  AVScrollBar: PScrollBar);
begin
  TScroller.Init(Bounds, AHScrollBar, AVScrollBar);
  GrowMode := gfGrowHiX + gfGrowHiY;
  SetLimit(128, LineCount);     { horizontal, vertical scroll limits }
end;

procedure TInterior.Draw;
var
  Color: Byte;
  Y, I: Integer;
  B: TDrawBuffer;
begin
  Color := GetColor($01);                      { use normal text color }
  for Y := 0 to Size.Y - 1 do              { still need to count lines }
  begin
    MoveChar(B, ' ', Color, Size.X);          { fill buffer with spaces }
    I := Delta.Y + Y;                        { Delta is scroller offset }
    if (I < LineCount) and (Lines[I] <> nil) then
      MoveStr(B, Copy(Lines[I]^, Delta.X + 1, Size.X), Color);
    WriteLine(0, Y, Size.X, 1, B);
  end;
end;

procedure TDemoWindow.MakeInterior(Bounds: TRect);
var
  HScrollBar, VScrollBar: PScrollBar;
  Interior: PInterior;
  R: TRect;
begin
  VScrollBar := StandardScrollBar(sbVertical);
  HScrollBar := StandardScrollBar(sbHorizontal);
  Interior := New(PInterior, Init(Bounds, HScrollBar, VScrollBar));
  Insert(Interior);
end;

constructor TDemoWindow.Init(Bounds: TRect; WinTitle: String;
```

```
    WindowNo: Integer);
var
  S: string[3];
begin
  Str(WindowNo, S);
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  GetExtent(Bounds);
  Bounds.Grow(-1,-1);
  MakeInterior(Bounds);
end;
```

Figure 2.5
File viewer with scrolling interior



The horizontal and vertical scroll bars are initialized and inserted in the group, and then are passed to *TScroller* in its initialization.

A scroller is a view designed to display part of a larger virtual view. A scroller and its scroll bars cooperate to produce a scrollable view with remarkably little work by you. All you have to do is provide a *Draw* method for the scroller so it displays the proper part of the virtual view. The scroll bars automatically control the scroller values *Delta.X* (the column to begin displaying) and *Delta.Y* (the first line to begin displaying).

You must override a *TScroller's* *Draw* method in order to make a useful scroller. The *Delta* values will change in response to the scroll bars, but it won't display anything by itself. The *Draw* method will be called whenever *Delta* changes, so that is where you need to put the response to *Delta*.

Multiple views in a window

Next, you duplicate the interior and create a window with two scrolling views of the text file. The mouse or the tab key automatically selects one of the two interior views. Each view scrolls independently and has its own cursor position.

To do this, you add a bit to the *MakeInterior* method so it knows which side of the window the interior is on (since the different sides behave a bit differently), and you make two calls to *MakeInterior* in *TDemoWindow.Init.*

☞

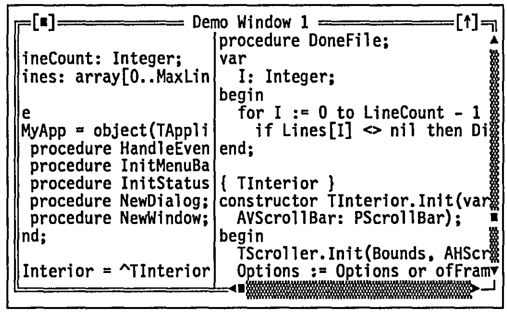
Be sure to change the declaration of MakeInterior!

```
procedure TDemoWindow.MakeInterior(Bounds: TRect; Left: Boolean);
var
  Interior: PInterior;
  R: TRect;
begin
  Interior := New(PInterior, Init(Bounds,
    StandardScrollBar(sbHorizontal),
    StandardScrollBar(sbVertical)));
  if Left then Interior^.GrowMode := gfGrowHiY
  else Interior^.GrowMode := gfGrowHiX + gfGrowHiY;
  Insert(Interior);
end;
```

This is TVGUID09.PAS.

```
constructor TDemoWindow.Init(Bounds: TRect; WinTitle: String;
  WindowNo: Word);
var
  S: string[3];
  R: TRect;
begin
  Str(WindowNo, S);
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  GetExtent(Bounds);
  R.Assign(Bounds.A.X, Bounds.A.Y, Bounds.B.X div 2 + 1, Bounds.B.Y);
  MakeInterior(R, True);
  R.Assign(Bounds.B.X div 2, Bounds.A.Y, Bounds.B.X, Bounds.B.Y);
  MakeInterior(R,False);
end;
```

Figure 2.6
Window with multiple panes



Note that you've changed *MakeInterior* both in style and in substance. Instead of declaring two static scroll bars and then passing them to the *Init* method, you simply included the *StandardScrollBar* calls as parameters to *Init*. The earlier style is somewhat clearer; the latter is a bit more efficient.

If you shrink down the windows in TVGUID09.PAS, you'll notice that the vertical scroll bar gets overwritten by the left interior view if you move the right side of the window too close to the left. To get around this, you can set a limit on how small you're allowed to make the window. You do this by overriding the *TWindow* method *SizeLimits*.

☞

Remember to add SizeLimits to TDemoWindow. It's virtual! This is TVGUID10.PAS.

```
procedure TDemoWindow.SizeLimits(var Min, Max: TPoint);
var R: TRect;
begin
  TWindow.SizeLimits(Min, Max);
  GetExtent(R);
  Min.X := R.B.X div 2;
end;
```

Note that you do not have to call *SizeLimits*. You just override it, and it will be called at the appropriate times. This is the same thing you did with the *Draw* method: You told the view *how* to draw itself, but not *when*. Turbo Vision already knew when to call *Draw*. The same applies to *SizeLimits*: You set the limits, and the view knows the appropriate times to check them.

Where to put the functionality

You've now created a window with a number of views: a frame and two scrolling interiors, each with two scroll bars. You're on your way to creating a window that can carry out specific functions in an application.

How do you proceed? Suppose you want to turn your window into a full-fledged text editor. Since the window has two views, you may be tempted to put some of the text-editing functionality into the group, and then have the group communicate with the two views. After all, a group's job is to manage views. Isn't it natural for it to be involved in all the work?

While a group is as capable of being extended as any view, and you can put any functionality in it that you wish, your Turbo Vision applications will be more robust and flexible if you follow these two pointers: *keep objects as autonomous as possible,* and *keep groups (such as windows) as dumb and devoid of additional functionality as possible.*

Thus, you'd build the text editor by putting all the functionality into the interior view: Create a text editor view type. Views can be easily reusable if you design them properly, and moving your text editor into a different environment wouldn't be very easy if its editing functionality were divided between a group and a view.

Making a dialog box

Objects used:
TView
TGroup
TDialog
TCluster
TCheckBoxes
TRadioButtons
TLabel
TInputLine

A dialog box is just a special kind of window. In fact, *TDialog* is a descendant of *TWindow,* and though you *can* treat it as just another window, you will *usually* do some things differently.

Building on your demonstration program, you'll add a new menu item that generates a command to open a dialog box, add a method to your application that knows how to do that, and add a line to the application's *HandleEvent* method to link the command to the action.

Note that you do not need to derive a new object type from *TDialog* as you did with *TWindow* (to produce *TDemoWindow*). Rather than creating a special dialog box type, you'll add the intelligence to the application: Instead of instantiating a dialog box object that knows what you want it to do, you'll instantiate a generic dialog box and *tell* it what you want it to do.

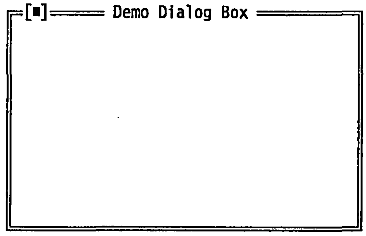
You will rarely find it necessary to create a descendant of *TDialog,* since the only difference between any two dialog boxes is what they contain, not how the dialog boxes themselves work.

This is TVGUID11.PAS

```
const
  cmNewDialog = 200;
```

```
procedure TMyApp.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(
      NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
      NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
      NewLine(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
      nil))))),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
      NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
      NewItem('~D~ialog', 'F2', kbF2, cmNewDialog, hcNoContext,
      nil)))),
    nil))
  )));
end;

procedure TMyApp.NewDialog;
var
  Dialog: PDialog;
  R: TRect;
begin
  R.Assign(0, 0, 40, 13);
  R.Move(Random(39), Random(10));
  Dialog := New(PDialog, Init(R, 'Demo Dialog'));
  DeskTop^.Insert(Dialog);
end;

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewWin: NewWindow;
      cmNewDialog: NewDialog;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
end;
```

Figure 2.7
Simple dialog box



There are really very few differences between this dialog box and your earliest windows, except for the following:

- The default color of the dialog box is gray instead of blue.
- The dialog box is not resizable or zoomable.
- The dialog box has no window number.

Note that you can close the dialog box either by clicking on its close icon, clicking the *Alt-F3* status line item, or pressing the *Esc* key. By default, the *Esc* key cancels the dialog box.

Modal views are discussed in Chapter 4, "Views."

This is an example of what is called a non-modal (or "modeless") dialog box. Dialog boxes are usually *modal*, which means that they define a mode of operation. Usually when you open a dialog box, the dialog box is the only thing active: it is the *modal view*. Clicking on other windows or the menus will have no effect as long as you are in the dialog box's mode. There may be occasions when you want to use non-modal dialog boxes, but in the vast majority of cases, you will want to make your dialog boxes modal.

Executing a modal dialog box

So how do you make your dialog box modal? It's really very easy. Instead of *inserting* the dialog box object into the desktop, you *execute* it, by calling the *DeskTop^.ExecView* function:

This is TVGUID12.PAS

```
procedure TMyApp.NewDialog;
var
  Dialog: PDialog;
  R: TRect;
  Control: Word;
begin
  R.Assign(0, 0, 40, 13);
  R.Move(Random(39), Random(10));
  Dialog := New(PDialog, Init(R, 'Demo Dialog'));
  Control := DeskTop^.ExecView(Dialog);
end;
```

A *TDialog* already knows how to respond to an Esc key event (which it turns into a *cmCancel* command) and an Enter key event (which will be handled by the dialog box's default *TButton*). A dialog box always closes in response to a *cmCancel* command.

Calling *ExecView* both inserts the dialog box into the group and makes the dialog box modal. Execution remains in *ExecView* until the dialog box is closed or canceled. *ExecView* then removes the dialog box from the group and exits. For the moment, you'll ignore the value returned by the *ExecView* function and stored in *Control*. You'll make use of this value in *TVGUID16*.

Taking control

Of course, a dialog box with nothing in it is not much of a dialog box! To make this interesting, you need to add *controls*. Controls are various elements within a dialog box that allow you to manipulate information. The important thing to remember about controls is that they only affect things within the dialog box.

Command handling is explained more in Chapter 5, "Event-driven programming."

The only exception to this rule is the case of a button in a modeless dialog box. Because buttons generate commands, those commands will spread downward from the current modal view. If the dialog box is not the modal view, those commands will go to places outside the dialog box, which may have unintended effects.

In general, when setting up controls in a dialog box, you can separate the visual presentation from the handling of data. This means you can easily design an entire dialog box without having to create the code that sets up or uses the data provided in the dialog box, just as you were able to set up menus and status items without having code that acted on the commands generated.

Button, button...

One of the simplest control objects is the *TButton*. It works very much like a fancy status line item: It's a colored region with a text label on it, and if you click on it, it generates a command. There is also a shadow "behind" the button, so that when you click on the button it gives a sort of three-dimensional movement effect.

Most dialog boxes have at least one or two buttons. The most common are buttons for "OK" (meaning "I'm done. You may close the dialog box and accept the results.") and "Cancel" (meaning "I want to close the dialog box and ignore any changes made in it."). A Cancel button will usually generate the same *cmCancel* command that the close icon produces.

The *Dialogs* unit defines five standard dialog commands that can be bound to a *TButton*: *cmOK, cmCancel, cmYes, cmNo,* and *cmDefault*. The first four commands also close the dialog box by having *TDialog* call its *EndModal* method, which restores the previous modal view to modal status.

You can also use buttons to generate commands specific to your application.

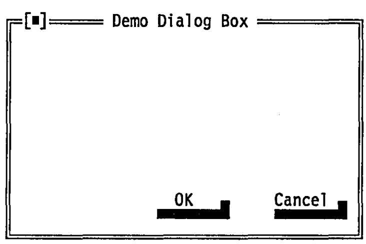
This is TVGUID13.PAS

```
procedure TMyApp.NewDialog;
var
  Dialog: PDialog;
  R: TRect;
  Control: Word;
begin
  R.Assign(20, 6, 60, 19);
  Dialog := New(PDialog, Init(R, 'Demo Dialog'));
  with Dialog^ do
  begin
    R.Assign(15, 10, 25, 12);
    Insert(New(PButton, Init(R, '~O~K', cmOK, bfDefault)));
    R.Assign(28, 10, 38, 12);
    Insert(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));
  end;
  Control := DeskTop^.ExecView(Dialog);
end;
```

Creating a button requires four parameters for the *Init* constructor:

1. the region the button will cover (Remember to leave room for the shadow!)
2. the text that will appear on the button
3. the command to be bound to the button
4. a flag indicating the type of button (normal or default)

Figure 2.8
Dialog box with buttons



Notice that you didn't highlight the "C" in "Cancel" because there is already a hot key (*Esc*) for canceling the dialog box. This leaves *C* available as a shortcut for some other control.

Normal and default buttons

Whenever you create a button, you give it a flag, either *bfNormal* or *bfDefault*. Most buttons will be *bfNormal*. A button flagged with *bfDefault* will be the default button, meaning that it will be "pressed" when you press the *Enter* key. Turbo Vision does not check to ensure that you have only one default button—that is your responsibility. If you designate more than one default control, the results will be unpredictable.

Usually, the "OK" button in a dialog box is the default button, and users become accustomed to pressing *Enter* to close a dialog box and accept changes made in it.

Focused controls

Notice that when a dialog box is open, one of the controls in it is always highlighted. That is the active, or *focused*, control. Focus of controls is most useful for directing keyboard input and for activating controls without a mouse. For example, if a button has the focus, the user can "press" the button by pressing *Spacebar*. Characters can only be typed into an input line if the input line has the focus.

Labels are discussed later in this chapter.

The user can press the *Tab* key to move the focus from control to control within the dialog box. Labels won't accept the focus, so the *Tab* key skips over them.

Tab order is important!

You will want the user to be able to *Tab* around the dialog box in some logical order. The *Tab* order is the order in which the objects were inserted into the dialog box. Internally, the objects owned by the dialog box are maintained in a circular linked list, with the last object inserted linked to the first object.

By default, the focus ends up at the last object inserted. You can move the focus to another control either by using the dialog box's *SelectNext* method or by calling the control's *Select* method directly. *SelectNext* allows you to move either forward or backward through the list of controls. *SelectNext(False)* moves you forward through the circular list (in *Tab* order); *SelectNext(True)* moves you backward.

Take your pick

Often, the choices you want to offer your users in a dialog box are not simple ones that can be handled by individual buttons. Turbo Vision provides several useful standard controls for allowing the user to choose among options. Two of the most useful are *check boxes* and *radio buttons.*

Check boxes and radio buttons function almost identically, with the exception that you can pick as many (or as few) of the check boxes in a set as you want, but you can pick only one (and *exactly* one) radio button. The reason the two sets appear and behave so similarly is that they both derive from a single Turbo Vision object, the *TCluster*.

If you're not familiar with the concept of check boxes and radio buttons, you might look at the **O**ptions menu in the Turbo Pascal integrated environment. Many of the dialog boxes brought up by that menu feature cluster controls.

Creating a cluster

There is probably no reason you would ever want to create an instance of a plain *TCluster*. Since the process for setting up a check box cluster is the same as that for setting up a cluster of radio buttons, you only need to look at the process in detail once.

Add the following code to the *TMyApp.NewDialog* method, *after* the dialog box is created but *before* the buttons are added. Keep the buttons as the last items inserted so they will also be last in *Tab* order.

```
[ ] Hvarti
[ ] Tilset
[ ] Jarlsberg
```

```
var
  B: PView;
R.Assign(3, 3, 18, 6);
B := New(PCheckBoxes, Init(R,
  NewSItem('~H~varti',
  NewSItem('~T~ilset',
  NewSItem('~J~arlsberg',
  nil)))
));
Insert(B);
```

The initialization is quite simple. You designate a rectangle to hold the items (remembering to allow room for the check boxes themselves), and then create a linked list of pointers to strings that will show up next to the check boxes, terminated by a **nil**.

Check box values

The preceding code creates a set of check boxes with three choices. You may have noticed that you gave no indication of the settings for each of the items in the list. By default, they will all be unchecked. But often you will want to set up boxes where some or all of the entries are already checked. Rather than assigning values when you set up the list, Turbo Vision provides a way to set and store values easily, outside the visual portion of the control.

A set of check boxes may have as many as 16 entries. Since you have up to 16 items that may be checked either on or off, you can represent the information as a single 16-bit word, with each bit corresponding to one item to be checked.

After you finish constructing the dialog box as a whole, you will look at how to set and read the values of all the controls. For now, concentrate on getting the proper controls in place.

One more cluster

Before moving on, however, add a set of radio buttons to the dialog box so you can compare them with check boxes. The following code sets up a set of three radio buttons next to your check boxes:

```
(•) Solid
( ) Runny
( ) Melted
```

```
R.Assign( , , , );
B := New(PRadioButtons, Init(R,
  NewSItem('~S~olid',
  NewSItem('~R~unny',
  NewSItem('~M~elted',
  nil)))
));
Insert(B);
```

The main differences you will note between the check boxes and the radio buttons are that you can only select one radio button in the group, and the first item in the list of radio buttons is selected by default.

Since you don't need to know the state of every radio button (only one can be on, so you only need to know *which* one it is), radio button data is not bitmapped. This means you can have more than just 16 radio buttons, if you choose, but since the data is still stored, you are limited to 65,536 radio buttons per cluster. This should not be a serious impediment to your design. A value of zero indicates the first radio button is selected, a one indicates the second button, a two the third, and so on.

Labeling the controls

Of course, setting up controls may not be sufficient. Simply offering a set of choices may not tell the user just *what* he is choosing! Turbo Vision provides a handy method for labeling controls in the form of another control, the *TLabel.*

There's more to the *TLabel* than appears at first glance. A *TLabel* not only displays text, it is also bound to another view. Clicking on a label will move the focus to the bound view. You can also define a shortcut letter for a label by surrounding the letter with tildes (~).

To label your check boxes, add the following code right after you insert the check boxes into the dialog box:

```
R.Assign(2, 2, 10, 3);
Insert(New(PLabel, Init(R, 'Cheeses', B)));
```

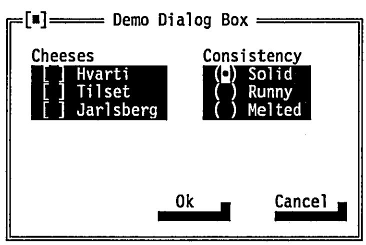
You can now activate the set of check boxes by clicking on the word "Cheeses." This also lets the uninformed know that the items in the box are, in fact, cheeses.

Similarly, you can add a label to your radio buttons with the following code:

This is TVGUID14.PAS

```
R.Assign(21, 2, 33, 3);
Insert(New(PLabel, Init(R, 'Consistency', B)));
```

Figure 2.9
Dialog box with labeled clusters added



The input line object

There is one other fairly simple kind of control that you can add to your dialog box: an item for editing string input, called an *input line.* Actually, the workings of the input line are fairly complex, but from your perspective as a programmer, *TInputLine* is a very simple object to use.

Add the following code after the code for labeling the radio buttons and before you execute the dialog box:

This is TVGUID15.PAS

```
R.Assign(3, 8, 37, 9);
B := New(PInputLine, Init(R, 128));
Insert(B);
R.Assign(2, 7, 24, 8);
Insert(New(PLabel, Init(R, 'Delivery instructions', B)));
```

Setting up an input line is simplicity itself: You assign a rectangle that determines the length of the input line within the screen. The only other parameter required is one defining the maximum length of the string to be edited. That length may exceed the displayed length because the *TInputLine* object knows how to scroll the string forward and backward. By default, the input line can handle keystrokes, editing commands, and mouse clicks and drags.

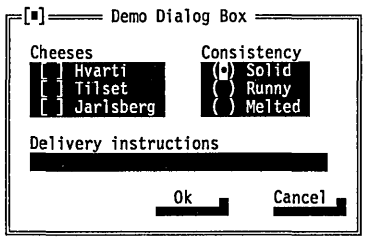


Figure 2.10
Dialog box with input line added

The input line also has a label for clarity, since unlabeled input lines can be even more confusing to users than unlabeled clusters.

Setting and getting data

Now that you have constructed a fairly complex dialog box, you need to figure out how to *use* it. You have set up the user interface end; now you need to set up the program interface. Having controls isn't much help if you don't know how to get information from them!

There are basically two things you need to be able to do: Set the initial values of the controls when the dialog box is opened, and read the values back when the dialog box is closed. Note that you don't want to modify any data outside the dialog box until you successfully close the box. If the user decides to cancel the dialog box, you have to be able to ignore any changes made while the dialog box was open.

Luckily, Turbo Vision facilitates doing just that. Your program hands a record of information to a dialog box when it is opened. When the user ends the dialog box, your program needs to check to see if the dialog box was canceled or closed normally. If it was canceled, you can simply proceed, without modifying the record. If the dialog box closed successfully, you can read back a record from the dialog box in the same form as the one given to it.

The methods *SetData* and *GetData* are used to copy data to and from a view. Every view has both a *SetData* and *GetData* method.

When a group (such as *TDialog*) is initialized through a *SetData* call, it passes the data along by calling each of its subviews' *SetData* methods.

When you call a group's *SetData*, you pass it a data record that contains the data for each view in the group. You need to arrange each view's data in the same order as the group's views were inserted.

You also need to make the data the proper size for each view. Every view has a method called *DataSize* which returns the size of the view's data space. Each view copies *DataSize* amount of data from the data record, then advances a pointer to tell the next view where to begin. If a subview's data is the wrong size, each subsequent subview will also copy invalid data.

If you create a new view and add data fields to it, don't forget to override *DataSize*, *SetData*, and *GetData* so that they handle the proper values. The order and sizes of the data in the data structure is entirely up to you. The compiler will return no errors if you make a mistake.

After the dialog box executes, your program should first make sure the dialog box wasn't canceled, then call *GetData* to import the dialog box's information back into your application.

So, in your example program, you initialize in turn a cluster of check boxes, a label, a cluster of radio buttons, a label, an input line of up to 128 characters, a label, and two buttons (Ok and Cancel). Table 2.1 summarizes the data requirements for each of these.

Table 2.1
Data for dialog box controls

Control	Data required
check boxes	*Word*
label	none
radio buttons	*Word*
label	none
input line	**string**[128]
label	none
button	none
button	none

Views that have no data (such as labels and buttons) use the *GetData* method they inherit from *TView*, which does nothing at all, so you don't need to concern yourself with them here. This means that when getting and setting data, you can skip over labels and buttons.

Thus, you are only concerned with three of the views in the dialog box: the check boxes, the radio buttons, and the input line. As noted earlier, each of the cluster items stores its data in a *Word*-type field. The input line's data is stored in a string. You can set up a data record for this dialog box in a global type declaration:

```
DialogData = record
  CheckBoxData: Word;
  RadioButtonData: Word;
  InputLineData: string[128];
end;
```

Now all you have to do is initialize the record when you start up the program (*MyApp.Init* is a good place), set the data when you enter the dialog box, and read it back when the dialog box closes successfully. It's almost easier to say that in Pascal than it was in English! Once you've declared the type as we did here, you declare a global variable:

```
var
  DemoDialogData: DialogData;
```

then add one line before executing the dialog box and one after:

```
Dialog^.SetData(DemoDialogData);
Control := DeskTop^.ExecView(Dialog);
if Control <> cmCancel then Dialog^.GetData(DemoDialogData);
```

and add six lines to the *TMyApp.Init* method to set the initial values for the dialog box:

This is TVGUID16.PAS

```
with DemoDialogData do
```

```
begin
  CheckboxData := 1;
  RadioButtonData := 2;
  InputLineData := 'Phone home.';
end;
```

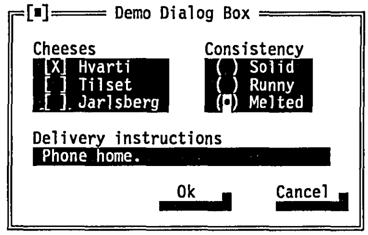


Figure 2.11
Dialog box with initial values set

Now any changes you make to the dialog box should be there when you reopen it, as long as you didn't cancel the dialog.

One of the things we learned as we wrote the Turbo Pascal integrated environment was that it is a good idea to have your program store information that gets altered by a dialog box in the form of a record that can be used for setting or getting data from the dialog box. This keeps you from having to construct lots of data records from discrete variables every time you want to open a dialog box, and from having to disperse the information returned from a dialog box to various variables when it's done.

Shortcut keys and conflicts

By default, labels, check boxes and radio buttons can respond to shortcut keys even when the focus is elsewhere within the dialog. For example, when your example dialog box first opens, the focus is in the check boxes, and the cursor is on the first check box. Pressing an *M* for "Melted" will immediately move the focus to the **Melted** radio button and turn it on.

While you obviously want shortcut keys to be as mnemonic as possible, there are only 26 letters and 10 digits available. This may cause some conflicts. For example, in your little dialog box it would make sense to have *C* as the shortcut for "Cheeses," "Consistency," and maybe a cheese called "Cheddar." There are a couple of ways to deal with such situations.

First, while it is nice to have the first letter of a word be the shortcut, it is not always possible. You can resolve the conflict between "Cheeses" and "Consistency," for example, by making *O* the shortcut for "Consistency," but the result is not as easy to remember. Another way, of course, is to relabel something. Instead of the label "Cheeses," you could label that cluster "Kind of Cheese," with *K* as the shortcut.

This sort of manipulation is the only way around conflicts of shortcut keys at the same level. However, there is another approach you can take if the conflict is between, say, a label and a member of a cluster: Shortcut keys can be made local within a dialog box item. In the previous example, for example, if you localize the shortcuts within each cluster, pressing *M* when the check boxes are focused will *not* activate the "Consistency" buttons or the "Melted" button. *M* would only function as a shortcut if you clicked or *Tab*bed into the "Consistency" cluster first.

The Options field and the ofPostProcess bit are both explained in Chapter 4.

By default all shortcut keys are active over the entire dialog box. If you want to localize shortcuts, change the default *Options* field for the object you are about to insert into the dialog box. For example, if you want to make the shortcuts in your check boxes local, you would add another line before inserting into the dialog box:

```
R.Assign(3, 3, 18, 6);
B := New(PCheckBoxes, Init(R,
  NewSItem('~H~varti',
  NewSItem('~T~ilset',
  NewSItem('~J~arlsberg',
  nil)))
));
B^.Options := B^.Options and not ofPostProcess;
Insert(B);
```

Now the *H, T,* and *J* shortcut keys only operate if you click or *Tab* into the "Cheeses" cluster first. *Alt-H, Alt-T,* and *Alt-J* will continue to function as before, however.

See the "Phase" section in Chapter 5 for more explanation.

Keep in mind that a label never gets the focus. Therefore, a label must have its *ofPostProcess* bit on for its shortcut to operate.

Having *ofPostProcess* set means that the user can enter information in a dialog box quickly. However, there are some possible drawbacks. A user may press a shortcut key expecting it to go to one place, but because of a conflict it goes somewhere else. Similarly, if the user *expects* shortcut keys to be active, but they're only active locally, it could be confusing to have a shortcut key do nothing when it is pressed outside the area where it is active.

The best advice we can give you is to test your dialog boxes carefully for conflicts. Avoid having duplicate shortcut keys when possible, and always make it clear to the user which options are available.

Ending the dialog box

When you are through with the dialog box, you call *Dispose(D, Done)*. Calling *Done* also removes the dialog box from the desktop.

Other dialog box controls

The *Dialogs* unit has some additional ready-made parts that weren't used in this example. They are used in the same way as the items you did use: You create a new instance, insert it into the dialog box, and include any appropriate data in the data record. This section will just describe briefly the functions and usage of each one. Much more detail is contained in Chapter 13, "Object reference."

Static text

TStaticText is a view that simply displays the string passed to it. The string is word wrapped within the view's rectangle. The text will be centered if the string begins with a *Ctrl-C* and line breaks can be forced with *Ctrl-M*. By default, the text can't get the focus, and of course, the object gets no data from the data record.

List viewer

A *TListViewer* will display a single or multiple column list, from which the user can select items. A *ListViewer* can also communicate with two scroll bars.

TListViewer is meant to be a building block, and is not usable by itself. It has the ability to handle a list, but does not itself contain a list. Its abstract method *GetText* loads the list members for its *Draw* method. A working descendant of *TListViewer* needs to override *GetText* to load actual data.

List box

TListBox is a working descendant of *TListViewer*. It owns a *TCollection* that is assumed to be pointers to strings. *TListBox* only supports one scroll bar. An example of a list box is the file selection list in the Turbo Pascal integrated environment, or the file list used by *TFileDialog* in STDDLG.PAS.

Getting and setting data with list boxes is greatly facilitated by the use of the *TListBoxRec* record type, which holds a pointer to a collection containing the list of strings to be displayed and a word indicating which item is currently selected in the list.

History

THistory implements an object that works together with an input line and a related list box. By clicking on the arrow icon next to the input line, the user brings up a list of previous values given for the input line, any of which may then be selected. This saves on repetitive typing.

THistory objects are used in many places in the Turbo Pascal integrated environment, such as the **F**ile | **O**pen dialog box and in the **S**earch | **F**ind dialog box.

Standard dialog boxes

The *StdDlg* unit contains a pre-built dialog called *TFileDialog*. You use this dialog box in the integrated environment when you open a file. *TFileDialog* uses a number of further objects, also in the *StdDlg* unit, which you may find useful:

```
TFileInputLine = object(TInputLine)
TFileCollection = object(TSortedCollection)
TSortedListBox = object(TListBox)
TFileList = object(TSortedListBox)
TFileInfoPane = object(TView)
```

Because the source for the entire standard *Dialogs* unit is included, we will not describe the objects in detail here.

PART 2

Programming Turbo Vision

C H A P T E R

3

The object hierarchy

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

After some general comments on OOP and hierarchies, this chapter takes you quickly through the Turbo Vision object hierarchy, stressing how the objects are related through the inheritance mechanism. By learning the main properties of each standard object type (many of which are related to the object's name in an obvious way), you will gain an insight into how the inherited and new fields and methods of each object combine to provide the object's functionality.

The complete hierarchy tree is shown in Figure 3.1. You'll find that this picture repays careful study. 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 3.1
Turbo Vision object hierarchy

```
TObject─┬─TView────────┬─TBackground
        │              ├─TButton
        │              ├─TCluster─────────┬─TCheckBoxes
        │              │                  └─TRadioButtons
        │              ├─TFrame
        │              ├─TGroup───────────┬─TDeskTop
        │              │                  ├─TProgram──────────TApplication
        │              │                  └─TWindow─────────┬─TDialog
        │              │                                    └─THistoryWindow
        │              ├─THistory
        │              ├─TInputLine
        │              ├─TListViewer──────┬─TListBox
        │              │                  └─THistoryViewer
        │              ├─TMenuView────────┬─TMenuBar
        │              │                  └─TMenuBox
        │              ├─TScroller──────────TTextDevice───────TTerminal
        │              ├─TScrollBar
        │              ├─TStaticText──────┬─TLabel
        │              │                  └─TParamText
        │              └─TStatusLine
        ├─TCollection────TSortedCollection──TStringCollection─TResourceCollection
        ├─TStream──────┬─TDosStream─────────TBufStream
        │              └─TEmsStream
        ├─TResourceFile
        ├─TStringList
        └─TStrListMaker
```

As you develop your own Turbo Vision applications, you will 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.

There is no "perfect" hierarchy for any application. Every object hierarchy is something of a compromise obtained by careful experiment (and a fair amount of intuition acquired with practice). You can benefit from our experience in developing object type hierarchies. Naturally, you can create your own base object types to achieve special effects beyond the standard objects provided.

Chapter 13, "Object 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. Part 3 provides alphabetical reference material.

Object typology

Not all object types are created equal in Turbo Vision. You can separate their functions into three distinct groups: primitive objects, view objects, and mute objects. Each of these is described in a separate section of this chapter.

Within each of these groups there are also different sorts of objects, some of which are useful objects that you can instantiate and use, and others of which are abstract objects that serve as the basis for deriving related, useful objects. Before we look at the objects in the Turbo Vision hierarchy, it will be helpful to understand a little about these abstract objects.

Abstract objects

Many object types exist as "abstract" bases from which more specialized and immediately useful object types can be derived. The reason for having abstract types is partly conceptual but largely serves the practical aim of reducing coding effort.

Take the *TRadioButtons* and *TCheckBoxes* types, for example. They 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 within which only one box can be checked, although there are a few other technical differences. This commonality warrants an abstract class called *TCluster*. *TRadioButtons* and *TCheckBoxes* are then derived from *TCluster* with the addition of a few specialized methods to provide their individual functionalities.

Abstract types are never usefully instantiated. An instance of *TCluster*, *MyCluster*, for example, would not have a useful *Draw* method: It inherits *TView.Draw* without overriding, so *MyCluster.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 would 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.

Abstract methods

Whether you can usefully instantiate an object type depends entirely on the circumstances. Many of Turbo Vision's standard types have abstract methods that must be defined in descendant types. Standard types may also have pseudo-abstract methods offering minimal default actions that may suit your purposes—if not, a derived type will be needed.

A general rule is that as you travel down the Turbo Vision hierarchy, the standard types become more specialized and less "abstract." Their names reveal the functionality encapsulated in their fields and methods. For most applications there will be obvious base types from which you can create a "standard" interface: a desktop, menu bar, status line, dialog boxes, and so on.

Object instantiations and derivations

Given any object type there are two basic operations available: You can create an instance of that type ("instantiate" it), or you can derive a descendant object type. In the latter case, you have a new object type on which the previous two operations can again be applied. Let's examine these operations in more detail.

Instantiation

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

```
MyScrollBar: TScrollBar;
SomeButton: PButton;
```

MyScrollBar would be initialized by *TScrollBar.Init* with certain default field values. These can be found by consulting the *TScrollBar.Init* entry in Chapter 13, "Object reference." Since *TScrollBar* is a descendant of *TView*, *TScrollBar.Init* calls *TView.Init* to set the fields inherited from *TView*. Similarly, *TView.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 *MyScrollBar* object now has default field values which you may need to change. It also has all the methods of *TScrollBar* plus the methods (possibly overridden) of *TView* and *TObject*. To make use of *MyScrollBar*, you need to know what its methods *do*, especially *HandleEvent* and *Draw*. If the required functionality is not defined in *TScrollBar*, you need to derive a new descendant type.

Derivation

You can easily derive a new object type from an existing one:

```
PNewScrollBar = ^TNewScrollBar;
TNewScrollBar = object(TScrollBar)
  end;
```

You do not yet have any instances of this new object type. Before declaring any *TNewScrollBar* objects, you need to define new methods or override some of *TScrollBar's* methods and possibly add some new fields; otherwise there would be no reason to create a new scroll bar object type. The new or revised methods and fields you define constitute the process of adding functionality to *TScrollBar*. Your new *Init* method would determine the default values for your new scroll bar objects.

Turbo Vision methods

Turbo Vision methods can be characterized in four (possibly overlapping) ways, each described here.

Abstract methods

In the base object type, an abstract method has no defining 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. Abstract methods are always virtual methods. An example of this is *TStream.Read*.

Pseudo-abstract methods

In the base object type, a pseudo-abstract method has a minimal action defined. It will almost always be overridden by a descendant to be useful, but the method provides a reasonable default for all objects in the inheritance chain. An example is *TSortedCollection.Compare.*

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.*

Static methods

A static method cannot be overridden *per se.* A descendant type may 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. In other words, *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.*

Turbo Vision 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 (recall that object trees grow downward from the root!).

Table 3.1
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*.

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. *TPoint* and *TRect* are used by all the visible objects in the Turbo Vision hierarchy. *TObject* is the basis of the hierarchy.

Note that objects of these types are not directly displayable. *TPoint* is simply a screen-position object (X, Y coordinates). *TRect* sounds like a view 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 topmost, leftmost point on the screen. *X* increases horizontally to the right; *Y* increases vertically downwards. *TPoint* has no methods, but other types have methods that convert between global (whole screen) and local (relative to a view's origin) coordinates.

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 cannot 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 an abstract destructor that must be overriden by descendants. Any objects that 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.

Views overview

A view is any object that can be drawn (displayed) in a rectangular portion of the screen. The type of a view object must be a descendant of *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 minimal virtual *Draw* method (forcing all immediate descendants to supply a specific *Draw* method).

Most of your Turbo Vision programming will use the more specialized descendants of *TView*, but the functionality of *TView* permeates the whole of Turbo Vision, so you'll need to understand what it offers.

Groups

The importance of *TView* is literally apparent from the hierarchy chart shown in Figure 3.1. 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 several objects to act in concert.

The abstract group

TGroup lets you handle dynamically chained lists of related, interacting subviews via a designated view called the *owner* of the group. Each view has an *Owner* field of type *PView* that points to the owning *TGroup* object. A **nil** pointer means that the view has no owner. A field called *Next* provides a link to the next view in the view chain. Since a group is a view, there can be subviews that are 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. During its lifespan, a subview can be hidden or exposed by actions performed on other subviews, so the group needs to coordinate many activities.

Desktops *TDesktop* is the normal startup background view, providing the familar user's desktop, usually surrounded by a menu bar and status line. Typically, *TApplication* will be the owner of a group containing *TDesktop*, *TMenuBar* and *TStatusLine* objects. 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.

Programs *TProgram* provides a set of virtual methods for its descendant, *TApplication*.

Applications *TApplication* provides a program template object for your Turbo Vision application. It is a descendant of *TGroup* (via *TProgram*). Typically, it will own *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.

Windows *TWindow* objects, with help from associated *TFrame* objects, are the popular bordered rectanglar displays that you can drag, resize, and hide using methods inherited from *TView*. A field called *Frame* points to the window's *TFrame* object. A *TWindow* object can also zoom and close itself using its own methods. *TWindow* handles the *Tab* and *Shift-Tab* key method for selecting the next and previous selectable subviews in a window. *TWindow's* event handler takes care of close, zoom, and resize commands. Numbered windows can be selected with *Alt-n* hot keys.

Dialog boxes *TDialog* is a descendant of *TWindow* used to create dialog boxes to handle a variety of user interactions. Dialog boxes typically contain controls such as buttons and check boxes. The parent's *ExecView* method is used to save the previous context, insert a *TDialog* object into the group, and then make the dialog box modal. The *TDialog* object then handles user-generated events such as button clicks and keystrokes. The *Esc* key is treated specially as an exit (*cmCancel*). The *Enter* key is specially treated as a broadcast *cmDefault* event (usually meaning that the default button has been selected). Finally, *ExecView* restores the previously saved context.

Terminal views

Terminal views are all views that are not groups. That is, they cannot own other views. They are therefore the endpoints of any chains of views.

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 explanatory label. Additional fields are *Value*, giving a user-defined value, and *Sel*, indexing the selected control of the cluster. Methods for drawing text-based icons and mark characters are provided. The cursor keys or mouse clicks can be used to mark controls in the cluster.

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, *TMenuBar* and *TMenuBox*, 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. The *HandleEvent* methods take care of the mechanics of mouse and/or keyboard (including shortcut and hot key) menu selection.

Menu selections are displayed using a *TMenuBar* object, usually owned by a *TApplication* object. Menu selections are displayed in objects of type *TMenuBox*.

For most applications, you will not be involved directly with menu objects. By overriding *TApplication.InitMenuBar* with a suitable set of nested *New, NewSubMenu, NewItem* and *NewLine* calls, Turbo Vision builds, displays, and interacts with the required menus.

Histories

The abstract type *THistory* implements a generic pick-list mechanism. Its two additional fields, *Link* and *HistoryId,* give each *THistory* object an associated *TInputLine* and the ID of a list of previous entries in the input line. *THistory* works in conjunction with *THistoryWindow* and *THistoryViewer.*

Input lines

TInputLine is a specialized view that provides a basic input line string editor. It handles all the usual keyboard entries and cursor movements (including *Home* and *End*). It offers deletes and inserts with selectable insert and overwrite modes and automatic cursor shape control. The mouse can be used to block mark text.

List viewers

The *TListViewer* object type is an abstract base type from which to derive list viewers of various kinds, such as *TListBox. TListViewer's* fields and methods let you display linked lists of strings with control over one or two scroll bars. The event handler permits mouse or key selection (with highlight) of items on the list. The *Draw* method 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.

TListBox, derived from *TListViewer,* implements the most commonly used list boxes, namely those displaying lists of strings such as file names. *TListBox* objects represent displayed lists of such items 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 select (and highlight) items by mouse and keyboard cursor actions. *TListBox* has an additional field called *List,* pointing to a *TCollection* object. This provides the items to be listed and selected. The contents of the collection are your responsibility, as are the actions to be performed when an item is selected.

Scrolling objects

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 have two fields, *HScrollId* and *VScrollId*, identifying their controlling horizontal and vertical scroll-bars. The *Delta* field in *TScroller* determines the unit amount of *X* and *Y* scrolling in conjunction with fields in the associated scroll bars.

TScrollBar objects provide either vertical or horizontal control. The key fields are *Value* (the position of the scroll bar indicator), *PgStep* (the amount of scrolling needed in response to mouse clicks and *PgUp*, *Pg↓* keys) and *ArStep* (the amount of scrolling needed in response to mouse clicks and arrow keys).

A scroller and its scroll bars are usually owned by a *TWindow* object leading to a complex set of events to be handled. For example, resizing the window must trigger appropriate redraws by the scroller. The values of the scroll bar must also be changed and redrawn.

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. *TTextDevice* uses *TScroller's* constructor and destructor.

TTerminal implements a "dumb" terminal with buffered string reads and writes. The size of the buffer is determined at initialization.

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. The additional field *Link* associates the label with another view, usually a control view that handles all label events. Selecting the label selects the linked control and selecting the linked control highlights the label as well as the control.

Status lines

A *TStatusLine* object is intended for various status and hint (help) 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. Items on the status line can be mouse or hot key selected rather like *TLabel* objects. Most application objects will start life owning a *TMenuBar* object, a *TDesktop* object, and a *TStatusLine* object. The added fields for *TStatusLine* provide an Items pointer and a *Defs* pointer.

The *Items* field points to the current linked list of *TStatusItem* records. These hold the strings to be displayed, the hot key mappings, and the associated *Command* word. The *Defs* field points to a linked list of *PStatusDef* records used to determine the current help context so you can display short "hints." *TStatusLine* can be instantiated and initialized using *TApplication.InitStatusLine.*

Non-visible elements

The non-view families derived from *TObject* provide streams, resource files, collections, and string lists.

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. *TStream* provides a *Status* field indicating the access mode (read only, write only, read/write) and an *ErrorInfo* field to report I/O failures. There are seven virtual methods: *Flush, GetPos, GetSize, Read, Seek, Truncate,* and *Write.* These must be overridden to derive specialized stream types. You'll see that Turbo Vision adopts this strategy to derive *TDosStream, TEmsStream,* and *TBufStream.* Other methods include *CopyFrom, Error, Get, ReadStr, Reset,* and *WriteStr.*

Object types must be registered using *RegisterType* before they can be used with streams. Turbo Vision's standard object types are preregistered (see "*RegisterType* procedure" in Chapter 14, "Global reference").

DOS streams

TDosStream is a specialized *TStream* derivative implementing unbuffered DOS file streams. A *Handle* field is provided, corresponding to the familiar DOS file handle. The *Init* constructor creates a DOS stream with a given file name and access mode. *TDosStream* defines all the abstract methods of *TStream* except for *Flush*, which is needed only for buffered streams.

Buffered streams

TBufStream implements a buffered version of *TDosStream*. The *Buffer* and *BufSize* fields are added to specify the location and size of the buffer. The fields *BufPtr* and *BufEnd* define a current position and final position within the buffer. The abstract *TStream.Flush* method is defined to flush the buffer. Flushing means writing out and clearing any residual buffer data before a stream is closed.

EMS streams

A further specialized stream, *TEmsStream* implements streams in EMS memory. New fields provide an EMS handle, the number of pages, the stream size, and the current position within the stream.

Resources

A resource file is a special kind of stream where generic objects ("items") 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. Resource items are accessed with *Get(Key)* calls where *Key* is the string index. Other methods provided are *Put* (store an item with a given key), *KeyAt* (get the index to a given item), *Flush* (write all changes to the stream), *Delete* (erase the item at a given key), and *Count* (return the number of items on file).

Collections

TCollection implements a general set of items, including arbitrary objects of different types. Unlike the arrays, sets, and lists of non-OOP languages, a Turbo Vision collection allows for dynamic sizing. *TCollection* is an abstract base for more specialized collections, such as *TSortedCollection*. The chief field is *Items*, a pointer to an array of items. Apart from the indexing, insertion, and deletion methods, *TCollection* offers several iterator routines. A collection can be scanned for the first or last item that meets a condition specified in a user-supplied test function. With the *ForEach* method you can also trigger user-supplied actions on each item in the collection.

Sorted collections

TSortedCollection implements collections that are sorted by keys. Sorting is defined via a virtual, abstract *Compare* method. Your derived types can therefore specify particular ordering for collections of objects of any type. The *Insert* method adds items to maintain this ordering, and keys can be located quickly with a binary *Search* method.

String collections

TStringCollection is a simple extension of *TSortedCollection* for handling sorted collections of Turbo Pascal strings. The secret ingredient is the overriding of the *Compare* method to provide alphabetical ordering. A *FreeItem* method removes a given string item from the collection. For writing and reading string collections on streams, the virtual *PutItem* and *GetItem* methods are provided.

Resource collections

TResourceCollection implements a collection of sorted resource indexes used by resource files. The *TStringCollection* methods, *FreeItem*, *GetItem*, *KeyOf*, and *PutItem* are all overriden to handle resources.

String lists

TStringList implements a special kind of string resource in which strings can be accessed via a numerical index using the *Get* method. A *Count* field holds the number of strings in the object. *TStringList* simplifies internationalization and multilingual text applications. String lists can be read from a stream using the *Load* constructor. To create and add to string lists, you use *TStrListMaker*.

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.

C H A P T E R

4

Views

By now, you should have a sense, from reading Chapters 1 and 2 and from looking at the integrated environment, of what a Turbo Vision application looks like from the outside. But what's behind the scenes? That's the subject of the next two chapters.

"We have taken control of your TV..."

One of the adjustments you make when you use Turbo Vision is that you give up writing directly to the screen. Instead of using *Write* and *Writeln* to convey information to the user, you give the information to Turbo Vision, which makes sure the information appears in the right places at the right time.

The basic building block of a Turbo Vision application is the *view*. A view is a Turbo 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.

Menus are views, as are windows, the status line, buttons, scroll bars, dialog boxes, and usually even a simple line of text. In general, anything that shows up on the screen of a Turbo Vision program *must* be a view, and the most important property of a view is that it knows how to represent itself on the screen. So, for example, when you want to make a menu system, you simply tell

Turbo Vision that you want to create a menu bar containing certain menus, and Turbo Vision handles the rest.

The most visible example of a view, but one you probably would not think of as a view, is the program itself. It controls the entire screen, but you don't notice that because the program sets up other views (called its *subviews*) 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.

Simple view objects

As you can see from the hierarchy chart in Figure 4.6, all Turbo Vision views have *TObject* as an ancestor. *TObject* is little more than a common ancestor for all the objects. Turbo Vision itself really starts at *TView*.

A *TView* itself just appears on the screen as a blank rectangle. There is little reason to instantiate a *TView* itself unless you want to create a blank rectangle on the screen for prototyping purposes. But even though *TView* is visually simple, it contains all of Turbo Vision's basic screen management methods and fields.

There are two things any *TView*-derived object must be able to do:

The first is draw itself at any time. *TView* defines a virtual method called *Draw*, and each object derived from *TView* must also have a *Draw* method. This is important, because often a view will be covered or overlapped by another view, and when that other view goes away or moves, the view must be able to show the part of itself that was hidden.

The second is handle any events that come its way. As noted in Chapter 1, Turbo Vision programs are event-driven. This means that Turbo Vision gathers input from the user and parcels it out to the appropriate objects in the application. Views need to know what to do when events affect them. Event handling is covered in detail in Chapter 5.

Setting your sights

Before discussing what view objects *do*, you need to learn a bit about what they *are*—how they represent themselves on the screen.

TPoint is described in the next section.

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 the type *TPoint*. The *Origin* field is a *TPoint* indicating the origin of the view, and the *Size* field represents the lower right corner.

Note that *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.

Getting the TPoint

The *TPoint* type is *extremely* simple. It has only two fields, called *X* and *Y*, which are its coordinates. It has no methods. Turbo Vision uses the *TPoint* object to allow views to specify their coordinates as a single field.

Getting into a TRect

For convenience, *TPoint*s are rarely dealt with directly in Turbo Vision. Since each view object has both an origin and a size, they are usually handled together in an object called *TRect*. *TRect* has two fields, *A* and *B*, each of which is a *TPoint*. When specifying the boundaries of a view object, those boundaries are passed to the constructor in a *TRect*.

TRect and *TView* both provide useful methods for manipulating the size of a view. For example, if you want to create a view that fits just inside a window, you can get the window to tell you how big it is, then shrink that size and assign it to the new inside view.

ThisWindow and PInsideView are just made up for this example.

```
procedure ThisWindow.MakeInside;
var
  R: TRect;
  Inside: PInsideView;
begin
  GetExtent(R);                          { sets R to size of ThisWindow }
  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;
```

GetExtent is a *TView* method that sets the argument *TRect* to the coordinates of a rectangle covering the entire view. *Grow* is a

TRect method that increases (or with negative parameters, decreases) the horizontal and vertical sizes of a rectangle.

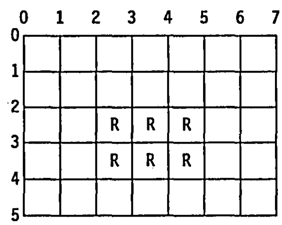
Turbo Vision coordinates

Turbo Vision's method of assigning coordinates may be different from what you're used to. The difference is that, unlike most coordinate systems that designate the character spaces on the screen, Turbo Vision coordinates specify the grid *between* the characters.

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

Figure 4.1
Turbo Vision coordinate system



Thus, `R.Assign(2,2,4,5)` produces a rectangle that contains six character spaces. Although this coordinate system is slightly unconventional, it makes it *much* easier to calculate the sizes of rectangles, the coordinates of adjacent rectangles, and some other things as well.

Making an appearance

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, and
- be able to draw itself at any time.

Both of these properties are very important and deserve some discussion.

Territoriality

There are good reasons for each view to take responsibility for its own territory. A view is assigned a rectangular region of the screen. If it does not fill in that whole area, the contents of the unfilled area are undefined: Just about anything could show up there, and you would have no control over it. The program TVDEMO05.PAS demonstrates what happens if a view leaves some of its appearance to chance.

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 may mean that the view does nothing at all: It may be entirely covered, or it may 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 a lot of 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 correct—after all, something may have told it to change while it was covered!

Putting on your best behavior

The behavior of a view is almost entirely determined by a method called *HandleEvent*. *HandleEvent* is passed an event record, which it must process in one of two ways. It can either perform some action in response to the event and then mark the event as having been handled, or it can pass the event along to the next view (if any) that should see it.

Event handling is covered in detail in Chapter 5, "Event-driven programming."

The key to behavior, really, is how the view responds to certain events. For example, if a window receives an event containing a *cmClose* command, the expected behavior is that the window would close. It is possible that you might devise some other response to that command, but not likely.

Complex views

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.

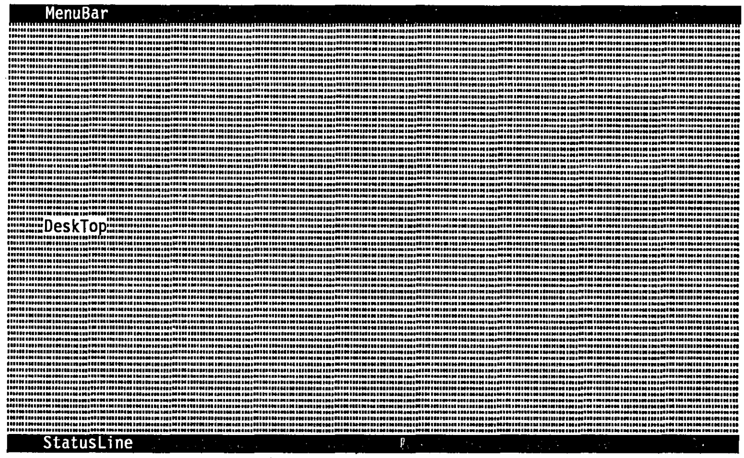
Basically a group is just an empty box that contains and manages other views. Technically, it is a view, and therefore responsible for all the things that any view must be able to do: manage a rectangular area of the screen, visually represent itself at any time, and handle events in its screen region. The difference is really in *how* it accomplishes these things: most of it is handled by *subviews*.

Groups and subviews

A subview is a view that is owned by another view. That is, some view (a group) has delegated part of its region on the screen to be handled by another view, called a subview, which it will manage.

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 4.2 shows a typical *TApplication* screen.

Figure 4.2
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.

Getting into a group

How does a subview get attached to a group? The process is called *insertion*. Subviews are created and then inserted into groups. In the previous example, 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);
```

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 do not overlap at all.

There is no reason, however, that views cannot 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.

Groups keep track of the order in which subviews are inserted. That order is referred to as *Z-order*. As you will see, Z-order determines the order in which subviews get drawn and the order in which events get passed to them.

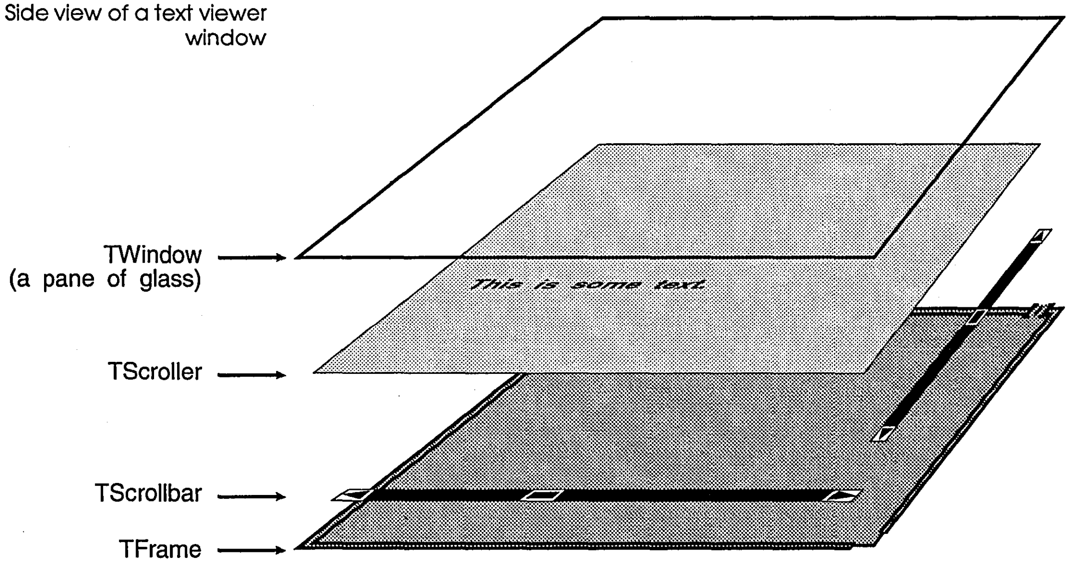
Another angle on Z-order

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, 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.

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 4.3.

Figure 4.3
Side view of a text viewer window



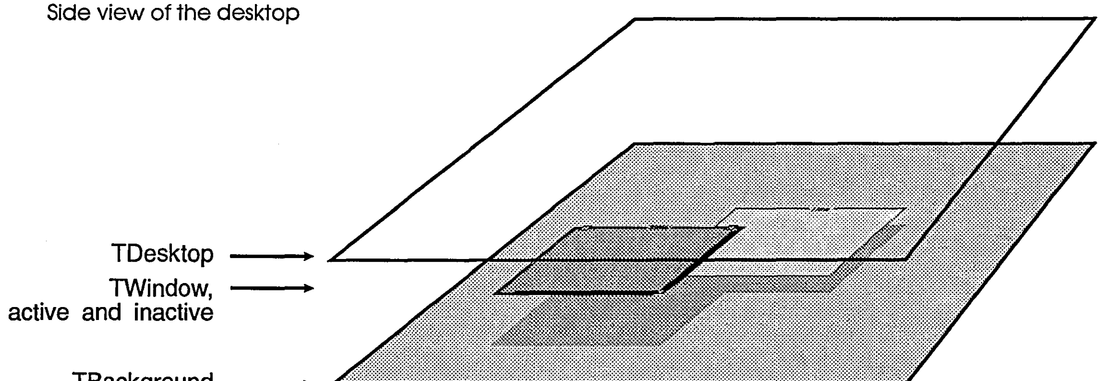
The window itself is just a pane of glass covering a group of views. Since all you see is a projection of the views behind the glass on the screen, you can't see which views are in front of others unless they overlap.

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 get 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 gets inserted, covering the entire area inside the border of the frame. Text gets 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 4.4.

Figure 4.4
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.

Group portraits

Groups are sort of 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* asks its subviews to draw themselves.

The subviews are called upon to draw themselves in Z-order, meaning that the first subview inserted into the group is the first one drawn. That way, if subviews overlap, the one most recently inserted will be in front of any others.

The subviews owned by a group must cooperate to cover the entire region controlled by 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 would appear, with unpredictable (and unpleasant!) results.

When the subviews of a group draw themselves, their 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.

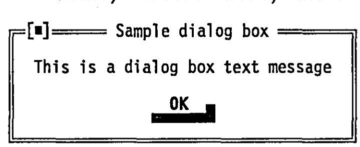
You may wonder where the desktop gets its visible background if it is a *TGroup*. At its initialization, the desktop creates and owns a subview called *TBackGround*, whose sole purpose is to draw in a uniform background for the whole screen. Since the background is the first subview inserted, it is obscured by the other views drawn in front of it.

Relationships between views

Views are related to each other in two distinct ways: They are members of the Turbo Vision object hierarchy, and they are members of the *view tree*. When you are new to Turbo Vision, it is important to remember the distinction.

For example, consider the simple dialog box in Figure 4.5. It has a frame, a one-line text message, and a single button that closes the dialog box. In Turbo Vision terms, that's a *TDialog* view that owns a *TFrame*, a *TStaticText*, and a *TButton*.

Figure 4.5
A simple dialog box



The object hierarchy

One way views are related is as parent and child in the object hierarchy. Notice in the hierarchy diagram (Figure 4.6) that *TButton* is a descendant of the *TView* object type. The *TButton* actually *is* a *TView*, but it has additional fields and methods that make it a button. *TDialog* is also a descendant of *TView* (through *TGroup* and *TWindow*), so it has much in common with *TButton*. The two are distant "cousins" in the Turbo Vision hierarchy.

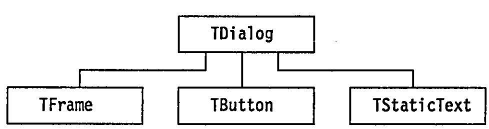
Figure 4.6
Turbo Vision object hierarchy

```
TObject─┬─TView──────────┬─TBackground
        │                ├─TButton
        │                ├─TCluster──────────┬─TCheckBoxes
        │                │                   └─TRadioButtons
        │                ├─TFrame
        │                ├─TGroup────────────┬─TProgram──────────TApplication
        │                │                   ├─TDeskTop
        │                │                   └─TWindow──────────┬─TDialog
        │                │                                      └─THistoryWindow
        │                ├─THistory
        │                ├─TInputLine
        │                ├─TListViewer───────┬─TListBox
        │                │                   └─THistoryViewer
        │                ├─TMenuView─────────┬─TMenuBar
        │                │                   └─TMenuBox
        │                ├─TScroller─────────TTextDevice─────────TTerminal
        │                ├─TScrollBar
        │                ├─TStaticText───────┬─TLabel
        │                │                   └─TParamText
        │                └─TStatusLine
        ├─TCollection────TSortedCollection───TStringCollection───TResourceCollection
        ├─TStream────────┬─TDosStream────────TBufStream
        │                └─TEmsStream
        ├─TResourceFile
        ├─TStringList
        └─TStrListMaker
```

Ownership

The other way that views are related is in a view tree. In the view tree diagram (Figure 4.7), the *TDialog* owns the *TButton*. Here the relationship is not between hierarchical object types (*TDialog* is *not* an ancestor of *TButton*!), but between instances of objects, between owner and subview.

Figure 4.7
A simple dialog box's view tree



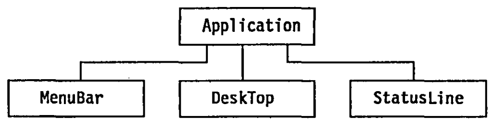
As you program, you'll need to make a *TButton* interact with its owner in the view tree (*TDialog*), and the *TButton* will also draw upon attributes inherited from its ancestor (*TView*). Don't confuse the two relationships.

A running Turbo Vision application looks like a tree, with views instantiating and owning other views. As your Turbo Vision application opens and closes windows, the view tree grows and shrinks as object instances are inserted and removed. Of course, the object hierarchy only grows when you derive new object types from the standard objects.

Subviews and view trees

As noted earlier, the *TApplication* view *owns* and manages the three subviews that it creates. You can think of this relationship as forming a *view tree*. *Application* is the trunk, and *MenuBar*, *DeskTop*, and *StatusLine* form the branches, as shown in Figure 4.8.

Figure 4.8
Basic Turbo Vision view tree



Remember, the relationship illustrated in Figure 4.8 is *not* an object hierarchy, but a model of a data structure. The links indicate *ownership*, not inheritance.

In a typical application, as the user clicks with the mouse or uses the keyboard, he creates more views. These views will normally appear on the desktop, and so form further branches of the tree.

It is important to understand these relationships between owners and subviews, as both the appearance and the behavior of a view depend a great deal on who owns the view.

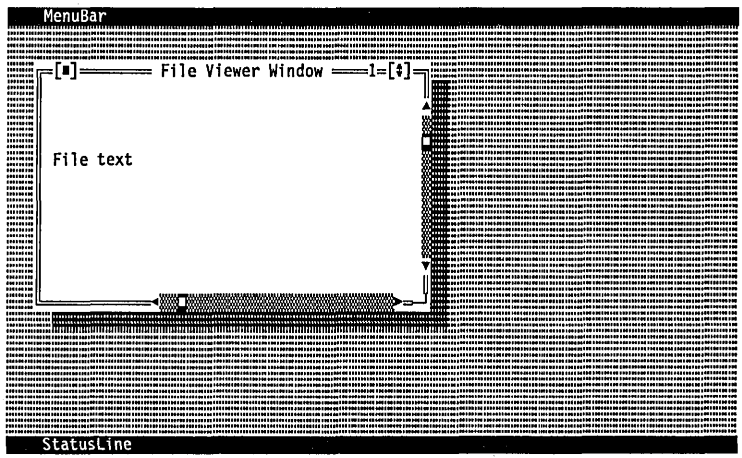
Let's follow the process. Say, for instance, that the user clicks on a menu selection that calls for a file viewer window. The file viewer window will be a view. Turbo Vision will create the window and attach it to the desktop.

This same kind of object is depicted somewhat differently in Figure 4.3.

A window will most likely own a number of subviews: a *TFrame* (the frame around the window), a *TScroller* (the interior view that holds a scrollable array of text), and a couple of *TScrollbars*. When the window is called into being, it creates, owns, and manages its subviews.

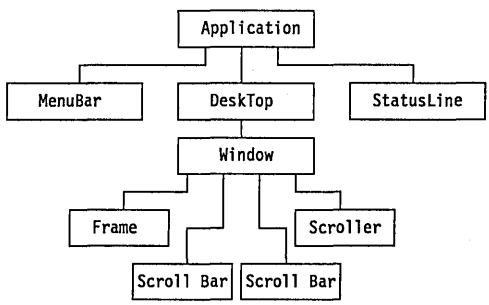
More views are now attached to our growing application, which now looks something like Figure 4.9.

Figure 4.9
Desktop with file viewer added



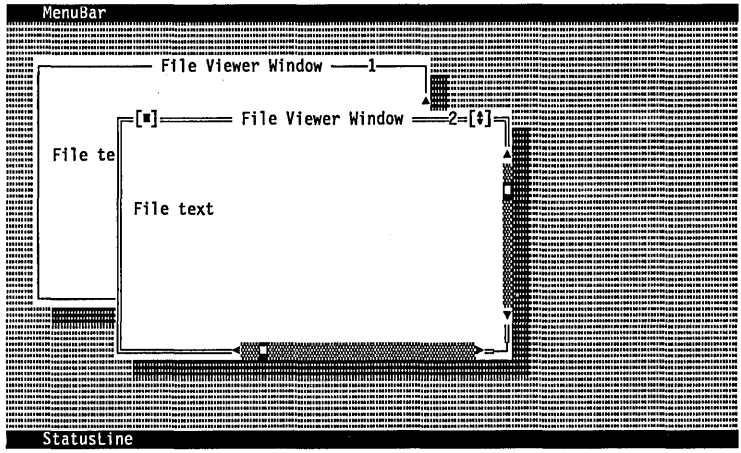
The view tree has also become somewhat more complex, as shown in Figure 4.10. (Again, these are *ownership* links.)

Figure 4.10
View tree with file viewer added



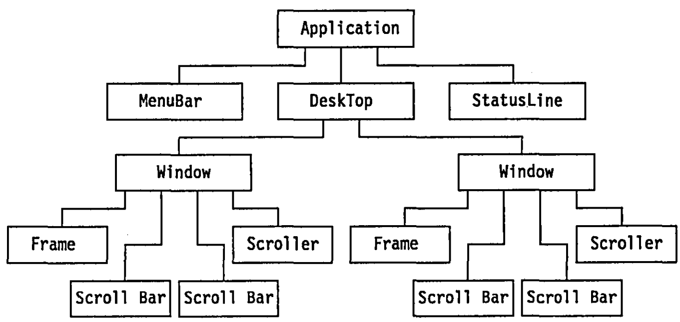
Now suppose the user clicks on the same menu selection and creates another file viewer window. Turbo Vision will create a second window and attach it to the desktop, as shown in Figure 4.11.

Figure 4.11
Desktop with file viewer added



The view tree also becomes correspondingly more complex, as shown in Figure 4.12.

Figure 4.12
View tree with two file viewers added



As you'll see in Chapter 5, program control flows down this view tree. In the preceding example, suppose you click on a scroll bar in the file viewer window. How does that click arrive at the right place?

The *Application* program sees the mouse click, realizes that it's within the area controlled by the desktop, and passes it to the desktop object. The desktop in turn sees that the click is within the area controlled by the file viewer, and passes it off to that view. The file viewer now sees that the click was on the scroll bar, and lets the scroll bar view handle the click, generating an appropriate response.

Event routing is explained in Chapter 5.

The actual mechanism for this is unimportant at this point. The important thing to remember is how views are connected. No matter how complex the structure becomes, all views are ultimately connected to your application object.

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 all of its subviews, then be disposed itself.

Eventually, the user will trim the views down to just the original four, and will indicate at some point that he is finished by pressing *Alt-X* or by selecting Exit from a menu. *TApplication* will dispose its three subviews, then dispose itself.

Selected and focused views

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.

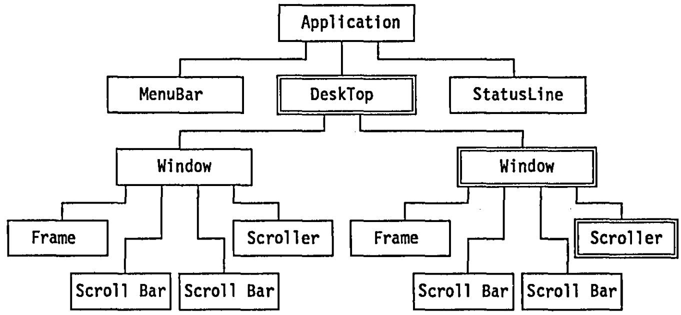
When you have several windows open on the desktop, the selected window is the one in which you're currently working. This is also called the *active* window (typically the topmost window).

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 would be 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 4.12, *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 4.13 depicts the same view tree with the chain of focused views highlighted by double-lined boxes.

Figure 4.13
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 5, "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 will be single-lined. Within a dialog box, the focused control (controls are views, too!) is brighter than the others, indicating that it is the one that will be 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 gets 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 may wish to change which view currently has the focus. A common way to do this is to click the mouse on a different view. For instance, if you have several windows open on the desktop, you can select different ones simply by clicking on them. In a dialog box, you can move the focus among views by pressing *Tab,* which cycles through all the available views, or by clicking the mouse on a particular view, or by pressing a hot key.

Note that there are some views that are not selectable, including the background of the desktop, frames of windows, and scroll bars. When you create a view, you may designate whether that view is selectable, after which the view will determine whether it lets itself be selected. If you click on the frame of a window, for example, the frame does not get the focus, because the frame knows it cannot be the focused view.

The focus chain

See Chapter 5, "Event-driven programming," for a full explanation.

If you start with the main application and trace to its selected subview, and continue following to each subsequent selected subview, you will eventually end up at the focused view. This chain of views from the *TApplication* object to the focused view is called the *focus chain*. The focus chain is used for routing focused events, such as keystrokes.

Modal views

A *mode* is a way of acting or functioning. A program may have a number of modes of operation, usually distinguished by different control functions or different areas of control. Turbo Pascal's integrated environment, for example, has an editing and debugging mode, a compiler mode, and a run mode. Depending on which of these modes is active, keys on the keyboard may have varying effects (or no effect at all).

A Turbo Vision view may define a mode of operation, in which case it is called a *modal view*. 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 closed. (Some dialog boxes are non-modal, but these are rare exceptions.)

When you instantiate a view and make it 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 Turbo 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 actually one exception to this rule, and that is the status line. Turbo Vision "cheats" a little, and keeps the status line 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.

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.

Modifying default behavior

Up to this point, you have seen mostly the default behavior of the standard views. But sometimes you will want to make your views look or act a little different, and Turbo Vision provides for that. This section explains the ways you can modify the standard views.

Every Turbo Vision view has four bitmapped fields that you can use to change the behavior of the view. Three of them are covered here: the *Options* word, the *GrowMode* byte, and the *DragMode* byte. The fourth, the *EventMask* word, is covered in Chapter 5, "Event-driven programming."

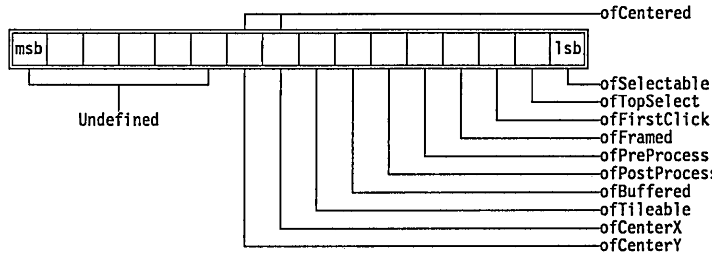
There is also a *State* word that contains information about the current state of the view. Unlike the others, State is essentially read-only. Its value should only be changed by the *SetState* method. For more details, see the "State flag and SetState" section in this chapter.

The Options flag word

Options is a bitmapped word in every view. Various descendants of *TView* have different *Options* set by default.

The *Options* bits are defined in Figure 4.14; explanations of the possible *Options* follow.

Figure 4.14
Options bit flags



ofSelectable If set, the user can select the view with the mouse. If the view is in a group, the user can select it with the mouse or *Tab* key. If you put a purely informational view on the screen, you might not want the user to be able to select it. Static text objects and window frames, for example, are usually not selectable.

ofTopSelect The view will be moved to the top of the owner's subviews if the view is selected. This option is designed primarily for windows on the desktop. You shouldn't use it for views in a group.

ofFirstClick The mouse click that selects the view is sent on to the view. If a button is clicked, you definitely want the process of selecting the button and operating it to happen with one click, so a button has *ofFirstClick* set. But if someone clicks on a window, you may or may not want the window to respond to the selecting mouse click other than by selecting itself.

ofFramed If set, the view has a visible frame around it. This is useful if you create multiple "panes" within a window, for example.

ofPreProcess If set, allows the view to process focused events before the focused view sees them. See the "Phase" section in Chapter 5, "Event-driven programming" for more details.

ofPostProcess If set, allows the view to handle focused events after they have been seen by the focused view, assuming the focused view has not cleared the event. See the "Phase" section in Chapter 5, "Event-driven programming" for more details.

ofBuffered When this bit is set, groups can speed their output to the screen. When a group is first asked to draw itself, it automatically stores the image of itself in a buffer if this bit is set and if enough memory is available. The next time the group is asked to draw itself, it copies the buffered image to the screen instead of asking all its subviews to draw themselves. If a *New* or *GetMem* call runs out of memory, Turbo Vision's memory manager will begin disposing of these group buffers until the memory request can be satisfied.

If a group has a buffer, a call to *Lock* will stop all writes of the group to the screen until the method *Unlock* is called. When *Unlock* is called, the group's buffer is written to the screen. Locking can decrease flicker during complicated updates to the screen. For example, the desktop locks itself when it is tiling or cascading its subviews.

ofTileable The desktop can tile or cascade the windows that are currently open. If you don't want a window to be tiled, you can clear this bit. The window will then stay in the same position, while the rest of the windows will be automatically tiled.

Tiling or cascading views from *TApplication.HandleEvent* is simple:

```
cmTile:
  begin
    DeskTop^.GetExtent(R);
    DeskTop^.Tile(R);
  end;
cmCascade:
  begin
    DeskTop^.GetExtent(R);
    DeskTop^.Cascade(R);
  end;
```

If there are too many views to be successfully cascaded, the desktop will do nothing.

ofCenterX When the view is inserted in a group, center it in the x dimension.

ofCenterY When the view is inserted in a group, center it in the y dimension. You may find this an important step in making a window work well with 25- or 43-line text modes.

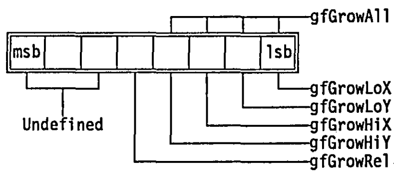
ofCentered Center the view in both the x and y dimensions when it is inserted in the group.

The GrowMode flag byte

A view's *GrowMode* field determines how the view will change when its owner group is resized.

The *GrowMode* bits are defined as follows:

Figure 4.15
GrowMode bit flags



gfGrowLoX If set, the left side of the view will maintain a constant distance from its owner's left side.

gfGrowLoY If set, the top of the view will maintain a constant distance from the top of its owner.

gfGrowHiX If set, the right side of the view will maintain a constant distance from its owner's right side.

gfGrowHiY If set, the bottom of the view will maintain a constant distance from the bottom of its owner.

gfGrowAll If set, the view will always remain the same size, and will move with the lower right corner of the owner.

gfGrowRel If set, the view will maintain its size relative to the owner's size. You should only use this option with *TWindows* (or descendants of *TWindow*) that are attached to the desk top. The window will maintain its relative size when the user switches the application between 25- and 43/50-line mode. This flag isn't designed to be used with views within a window.

The DragMode flag byte

A view's *DragMode* field determines how the view will behave when it is dragged.

The *DragMode* bits are defined as follows:

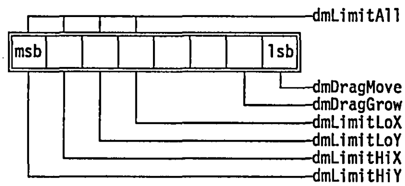


Figure 4.16
DragMode bit flags

The *DragMode* settings include the following:

dmDragMove When this bit is set, when you click on the top of a window's frame, you can drag it.

dmDragGrow When this bit is set, the view can grow.

dmLimitLoX If set, the left side of the view cannot go out of the owner view.

dmLimitLoY If set, the top of the view is not allowed to go out of the owner view.

dmLimitHiX If set, the right side of the view cannot go out of the owner view.

dmLimitHiY If set, the bottom of the view cannot go out of the owner view.

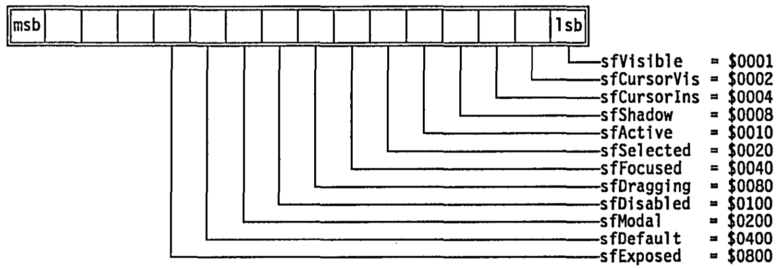
dmLimitAll If set, no part of the view can go out of the owner view.

State flag and SetState

A view also has a bitmapped flag called *State* which keeps track of various aspects of the view, such as whether it is visible, disabled, or being dragged.

The *State* flag bits are defined in Figure 4.17.

Figure 4.17
State flag bit mapping



The meanings of each of the state flags is covered in Chapter 14, "Global reference," under "sfXXXX state flag constants." This section focuses on the mechanics of manipulating the *State* field.

Turbo Vision changes a view's state flag through its *SetState* method. If the view gets the focus, gives up the focus, or becomes selected, Turbo Vision calls *SetState*. This differs from the way the other bitmapped flags are handled, because those are set on initialization and then not changed (if a window is resizable, it is *always* resizable, for example). The state of a view, however, will often change during the time it is on the screen. Because of this, Turbo Vision provides a mechanism in *SetState* that allows you not only to change the state of a view, but also to react to those changes in state.

SetState receives a state (*AState*) and a flag (*Enable*) indicating whether the state is being set or cleared. If *Enable* is True, the bits in *AState* are set in *State*. If *Enable* is False, the corresponding *State* bits are cleared. That much is essentially like what you would do with any bitmapped field. The difference comes when you want a view to *do* something when you change its state.

Acting on a state change

Views often take some action when *SetState* is called, depending on the resulting state flags. A button, for example, watches *State* and changes its color to cyan when it gets the focus. Here's a typical *SetState* for a descendant of *TView*:

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

Notice that you should always call *TView.SetState* from within a new *SetState* method. *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.

If you need to make changes in the view or the application when the state of a particular view changes, you can do it by overriding the view's *SetState*. Suppose your application includes a text editor, and you want to enable or disable all the menu bar's text editing commands depending on whether or not an editor is open. The text editor's *SetState* is defined like this:

This is the code used by the IDE's editor view.

```
procedure TEditor.SetState(AState: Word; Enable: Boolean);
const
  EditorCommands = [cmSearch, cmReplace, cmSearchAgain, cmGotoLine,
    cmFindProc, cmFindError, cmSave, cmSaveAs];
begin
  TView.SetState(AState, Enable);
  if AState and sfActive <> 0 then
    if Enable then EnableCommands(EditorCommands)
    else DisableCommands(EditorCommands);
end;
```

This code comes directly from the Turbo Pascal integrated environment, so the behavior it describes should be familiar.

The programmer and Turbo Vision often cooperate when the state changes. Suppose you want a block cursor to appear in your text editor when the editor's insert mode is toggled on, for example.

First, the editor insert mode will have been bound to a keystroke—say, the *Ins* key. When the text editor is the focused view and the *Ins* key is pressed, the text editor receives the *Ins* key event. The text editor's *HandleEvent* method responds to the *Ins* event by toggling some internal state of the view saying that the insert mode has changed, and by calling the *BlockCursor* method. Turbo Vision does the rest. *BlockCursor* calls the view's *SetState* to set the *sfCursorIns* state true.

What color is your view?

No one ever seems to agree on what colors are "best" for any computer screen. Because of this, Turbo Vision allows you to change the colors of the views you put on the screen. In order to facilitate this, Turbo Vision provides you with color palettes.

Color palettes

Palettes for all standard views are listed in Chapter 13, "Object reference."

When a Turbo Vision view draws itself, it asks to be drawn, not with a specific color, but with a color indicated by a position in its palette. For example, the palette for *TScroller* looks like this:

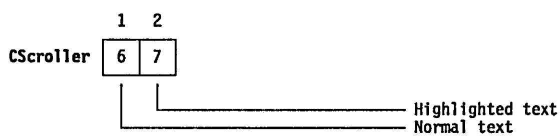
```
CScroller   = #6#7;
```

Color palettes are actually stored in strings, which 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 might be more useful to look at it this way:

Figure 4.18
TScroller's default color palette



GetColor is a TView method.

This means there are two kinds of text a scroller object knows how to display: normal and highlighted. The default color of each is determined by the palette entries. When displaying normal text, the *Draw* method needs to call *GetColor(1)*, meaning it wants the color indicated by the first palette entry. To show highlighted text, the call would be *GetColor(2)*.

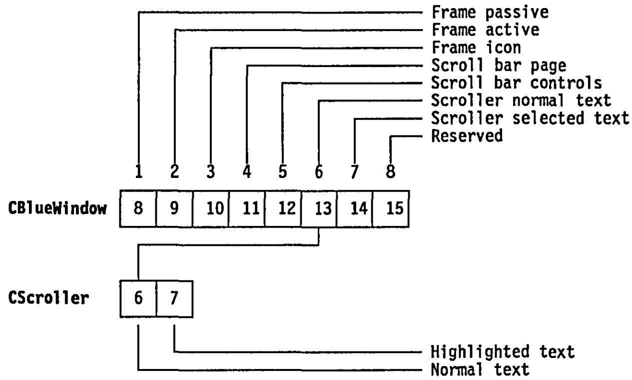
Selecting non-default colors is described in the next section.

If all you want to do is display the default colors, that's really all you need to know. The palettes are set up so that any reasonable combination of objects should produce decent looking colors.

Inside color palettes

Palette entries are actually indexes into their owner's palette, not the colors themselves. If a scroller is inserted into a window, you get normal text by calling for the normal text color in the scroller's palette, which contains the number 6. To translate that into a color, you find the sixth entry in the owner's palette. Figure 4.19 shows *TWindow*'s palette.

Figure 4.19
Mapping a scroller's palette onto a window



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 anything—you can think of it as a "straight" or "transparent" palette, with the first entry being the number 1, the second being 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. The application is the end of the line (it has no owner), so the mapping stops there.

So now you are left with $1E, which is a text attribute byte corresponding 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, but rather say that you want your text displayed as the normal color for window text.

Don't think of palettes as *colors*. They are *kinds of things to display*.

The GetColor method

Color palette mapping is done by the virtual *TView* function *GetColor*. *GetColor* climbs up the view tree from the object being drawn to its owner, to the owner's owner, and so on, until it gets to the application object. At each object along that chain, *GetColor* calls *GetPalette* for that object. The end result is a color attribute.

A view's palette contains offsets into its owner's palette, except the application, whose palette contains color attributes.

Overriding the default colors

The obvious way to change colors is to change the 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!

So what if you don't want bright yellow on blue? Change the palette entry for normal window text in *TApplication*. 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. Make this your color mantra: Colors are not absolute, but are determined by the owner's palettes.

This makes sense: Presumably you want 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 would have to change the entries for *each* 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.

To change a view's palette, override its *GetPalette* method. To create a new scroller object type that draws itself in the window's frame color instead of the 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;
  PMyScroller: string[Length(CMyScroller)] = CMyScoller;
begin
  GetPalette := @PMyScroller;
end;
```

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 allows for easier manipulation of the palettes, since all the string functions and the like can also be used with palettes.

Adding new colors

You may want to add additional colors to the window object type, which will allow for a variety of colors to be used for new views you create. For example, you might decide you want a third color in your scroller for a different type of highlight, such as the one used for the breakpoints in the IDE editor. This can be done by deriving a new object type from the existing *TWindow*, and adding to the default palette, as shown here:

```
type
  TMyWindow = object(TWindow)
    function GetPalette: PPalette; virtual;
  end;

function TMyWindow.GetPalette: PPalette;
const
  CMyWindow = CBlueWindow + #84;
  P: string[Length(CMyWindow)] = CMyWindow;
begin
  GetPalette := @P;
end;
```

Palettes are strings, so you can use string operations like "+."

Now *TMyWindow* has a new palette entry that contains this new type of hightlight. *CWindow* is a string constant containing *TWindow*'s default palette. You will have to change the *GetPalette* routine of *MyScroller* to take advantage of this:

```
function TMyScroller.GetPalette: PPalette;
const
  CMyScroller = #6#7#9;
  P: string[Length(CMyScroller)] = CMyScoller;
begin
  GetPalette := @P;
end;
```

The scroller's palette entry 3 is now the new highlight color (in this case bright white on red). If you use this new *GetPalette* using the *CMyScroller* that accesses the ninth element in its owner's palette, be sure that the owner is indeed using the *CMyWindow* palette. If you try to access the ninth element in an eight-element palette, the results are undefined.

C H A P T E R

5

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 4 explained views, and this chapter will deal with 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, and you make decisions based on that input within the loop. You'll call procedures or functions, or branch to a code loop somewhere else that again begins reading 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 really 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 on 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 will define new views with new actions, which will need to know about certain kinds of events that you'll define. You'll 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 what to do when they see 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; thus, 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 event: 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...
```

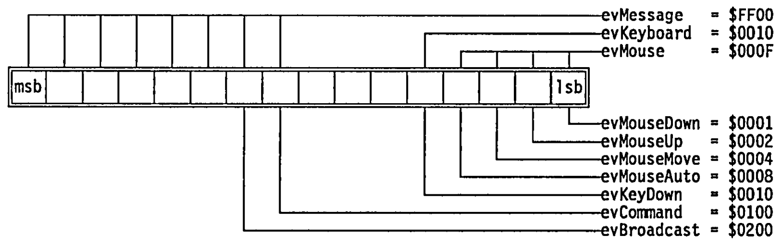
you can use

```
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 5.1.

Figure 5.1
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, as they have already been dealt with by another object.

Events and commands

Ultimately, most events end up being translated into commands of some sort. For example, clicking the mouse on 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. A mouse click on `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 1, "Inheriting the wheel," 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 124, 121, and 123, 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 what 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 4, "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 on!

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 on 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 4, "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 intend to get 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.

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 10, "Hints and tips."

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.

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. If you look at the example program TVDEMO09.PAS, you will notice that the scroll bars for the interior views all have their *ofPostProcess* bits set. If you modify the code to *not* set those bits, keyboard scrolling will be disabled.

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, you would want to use *ofPostProcess* in a case like this, because it provides greater flexibility. Later on you may wish to add functionality to the interior that checks keystrokes, but if the keystrokes have been taken by the scroll bar before they get to the focused view (*ofPreProcess*), your interior will never get to act on them.

☞ Although there are times when you will *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 so that 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 actually rather simple: Have the button check for different shortcut keys before and after the focused view handles the event. Specifically, by default, a button will look for its shortcut key in *Alt*-letter form pre process, and in letter form post process. 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 only checks 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 wind up getting 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 will have 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 whatever command was 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 will also create a command that will be used to invoke the view. Commands may be called anything, but Turbo Vision's convention is that a command identifier should start with "cm." The actual mechanics of creating a command are 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 may use the numbers 100 through 255 and 1000 through 65,535 for commands.

The reason for having two ranges of commands is that only the 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 5.1.

Table 5.1
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.

☞ The important thing to remember is that defining the command does not specify what action will be taken when that command appears in an event record. You will 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.

Specifically, to enable or disable a group of commands, you 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 NewDescendant.HandleEvent(var Event: TEvent);
begin
  { code to change or eliminate parental behavior }
  Parent.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;
```

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, and when they do, they often use the *InfoPtr* field. Communication among views and the *InfoPtr* field are both covered in the "Inter-view communication" section of this chapter.

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), 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 could either override *Idle* to look for characters from the serial port and generate events based on them, or override *GetEvent* itself to add a *GetComEvent(Event)* call to the loop, where *GetComEvent* returns an event record if there is a character available at the designated 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 just 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
  TApplication.GetEvent(Event);
  { 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, no matter where he is in your program.

```
procedure TMyApp.Idle;
begin
  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
  TSortedListBox.FocusItem(Item);      { call inherited method first }
  Message(TopView, evBroadcast, cmFileFocused, List^.At(Item));
end;

procedure TFileInputLine.HandleEvent(var Event: TEvent);
var
  Name: NameStr;
begin
  TInputLine.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);
```

In the code for a watch window's *HandleEvent* method is a test to respond to *cmFindWindow* by clearing the event:

```
case Event.Command of
  ...
  cmFindWindow: ClearEvent(Event);
  ...
end;
```

ClearEvent, remember, 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, for example, 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 be 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 will handle 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.

Help context

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.

C H A P T E R

6

Writing safe programs

Handling errors in an interactive user interface is much more complicated than in a command line utility. In a non-interactive application, it is quite acceptable (and indeed, expected) that errors cause the program to display an error message and terminate the program. In an interactive setting, however, the program needs to recover from errors and leave the user in an acceptable state. Errors should not be allowed to corrupt the information the user is working on, nor should they terminate the program, regardless of their nature. A program that meets these programming criteria can be considered "safe."

Turbo Vision facilitates writing safe programs. It promotes a style of programming that makes it easier to detect and recover from errors, especially the wily and elusive "Out of memory" error. It does this by promoting the concept of *atomic* operations.

All or nothing programming

An atomic operation is an operation that cannot be broken down into smaller operations. Or, more specific to our use, it is an operation that either completely fails, or completely succeeds. Making operations atomic is especially helpful when dealing with memory allocation.

Typically, programs allocate memory in many small chunks. For example, when constructing a dialog box, you allocate memory for the dialog box, then allocate memory for each of the controls. Each of these allocations could potentially fail, and each possible failure requires a test to see if you should proceed with the next allocation or stop. If any allocation does fail, you need to deallocate any memory allocated successfully. Ideally, you would allocate everything and then check to see if any of your allocations failed. Enter the *safety pool.*

The safety pool

Turbo Vision sets aside a fixed amount of memory (4K by default) at the end of the heap, called the safety pool. If allocating memory on the heap reaches into the safety pool, the Turbo Vision function *LowMemory* returns *True.* This indicates that further allocations are not safe and might fail.

The size of the safety pool is set by the variable LowMemSize.

For the safety pool to be effective, the pool must be as large as the largest atomic allocation. In other words, it needs to be large enough to make sure that all allocations between checks of *LowMemory* will succeed; 4K should suffice in most applications.

Using the traditional approach to memory allocation, constructing a dialog box would look something like this:

```
OK := True;
R.Assign(20,3,60,10);
D := New(Dialog, Init(R, 'My dialog'));
if D <> nil then
begin
  with D^ do
  begin
    R.Assign(2,2,32,3);
    Control := New(PStaticText, Init(R,
      'Do you really wish to do this?'));
    if Control <> nil then Insert(Control)
    else OK := False;
    R.Assign(5,5,14,7);
    Control := New(PButton, Init(R, '~Y~es', cmYes));
    if Control <> nil then Insert(Control)
    else OK := False;
    R.Assign(16,6,25,7);
    Control := New(PButton, Init(R, '~N~o', cmNo));
    if Control <> nil then Insert(Control)
    else OK := False;
    R.Assign(27,5,36,7);
    Control := New(PButton, Init(R, '~C~ancel', cmCancel));
    if Control <> nil then Insert(Control)
```

```
    else OK := False;
  end;
  if not OK then Dispose(D, Done);
end;
```

Note that the variable *OK* is used to indicate if any of the allocations failed. If any did, the whole dialog box needs to be disposed. Remember, disposing of a dialog box also disposes of all its subviews. On the other hand, with a safety pool this entire block of code can be treated as an atomic operation, changing the code to this:

```
R.Assign(20,3,60,10);
D := New(Dialog, Init(R, 'My dialog'));
with D^ do
begin
  R.Assign(2,2,32,3);
  Insert(New(PStaticText, Init(R,
   'Do you really wish to do this?')));
  R.Assign(5,5,14,7);
  Insert(New(PButton, Init(R, '~Y~es', cmYes)));
  R.Assign(16,6,25,7);
  Insert(New(PButton, Init(R, '~N~o', cmNo)));
  R.Assign(27,5,36,7);
  Insert(New(PButton, Init(R, '~C~ancel', cmCancel)));
end;
if LowMemory then                    { check if we hit the safety pool }
begin
  Dispose(D, Done);
  OutOfMemory;                            { report out of memory error }
  DoIt := False;
end
else
  DoIt := Desktop^.ExecView(D) = cmYes;
```

Since the safety pool is large enough to allocate the entire dialog box, which takes up much less than 4k, the code can assume that all the allocations succeeded. After the dialog box is completely allocated, the *LowMemory* variable is checked, and if *True*, the entire dialog box is disposed of; otherwise, the dialog box is used.

The ValidView method

Since the *LowMemory* check is done quite often, *TApplication* has a method called *ValidView* that can be called to perform the necessary check. Using *ValidView*, the **if** test in the last eight lines of the code can be condensed into two:

```
DoIt := (ValidView(D) <> nil) and
  (Desktop^.ExecView(D) = cmYes);
```

ValidView returns either a pointer to the view passed or **nil** if the view was invalid. If *LowMemory* returns *True, ValidView* takes care of disposing the view in question and calling *OutOfMemory*.

Non-memory errors

Of course, not all errors are memory related. For example, a view could be required to read a disk file for some information, and the file might be missing or invalid. This type of error must also be reported to the user. Fortunately, *ValidView* has a "hook" built in for handling non-memory errors: It calls the view's *Valid* method.

TView.Valid returns *True* by default. *TGroup.Valid* only returns *True* if *all* the subviews owned by the group return *True* from their *Valid* functions. In other words, a group is valid if all the subviews of the group are valid. When you create a view that may encounter non-memory errors, you will need to override *Valid* for that view to return *True* only if it has been successfully instantiated.

Valid can be used to indicate that a view should not be used for any reason; for example, if the view could not find its file. Note that what *Valid* checks for and *how* it checks are entirely up to you. A typical *Valid* method would look something like this:

```
function TMyView.Valid(Command: Word): Boolean;
begin
  Valid := True;
  if Command = cmValid then
  begin
    if ErrorEncountered then
    begin
      ReportError;
      Valid := False;
    end;
  end;
end;
```

When a view is first instantiated, its *Valid* method should be called with a *Command* parameter of *cmValid* to check for any non-memory related errors involved in the creation of the view. *ValidView(X)* calls *X.Valid(cmValid)* automatically, as well as checking the safety pool, so calling *ValidView* before using any new view is a good idea.

Valid is also called whenever a modal state terminates, with the *Command* parameter being the command that terminated the modal state (see Chapter 4, "Views"). This gives you a chance to trap for conditions like unsaved text in an editor window before terminating your application.

ErrorEncountered could be, and most likely is, a (Boolean) instance variable of the object type that is specified at the call to *Init*.

Reporting errors

Before a *Valid* method returns *False*, it should let the user know about whatever error occurred, since the view is not going to show up on the screen. This is what the *ReportError* call in the previous example does. Typically this involves popping up a message dialog box. Each individual view, then, is responsible for reporting any errors, so the program itself does not have to know how to check each and every possible condition.

This is an important advance in programming technique, because it lets you program as if things were going right, instead of always looking for things going wrong. Group objects, including applications, don't have to worry about checking for errors at all, except to see if any of the views they own were invalid, in which case the group simply disposes of itself and its subviews and indicates to *its* owner that it was invalid. *The group can assume that its invalid subview already notified the user of the problem.*

Using *Valid* allows the construction of windows and dialog boxes to be treated as atomic operations. Each subview that makes up the window can be constructed without checking for failure; if the constructor fails, it simply sets *Valid* to *False*. The window then goes through its entire construction, at which point the entire window can be passed to *ValidView*. If any of the subviews of the window are invalid, the entire window returns *False* from the valid check. *ValidView* will dispose of the window and return **nil**. All that needs to be done is to check the return result from *ValidView*.

Major consumers

The *Valid* function can also handle *major consumers*, which are views that allocate memory greater than the size of the safety pool, such as reading the entire contents of a file into memory. Major consumers should check *LowMemory* themselves, instead of waiting until they have finished all construction and then allowing *ValidView* to do so for them.

If a major consumer runs out of memory in the middle of constructing itself, it sets a flag in itself that indicates that it encountered an error (such as the *ErrorEncountered* flag in the earlier example) and stops trying to allocate more memory. The flag would be checked in *Valid* and the view would call *Application^.OutOfMemory* and return *False* from the *Valid* call.

Obviously, this is not quite as nice as being able to assume that your constructors work, but it is the only way to manage the construction of views that exceed the size of your safety pool.

The program FILEVIEW.PAS included on the Turbo Pascal distribution disks demonstrates the use of these techniques to implement a safe file viewer.

C H A P T E R

7

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 only provides you with 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 TVGUID17.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 *TVGUID17* 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 you're going 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;

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;
```

Keys must be typecast because they are untyped pointers.

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 TVGUID18.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 TVGUID19.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 de-allocating 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 TVGUID20.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 TVGUID20.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.

C H A P T E R

8

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
  TWindow.Load(S);                        { load the ancestor type }
  S.Read(Painted, SizeOf(Boolean));       { read additional fields }
end;

procedure TMagritte.Store(var S: Stream);
begin
  TWindow.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*.

Turbo Vision registers all the standard objects, so you don't have to.

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 7, "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 TVGUID20.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
  TGraphObject.Store(S);
  S.Write(Radius, SizeOf(Radius));
end;

procedure TGraphRect.Store(var S: TStream);
begin
  TGraphObject.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 (TVGUID22.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 TVGUID21.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 TVGUID22.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 Load(var S: TStream);
begin
  TWindow.Load(S);
  GetSubViewPtr(S, Button);
end;

procedure Store(var S: TStream);
begin
  TWindow.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.

Storing and loading the desktop

If the object you save to a stream is the desktop, the desktop will in turn save everything it owns: the entire desktop environment, including all current views.

If you intend to let the user save the desktop, you need to ensure that all possible views have proper *Store* and *Load* methods, and that all views are registered, since what the desktop contains at any moment will most likely be up to the user.

To do this, you can use something like the following code:

```
procedure TMyApp.RestoreDeskTop;
var
  SaveFile: TBufStream;
  Temp: PDeskTop;
begin
  SaveFile.Init('T.DSK',stOpen, 1024);          { Open a buffered file }
  Temp := PDeskTop(SaveFile.Get);              { Read a desktop object }
  SaveFile.Done;                                       { Close the file }
  if Temp <> nil then                         { If we got something... }
  begin
    Dispose(DeskTop, Done);              { ...get rid of the old desktop }
    DeskTop := Temp;               { ...assign the one we read to DeskTop }
    Insert(DeskTop);              { ...and insert it into the application }
    DeskTop^.DrawView;                             { Show us what we got! }
  end;
  if SaveFile.Status <> 0 then ErrorReadingFile;
end;
```

You can even go a step further and save and restore whole applications. A *TApplication* object can save and restore itself.

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 9, "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 14, "Global 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.

C H A P T E R

9

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 *Inits* in a separate program. *Inits* 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 *Inits*. 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.REZ containing a single resource: a status line with the key 'Waldo.'

```
program BuildResource;

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.REZ', 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:

```
program MyApp;

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.REZ';
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 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.

C H A P T E R

10

Hints and tips

This chapter contains a few additional suggestions on how to use Turbo Vision more effectively. Because object-oriented programming and event-driven programming are fairly new concepts to even experienced programmers, we want to try to provide some guidance in using these new paradigms.

Debugging Turbo Vision applications

If you have tried stepping or tracing through any of the example programs provided in this cookbook, you have probably noticed that you don't get very far. Because Turbo Vision programs are event-driven, much (or even most) of the program's time is spent running through a rather tight loop in *TGroup.Execute,* waiting for some sort of event to occur. As a result, stepping and tracing is not very meaningful at that point.

☞ The key to debugging Turbo Vision applications is breakpoints, breakpoints, and breakpoints.

Let's look at how well-placed breakpoints can help you find problems in Turbo Vision programs.

It doesn't get there

One problem in debugging your application might be that some portion of your code is not being executed. For example, you might click on a status line item or select a menu option that you *know* is supposed to bring up a window, but it doesn't.

Your normal instinct might tell you to step through your program until you get to that command, and then figure out where execution *does* go instead of where you expected. But if you try it, it doesn't help. You step, and you end up right back where you were.

The best approach in this situation is to set a breakpoint in the *HandleEvent* method that should be calling the code that isn't getting executed. Set the breakpoint at the beginning of the *HandleEvent* method and when it breaks, inspect the event record that's being processed to make sure it's the event you expected. At this point you can also start stepping through your code, because the *HandleEvent* and any code responding to your own commands will be code you have written, and therefore code you can trace through.

Hiding behind a mask

Keep in mind, however, that there are a couple of reasons why your object may never get to see the event you intend it to handle. The first and simplest mistake is leaving a type of event out of your object's event mask. If you haven't told your object that it is allowed to handle a certain kind of event, it won't even look at those events!

Stolen events

A second possibility you need to consider is that some other object is "stealing" the event. That is, the event is being handled and cleared by some object other than the one you intended to give it to.

There are a couple of things which could cause this. The first is duplicate command declarations: if two commands have been assigned the same constant value, they could be handled interchangeably. This is why it is *crucial* to keep track of which constants you have assigned which values, particularly in a situation when you are reusing code modules.

Another possible cause of this would be duplicate command labels, particularly in reused code. Thus, if you assign a command *cmJump*, and there is a *HandleEvent* method in some other object that already responds to a command *cmJump* that you have forgotten about and never deleted, you could have conflicts. Always check to see if some other object is handling the events that seem to get "lost."

Blame your parents

Finally, check to make sure that the event isn't being handled in a call to the object's ancestor. Often, the *HandleEvent* method of a derived type will rely on the event handler of its ancestor to deal with most events, and it may be handling one that you didn't expect. Try trapping the event before the call to the ancestor's *HandleEvent*.

It doesn't do what I expect

Perhaps your window does show up, but it displays garbage, or something other than what you expected. That indicates that the event is being handled properly, but the code that responds to the event is either incorrect or perhaps overridden. In this instance, it is best to set a breakpoint in the routine that gets called when the event occurs. Once execution breaks, you can step or trace through your code normally.

It hangs

Hang bugs are among the most difficult to track down, but they *can* be found. First you might try some combination of the breakpointing methods suggested previously to narrow down just where the hang occurs. The second thing to look for is pointers being disposed of twice. This can happen when a view is disposed of by its owner, and then you try to dispose of it directly. For example:

Warning!
This code will hang your system. Do not run it! It is only an illustration.

```
var
  Bruce, Pizza: PGroup;
  R: TRect;
begin
  R.Assign(5, 5, 10, 10);
  Pizza := New(PGroup, Init(R));
  R.Assign(10, 10, 20, 20);
  Bruce := New(PGroup, Init(R));
  Bruce^.Insert(Pizza);
  Dispose(Bruce, Done);              { dispose of Bruce and subviews }
  Dispose(Pizza, Done);                 { This will hang your system }
end;
```

Disposing of the group *Bruce* also disposes of *Bruce*'s subview, *Pizza*. If you then try to dispose of *Pizza*, your program will hang.

Hangs can also be cause by such things as reading stream data into the wrong type of object and incorrectly typecasting data from collections.

Porting applications to Turbo Vision

If you want to port an existing application to Turbo Vision, your first inclination might be to try to port the Turbo Vision interface into the application, or to put a Turbo Vision layer on top of your application. This will be an exercise in frustration. Turbo Vision applications are event-driven, and most existing applications will not shift easily, if at all, to that paradigm.

Scavenge your old code

There is an easier way. By now, you know that the essence of programming a specific application in Turbo Vision is concentrated in the application's *Init*, *Draw*, and *HandleEvent* methods. The better approach to porting an existing application is first to write a Turbo Vision interface that parallels your existing one, and then scavenge your old code into your new application. Most of the scavenged code will end up in new view's *Init*, *Draw*, and *HandleEvent* methods.

You need to spend some time thinking about the essence of your application, so you can divide your interface code from the code that carries out the work of your application. This can be difficult, because you have to think differently about your application.

The job of porting will involve some rewriting to teach the new objects how to represent themselves, but it will also involve a lot of throwing away of old interface code. This shouldn't introduce a lot of new bugs, and can actually be a *fun* thing to do.

If you port an application, you will be amazed to discover how much of your code is dedicated to handling the user interface. When you let Turbo Vision work for you, a lot of the user interface work you did before will simply disappear.

We discovered how rewarding this can be when we ported Turbo Pascal's integrated environment to Turbo Vision. We scavenged the compiler, the editor, the debugger—all the various engines—

from the old user interface, and brought them into a user interface written in Turbo Vision.

Rethink your organization

Programming in this new paradigm takes some getting used to. In traditional programming, we tend to think of the program from the perspective of the code. We are the code, and the data is "out there," something on which we operate. At first glance, we might be tempted to organize a program such as Turbo Pascal's integrated environment around an editor object. After all, that's what you're doing most of the time in the environment, editing. The editor would edit, and at intervals, it would call the compiler.

But we need to make some shifts in perspective to use the true power of object-oriented programming. It makes more sense in the integrated environment to make the application itself the organizing object. When it's time to edit, the application calls up an editor. When it's time to compile, the application brings up the compiler, initializes it, and tells it what files to compile.

If the compiler hits an error, how is the user returned to the point of error in the source code? The application calls the compiler, and it gets a result back from it. If the compiler returns an error result, it also returns a file name and a line number. The application looks to see if it already has an editor open for that file, and if not, it opens it. It passes the error information, including the line number, to the editor and constructs an error message string for the editor.

There's no reason for the editor to know anything about a compiler, or the compiler to know about an editor. The center of it all is the application itself. It's the application that needs the editor and the application that needs the compiler. After all, what is an application but something that binds things together? If we had continued to look on the application as just a lump of data that should be "out there" somewhere, and we might have been tempted to put the center of the application elsewhere. We would then have had to carry a burden of excessive and strained communications among parts of the program.

All in all, the job of writing the integrated environment in Turbo Vision took a fraction of the time that writing the environment from scratch would have taken. We look forward to you discovering the same strengths when you write your next application.

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. Definitions of the bits in the *Options* word follow:

Flag values

In the diagram, *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.

As it turns out, you really don't have to worry about what the actual values of the flag bits are unless you plan to define your own, and even in that case, you really only need to be concerned that your definitions be unique. For instance, the six highest bits in the *Options* word are presently undefined by Turbo Vision. You may define any of them to mean anything to the views you derive.

Bit masks

A *mask* is simply a shorthand 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 a bit

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 would use this code:

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

Note that 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 would set two different grow mode flags at once in a scrolling view called *MyScroller*:

```
MyScroller.GrowMode := MyScroller.GrowMode or (gfGrowHiX +
gfGrowHiY);
```

Clearing a bit

Clearing a bit is just as easy as setting it. You just use a different operation. The best way to do this is actually 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 would use

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

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

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, you could check

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

Summary

The following list summarizes the bitmap operations:

Setting a bit:

```
field := field or flag;
```

Clearing a bit:

```
field := field and not flag;
```

Checking if a flag is set:

```
if field and flag = flag then ...
```

Checking if a flag is in a mask:

```
if flag and mask <> 0 then ...
```

P A R T 3

Turbo Vision Reference

C H A P T E R

11

How to use the reference

The Turbo Vision reference describes all the *standard* objects and methods in the Turbo Vision hierarchy together with the mnemonic identifiers and miscellaneous constants and records needed to develop Turbo Vision applications. It is not intended as a tutorial.

By their nature, complex libraries of objects like those in Turbo Vision have a multitude of components. In order to avoid endless repetition of material, we have put as much complete information into the alphabetical lookup sections (Chapters 13 and 14), along with other, less detailed material that allows you to see Turbo Vision's components in their hierarchical and physical relationships, with references to the more detailed information.

How to find what you want

Chapter 12, "Unit cross reference" describes the various units that comprise Turbo Vision. It includes lists of all the types, constants, variables, procedures and functions declared in each unit.

Chapter 13, "Object reference," is an alphabetical lookup chapter for all the Turbo Vision standard object types, including all their fields and methods.

Chapter 14, "Global reference," is an alphabetical lookup chapter for all the global constants, variables, procedures and functions in

Turbo Vision. In general, if it's not an object or a part of an object, you'll find it listed here.

Keep in mind that the lookup chapter only covers the aspects of each object that are particular to it. Most of the objects will have fields and methods inherited from other objects. Thus, if you want to find a method for an object, check that object first. If you don't find the method listed for that object, check it's immediate ancestor object type. There is a diagram at the beginning of the entry for each object that depicts its relationships to its ancestors and immediate descendants.

Objects in general

Remember that each object (apart from the base object *TObject*, and the two special objects *TPoint* and *TRect*) inherits the fields and methods of its parent object. New objects that you derive will also inherit their parents' methods and fields. Many of the standard objects have *abstract* methods which *must* be overridden by your derived objects. Other methods are marked *virtual*, meaning that you will normally want to override them. There are other methods that provide useful default actions in the absence of overrides.

Naming conventions

All the standard Turbo Vision object types have a set of names using a mnemonic set of prefixes. The first letter of the identifier tells you whether you are dealing with the object type, a pointer to it, its stream registration record, or its color palette.

- Object types start with **T**: *TObject*
- Pointers to objects start with **P**: *PObject* = ^*TObject*
- Stream registration records start with **R**: *RObject*
- Color palettes start with **C**: *CObject*

All Turbo Vision constants have two-letter mnemonic prefixes that indicate their usage.

Table 11.1
Turbo Vision constant prefixes

Prefix	Meaning	Example
ap	Application palette	*apColor*
bf	Button flag	*bfNormal*
cm	Command	*cmQuit*
co	Collection code	*coOverFlow*
dm	Drag mode	*dmDragGrow*
ev	Event constant	*evMouseDown*
gf	Grow mode flag	*gfGrowLoX*
hc	Help context	*hcNoContent*
kb	Keyboard constant	*kbAltX*
mb	Mouse button	*mbLeftButton*
of	Option flag	*ofTopSelect*
sb	Scroll bar	*sbLeftArrow*
sf	State flag	*sfVisible*
sm	Screen mode	*smMono*
st	Stream code	*stOK*
wf	Window flag	*wfMove*
wn	Window numbers	*wnNoNumber*
wp	Window palette	*wpBlueWindow*

C H A P T E R

12

Unit cross reference

This chapter describes briefly the contents of each of the modules that make up Turbo Vision. First we'll take an overview of the Turbo Vision units, then each of the units will be described in more detail.

Turbo Vision consists of nine units:

Table 12.1
Turbo Vision units

Unit	Contents
App	All object definitions for writing event-driven applications
Dialogs	Tools and controls for use in dialog boxes
Drivers	Mouse support, keyboard handler, system error handler, etc.
HistList	History lists for input lines
Memory	Memory management system
Menus	Objects for adding menus and status bars to Turbo Vision applications
Objects	Basic object definitions, including all object types for streams, collections and resources
TextView	More specialized views for presenting text
Views	Base objects for using windows in your applications: views, windows, frames, scroll bars, etc.

The Objects unit

The *Objects* unit contains the basic object definitions for Turbo Vision, including the base object for the Turbo Vision hierarchy, *TObject*, as well as all the non-visible elements of Turbo Vision: streams, collections, and resources.

Types

Type conversion records

Type	Use
FNameStr	String to hold a DOS file name
LongRec	Converts a *Longint* into low- and high-order *Words*
PChar	Pointer for dynamic character allocation
PString	Pointer for dynamic strings
PtrRec	Converts a *Pointer* value into offset and segment parts
TByteArray	Array of *Byte* values used for typecasting
TWordArray	Array of *Word* values used for typecasting
WordRec	Converts a *Word* into low- and high-order *Bytes*

Objects unit types

Type	Use
TBufStream	A buffered Turbo Vision DOS stream
TCollection	Basically a polymorphic array
TDosStream	A Turbo Vision stream on a DOS file
TEmsStream	A Turbo Vision stream in EMS memory
TItemList	An array of pointers, used by collections
TObject	Base object for the Turbo Vision hierarchy
TPoint	Object designating a point on the screen
TRect	Simple object composed of two points for defining a region on the screen
TResourceCollection	Specialized *TCollection* for resources
TResourceFile	Object for storing resources on disk
TSortedCollection	Specialized *TCollection* that sorts automatically
TStream	Basic object defining a Turbo Vision stream
TStreamRec	Stream registration record
TStrIndex	Array of *TStrIndexRec*
TStrIndexRec	Record of string indexes used by *TStrIndex*
TStringCollection	Specialized *TSortedCollection* for strings
TStringList	String list object used for string resources
TStrListMaker	Special object for constructing string lists

Constants

Stream access modes

Constant	Value	Meaning
stCreate	$3C00	Creates new file
stOpenRead	$3D00	Read access only
stOpenWrite	$3D01	Write access only
stOpen	$3D02	Read and write access

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

Maximum collection size

Constant	Value	Meaning
MaxCollectionSize	16,380	Maximum size of a *TCollection*

Collection error codes

Error code	Value	Meaning
coIndexError	–1	Index out of range
coOverflow	–2	Overflow

Variables

Variable	Type	Initial value	Meaning
EmsCurHandle	Word	$FFFF	Current EMS handle
EmsCurPage	Word	$FFFF	Current EMS page

Procedures and functions

Procedure	Operation
Abstract	Default routine for methods that must be overridden
DisposeStr	Disposes of a string created with *NewStr*
RegisterType	Registers an object type with Turbo Vision streams

Function	Operation
LongDiv	Divides a long integer by an integer
LongMul	Multiplies two integers into a long integer
NewStr	Allocates a string on the heap

The Views unit

The *Views* unit contains the basic elements of views, the visible portions of Turbo Vision. Included are both abstract types such as *TView* and *TGroup* and useful components of more complex groups, such as window frames and scroll bars. More complex visible elements are found in the *Dialogs* and *TextView* units.

Types

Type	Use
TCommandSet	Allows groups of commands to be enabled or disabled
TDrawBuffer	Buffer used by draw methods
TFrame	Frame object used by windows
TGroup	Abstract object for complex views
TListViewer	Base type for list boxes and such
TPalette	Color palette type used by all views
TScrollBar	Object defining a scroll bar
TScrollChars	Scroll bar component characters
TScroller	Base object for scrolling text windows
TTitleStr	Title string used by *TFrame*
TVideoBuf	Video buffer used by screen manager
TView	Abstract object; base of all visible objects
TWindow	Base object for resizable windows

Constants

TView State masks

Constant	Value	Meaning
sfVisible	\$0001	View is visible
sfCursorVis	\$0002	View has visible cursor
sfCursorIns	\$0004	View's cursor is block for insert mode
sfShadow	\$0008	View has a shadow
sfActive	\$0010	View is, or is owned by, the active window
sfSelected	\$0020	View is owner's selected view
sfFocused	\$0040	View has the focus
sfDragging	\$0080	View is being dragged
sfDisabled	\$0100	View is disabled
sfModal	\$0200	View is in modal state
sfExposed	\$0800	View is attached to the application

Views unit constants

Constant	Value	Meaning
hcNoContext	0	Neutral help context code
hcDragging	1	Help context while view is dragged
MaxViewWidth	132	Maximum width in characters of a view
wnNoNumber	0	*TWindow* number constant

TView Option masks

Constant	Value	Meaning
ofSelectable	$0001	View can be selected
ofTopSelect	$0002	Selecting view moves it to top of owner's subviews
ofFirstClick	$0004	Mouse click selects *and* performs action
ofFramed	$0008	View has a visible frame
ofPreProcess	$0010	View sees focused events before focused view
ofPostProcess	$0020	View sees focused events after focused view
ofBuffered	$0040	Group should have a cache buffer
ofTileable	$0080	View can be tiled on the desktop
ofCenterX	$0100	Center view horizontally within owner
ofCenterY	$0200	Center view vertically within owner
ofCentered	$0300	Center view both horizontally and vertically within owner

TView GrowMode masks

Constant	Value	Meaning
gfGrowLoX	$01	Left side follows owner's right side
gfGrowLoY	$02	Top follows owner's bottom
gfGrowHiX	$04	Right side follows owner's right side
gfGrowHiY	$08	Bottom follows owner's bottom
gfGrowAll	$0F	View follows owner's lower-right corner
gfGrowRel	$10	Keep relative size when screen size changes

TView DragMode masks

Constant	Value	Meaning
dmDragMove	$01	View can move
dmDragGrow	$02	View can change size
dmLimitLoX	$10	View's left side cannot move outside *Limits*
dmLimitLoY	$20	View's top cannot move outside *Limits*
dmLimitHiX	$40	View's right side cannot move outside *Limits*
dmLimitHiY	$80	View's bottom cannot move outside *Limits*
dmLimitAll	$F0	No part of view can move outside *Limits*

Scroll bar part codes

Constant	Value	Meaning
sbLeftArrow	0	Horizontal bar's left arrow
sbRightArrow	1	Horizontal bar's right arrow
sbPageLeft	2	Horizontal bar's left paging area
sbPageRight	3	Horizontal bar's right paging area
sbUpArrow	4	Vertical bar's top arrow
sbDownArrow	5	Vertical bar's bottom arrow
sbPageUp	6	Vertical bar's upward paging area
sbPageDown	7	Vertical bar's downward paging area
sbIndicator	8	Scroll bar indicator tab

Window flag masks

Constant	Value	Meaning
wfMove	$01	Window frame's top line can move window
wfGrow	$02	Window frame has resize corner
wfClose	$04	Window frame has close icon
wfZoom	$08	Window frame has zoom icon

TWindow palette entries

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

Standard view commands

Constant	Value	Meaning
cmReceivedFocus	50	View has recieved focus
cmReleasedFocus	51	View has released focus
cmCommandSetChanged	52	Command set has changed
cmScrollBarChanged	53	Scroll bar has changed value
cmScrollBarClicked	54	Scroll bar was clicked on
cmSelectWindowNum	55	User wants to select a window by number
cmRecordHistory	56	History list should save contents of input line

Variables

Variable	Type	Initial value	Meaning
MinWinSize	*TPoint*	(X: 16; Y: 6)	Minimum window size
ShadowSize	*TPoint*	(X: 2; Y: 1)	Window shadow size
ShadowAttr	*Byte*	$08	Window attribute

Function

Function	Operation
Message	Sends user-defined messages between views

The Dialogs unit

The *Dialogs* unit defines most of the elements most often used in constructing dialog boxes. These include dialog boxes themselves (which are specialized windows) as well as various controls such as buttons, labels, check boxes, radio buttons, input lines and history lists.

Types

Type	Use
TButton	Pushbuttons to generate commands
TCheckBoxes	Clusters of on/off toggle switches
TCluster	Abstract type for check boxes and radio buttons
TDialog	Specialized window for dialog boxes
THistory	List of previous entries for an input line
TInputLine	Text input editor
TLabel	Smart label for a cluster or an input line
TListBox	Scrollable list for user choices
TParamText	Formatted static text
TRadioButtons	Cluster of buttons, only one of which may be pressed at a time
TSItem	String items in a linked list, used by clusters
TStaticText	Plain text

Constants

Button flags

Constant	Value	Meaning
bfNormal	$00	Button is a normal button
bfDefault	$01	Button is the default button
bfLeftJust	$02	Button text should be left-justified

Procedures and functions

Function	Operation
NewSItem	Creates a new string item for a list box

Procedure	Operation
RegisterDialogs	Registers all objects in the *Dialogs* unit for use with streams

The App unit

The *App* unit (provided in source code form) provides the elements of the Turbo Vision application framework. Four very powerful object types are defined in *App*, including the *TProgram* and *TApplication* objects which actually serve as Turbo Vision programs, and the desktop object that controls most of the other elements in a windowing application.

Types

Type	Use
TApplication	Application object with event manager, screen manager, error handling, and memory management
TBackground	Colored background for desktop
TDeskTop	Group object to hold windows and dialog boxes
TProgram	Abstract application object

Variables

Variable	Type	Initial value	Meaning
Application	*PProgram*	**nil**	Pointer to current application
DeskTop	*PDeskTop*	**nil**	Pointer to current desktop
StatusLine	*PStatusLine*	**nil**	Pointer to current status line
MenuBar	*PMenuView*	**nil**	Pointer to current menu bar

The Menus unit

The *Menus* unit provides all the objects and support routines for the Turbo Vision menuing systems, including pull-down and pop-up menus and active status line items.

Types

Type	Use
TMenu	Linked list of *TMenuItem* records
TMenuBar	Horizontal menu header, connected to menus
TMenuBox	Pull-down or pop-up menu box
TMenuItem	Record linking a label text, a hot key, a command, and a help context for use within a menu
TMenuStr	String type for menu labels
TMenuView	Abstract object type for menu bars and menu boxes
TStatusDef	Record linking a range of help contexts with a list of status line items
TStatusItem	Record linking a label text, a hot key, and a command for use on a status line
TStatusLine	Message line for the bottom of the application screen, including a list of *TStatusDef* records

Procedures and functions

TMenuItem functions

Function	Operation
NewItem	Creates a new menu item
NewLine	Creates a line across a menu box
NewSubMenu	Creates a menu off a menu bar or menu box

TMenu routines

Routine	Operation
NewMenu function	Allocates a menu on the heap
DisposeMenu procedure	Deallocates menu from heap

TStatusLine functions

Function	Operation
NewStatusDef	Defines a range of help contexts and a pointer to a list of status items
NewStatusKey	Defines a status line item and binds it to a command and an optional hot key

The Drivers unit

The *Drivers* unit contains all the specialized drivers used by Turbo Vision, including mouse and keyboard drivers, video support, and system error handling along with the event manager for event-driven programs.

Types

Type	Use
TEvent	Event record type
TSysErrorFunc	System error handler function type

Constants

Mouse button state masks

Constant	Value	Meaning
mbLeftButton	$01	Left mouse button
mbRightButton	$02	Right mouse button

Event codes

Constant	Value	Meaning
evMouseDown	$0001	Mouse button pressed
evMouseUp	$0002	Mouse button released
evMouseMove	$0004	Mouse changed location
evMouseAuto	$0008	Automatic mouse repeat event
evKeyDown	$0010	Event is a keystroke
evCommand	$0100	Event is a command
evBroadcast	$0200	Event is a broadcast

Event masks

Constant	Value	Meaning
evNothing	$0000	Event has been cleared
evKeyboard	$0010	Event came from keyboard
evMouse	$000F	Event came from mouse
evMessage	$FF00	Event is a message or command

Keyboard state and shift masks

Constant	Value	Meaning
kbRightShift	$0001	Right shift key pressed
kbLeftShift	$0002	Left shift key pressed
kbCtrlShift	$0004	Ctrl and shift keys pressed
kbAltShift	$0008	Alt and shift keys pressed
kbScrollState	$0010	Scroll lock set
kbNumState	$0020	Num lock set
kbCapsState	$0040	Caps lock set
kbInsState	$0080	Insert mode on

Standard command codes

Command	Value	Meaning
cmValid	0	Check validity of a new view
cmQuit	1	Terminate the application
cmError	2	Undefined
cmMenu	3	Move focus to menu bar
cmClose	4	Close the current window
cmZoom	5	Zoom (or unzoom) a window
cmResize	6	Resize a window
cmNext	7	Make the next window active
cmPrev	8	Make the previous window active

TDialog standard commands

Command	Value	Meaning
cmOK	10	Ok button pressed
cmCancel	11	Cancel button or *Esc* key pressed
cmYes	12	Yes button pressed
cmNo	13	No button pressed
cmDefault	14	Default button or *Enter* pressed

Screen modes

Constant	Value	Meaning
smBW80	$0002	Black and white screen mode
smCO80	$0003	Color screen mode
smMono	$0007	Monochrome screen mode
smFont8x8	$0100	43- or 50-line mode (EGA/VGA)

Variables

Initialized variables

Variable	Type	Initial value	Meaning
ButtonCount	Byte	0	Number of buttons on the mouse
MouseEvents	Boolean	False	Indicates whether a mouse was detected
DoubleDelay	Word	8	Maximum delay time between double clicks
RepeatDelay	Word	8	Delay between automatic mouse repeats

Uninitialized variables

Variable	Type	Meaning
MouseIntFlag	Byte	Internal use only
MouseButtons	Byte	Which button was pressed
MouseWhere	*TPoint*	Position of the mouse cursor
StartupMode	Word	Screen mode when program was started
ScreenMode	Word	Current screen mode
ScreenWidth	Byte	Width of screen in columns
ScreenHeight	Byte	Height of screen in lines
CheckSnow	Boolean	Determines whether to slow output for CGA adapters
HiResScreen	Boolean	Screen can display 43 or 50 lines (EGA/VGA)
ScreenBuffer	Pointer	Pointer to video screen buffer
CursorLines	Word	Beginning and ending scan lines, for setting cursor type

System error handler variables

Variable	Type	Initial value	Meaning
SysErrorFunc	*TSysErrorFunc*	*SystemError*	Function called by the system error manager when a system error occurs
SysColorAttr	Word	$4E4F	Video attributes for error messages on color screen
SysMonoAttr	Word	$7070	Video attributes for error messages on monochrome screen
CtrlBreakHit	Boolean	*False*	Indicates whether user pressed *Ctrl-Break*
SaveCtrlBreak	Boolean	*False*	Status of *Ctrl-Break* checking at startup of program

Procedures and functions

Event manager procedures

Procedure	Operation
InitEvents	Initializes the event manager
DoneEvents	Shuts down the event manager
ShowMouse	Displays the mouse cursor
HideMouse	Hides the mouse cursor
GetMouseEvent	Creates event record from mouse action
GetKeyEvent	Creates event record from keyboard input

Screen manager procedures

Procedure	Operation
InitVideo	Initializes the screen manager
DoneVideo	Shuts down the screen manager
SetVideoMode	Selects screen mode (color, black & white, monochrome, high resolution)
ClearScreen	Clears the screen in any video mode

Default system error handler function

Function	Operation
SystemError	Displays an error message on the bottom line of the screen and prompts for abort or retry

System error handler procedures

Procedure	Operation
InitSysError	Initializes the system error manager
DoneSysError	Shuts down the system error manager

Keyboard support functions

Function	Operation
GetAltChar	Returns character from keyboard
GetAltCode	Returns scan code from keyboard

String formatting procedure

Procedure	Operation
FormatStr	Formats a string and the parameters passed with it

Buffer move procedures

Procedure	Operation
MoveBuf	Moves a buffer into another buffer
MoveChar	Moves one or more copies of a character into a buffer
MoveCStr	Moves a control string into a buffer
MoveStr	Moves a string into a buffer

String length function

Function	Operation
CStrLen	Returns length of a control string, ignoring tildes

Driver initialization

Procedure	Operation
InitDrivers	Initialize drivers unit

The TextView unit

The *TextView* unit contains several specialized views for displaying text in a scrolling window.

Types

Type	Use
TTerminal	TTY-like scrolling text device
TTerminalBuffer	Circular text buffer for *TTerminal*
TTextDevice	Abstract text device object

Procedure

Procedure	Operation
AssignDevice	Assigns a text file device for input and/or output

The Memory unit

The *Memory* unit contains Turbo Vision's memory management routines, which provide heap management functions that facilitate safe programming.

Variables

Variable	Type	Initial value	Meaning
LowMemSize	Word	4096 **div** 16	Size of safety pool
MaxBufMem	Word	65536 **div** 16	Maximum memory for cache buffers

Procedures and functions

Procedure	Operation
DoneMemory	Shuts down the memory manager
FreeBufMem	Deallocate cache buffer for a group
GetBufMem	Allocate cache buffer for a group
InitMemory	Initializes the memory manager
SetMemTop	Sets top of application's memory block

Function	Operation
LowMemory	Indicates whether safety pool has been eaten into
MemAlloc	Allocates memory with safety pool check
MemAllocSeg	Allocates segment-aligned memory block

The HistList unit

The *HistList* unit contains all the variables, procedures and functions needed to implement history lists.

Variables

Variable	Type	Initial value	Meaning
HistoryBlock	Pointer	**nil**	Memory buffer to hold all history list items
HistorySize	Word	1024	Size of history block
HistoryUsed	Word	0	Offset into history block indicating amount of block used

Procedures and functions

Procedure	Operation
ClearHistory	Clears all history lists
DoneHistory	Shuts down the history list manager
HistoryAdd	Adds a string to a history list
InitHistory	Initialized the history list manager

Function	Operation
HistoryCount	Returns the number of strings in a history list
HistoryStr	Returns a particular string from a history list

C H A P T E R

13

Object reference

This chapter contains an alphabetical listing of all the standard Turbo Vision object types, with explanations of their general purposes and usage, their fields, methods and color palettes.

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.

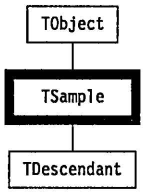
To save you some hunting, all fields and methods are indexed.

For example, if you want to know about the *Owner* field of a *TLabel* object, you might first look under *TLabel*'s fields, where you won't find *Owner* listed. You would then check *TLabel*'s immediate ancestor in the hierarchy, *TStaticText*. Again, *Owner* will not be listed. You would next check *TStaticText*'s immediate ancestor, *TView*. There you will find complete information about *Owner*, which is inherited unchanged by *TLabel*.

Each object's entry in this chapter has a graphical representation of the object's ancestors and immediate descendants, so it should be easy for you to find the objects from which fields and methods are inherited.

Each object's entry is laid out in the following format:

TSample object — Object's unit



Fields

This section will list all fields for each object, alphabetically. In addition to showing the declaration of the field and an explanation of its use, there is a **Read only** or **Read/write** designation. Read-only fields are generally fields that are set up and maintained by the object's methods, and they should *not* be on the left side of an assignment statement.

AField

```
AField: SomeType;
```

Read only

AField is a field that holds some information about this sample object. This text explains how it functions, what it means, and how you use it.

See also: related fields, methods, objects, global functions, etc.

AnotherField

```
AnotherField: Word;
```

Read/write

AnotherField has similar information to that for *AField*.

Methods

This section lists all methods which are either newly defined for this object or which override inherited methods. For virtual methods, an indication will be given as to how often you will probably need to override the method: Never, Seldom, Sometimes, Often, or Always.

Init

```
constructor Init(AParameter: SomeType);
```

Init creates a new sample object, setting the *AField* field to *AParameter*.

Zilch

```
procedure Zilch; virtual;
```

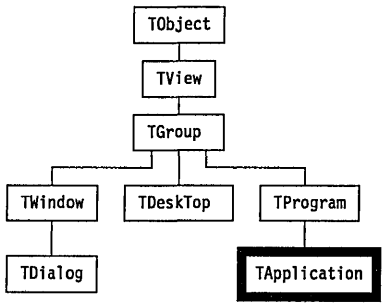
Override: Sometimes

The *Zilch* procedure causes the sample object to perform some action.

See also: *TSomethingElse.Zilch*

TApplication

App



TApplication is a simple "wrapper" around *TProgram,* and only differs from *TProgram* in its constructor and destructor methods. *TApplication.Init* first initializes all Turbo Vision subsystems (the memory, video, event, system error, and history list managers) and then calls *TProgram.Init.* Likewise, *TApplication.Done* first calls *TProgram.Done* and then shuts down all Turbo Vision subsystems.

Normally you will want to derive your own applications 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.

Methods

Init `constructor Init;`

The actual implementation of *TApplication.Init* is shown below:

```
constructor TApplication.Init;
begin
  InitMemory;
  InitVideo;
  InitEvents;
  InitSysError;
  InitHistory;
  TProgram.Init;
end;
```

See also: *TProgram.Init*

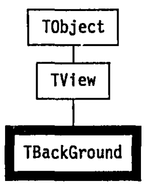
Done `destructor Done; virtual;`

Override: Sometimes

The actual implementation of *TApplication.Done* is shown below:

```
destructor TApplication.Done;
begin
  TProgram.Done;
  DoneHistory;
  DoneSysError;
  DoneEvents;
  DoneVideo;
  DoneMemory;
end;
```

TBackground App



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 *TBackground* object with the given *Bounds* by calling *TViewInit*. *GrowMode* is set to *gfGrowHiX* + *gfGrowHiY*, and the *Pattern* field is set to *APattern*.

See also: *TView.Init, TBackground.Pattern*

Load `constructor Load(var S: TStream);`

Creates a *TBackground* object and loads it from the stream *S* by calling *TView.Load* and then reading the *Pattern* field.

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*.

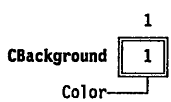
Store `procedure Store(var S: TStream);`

Stores the *TBackground* view on the stream by calling *TView.Store* and then writing the *Pattern* field.

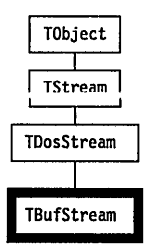
See also: *TView.Store, TBackground.Load*

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

Buffer `Buffer: Pointer;` **Read only**

A pointer to the start of the stream's buffer

BufSize `BufSize: Word;` **Read only**

The size of the buffer in bytes

BufPtr `BufPtr: Word;` **Read only**

An offset from the *Buffer* pointer indicating the current position within the buffer.

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.

Methods

Init `constructor Init(FileName: FNameStr; Mode, Size: Word);`

Creates and opens the named file with access mode *Mode* by calling *TDosStream.Init*. Also creates a buffer of *Size* bytes with a *GetMem* call. The *Handle, Buffer* and *BufSize* fields are suitably initialized. Typical buffer sizes range from 512 bytes to 2,048 bytes.

See also: *TDosStream.Init*

Done `destructor Done; virtual;`

Override: Never Closes and disposes of the file stream; flushes and disposes of its buffer.

See also: *TBufStream.Flush*

Flush `procedure Flush; virtual;`

Override: Never Flushes the calling file stream's buffer provided the stream is *stOK*.

See also: *TBufStream.Done*

TB

GetPos `function GetPos: Longint; virtual;`

Override: Never Returns the value of the calling 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 calling stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never If *stOK*, reads *Count* bytes into the *Buf* buffer starting at the calling stream's current position.

Note that *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.GetPos, TBufStream.GetPos*

Truncate `procedure Truncate; virtual;`

Override: Never Flushes the buffer then deletes all data on the calling stream from the current position to the end. The current position is set to the new end of the stream.

See also: *TBufStream.GetPos, TBufStream.Seek*

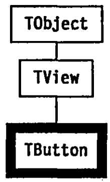
Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Never If *stOK*, writes *Count* bytes from the *Buf* buffer to the calling stream, starting at the current position.

Note that *Buf* is *not* the stream's buffer, but an external buffer to hold the data being written to the stream. When *Write* is called, *Buf* will point to the variable whose value is being written.

See also: *TBufStream.Read, stWriteError*

TButton — Dialogs



A *TButton* object is a box with a title and a shadow that generates a command when pressed. These are the buttons that are used extensively in the IDE (e.g., OK and Cancel on dialog boxes). A button can be selected 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 on the button with a mouse.

With color and black & white palettes, a button has a three-dimensional look that moves when selected. On monochrome systems, a button is bordered by brackets, and other ASCII characters are used to indicate whether the button is default, selected, etc.

Like the other controls defined in the *Dialogs* unit, *TButton* is a "terminal" object. It can be inserted into any group and is intended for use without having to override any of its methods.

A button is initialized by passing it a *TRect*, a title string, the command to generate when the button is pressed, and byte of flags. To define a shortcut key for the button, the title string may contain tildes (~) around one of its characters, which becomes the shortcut. The *AFlag* parameter indicates whether the title should be centered or left justified, and whether or not the button should be the default (and therefore selectable by *Enter*).

There can only be one default button in a window or dialog at any given time. Buttons that are peers in a group grab and release the default state via *evBroadcast* messages. Buttons can be enabled or disabled using *SetState* and the *CommandEnabled* methods.

Fields

Title `Title: PString;` **Read only**

A pointer to the button label's text.

Command `Command: Word;` **Read only**

The command word of the event generated when this button is pressed.

See also: *TButton.Init, TButton.Load*

Flags `Flags: Byte;` **Read/write**

Flags is a bitmapped field used to indicate whether button text is left-justified or centered. The individual flags are described in Chapter 14, under "bf*XXXX* button flag constants."

See also: *TButton.Draw,* bf*XXXX* button flag constants

AmDefault `AmDefault: Boolean;` **Read only**

If *True,* the button is the default (and therefore selected when *Enter* is pressed). Otherwise the button is "normal."

See also: bf*XXXX* button flag constants

Methods

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr; ACommand: Word;
AFlags: Byte);
```

Creates a *TButton* object with the given size by calling *TView.Init*. *NewStr(ATitle)* is called and assigned to *Title*. *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-justified 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.

See also: *TView.Init,* bf*XXXX* button flag constants

Load

```
constructor Load(var S: TStream);
```

Creates a *TButton* object and initializes it from the given stream by calling *TView.Load(S)*. Other fields are set via *S.Read* calls, and *State* is set according to whether the command in the *Command* field is enabled. Used in conjunction with *TButton.Store* to save and retrieve *TButton* objects on a *TStream*.

See also: *TView.Load, TButton.Store*

Done

Override: Never

```
destructor Done; virtual;
```

Disposes the memory assigned to the button's *Title*, then calls *TView.Done* to destroy 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.

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 button is pressed, a command event is generated with *TView.PutEvent*, with the *TButton.Command* field assigned to *Event.Command* and *Event.InfoPtr* set 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.

See also: *TView.HandleEvent*

MakeDefault `procedure MakeDefault(Enable: Boolean);`

This method does nothing if the button is already the default button. Otherwise, the button's *Owner* is told of the change in the button's default status. If *Enable* is *True* the *cmGrabDefault* command is broadcast, otherwise the *cmReleaseDefault* is broadcast. The button is redrawn to show the new status.

See also: *TButton.AmDefault*, *bfDefault*

SetState `procedure SetState(AState: Word; Enable: Boolean); virtual;`

Override: Seldom

Calls *TView.SetState*, then *DrawView*'s the button if the button has been made *sfSelected* or *sfActive*. If focus is received (i.e., if *AState* is *sfFocused*), the button grabs or releases default from the default button by calling *MakeDefault*.

See also: *TView.SetState*, *TButton.MakeDefault*

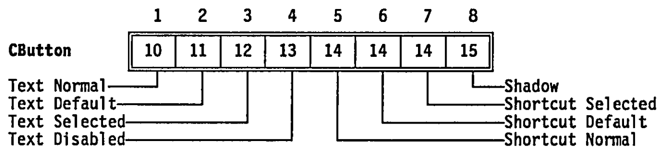
Store `procedure Store(var S: TStream);`

Stores the *TButton* object on the given *TStream* by calling *TView.Store(S)* followed by *S.Write* calls 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*

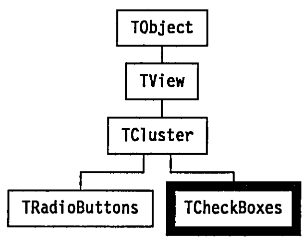
Palette

Button objects use the default palette *CButton* to map onto *CDialog* palette entries 10 through 15.



TCheckBoxes

Dialogs



TCheckBoxes is a specialized cluster of one to sixteen controls. Unlike radio buttons, any number of check boxes can be marked independently, so there is no default check box in the group. Marking can be made with mouse clicks, cursor movements, and *Alt*-letter shortcuts. Each check box can be highlighted and toggled on/off (with the *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 have been chosen by the user (the IDE, for example, has compiler/linker options selected in this way). Check box clusters are often associated with *TLabel* objects.

Fields

None apart from *Value* and *Sel*, which are inherited from *TCluster*. The *Value* word is interpreted as a set of 16 bits (0 through 15), with a 1 in the *Item*'th bit position meaning that the *Item*'th check box is marked.

Methods

Note that *TCheckBoxes* does not override the *TCluster* constructors, destructor, or event handler. Derived object types, however, may 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*

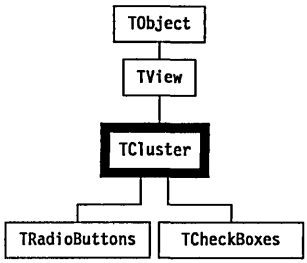
Palette

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



A cluster is a group of controls that all respond in the same way. *TCluster* is an abstract object type from which the useful group controls *TRadioButtons* and *TCheckBoxes* are derived. Cluster controls are often associated with *TLabel* objects, letting you select the control by selecting on the adjacent explanatory label.

While buttons are used to generate commands and input lines are used to edit strings, clusters are used to toggle bit values in the *Value* field, which is of type *Word*. 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

Value `Value: Word;` **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*.

Sel `Sel: Integer;` **Read only**

The currently selected item of the cluster.

Strings `Strings: TStringCollection;` **Read only**

The list of items in the cluster.

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*. In this way, an entire cluster of radio buttons or check boxes may be created 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))));
...
```

When adding additional radio buttons or check boxes to a cluster (or menus and status lines, for that matter), 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. Alternatively, just keep re-compiling and adding one additional closing parenthesis until the compiler accepts the statement.

See also: *TSItem* type

Load `constructor Load(var S: TStream);`

Creates a *TCluster* object by calling *TView.Load(S)* then setting the *Value* and *Sel* fields with *S.Read* calls. Finally the *Strings* field for the cluster is loaded from *S* with *Strings.Load(S)*. Used 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 with a *TView.Done* call.

See also: *TView.Done*

DataSize `function DataSize: Word; virtual;`

TC

Override: Seldom Returns the size of *Value*. Must be overridden in derived object types that change *Value* or add other data fields, in order to work with *GetData* and *SetData*.

See also: *TCluster.GetData, TCluster.SetData*

DrawBox `procedure DrawBox(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*

GetData `procedure GetData(var Rec); virtual;`

Override: Seldom Writes the *Value* field to the given record and *DrawView*'s the cluster. Must be overridden in derived object types that change the *Value* field, in order to work with *DataSize* and *SetData*.

See also: *TCluster.DataSize, TCluster.SetData, TView.DrawView*

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 *TView.HandleEvent* then handles all mouse and keyboard events appropriate to this cluster. Controls are selected by mouse click or cursor movement keys (including *Spacebar*). The cluster is redrawn to show the selected controls.

See also: *TView.HandleEvent*

Mark `function Mark(Item: Integer): Boolean; virtual;`

Override: Always Called by *Draw* to determine which items are marked. The default *TCluster.Mark* returns *False*. *Mark* should be overridden to return *True* if the *Item*'th control in the cluster is marked, otherwise *False*.

MovedTo `procedure MovedTo(Item: Integer); virtual;`

Override: Seldom Called by *HandleEvent* to move the selection bar to the *Item*'th control of the cluster.

Press `procedure Press(Item: Integer); virtual;`

Override: Always Called by *HandleEvent* when the *Item*'th control in the cluster is pressed either by mouse click or keyboard event. This abstract method must be overridden.

SetData `procedure SetData(var Rec); virtual;`

Override: Seldom Reads the *Value* field from the given record and *DrawView*'s the cluster. Must be overridden in derived cluster types that require other fields 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 *TView.SetState*, then *DrawView*'s the cluster if *AState* is *sfSelected*.

See also: *TView.SetState, TView.DrawView*

Store `procedure Store(var S: TStream);`

Stores the *TCluster* object on the given stream by calling *TView.Store(S)*, writing *Value* and *Sel* to *S*, then storing the cluster's *Strings* field by using its *Store* method. Used in conjunction with *TCluster.Load* to save and retrieve *TCluster* objects on a stream.

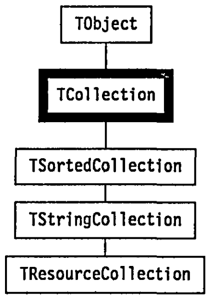
See also: *TCluster.Load, TStream.Write*

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.

```
                  1    2    3    4
              ╔════╤════╤════╤════╗
CCluster      ║ 16 │ 17 │ 18 │ 18 ║
              ╚════╧════╧════╧════╝
Text Normal────────┘    │    │    └───Shortcut Selected
Text Selected───────────┘    └────────Shortcut Normal
```

TCollection — Objects



TCollection is an abstract type for implementing any collection of items, including other objects. *TCollection* is a more general concept than the traditional array, set, or list. *TCollection* 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.

Fields

Items `Items: PItemList;` **Read only**

A pointer to an array of item pointers.

See also: *TItemList* type

Count `Count: Integer;` **Read only**

The current number of items in the collection, up to *MaxCollectionSize*.

See also: *MaxCollectionSize* variable

Limit `Limit: Integer;` **Read only**

The currently allocated size (in elements) of the *Items* list.

See also: *Delta, TCollection.Init*

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*

Methods

Init `constructor Init(ALimit, ADelta: Integer);`

Creates a collection with *Limit* set to *ALimit* and *Delta* set to *ADelta*. The initial number of items will be limited to *ALimit*, but the collection is allowed to 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);`

Creates and loads a collection from the given stream. *TCollection.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 *TCollection.FreeAll* and setting *Limit* to 0

See also: *TCollection.FreeAll, TCollection.Init*

At `function At(Index: Integer): Pointer;`

Returns a pointer to the item indexed by *Index* in the collection. This method lets you treat a collection as an indexed array. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*, and a value of **nil** is returned.

See also: *TCollection.IndexOf*

AtDelete `procedure AtDelete(Index: Integer);`

Deletes the item at the *Index*'th position and moves the following items up by one position. *Count* is decremented by 1, but the memory allocated to the collection (as given by *Limit*) is not reduced. If *Index* is less than zero

TC

or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*.

See also: *TCollection.FreeItem, TCollection.Free, TCollection.Delete*

AtInsert

```
procedure AtInsert(Index: Integer; Item: Pointer);
```

Inserts *Item* at the *Index*'th position and moves the following items down by one position. If *Index* is less than zero or greater than *Count*, the *Error* method is called with an argument of *coIndexError* and the new *Item* is not inserted. If *Count* is equal to *Limit* before the call to *AtInsert*, the allocated size of the collection is expanded by *Delta* items using a call to *SetLimit*. If the *SetLimit* call fails to expand the collection, the *Error* method is called with an argument of *coOverflow* and the new *Item* is not inserted.

See also: *TCollection.At, TCollection.AtPut*

AtPut

```
procedure AtPut(Index: Integer; Item: Pointer);
```

Replaces the item at index position *Index* with the item given by *Item*. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called 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))*.

See also: *TCollection.AtDelete, TCollection.DeleteAll*

DeleteAll

```
procedure DeleteAll;
```

Deletes all items from the collection by setting *Count* to zero.

See also: *TCollection.Delete, TCollection.AtDelete*

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Override: Sometimes

Called whenever a collection error is encountered. By default, this method produces a run-time error of (212 – *Code*).

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*. The result is the item pointer for which *Test* returned *True*, or **nil** if the *Test* function 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

```
function 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 and disposes of the given *Item.* Equivalent to

```
Delete(Item);
FreeItem(Item);
```

See also: *TCollection.FreeItem, TCollection.Delete*

FreeAll

```
procedure FreeAll;
```

Deletes and disposes of all items in the collection.

See also: *TCollection.DeleteAll*

FreeItem

```
procedure FreeItem(Item: Pointer); virtual;
```

TC

Override: Sometimes

The *FreeItem* method must dispose the given *Item*. The default *TCollection.FreeItem* assumes that *Item* is a pointer to a descendant of *TObject*, and thus calls the *Done* destructor:

```
if Item <> nil then Dispose(PObject(Item), Done);
```

FreeItem is called by *Free* and *FreeAll*, but it should never be called directly.

See also: *TCollection.Free*, *TCollection.FreeAll*

GetItem

```
function TCollection.GetItem(var S: TStream): Pointer; virtual;
```

Override: Sometimes

Called by *TCollection.Load* for each item in the collection. This method can be overridden but should not be called directly. The default *TCollection.GetItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Get* to load the item:

```
GetItem := S.Get;
```

See also: *TStream.Get*, *TCollection.Load*, *TCollection.Store*

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Override: Never

Returns the index of the given *Item*. The converse operation to *TCollection.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, insertions are made at the end of the collection by calling *AtInsert(Count, Item);*

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*. The result is the item pointer for which *Test* returned *True*, or **nil** if the *Test* function 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, TCollection.DeleteAll*

PutItem

```
procedure PutItem(var S: TStream; Item: Pointer); virtual;
```

Override: Sometimes

Called by *TCollection.Store* for each item in the collection. This method can be overridden but should not be called directly. The default *TCollection.PutItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Put* to store the item:

```
S.Put(Item);
```

See also: *TCollection.GetItem, TCollection.Store, TCollection.Load*

SetLimit

```
procedure SetLimit(ALimit: Integer); virtual;
```

Override: Seldom

Expands or shrinks the collection by changing the allocated size to *ALimit*. 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*, a new *Items* array of *ALimit* elements is allocated, the old *Items* array is copied into the new array, and the old array is disposed.

See also: *TCollection.Limit, TCollection.Count, MaxCollectionSize* variable

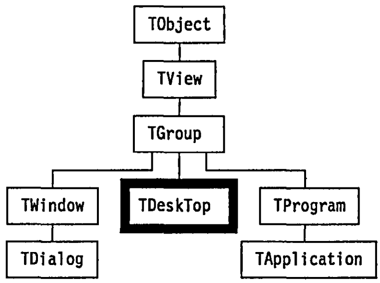
Store

```
procedure Store(var S: TStream);
```

Stores the collection and all its items on the stream *S*. *TCollection.Store* calls *TCollection.PutItem* for each item in the collection.

See also: *TCollection.PutItem*

TDeskTop



TDeskTop is a simple group that owns the *TBackground* 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.

Methods

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

Creates a *TDeskTop* group with size *Bounds*. The default *GrowMode* is *gfGrowHiX* + *gfGrowHiY*. *Init* also calls *NewBackground* to insert a *TBackground* view into the group.

See also: *TDeskTop.NewBackground, 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.

See also: *ofTileable, TDeskTop.Tile*

NewBackground `function NewBackground: PView; virtual;`

Override: Sometimes

Returns a pointer to the background to be used in the desktop. This method is called in the *TDeskTop.Init* method. Descendant objects can change the background type by overriding this method.

See also: *TDeskTop.Init*

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Seldom

Calls *TGroup.HandleEvent* and 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

Tile

```
procedure Tile(var R: TRect);
```

Redisplays all *ofTileable* views owned by the desktop in tiled format.

See also: *TDeskTop.Cascade, ofTileable*

TileError

```
procedure TileError; virtual;
```

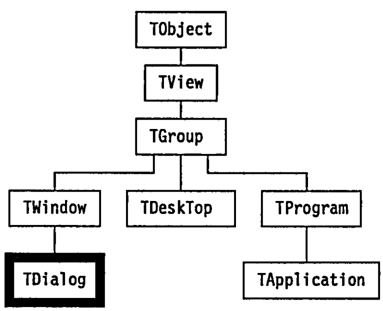
Override: Sometimes

TileError is called if an error occurs during *TDeskTop.Tile* or *TDeskTop.Cascade*. By default it does nothing. You may wish to override it to notify the user that the application is unable to rearrange the windows.

See also: *TDeskTop.Tile, TDeskTop.Cascade*

TDialog

Dialogs



TDialog is a simple child of *TWindow* with the following properties:

- *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 additionally handles the special cases of *Esc* and *Enter* key responses. The *Esc* key generates a *cmCancel* command, while *Enter* generates the *cmDefault* command.
- The *TDialog.Valid* method returns *True* on *cmCancel*, otherwise it calls its *TGroup.Valid*.

Methods

Init `constructor Init(var Bounds: TRect; ATitle: TTitleStr);`

Creates a dialog box with the given size and title by calling *TWindow.Init(Bounds, ATitle, wnNoNumber)*. *GrowMode* is set to 0, and *Flags* is set to *wfMove* + *wfClose*. This means that, by default, dialog boxes can move and close (via the close icon) but cannot grow (resize).

Note that *TDialog* does not define its own destructor, but uses *Close* and *Done* inherited via *TWindow*, *TGroup*, and *TView*.

See also: *TWindow.Init*

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Sometimes

Calls *TWindow.HandleEvent(Event)*, then handles Enter and Esc key events specially. In particular, Esc generates a *cmCancel* command, and the Enter key broadcasts a *cmDefault* command. This method also handles *cmOK*, *cmCancel*, *cmYes*, and *cmNo* command events by ending the modal state of the dialog box. For each of the above events handled successfully, this method calls *ClearEvent*.

See also: *TWindow.HandleEvent*

GetPalette `function GetPalette: PPalette; virtual;`

Override: Seldom

This method returns a pointer to the default palette, *CPalette*.

Valid `function Valid(Command: Word): Boolean; virtual;`

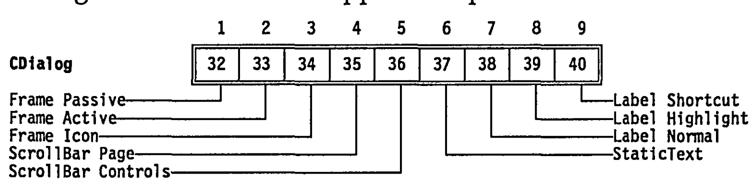
Override: Seldom

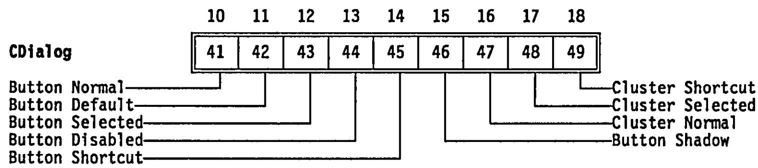
Returns *True* if the command given is *cmCancel* or if all the group controls return *True*.

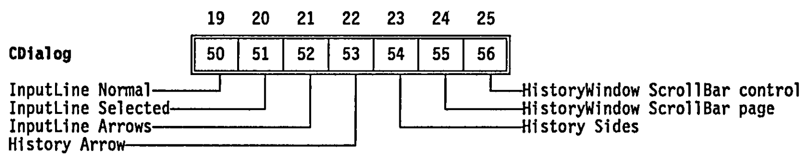
See also: *TGroup.Valid*

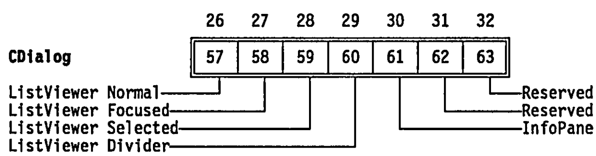
Palette

Dialog box objects use the default palette *CDialog* to map onto the 32nd through 63rd entries in the application palette.



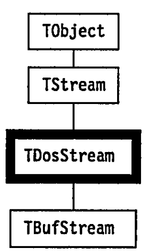






See also: *GetPalette* method for each object type

TDosStream — Objects



TDosStream is a specialized *TStream* 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* called *TBufStream*. *TDosStream* overrides all the abstract methods of *TStream* except for *TStream.Flush*.

Fields

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. Failure is signaled by a call to *Error* with an argument of *stInitError*.

The *Mode* argument must be set to one of the values *stCreate, stOpenRead, stOpenWrite,* or *stOpen*. These constant values are explained in Chapter 14 under "stXXXX stream 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 value of the calling stream's current position.

See also: *TDosStream.Seek*

GetSize

```
function GetSize: Longint; virtual;
```

Override: Never

Returns the total size in bytes of the calling stream.

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Override: Never

Reads *Count* bytes into the *Buf* buffer starting at the calling stream's current position.

See also: *TDosStream.Write, stReadError*

Seek

```
procedure Seek(Pos: Longint); virtual;
```

Override: Never

Resets the current position to *Pos* bytes from the beginning of the calling stream.

See also: *TDosStream.GetPos, TDosStream.GetSize*

Truncate

```
procedure Truncate; virtual;
```

Override: Never

Deletes all data on the calling stream from the current position to the end.

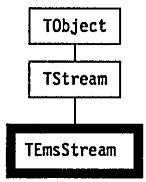
See also: *TDosStream.GetPos*, *TDosStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Writes *Count* bytes from the *Buf* buffer to the calling stream, starting at the current position.

See also: *TDosStream.Read*, *stWriteError*

TEmsStream Objects



TEmsStream is a specialized *TStream* derivative for implementing 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.

Size `Size: Longint;` **Read only**

The size of the stream in bytes.

Position `Position: Longint;` **Read only**

The current position within the stream. The first position is 0.

Methods

Init `constructor Init(MinSize: Longint);`

Creates an EMS stream with the given minimum size in bytes. Calls *TStream.Init* then sets *Handle, Size* and *PageCount*. Calls *Error* with an argument of *stInitError* if initialization fails.

TE

See also: *TEmsStream.Done*

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 value of the calling stream's current position.

See also: *TEmsStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never Returns the total size of the calling stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never Reads *Count* bytes into the *Buf* buffer starting at the calling stream's current position.

See also: *TEmsStream.Write, stReadError*

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Never Resets the current position to *Pos* bytes from the start of the calling stream.

See also: *TEmsStream.GetPos, TEmsStream.GetSize*

Truncate `procedure Truncate; virtual;`

Override: Never Deletes all data on the calling stream from the current position to the end. The current position is set to the new end of the stream.

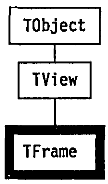
See also: *TEmsStream.GetPos, TEmsStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Never Writes *Count* bytes from the *Buf* buffer to the calling stream, starting at the current position.

See also: *TStream.Read, TEmsStream.GetPos, TEmsStream.Seek*

TFrame — Views



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 *TView.Init,* 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

Override: Seldom

```
procedure Draw; virtual;
```

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

Override: Seldom

```
function GetPalette: PPalette; virtual;
```

Returns a pointer to the default frame palette, *CFrame.*

HandleEvent

Override: Seldom

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Calls *TView.HandleEvent,* then handles mouse events. If the mouse is clicked on the close icon, *TFrame* generates a *cmClose* event. Clicking on the zoom icon or double-clicking on 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;`

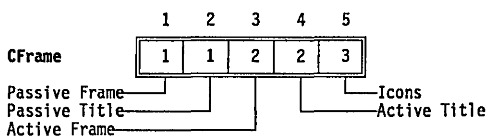
Override: Seldom

Calls *TView.SetState*, then if the new state is *sfActive* or *sfDragging*, calls *DrawView* to redraw the view.

See also: *TView.SetState*

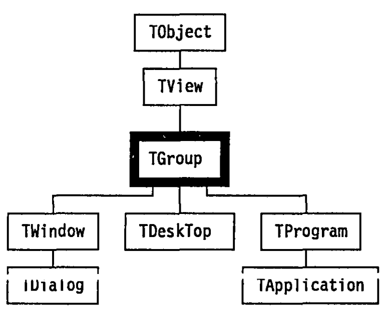
Palette

Frame objects use the default palette, *CFrame*, to map onto the first three entries in the standard window palette.



TGroup

Views



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 conceptually 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 and subgroups 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).

The three derived object types of *TGroup*, namely *TWindow*, *TDeskTop*, and *TApplication* (via *TProgram*) illustrate the group and subgroup concept. *TApplication* will typically own a *TDeskTop* object, a *TStatusLine* object, and a *TMenuView* 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 4, "Views" and Chapter 5, "Event-driven programming".

Many of the basic *TView* methods are overridden in *TGroup* in a natural way. For example, storing and loading groups on streams can be achieved with single calls to *TGroup.Store* and *TGroup.Load*.

TGroup objects are not usually instantiated; rather you would instantiate one or more of *TGroup*'s derived object types: *TApplication*, *TDeskTop*, and *TWindow*.

Fields

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.

Current `Current: PView;` **Read only**

Points to the subview that is currently selected, or is **nil** if no subview is selected.

See also: *sfSelected*, *TView.Select*

Buffer `Buffer: PVideoBuf;` **Read only**

Points to a buffer used to cache redraw operations, or is **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*

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);`

Calls *TView.Init,* sets *ofSelectable* and *ofBuffered* in *Options,* and sets *EventMask* to $FFFF.

See also: *TView.Init, TGroup.Load*

Load `constructor Load(var S: TStream);`

Loads an entire group from a stream by first calling the inherited *TView.Load* and then using *TStream.Get* to read each subview. Once all subviews have been loaded, a pass is performed over the subviews to fix up all pointers that were 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.

See also: *TView.Load, TGroup.Store, TGroup.GetSubViewPtr*

Done `destructor Done; virtual;`

Override: Often

Overrides *TView.Done.* Hides the group using *Hide,* disposes each subview in the group using a *Dispose(P, Done),* and finally calls the inherited *TView.Done.*

See also: *TView.Done*

ChangeBounds `procedure ChangeBounds(var Bounds: TRect); virtual;`

Override: Never

Overrides *TView.ChangeBounds.* 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

Overrides *TView.DataSize*. Returns total size of group 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. *P*'s *Owner* and *Next* fields are set to **nil**.

See also: *TGroup.Insert*

Draw `procedure Draw; virtual;`

Override: Never

Overrides *TView.Draw*. If a cache buffer exists (see *TGroup.Buffer* field) then the buffer is written to the screen using *TView.WriteBuf*. Otherwise, each subview is told to draw itself using a call to *TGroup.Redraw*.

See also: *TGroup.Buffer, TGroup.Redraw*

EndModal `procedure EndModal(Command: Word); virtual;`

Override: Never

If this group is the current modal view, it terminates its modal state. *Command* is passed to *ExecView* (which made this view modal in the first place), which returns *Command* as its result. If this group is *not* the current modal view, it calls *TView.EndModal*.

See also: *TGroup.ExecView, TGroup.Execute*

EventError `procedure EventError(var Event: TEvent); virtual;`

Override: Sometimes

EventError is called whenever the modal *TGroup.Execute* event-handling loop encounters an event that cannot be handled. The default action is: If the group's *Owner* is not **nil**, *EventError* calls its 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 to execute modal dialog boxes.

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** upon a call to *ExecView*, a value of *cmCancel* is returned.

See also: *TGroup.Execute, sfModal.*

Execute `function Execute: Word; virtual;`

Override: Seldom

Overrides *TView.Execute*. *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 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(Test: Pointer): PView;`

FirstThat applies a boolean function, given by the function pointer *Test*, to each subview in Z-order until *Test* returns *True*. The result is the subview pointer for which *Test* returned *True*, or **nil** if the *Test* function returned

False for all subviews. *Test* 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*

ForEach `procedure ForEach(Action: Pointer);`

ForEach applies an action, given by the procedure pointer *Action*, to each subview in the group in Z-order. *Action* must point to a **far** local procedure taking one *Pointer* parameter, for example:

```
procedure MyActionProc(P: PView); far;
```

The *MoveSubViews* method show 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

Overrides *TView.GetData*. 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 subviews' *GetHelpCtx* method. If no help context is specified by any subview, *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

Overrides *TView.HandleEvent*. A group basically 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 is done in three phases: First, the group's *Phase* field is set to *phPreProcess* and the event is passed to *HandleEvent* of all subviews that have the *ofPreProcess* flag set. Next, *Phase* is set to *phFocused* and the event is passed to *HandleEvent* of the currently selected view. Finally, *Phase* is set to *phPostProcess* and the event is passed to *HandleEvent* of all subviews that have the *ofPostProcess* flag set.

For positional events (by default *evMouse*, see *PositionalEvents* variable), the event is passed to the *HandleEvent* of the first subview whose bounding rectangle contains the point given by *Event.Where*.

For broadcast events (events that aren't focused or positional), the event is passed to the *HandleEvent* 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 given by *P* in the group's subview list. The new subview is placed on top of all other subviews. If the subview has the *ofCenterX* and/or *ofCenterY* flags set, it is centered accordingly in the group. If the view has the *sfVisible* flag set, it will be shown in the group—otherwise it remains invisible until specifically shown. If the view has the *ofSelectable* flag set, it becomes the currently selected subview.

See also: *TGroup.Delete, TGroup.ExecView, TGroup.Delete*

InsertBefore `procedure InsertBefore(P, Target: PView);`

Inserts the view given by *P* 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*, unpleasant "screen flicker" can be reduced if not eliminated. For example, the *TDeskTop.Tile* and *TDeskTop.Cascade* methods use *Lock* and *Unlock* in an attempt to reduce flicker.

☞ *Lock* and *Unlock* calls *must* be balanced, otherwise a group may 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*. *PutSubViewPtr* should only be used 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* will select (make 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.

See also: ofXXXX option flag constants

SetData

```
procedure SetData(var Rec); virtual;
```

Override: Seldom

Overrides *TView.SetData*. 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

Overrides *TView.SetState*. First calls the inherited *TView.SetState*, then updates the subviews as follows:

If *AState* is *sfActive, sfExposed*, or *sfDragging* then each subview's *SetState* is called to update the subview correspondingly.

If *AState* is *sfFocused* then the currently selected subview is called to focus itself correspondingly.

See also: *TView.SetState*

Store

```
procedure Store(var S: TStream);
```

Stores an entire group on a stream by first calling the inherited *TView.Store* and 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*

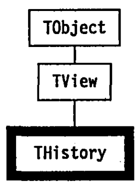
Valid

```
function Valid(Command: Word): Boolean; virtual;
```

Overrides *TView.Valid*. Returns *True* if all the subview's *Valid* calls return *True*. *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



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 a *TInputLine* 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 on 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.

Fields

Link `Link: PInputLine;` **Read only**

A pointer to the linked *TInputLine* object.

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.

Methods

Init `constructor Init(var Bounds: TRect; ALink: PInputLine; AHistoryId: Word);`

Creates a *THistory* object of the given size by calling *TView.Init*, then setting the *Link* and *HistoryId* fields with the given argument values. The *Options* field is set to *ofPostProcess* and *EventMask* to *evBroadcast*.

See also: *TView.Init*

Load `constructor Load(var S: TStream);`

Creates and initializes a *THistory* object from the given *TStream* by calling *TView.Load(S)* and reading *Link* and *HistoryId* from *S*.

See also: *TView.Store*

Draw `procedure Draw; virtual;`

Override: Seldom Draws the *THistory* icon in the default palette.

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes Returns a pointer to the default palette, *CHistory*.

Store `procedure Store(var S: TStream);`

Saves a *THistory* object on the target *TStream* by calling *TView.Store(S)* then writing *Link* and *HistoryId* to *S*.

See also: *TView.Load*

Palette

History icons use the default palette, *CHistory*, to map onto the 22nd and 23rd entries in the standard dialog box palette.

```
             1    2
CHistory  ╔════╤════╗
          ║ 22 │ 23 ║
          ╚════╧════╝
Arrow─────────┘    └───Sides
```

THistoryViewer — Dialogs

THistoryViewer is a rather straightforward descendant of *TListViewer*. It is used by the history list system, and appears inside the history window set up by clicking on the history icon. For details on how *THistory*, *THistoryWindow*, and *THistoryViewer* cooperate, see the entry for *THistory* in this chapter.

Field

HistoryId `HistoryId: Word;` **Read only**

HistoryID is the ID number of the history list to be displayed in the view.

Methods

Init `constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar; AHistoryId: Word);`

Initializes the viewer list by first calling *TListViewer.Init* to set up the boundaries, a single column, and the two scroll bars passed in *AHScrollBar* and *AVScrollBar*. The view is then linked to a history list, with the *HistoryID* field set to the value passed in *AHistory*. That list is then checked for length, so the range of the list is set to the number of items in the list. The first item in the history list is given the focus, and the horizontal scrolling range is set to accommodate the widest item in the list.

See also: *TListViewer.Init*

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes

Returns a pointer to the default palette, *CHistoryViewer*.

GetText `function GetText(Item: Integer; MaxLen: Integer): String; virtual;`

Override: Seldom

Returns the *Item*'th string in the associated history list. *GetText* is called by the virtual *Draw* method for each visible item in the list.

See also: *TListViewer.Draw*, *HistoryStr* function

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Sometimes

The history viewer handles two kinds of events itself; all others are passed to *TListViewer.HandleEvent*. Double clicking or pressing the *Enter* key will terminate the modal state of the history window with a *cmOK* command.

Pressing the *Esc* key, or any *cmCancel* command event, will cancel the history list selection.

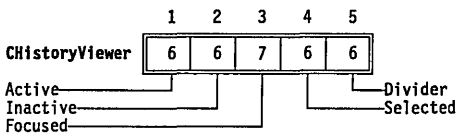
See also: *TListViewer.HandleEvent*

HistoryWidth `function HistoryWidth: Integer;`

Returns the length of the longest string in the history list associated with *HistoryID*.

Palette

History viewer objects use the default palette *CHistoryViewer* to map onto the 6th and 7th entries in the standard dialog box palette.



THistoryWindow — Dialogs

THistoryWindow is a specialized descendant of *TWindow* used for holding a history list viewer when the user clicks on 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.

Field

Viewer `Viewer: PListViewer;` **Read only**

Viewer points to a list viewer to be contained in the history window.

Methods

Init `constructor Init(var Bounds: TRect; HistoryId: Word);`

Calls *TWindow.Init* to set up a window with the given bounds, a null title string, and no window number (*wnNoNumber*). The *TWindow.Flags* field is set to *wfClose* to provide a close icon, and a history viewer object is created to show the items in the history list given by *HistoryID*.

See also: *TWindow.Init, THistoryWindow.InitViewer*

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes Returns a pointer to the default palette, *CHistoryWindow*.

GetSelection `function GetSelection: String; virtual;`

Override: Never Returns the string value of the focused item in the associated history viewer.

See also: *THistoryViewer.GetText*

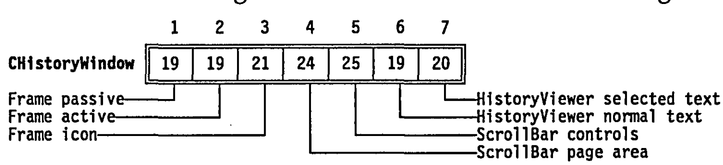
InitViewer `procedure InitViewer(HistoryId: Word); virtual;`

Override: Never Instantiates and inserts a *THistoryViewer* object inside the boundaries of the history window for the list associated with the ID *HistoryId*. Standard scroll bars are placed on the frame of the window to scroll the list.

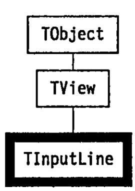
See also: *THistoryViewer.Init*

Palette

History window objects use the default palette *CHistoryWindow* to map onto the 19th through 25th entries in the standard dialog box palette.



TInputLine — Dialogs



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*.

An input line frequently has a *TLabel* and/or a *THistory* object associated with it.

TInputLine can be extended to handle data types other than strings. To do so, you'll generally add additional fields and then override the *Init*, *Load*, *Store*, *Valid*, *DataSize*, *GetData*, and *SetData* methods. For example, to define a numeric input line, you might want it to contain minimum and maximum allowable values which will be tested by the *Valid* function. These minimum and maximum fields would be *Load*ed and *Store*d on the stream. *Valid* would be modified to make sure the value was numeric and within range. *DataSize* would be modified to include the size of the new range fields (probably *SizeOf(Longint)* for each). Oddly enough, in this example it would not be necessary to add a field to store the numeric value itself. It could be stored as a string value (which is already managed by *TInputLine*) and converted from string to numeric value and back by *GetData* and *SetData* respectively.

TI

Fields

Data `Data: PString;` **Read/write**

Pointer to the string containing the edited information.

MaxLen `MaxLen: Integer;` **Read only**

Maximum length allowed for string to grow, excluding the length byte.

See also: *TInputLine.DataSize*

CurPos `CurPos: Integer;` **Read/write**

Index to insertion point (that is, to the current cursor position).

See also: *TInputLine.SelectAll*

FirstPos `FirstPos: Integer;` **Read/write**

Index to the first displayed character.

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*

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*

Methods

Init `constructor Init(var Bounds: TRect; AMaxLen: Integer);`

Creates an input box control with the given argument values by calling *TInputLine.Init*. *State* is set to *sfCursorVis*, *Options* is set to (*ofSelectable* + *ofFirstClick*), and *MaxLen* is set to *AMaxLen*. Memory is allocated and cleared for *AMaxlen*+1 bytes and the *Data* field set to point at this allocation.

See also: *TView.Init, TView.sfCursorVis, TView.ofSelectable, TView.ofFirstClick*

Load `constructor Load(var S: TStream);`

Creates and initializes a *TInputLine* object by calling *TView.Load(S)* to load the view off the given stream, then reads the integer fields off the stream using *S.Read*, allocates *MaxLen*+1 bytes at *Data* with *GetMem*, and finally sets the string-length byte and loads the data from the stream with two more *S.Read* calls. *Load* is used in conjunction with *TInputLine.Store* to save and retrieve *TInputLine* objects on a *TStream*.

Override this method if you define descendants that contain additional fields.

See also: *TView.Load, TInputLine.Store, TStream.Read*

Done `destructor Done; virtual;`

Override: Seldom

Deallocates the *Data* memory allocation, then calls *TView.Done* to destroy the *TInputLine* object.

See also: *TView.Done*

DataSize `function DataSize: Word; virtual;`

Override: Sometimes

Returns the size of the record for *TInputLine.GetData* and *TInputLine.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 `procedure Draw; virtual;`

Override: Seldom

Draws the input box and its data. The box is drawn with the appropriate colors depending on whether the box is *sfFocused* or not (that is, whether the box view owns the cursor or not), and arrows are drawn if the input string exceeds the size of the view (in either or both directions). Any selected (block marked) characters are drawn with the appropriate palette.

GetData `procedure GetData(var Rec); virtual;`

Override: Sometimes

Writes *DataSize* bytes from the string *Data^* to given record. Used with *TInputLine.SetData* for a variety of applications, e.g., temporary storage or passing on the input string to other views. Override this method if you define descendants to handle non-string data types. Use this method to convert from a string to your data type after editing by *TInputLine.*

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 *TView.HandleEvent*, then handles all mouse and keyboard events if the input box 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 set *True, SelEnd* is set to *Length(Data^)* thereby selecting the whole input line; if *Enable* is set *False, SelEnd* is set to 0, thereby deselecting the whole line. Finally, the view is redrawn by calling *DrawView.*

See also: *TView.DrawView*

SetData `procedure SetData(var Rec); virtual;`

Override: Sometimes

By default, reads *DataSize* bytes from given record to the *Data^* string and calls *SelectAll(True)* to reset *CurPos, FirstPos,* and *SelStart* to zero; *SelEnd* is set to the last character of *Data^* and the view is *DrawView'*d. Override this method if you define descendants to handle non-string data types. Use 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 box needs redrawing (for example, palette changes) following a change of *State.* Calls *TView.SetState* to set or clear the view's *State* field with the given *AState* bit(s). Then if *AState* is *sfSelected* or if *AState* is *sfActive* and the input box is *sfSelected, SelectAll(Enable)* is called.

See also: *TView.SetState, TView.DrawView*

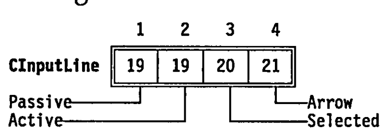
Store `procedure Store(var S: TStream);`

Stores the view on the given stream by calling *TView.Store(S),* then stores the five integer fields and the *Data* string with *S.Write* calls. Used in conjunction with *TInputLine.Load* for saving and restoring entire *TInputLine* objects. Override this method if you define descendants that contain additional fields.

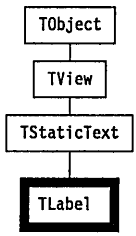
See also: *TView.Store, TInputLine.Load, TStream.Write*

Palette

Input lines use the default palette, *CInputLine,* to map onto the 19th through 21st entries in the standard dialog palette.



TLabel — Dialogs



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

Link `Link: PView;` **Read only**

Pointer to the control associated with this label.

Light `Light: Boolean;` **Read only**

If *True*, the label and its linked control has been selected and will be highlighted.

Methods

Init `constructor Init(var Bounds: TRect; AText: String; ALink: PView);`

Creates a *TLabel* object of the given size by calling *TStaticText.Init*, then sets the *Link* field to *ALink* for the associated control (make *ALink* **nil** if no control is needed). The *Options* field is set to *ofPreProcess* and *ofPostProcess*. The *EventMask* is set to *evBroadcast*. The *AText* field is assigned to the *Text* field by *TStaticText.Init*. *AText* can designate a shortcut letter for the label by surrounding the letter with tildes ('~').

See also: *TStaticText.Init*

Load `constructor Load(var S: TStream);`

Creates and loads a *TLabel* object from the given stream by calling *TStaticText.Load,* then calling *GetPeerViewPtr(S, Link)* to reestablish the link to the associated control (if any).

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 all events by calling *TStaticText.HandleEvent.* If an *evMouseDown* or shortcut key event is received, the appropriate linked control (if any) is selected. This method also handles *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 given stream by calling *TStaticText.Store,* then records the link to the associated control by calling *PutPeerViewPtr.*

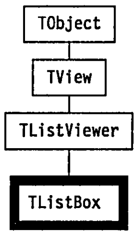
See also: *TLabel.Load*

Palette

Labels use the default palette, *CLabel,* to map onto the 7th, 8th and 9th entries in the standard dialog palette.

```
                 1   2   3   4
              ╔═══╤═══╤═══╤═══╗
CLabel        ║ 7 │ 8 │ 9 │ 9 ║
              ╚═╤═╧═╤═╧═╤═╧═╤═╝
Text Normal─────┘   │   │   └───Shortcut Selected
Text Selected───────┘   └───────Shortcut Normal
```

TListBox — Dialogs



TListBox is derived from *TListViewer* to help you set up the most commonly used list boxes, namely those displaying collections of strings such as file names. *TListBox* objects represent displayed lists of such items 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 select (and highlight) items by mouse and keyboard cursor actions. *TListBox* does not override *TListViewer.HandleEvent* or *TListViewer.Draw*, so you should refer to the sections describing these before using *TListBox* in your applications.

TListBox has an additional field called *List* not found in *TListViewer*. *List* points to a *TCollection* object that provides the items to be listed and selected. Inserting data into the *TCollection* is your responsibility, as are the actions to be performed when an item is selected.

TListViewer 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);`

Creates a list box control with the given size, number of columns, and a vertical scroll bar referenced by the *AScrollBar* pointer. This method calls *TListViewer.Init* with a **nil** horizontal scroll bar argument.

The *List* field is initially **nil** (empty list) and the inherited *Range* field is set to zero. Your application must provide a suitable *TCollection* holding the strings (or other objects) to be listed. The *List* field must be set to point to this collection using *NewList*.

See also: *TListViewer.Init, TListBox.NewList*

Load `constructor Load(var S: TStream);`

Creates a *TListBox* object and loads it with values from the given *TStream*. This method calls *TListViewer.Load* then sets *List* by reading a *List* pointer 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 *TListBox.GetData* and *TListBox.SetData*. These three methods are useful for initializing groups. By default *TListBox.DataSize* returns the size of a pointer plus the size of a word (for the *List* and the selected item). You may need to override this method for your own applications.

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*. You may need to override this method for your own applications.

See also: *TListBox.DataSize, TListBox.SetData*

GetText `function GetText(Item: Integer; MaxLen: Integer): String; virtual;`

Override: Sometimes

Returns a string from the calling *TListBox* object. By default, the returned string is obtained from the *Item*'th item in the *TCollection* 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*. The inherited *Range* field is set to the *Count* field of the new *TCollection*, and the first item is focused by calling *FocusItem(0)*. Finally, the new list is displayed with a *DrawView* call. Note that if the previous *List* field is non-**nil** it is disposed of before the new list values are assigned.

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 may 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 given *TStream* by calling *TListViewer.Store* and then puts the collection onto the stream by calling *S.Put(List)*.

See also: *TListBox.Load, TListViewer.Store, TStream.Put*

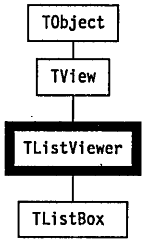
Palette

List boxes use the default palette, *CListViewer*, to map onto the 26th through 29th entries in the standard application palette.

```
                  1    2    3    4    5
CListViewer    ╔════╤════╤════╤════╤════╗
               ║ 26 │ 26 │ 27 │ 28 │ 29 ║
               ╚════╧════╧════╧════╧════╝
Active────────────┘    │    │    │    └───Divider
Inactive───────────────┘    │    └────────Selected
Focused─────────────────────┘
```

TL

TListViewer — Views



The *TListViewer* object type is essentially 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

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.

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.

NumCols `NumCols: Integer;` **Read only**

The number of columns in the list control.

TopItem `TopItem: Integer;` **Read/write**

The item number of the top item to be displayed. 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*

Focused `Focused: Integer;` **Read only**

The item number of the focused item. Items are numbered from 0 to *Range* – 1. Initially set to 0, the first item, *Focused* can be changed by mouse click or *Spacebar* selection.

See also: *Range*

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*

TL

Methods

Init

```
constructor Init(var Bounds: TRect; ANumCols: Integer; AHScrollBar,
AVScrollBar: PScrollBar);
```

Creates and initializes a *TListViewer* object with the given size by first calling *TView.Init*. The *NumCols* field is set *ANumCols*. *Options* is set to (*ofFirstClick* + *ofSelectable*) so that mouse clicks that select this view will be passed first to *TListViewer.HandleEvent*. The *EventMask* is set to *evBroadcast*. The initial values of *Range* and *Focused* are zero. Pointers to vertical and/or horizontal scroll bars can be supplied via the *AVScrollBar* and *AHScrollBar* arguments. Set either or both to **nil** if you do not want scroll bars. These two pointer arguments will be assigned to the *VScrollBar* and *HScrollBar* fields.

If you provide valid scroll bars, their *PgStep* and *ArStep* fields will be adjusted 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);`

Creates a *TListViewer* object by calling *TView.Load*. The scroll bars, if any, are also loaded from the given stream using calls to *GetPeerViewPtr*. All integer fields are also loaded, using *S.Read*.

See also: *TView.Load, TListViewer.Store*

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Override: Never

Changes the size of the *TListViewer* object by calling *TView.ChangeBounds*. If a horizontal scroll bar has been assigned, this method adjusts *PgStep* if necessary.

See also: *TView.ChangeBounds, TScrollBar.ChangeStep*

Draw

```
procedure Draw; virtual;
```

Override: Never

Draws the *TListViewer* object with the default palette by repeatedly 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 the *Focused* field to *Item*. The method 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 *TListViewer* palette.

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 not exceeding *MaxLen* given an item index given by *Item*.

See also: *TListViewer.Draw*

IsSelected

```
function IsSelected(Item: Integer): Boolean; virtual;
```

Override: Never

Returns true if the given *Item* is focused, that is, if *Item* = *Focused*.

See also: *TListViewer.FocusItem*

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Override: Seldom

Handles events by calling *TView.HandleEvent*. Mouse clicks and "auto" movements over the list will change the focused item. Items can be selected 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* keys 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*

SelectItem `procedure SelectItem(Item: Integer); virtual;`

Override: Sometimes

An abstract method for selecting the item indexed by *Item.*

See also: *TListViewer.FocusItem*

SetRange `procedure SetRange(ARange: Integer);`

Sets the *Range* field to *ARange.* If a vertical scroll bar has been assigned, its parameters are adjusted as necessary. If the currently focused item falls outside the new *Range,* the *Focused* field is set to zero.

See also: *TListViewer.Range, TScrollBar.SetParams*

SetState `procedure SetState(AState: Word; Enable: Boolean); virtual;`

Override: Seldom

Calls *TView.SetState* to change the *TListViewer* 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);`

Calls *TView.Store* to save the *TListViewer* object on the target stream, then stores the scroll bar objects (if any) using calls to *PutPeerViewPtr,* and finally saves the integer fields using *S.Write.*

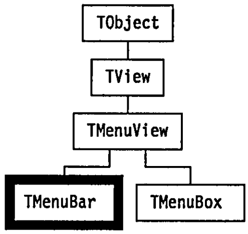
See also: *TView.Store, TListViewer.Load*

Palette

List viewers use the default palette, *CListViewer,* to map onto the 26th through 29th entries in the standard application palette.

```
                 1    2    3    4    5
CListViewer    | 26 | 26 | 27 | 28 | 29 |
Active-----------+    |    |    |    +---Divider
Inactive--------------+    |    +--------Selected
Focused--------------------+
```

TMenuBar — Menus



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. By overriding *TApplication.InitMenuBar* with a suitable set of nested *New, NewSubMenu, NewItem* and *NewLine* calls, Turbo Vision takes care of it.

Methods

Init

```
constructor Init(var Bounds: TRect; AMenu: PMenu);
```

Creates a menu bar with the given *Bounds* by calling *TMenuView.Init*. The grow mode is set to *gfGrowHiX*. The *Options* field is set to *ofPreProcess* to allow hot keys to operate. The *Menu* field is set to *AMenu*, providing the menu selections.

See also: *TMenuView.Init*, gfXXXX grow mode flags, ofXXXX option flags, *TMenuView.Menu*

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;
```

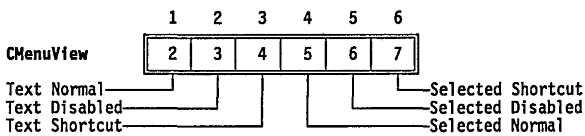
Override: Never

Overrides the abstract method in *TMenuView*. Returns the rectangle occupied by the given menu item in *R*. It is used to determine if a mouse click has occurred on a given menu selection.

See also: *TMenuView.GetItemRect*

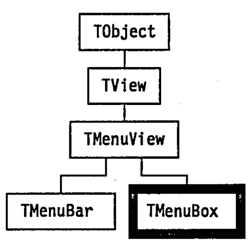
Palette

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



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 *TMenuView.Init.*

The *ofPreProcess* bit in the *Options* field is set so that hot keys will operate. *State* is set to include *sfShadow.* The *Menu* field is set to *AMenu,* which provides the menu selections. The *ParentMenu* field is set 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.

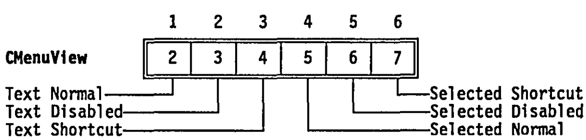
GetItemRect `procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;`

Override: Seldom Overrides the abstract method in *TMenuView.* Returns the rectangle occupied by the given menu item. It is used to determine if a mouse click has occurred on a given menu selection.

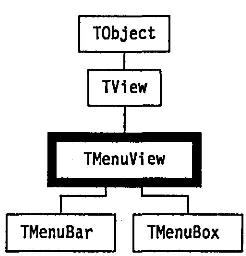
See also: *TMenuView.GetItemRect*

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.



TMenuView — Menus



TMenuView provides an abstract menu type from which menu bars and menu boxes (either pull-down or pop-up) are derived. You will probably never instantiate a *TMenuView* itself.

Fields

ParentMenu `ParentMenu: PMenuView;` **Read only**

A pointer to the *TMenuView* (or descendant) object 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*

Menu `Menu: PMenu;` **Read only**

A pointer 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

Current `Current: PMenuItem;` **Read only**

A pointer to the currently selected menu item.

TM

Methods

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

Calls *TView.Init* to create a *TMenuView* object of size *Bounds*. The default *EventMask* is set to *evBroadcast*. This method is not intended to be used for instantiating *TMenuView* objects. It is designed to be called by its descendant types, *TMenuBar* and *TMenuBox*.

See also: *TView.Init. evBroadcast. TMenuBar.Init. TMenuBox.Init*

Load `constructor TMenuView.Load(var S: TStream);`

Creates a *TMenuView* object and loads it from the stream *S* by calling *TView.Load* and then loading 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, this method returns the help context of the current menu selection. 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 mouse or keyboard selected (including hot keys) and generates the appropriate command event with *PutEvent.*

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.* This method is used by *TMenuView.HandleEvent* to determine whether a keystroke event selects an item in the menu.

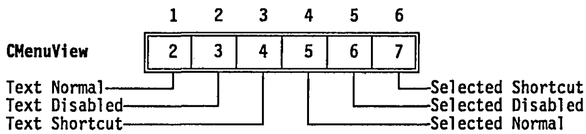
Store `procedure Store(var S: TStream);`

Saves the calling *TMenuView* object (and any of its submenus) on the stream *S* by calling *TView.Store* and then writing each menu item to the stream.

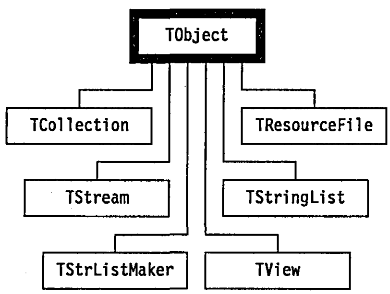
See also: *TMenuView.Load*

Palette

All menu views use the default palette *CMenuView* to map onto the 2nd through 7th entries in the standard application palette.



TObject Objects



TObject is the starting point of Turbo Vision's object hierarchy. As the base object, it has no parents 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 streams 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. Note that *TObject.Init* will zero all fields in descendants, so you should always call *TObject.Init* *before* initializing any fields in the derived objects' constructors.

Free `procedure Free;`

Disposes of the object and calls the *Done* destructor.

Done `destructor Done; virtual;`

Performs the necessary cleanup and disposal for dynamic objects.

TParamText — Dialogs

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; AText: String; AParamCount: Integer);`

Initializes a static text object by calling *TStaticText.Init* 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);`

Allocates a *TParamText* object on the heap and loads its value from the stream *S* by first calling *TStaticText.Load* 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 a call to *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*

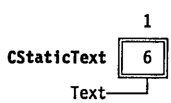
Store `procedure Store(var S: TStream);`

Stores the object on the stream *S* by first calling *TStaticText.Store* 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

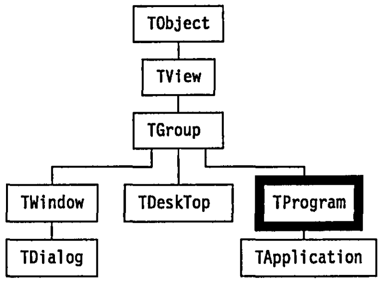
X `X: Integer`

X is the screen column of the point.

Y `Y: Integer`

Y is the screen row of the point.

TProgram App



TProgram provides the basic template for all standard Turbo Vision applications. All such programs must be derived from *TProgram* or its child, *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* derivative since it needs to contain your *TDeskTop*, *TStatusLine*, and *TMenuBar* objects

Methods

Init `constructor Init;`

Override: Sometimes

Sets the *Application* global variable to *@Self*; calls *TProgram.InitScreen* to initialize screen mode dependent variables; calls *TGroup.Init* passing a *Bounds* rectangle equal to the full screen; sets the *State* field to *sfVisible* + *sfSelected* + *sfFocused* + *sfModal* + *sfExposed*; sets the *Options* field to zero; sets the *Buffer* field 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**.

See also: *TGroup.Done*

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 *TProgram.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 *TProgram.Idle* to allow "background" tasks to be performed while the application is waiting 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 *Alt-1* through *Alt-9* keyboard 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.

TProgram.HandleEvent is almost always overridden to introduce handling of commands that are specific to your own application.

See also: *TGroup.HandleEvent*

Idle

```
procedure Idle; virtual;
```

Override: Sometimes

Idle is called by *TProgram.GetEvent* whenever the event queue is empty, allowing the application to perform background tasks while waiting for user input.

TP

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*, an *evBroadcast* 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* do not 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

Creates a *TDeskTop* object for the application and stores a pointer to it in the *DeskTop* global variable. *InitDeskTop* is called by *TProgram.Init* but should never be called directly. *InitDeskTop* can be overridden to instantiate a user-defined descendant of *TDeskTop* instead of the standard *TDeskTop*.

See also: *TProgram.Init*, *TDeskTop*, *TWindow.Init*

InitMenuBar `procedure InitMenuBar; virtual;`

Override: Always

Creates a *TMenuBar* object for the application and stores a pointer to it in the *MenuBar* global variable. *InitMenuBar* is called by *TProgram.Init* but should never be called directly. *InitMenuBar* is almost always overridden to instantiate a user defined *TMenuBar* instead of the default empty *TMenuBar*.

See also: *TProgram.Init*, *TMenuBar*, *TWindow.Init*

InitScreen `procedure InitScreen; virtual;`

Override: Sometimes

Called by *TProgram.Init* and *TProgram.SetScreenMode* 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 `procedure InitStatusLine; virtual;`

Override: Always

Creates a *TStatusLine* object for the application and stores a pointer to it in the *StatusLine* global variable. *InitStatusLine* is called by *TProgram.Init* but should never be called directly. *InitStatusLine* is almost always overridden to instantiate a user defined *TStatusLine* instead of the default *TStatusLine*.

See also: *TProgram.Init*, *TStatusLine*

OutOfMemory `procedure OutOfMemory; virtual;`

Override: Often

OutOfMemory is called by *TProgram.ValidView* 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 *StdDlg* unit:

```
procedure TMyApp.OutOfMemory;
begin
  MessageBox('Not enough memory to complete operation.',
    nil, mfError + mfOKButton);
end;
```

See also: *TProgram.ValidView*, *LowMemory*

PutEvent `procedure PutEvent(var Event: TEvent); virtual;`

Override: Seldom

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 *TProgram.GetEvent* will return that copy.

See also: *TProgram.GetEvent*, *TView.PutEvent*

Run `procedure Run; virtual;`

Override: Seldom

Runs the *TProgram* by calling the *Execute* method (which *TProgram* inherited from *TGroup*).

TP

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 screen mode 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. The following statement, for example, shows a typical sequence of instantiation, validation, and insertion of a new window on the desktop (both *TProgram.ValidView* and *TGroup.Insert* know how to ignore possible **nil** pointers resulting from errors).

```
DeskTop^.Insert(ValidView(New(TMyWindow, Init(...))));
```

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.

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
CColor	$71	$70	$78	$74	$20	$28	$24
CBlackWhite	$70	$70	$78	$7F	$07	$07	$0F
CMonochrome	$70	$07	$07	$0F	$70	$70	$70

Background (1)
Normal Text (2)
Disabled Text (3)
Shortcut text (4)
Normal selection (5)
Disabled selection (6)
Shortcut selection (7)

Entries 8 through 15 are used by blue windows.

	8	9	10	11	12	13	14	15
CColor	$17	$1F	$1A	$31	$31	$1E	$71	$00
CBlackWhite	$07	$0F	$07	$70	$70	$07	$70	$00
CMonochrome	$07	$0F	$07	$70	$70	$07	$70	$00

Frame Passive (8)
Frame Active (9)
Frame Icon (10)
ScrollBar Page (11)
ScrollBar Reserved (12)
Scroller Normal Text (13)
Scroller Selected Text (14)
Reserved (15)

Entries 16 through 23 are used by cyan windows.

	16	17	18	19	20	21	22	23
CColor	$37	$3F	$3A	$13	$13	$3E	$21	$00
CBlackWhite	$07	$0F	$07	$70	$70	$07	$70	$00
CMonochrome	$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
CColor	$70	$7F	$7A	$13	$13	$70	$7F	$00
CBlackWhite	$70	$7F	$7F	$70	$07	$70	$07	$00
CMonochrome	$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 dialog box objects. See *TDialog* for individual entries.

	32	33	34	35	36	37	38	39	40
CColor	$70	$7F	$7A	$13	$13	$70	$70	$7F	$7E
CBlackWhite	$70	$7F	$7F	$70	$07	$70	$70	$7F	$7F
CMonochrome	$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
CColor	$20	$2B	$2F	$78	$2F	$70	$30	$3F	$3F
CBlackWhite	$07	$0F	$0F	$78	$0F	$78	$07	$0F	$0F
CMonochrome	$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

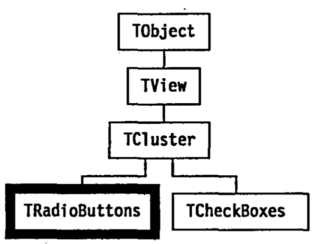
	50	51	52	53	54	55	56
CColor	$1F	$2F	$1A	$20	$72	$31	$31
CBlackWhite	$0F	$70	$0F	$07	$70	$70	$70
CMonochrome	$07	$70	$07	$07	$70	$07	$07

```
InputLine Normal         (50)
InputLine Selected       (51)
InputLine Arrows         (52)
History Arrow            (53)
History Sides            (54)
HistoryWindow ScrollBar page      (55)
HistoryWindow ScrollBar controls  (56)
```

	57	58	59	60	61	62	63
CColor	$30	$2F	$3E	$31	$13	$00	$00
CBlackWhite	$07	$70	$0F	$07	$07	$00	$00
CMonochrome	$07	$70	$0F	$07	$07	$00	$00

```
ListViewer Normal        (57)
ListViewer Focused       (58)
ListViewer Selected      (59)
ListViewer Divider       (60)
InfoPane                 (61)
Reserved                 (62)
Reserved                 (63)
```

TRadioButtons — Dialogs



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 are often associated with a *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

Override: Seldom

```
procedure Draw; virtual;
```

Draws buttons as " ( ) " surrounded by a box.

Mark

Overide: Never

```
function Mark(Item: Integer): Boolean; virtual;
```

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

Override: Never

```
procedure MovedTo(Item: Integer); virtual;
```

Assigns *Item* to *Value*.

See also: *TCluster.MovedTo*, *TRadioButtons.Mark*

Press

Override: Never

```
procedure Press(Item: Integer); virtual;
```

Assigns *Item* to *Value*. Called when the *Item*'th button is pressed.

SetData

Override: Seldom

```
procedure SetData(var Rec); virtual;
```

Calls *TCluster.SetData* 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.

```
                 1    2    3    4
CCluster      ║ 16 | 17 | 18 | 18 ║
Text Normal──────┘    |    |    └──Shortcut Selected
Text Selected─────────┘    └───────Shortcut Normal
```

TR

TRect Objects

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);`

This method assigns the parameter values to the rectangle's point fields. *XA* becomes *A.X*, *XB* becomes *X.B*, etc.

Copy `procedure Copy(R: TRect);`

Copy sets all fields equal to those in rectangle *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*.

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*.

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*.

Contains `function Contains(P: TPoint): Boolean;`

Returns true if the rectangle contains the point *P*.

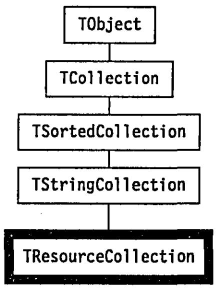
Equals `function Equals(R: TRect): Boolean;`

Returns true if *R* is the same as the rectangle.

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.

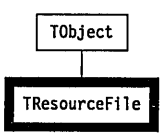
TResourceCollection — Objects



TResourceCollection is a derivative 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

Stream `Stream: PStream;` **Read only**

Pointer to the stream associated with this resource file

Modified `Modified: Boolean;` **Read/write**

Set *True* if the resource file has been modified.

See also: *TResourceFile.Flush*

Methods

Init `constructor Init(AStream: PStream);`

Override: Never

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* will look 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 instantiates 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 calling resource file.

See also: *TResourceFile.KeyOf*

Delete `procedure Delete(Key: String);`

Deletes the resource indexed by *Key* from the calling 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 the given *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;`

TR

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*.

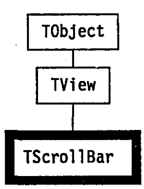
See also: *TResourceFile.Get*

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*, however, provides faster copying, but without the compression.

TScrollBar — Views



Fields

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 on 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 zero by default.

See also: *TScrollBar.SetValue, TScrollBar.SetParams, TScrollBar.ScrollDraw, TScroller.HandleEvent, TScrollBar.Init*

Min `Min: Integer;` **Read only**

Min represents the minimum value for the *Value* field. *TScrollBar.Init* sets *Min* to zero by default.

See also: *TScrollBar.SetRange, TScrollBar.SetParams*

Max `Max: Integer;` **Read only**

Max represents the maximum value for the *Value* field. *TScrollBar.Init* sets *Max* to zero by default.

See also: *TScrollBar.SetRange, TScrollBar.SetParams*

PgStep `PgStep: Integer;` **Read only**

PgStep is the amount added or subtracted to 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). *TScrollBar.Init* sets *PgStep* to 1 by default. *PgStep* can be changed using *TScrollBar.SetStep, TScrollBar.SetParams* or *TScroller.SetLimit*

See also: *TScrollBar.SetStep, TScrollBar.SetParams, TScroller.SetLimit, TScrollBar.ScrollStep*

ArStep `ArStep: Integer;` **Read only**

ArStep is the amount added or subtracted to the scroll bar's *Value* field when an arrow area is clicked (*sbLeftArrow, sbRightArrow, sbUpArrow*, or *sbDownArrow*) or the equivalent keystroke made. *TScrollBar.Init* sets *ArStep* to 1 by default.

See also: *TScrollBar.SetStep, TScrollBar.SetParam, TScrollBar.ScrollStep*

TS

Methods

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

Creates and initializes a scroll bar with the given *Bounds* by calling *TView.Init*. *Value, Max*, and *Min* are set to zero. *PgStep* and *ArStep* are set to 1. The shapes of the scroll bar parts are set 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);
```

Creates then loads the scroll bar on the stream *S* by calling *TView.Load* 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 *TView.HandleEvent* 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). *TScrollBar.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. This pseudo-abstract methods defaults by sending a *cmScrollBarChanged* message 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. The *Part* argument should be one of the sb*XXXX* scroll bar part constants described in Chapter 14.

See also: *TScrollBar.SetStep, TScrollBar.SetParams*

SetParams

```
procedure SetParams(AValue, AMin, AMax, APgStep, AArStep: Integer);
```

SetParams sets the *Value, Min, Max, PgStep,* and *ArStep* fields with the given argument values. 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 the given arguments *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 the fields *PgStep* and *ArStep* to the given arguments *APgStep* and *AArStep*. This method calls *SetParams* with the other arguments set to their current values.

See also: *TScrollBar.SetParams, TScrollBar.ScrollStep*

SetValue

```
procedure SetValue(AValue: Integer);
```

SetValue sets the *Value* field to *AValue* by calling *SetParams* with the other arguments 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

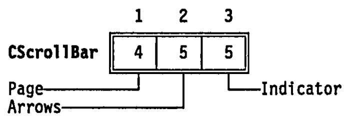
```
procedure Store(var S: TStream);
```

Stores the calling *TScrollBar* object on the stream *S* by calling *TView.Store* and then writing the five integer fields to the stream using *S.Write*.

See also: *TScrollBar.Load*

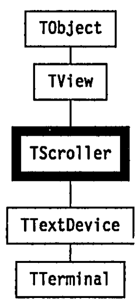
Palette

Scroll bar objects use the default palette, *CScrollBar*, to map onto the 4th and 5th entries in the standard application palette.



TScroller

Views



Fields

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**.

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**.

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*

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*

Methods

Init

```
constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar:
PScrollBar);
```

Creates and initializes a *TScroller* object with the given size and scroll bars. Calls *TView.Init* to set the view's size. *Options* is set to *ofSelectable* and *EventMask* is set 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 scroller view from the stream *S* by calling *TView.Load,* then restores pointers to the scroll bars using *GetPeerViewPtr,* and finally reads 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 *TView.HandleEvent.* Broadcast events with the command *cmScrollBarChanged,* if they come from either *HScrollBar* or *VScrollBar,* result in a call to *TScroller.ScrollDraw.*

See also: *TView.HandleEvent, TScroller.ScrollDraw*

ScrollDraw

```
procedure ScrollDraw; virtual;
```

TS

Override: Never

Checks to see if *Delta* matches the current positions of the scroll bars. If not, *Delta* is set to the correct value and *DrawView* is called 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 may trigger scroll bar redraws. Finally, *DrawView* is invoked to redraw the scroller if necessary.

See also: *TScroller.Limit, TScroller.HScrollBar, TScroller.VScrollBar, TScrollBar.SetParams*

SetState

`procedure SetState(AState: Word; Enable: Boolean); virtual;`

Override: Seldom

This method is called whenever the scroller's state changes. Calls *TView.SetState* 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.

See also: *TView.SetState*

Store

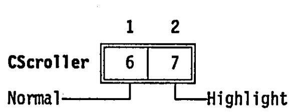
`procedure Store(var S: TStream);`

Writes the scroller to the stream *S* by calling *TView.Store,* then stores references to the scroll bars using *PutPeerViewPtr,* and finally writes the values of *Delta* and *Limit* using *S.Write.*

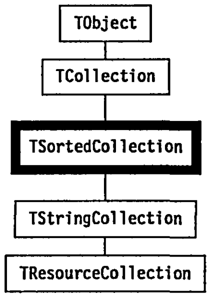
See also: *TScroller.Load, TStream.Write*

Palette

Scroller objects use the default palette, *CScroller,* to map onto the 6th and 7th entries in the standard application palette.



TSortedCollection Objects



TSortedCollection is a specialized derivative of *TCollection* implementing collections sorted by key without duplicates. Sorting is implied by a virtual *TStringCollection.Compare* method which you override to provide your own definition of element ordering. As new items are added they are automatically inserted in the order given by the *Compare* method. Items can be located using the binary search method, *TStringCollection.Search.* The virtual *KeyOf* method that returns a pointer for *Compare,* can also be overridden if *Compare* needs additional information.

Methods

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.

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

If the target item is not found in the sorted collection, it is inserted at the correct index position. Calls *TSortedCollection.Search* to determine if the item exists, and if not, where to insert it. The actual implementation of *TSortedCollection.Insert* is:

```
if not Search(KeyOf(Item), I) 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 *TSortedCollection.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

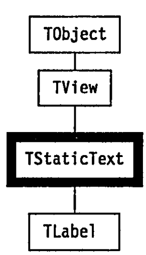
```
function Search(Key: Pointer; var Index: Integer): Boolean; virtual;
```

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.

See also: *TSortedCollection.Compare*, *TSortedCollection.Insert*

TStaticText — Dialogs



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; AText: String);`

Creates a *TStaticText* object of the given size by calling *TView.Init,* then sets *Text* to *NewStr(AText).*

See also: *TView.Init*

Load `constructor Load(var S: TStream);`

Creates and initializes a *TStaticText* object off the given stream. Calls *TView.Load* and sets *Text* with *S.ReadStr.* Used 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 *TView.Done* to destroy 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);`

TS

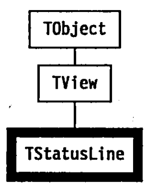
Stores *TStaticText* object on the given stream by calling *TView.Store* and *S.WriteStr*. Used in conjunction with *TStaticText.Store* to save and retrieve static text views on a stream.

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.

TStatusLine — Menus



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 using *InitStatusLine* called by *TApplication*, and the one 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 on 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

Items `Items: PStatusItem;` **Read only**

A pointer to the current linked list of *TStatusItem* records.

See also: *TStatusItem*

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*

Methods

Init `constructor Init(var Bounds: TRect; ADefs: PStatusDef);`

Creates a *TStatusLine* object with the given *Bounds* by calling *TView.Init*. The *ofPreProcess* bit in *Options* is set, *EventMask* is set to include *evBroadcast*, and *GrowMode* is set to *gfGrowLoY* + *gfGrowHiX* + *gfGrowHiY*. The *Defs* field is set to *ADefs*. If *ADefs* is **nil**, *Items* is set to **nil**, otherwise, *Items* is set to *ADefs^.Items*

See also: *TView.Init*

Load `constructor Load(var S: TStream);`

Creates a *TStatusLine* object and loads it from the stream *S* by calling *TView.Load* 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 *TStatusLine* object, then calls *TView.Done.*

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 events sent to the status line by calling TView.HandleEvent, 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

This pseudo-abstract method returns a null string. It must be overridden to provide a context-sensitive hint string for the *AHelpCtx* argument. 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);`

Saves the *TStatusLine* object on the stream *S* by calling *TView.Store* and 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.

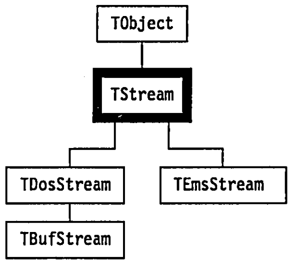
See also: *TStatusLine.Defs*

Palette

Status lines use the default palette *CStatusLine* to map onto the 2nd through 7th entries in the standard application palette.

```
                      1   2   3   4   5   6
                   ╔═══╤═══╤═══╤═══╤═══╤═══╗
CStatusLine        ║ 2 │ 3 │ 4 │ 5 │ 6 │ 7 ║
                   ╚═══╧═══╧═══╧═══╧═══╧═══╝
Text Normal──────────┘   │   │   │   │   └──Selected Shortcut
Text Disabled────────────┘   │   │   └──────Selected Disabled
Text Shortcut────────────────┘   └──────────Selected Normal
```

TStream — Objects



TStream is a general abstract object providing polymorphic I/O to and from a storage device. You can create your own derived 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 derived streams, you must also override *TStream.Flush*.

Fields

Status

`Status: Integer` **Read/write**

Indicates the current status of the stream as follows:

Table 13.1 Stream error codes

TStream error codes	
stOk	No error
stError	Access error
stInitError	Cannot initialize stream
stReadError	Read beyond end of stream
stWriteError	Cannot expand stream
stGetError	Get of unregistered object type
stPutError	Put of unregistered object type

If *Status* is not *stOk* all operations on the stream are suspended until *Reset* is called.

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.

Methods

CopyFrom `procedure CopyFrom(var S: TStream; Count: Longint);`

Copy *Count* bytes from stream *S* to the calling stream object. For example:

```
{Create a copy of entire stream}
NewStream := New(TEmsStream, 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 *TStream.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. The object must have been previously written to the stream by *TStream.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 create and load 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 value of the calling 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 calling stream. This is an abstract method that must be overridden.

Put

```
procedure Put(P: PObject);
```

Writes an object 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 the *P* argument passed to *Put* 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 zero.

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 zero. This method lets you continue stream processing following an error condition that you have corrected.

See also: *TStream.Status, TStream.ErrorInfo*, st*XXXX* 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. *TStream.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*.

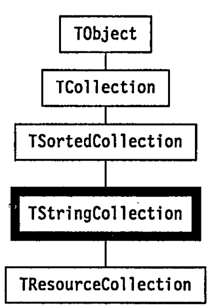
See also: *TStream.Read, TStream.Error.*

WriteStr `procedure WriteStr(P: PString);`

Writes the string *P^* to the calling stream, starting at the current position.

See also: *TStream.ReadStr*

TStringCollection Objects



TSortedCollection is a simple derivative of *TSortedCollection* implementing a sorted list of ASCII strings. The *TStringCollection.Compare* method is overridden to provide the conventional lexicographic 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: *TStringCollection.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 *TStream* by calling *S.ReadStr*.

See also: *TStream.ReadStr*

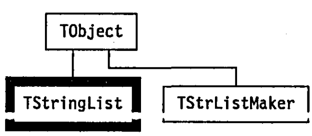
PutItem `procedure PutItem(var S: TStream; Item: Pointer); virtual;`

Override: Seldom

By default, writes the string *Item^* on to the *TStream* 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*

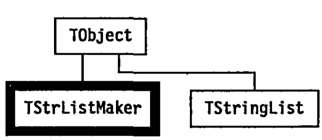
Get `function Get(Key: Word): String;`

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);
```

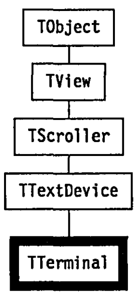
Add the given *String* to the calling string list (with the given numerical *Key*).

Store

```
procedure Store(var S: TStream);
```

Stores the calling string list on the target stream.

TTerminal — TextView



TTerminal implements a "dumb" terminal with buffered string reads and writes. The default is a cyclic buffer of 64K bytes.

Fields

BufSize

```
BufSize: Word;
```

Read only

The size of the terminal's buffer in bytes.

Buffer

```
Buffer: PTerminalBuffer;
```

Read only

Points to the first byte of the terminal's buffer.

QueFront

```
QueFront: Word;
```

Read only

Offset (in bytes) of the first byte stored in the terminal buffer.

QueBack

```
QueBack: Word;
```

Read only

Offset (in bytes) of the last byte stored in the terminal buffer.

Methods

Init

`constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar; ABufSize: Word);`

Creates a *TTerminal* object with the given *Bounds*, horizontal and vertical scroll bars, and buffer by calling *TTextDevice.Init* with the *Bounds* and scroller arguments, then creating a buffer (pointed to by *Buffer*) with *BufSize* equal to *ABufSize*. *GrowMode* is set to *gfGrowHiX* + *gfGrowHiY*. *QueFront* and *QueBack* are both initialized to 0, indicating an empty buffer. The cursor is shown at the view's origin, (0,0).

See also: *TScroller.Init*

Done

`destructor Done; virtual;`

Override: Sometimes

Deallocates the buffer and calls *TTextDevice.Done* to dispose 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 a 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;`

TT

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*

StrRead `function StrRead(var S: TextBuf): Byte; virtual;`

Override: Sometimes

Abstract method returning 0. You must override if you want a derived 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*

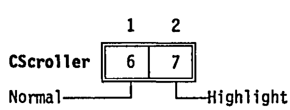
QueEmpty `function QueEmpty: Boolean;`

Returns true if *QueFront* is equal to *QueBack*.

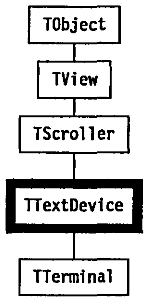
See also: *TTerminal.QueFront, TTerminal.QueBack*

Palette

Terminal objects use the default palette, *CScroller*, to map onto the 6th and 7th entries in the standard application palette.



TTextDevice — TextView



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 in any derived type to read a string from a text device into *S*. The method returns the number of lines read.

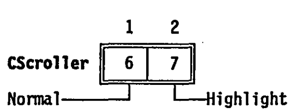
StrWrite `procedure StrWrite(var S: TextBuf; Count: Byte); virtual;`

Override: Always

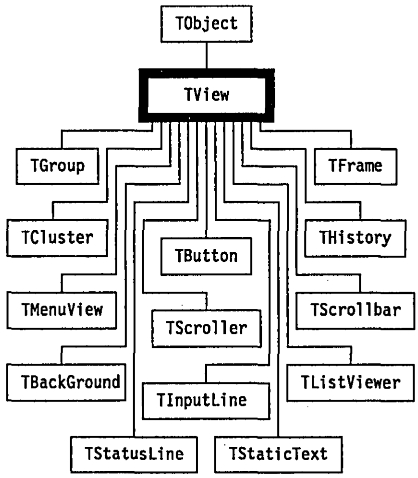
Abstract method to write a string to the device. It must be overridden by derived types. For example, *TTerminal.StrWrite* inserts *Count* lines of the text given by *S* into the terminal's buffer and redraws the view.

Palette

Text device objects use the default palette *CScroller* to map onto the 6th and 7th entries in the standard application palette.



TView — Views



Include the statement

```
uses Views;
```

in programs that make use of *TView*, *TFrame*, *TScrollBar*, *TScroller*, *TListViewer*, *TGroup* and *TWindow* objects. It is hard to envisage a Turbo Vision application that does not use some of these objects.

TView objects are rarely instantiated in Turbo Vision programs. The *TView* object type exists to provide basic fields and methods for its descendants.

Fields

Owner `Owner: PGroup;` **Read only**

Owner points to the *TGroup* object that owns this view. If **nil**, the view has no owner. The view is displayed within its owner's view and will be clipped by the owner's bounding rectangle.

Next `Next: PView;` **Read only**

Pointer to next peer view in Z-order. If this is the last subview, *Next* points to *Owner*'s first subview.

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*

Size `Size: TPoint;` **Read only**

The size of the view.

See also: *GrowTo, Locate*

Cursor `Cursor: TPoint;` **Read only**

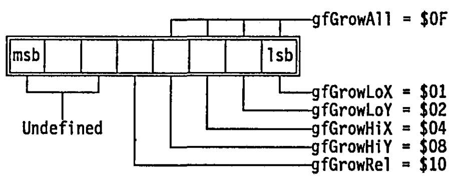
The location of the hardware cursor within the view. The cursor is visible only if the view is focused (*sfFocused*) and the cursor turned on (*sfCursorVis*). The shape of the cursor is either underline or block (determined by *sfCursorIns*).

See also: *SetCursor, ShowCursor, HideCursor, NormalCursor, BlockCursor*

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 13.1 GrowMode bit mapping



Example: `GrowMode := gfGrowLoX or gfGrowLoY;`

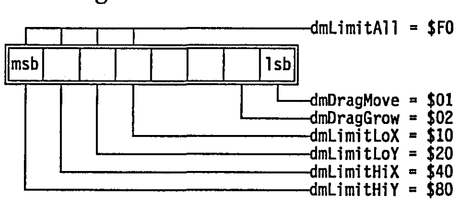
See also: gf*XXXX* grow mode constants

DragMode `DragMode: Byte;` **Read/write**

Determines how the view should behave when mouse-dragged.

The *DragMode* bits are defined as follows:

Figure 13.2 DragMode bit mapping



The *DragMode* masks are defined in Chapter 14 under "dm*XXXX* DragMode constants."

See also: *TView.DragView*

HelpCtx `HelpCtx: Word;` **Read/write**

The help context of the view. When the view is focused, this field will represent the help context of the application unless the context number is *hcNoContext*, in which case there is no help context.

See also: *TView.GetHelpCtx.*

State `State: Word;` **Read only**

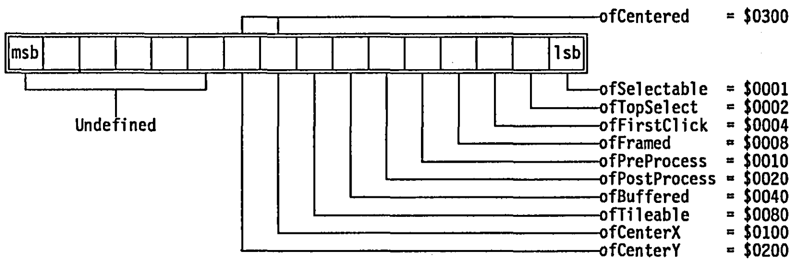
The state of the view is represented by bits set or clear in the *State* field. Many *TView* methods test and/or alter the *State* field by calling *TView.SetState. TView.GetState(AState)* returns true if the view's *State* is *AState.* The *State* bits are represented mnemonically by *sfXXXX* constants, described in Chapter 14 under "sfXXXX state flag constants."

Options `Options: Word;` **Read/write**

The *Options* word flags determine various behaviors of the view.

The *Options* bits are defined as follows:

Figure 13.3
Options bit flags



For detailed descriptions of the option flags, see "ofXXXX option flag constants" in Chapter 14.

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 Chapter 14.

See also: *HandleEvent* methods

Methods

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

Override: Often

Creates a *TView* object with the given *Bounds* rectangle. *Init* calls *TObject.Init* and 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*

Note that *TObject.Init* will zero all fields in *TView* descendants. Always call *TView.Init* *before* initializing any fields.

See also: *TObject.Init*

Load `constructor Load(var S: TStream);`

Override: Often

Creates a *TView* 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 peer view pointers, *Load* should use *GetPeerViewPtr* to read these pointers. An overridden *Load* constructor should always call its parent's *Load* constructor.

The default *TView.Load* sets the *Owner* and *Next* fields to **nil**, and reads the remaining fields from the stream.

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.

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.

TView.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*

BlockCursor

```
procedure BlockCursor;
```

Override: Never

Sets *sfCursorIns* to change the cursor to a solid block. The cursor will only be visible 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 repeatedly 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 the *TView.GrowMode* field.

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 the given *Command* is currently enabled, otherwise it returns false. Note that when you change a modal state, you can then disable and enable commands as you wish; 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 zero 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* argument.

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. *Draw* is seldom called directly, since it is more efficient to use *DrawView,* which draws only views that are exposed, that is, some or all 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;`

TV

Calls *Draw* if *TView.Exposed* returns *True*, indicating that the view is exposed (see *sfExposed*). You should call *DrawView* (not *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* argument.

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 *TGroup.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*

GetBounds `procedure GetBounds(var Bounds: TRect);`

Returns, in the *Bounds* variable, 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 the *Clip* variable, 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 indices 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);
```

Returns, in the *Commands* argument, 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

Returns the next available event in the *TEvent* argument. 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);
```

Returns, in the *Extent* variable, 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.

TV

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 indices 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, de-referencing a peer view pointer within a *Load* constructor does not produce the correct value.

See also: *TView.PutPeerViewPtr, TGroup.Load, TGroup.Store*

GetState `function GetState(AState: Word): Boolean;`

Returns *True* if the state(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 using a call to *TView.Locate.*

See also: *TView.Origin, TView.Size, TView.Locate, TView.MoveTo*

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 the *Event* variable, 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 those of the *Bounds* argument. The view is redrawn in its new location. *Locate* calls *SizeLimits* to verify that the given *Bounds* are 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* may 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, e.g., in drag 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
```

TV

```
    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: Event Masks, *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. The view's *Size* is unchanged.

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. A **nil** is returned if the calling view is the last one 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 *TView.Prev* treats the list as circular, whereas *TView.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. **nil** is returned if the calling view is the first one in its owner's list. Note that *TView.Prev* treats the list as circular, whereas *TView.PrevView* treats the list linearly.

See also: *TView.NextView, TView.Prev*

PutEvent `procedure PutEvent(var Event: TEvent); virtual;`

Override: Seldom

Puts the event given by *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 the *Target* view 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, then 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.

TV

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 (using a call to *TView.MakeFirst*).

See also: *sfSelected, sfFocused, ofTopSelect, TView.MakeFirst*

SetBounds `procedure SetBounds(var Bounds: TRect);`

Sets the bounding rectangle of the view to the value given by the *Bounds* parameter. The *Origin* field is set to *Bounds.A,* and the *Size* field is set to the difference between *Bounds.B* and *Bounds.A*. The *SetBounds* method 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*

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 a state flag in the *TView.State* field. The *AState* parameter specifies the state flag to modify (see *sfXXXX*), and the *Enable* parameter specifies whether to turn the flag off (*False*) or on (*True*). *TView.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*).

SetState is sometimes overridden to trigger additional actions that are 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
  TView.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
  TView.SetState(AState, Enable);
  if AState = sfSelected 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

Returns, in the *Min* and *Max* variables, the minimum and maximum values that the *Size* field may assume.

The default *TView.SizeLimits* returns (0, 0) in *Min* and *Owner^.Size* in *Max.*

See also: *TView.Size*

Store `procedure Store(var S: TStream);`

Override: Often

Stores the view on 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 at the point in time 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.

It is the responsibility of *Valid* to alert the user in case the view is invalid, for example by using the *MessageBox* routine in the *StdDlg* unit to show an error message.

The object types defined in the *StdDlg* unit contain a number of examples of overridden *Valid* methods.

The default *TView.Valid* simply returns *True*.

See also: *TGroup.Valid, TDialog.Valid, TProgram.ValidView*

WriteBuf

```
procedure TView.WriteBuf(X, Y, W, H: Integer; var Buf);
```

Writes the given buffer 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*

WriteChar

```
procedure TView.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 current 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 will be 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.

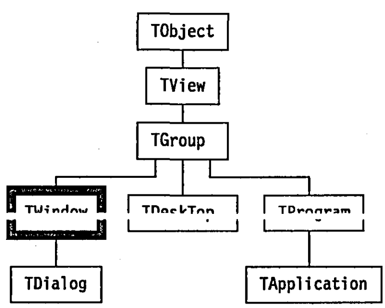
See also: *TView.Draw*

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*

TWindow — Views

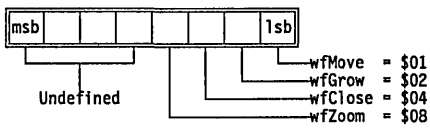


A *TWindow* object is a specialized group that typically owns a *TFrame* object, an interior *TScroller* object, and one or two *TScrollBar* objects. These attached subviews provide the "visibility" to the *TWindow* object. The *TFrame* object provides the familiar border, a place for an optional title 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 Chapter 14.

ZoomRect `ZoomRect: TRect;` **Read only**

The normal, unzoomed boundary of the window.

Number `Number: Integer;` **Read/write**

The number assigned to this window. If *TWindow.Number* is between 1 and 9, the number will appear 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

Frame `Frame: PFrame;` **Read only**

Frame is a pointer to this window's associated *TFrame* object

See also: *TWindow.InitFrame*

Title `Title: PString;` **Read/write**

A character string giving the (optional) title that appears on the frame.

Methods

Init `constructor Init(var Bounds: TRect; ATitle: TTitleStr; ANumber: Integer);`

Calls *TGroup.Init(Bounds)*. Sets default *State* to *sfShadow*. Sets default *Options* to (*ofSelectable* + *ofTopSelect*). Sets default *GrowMode* to *gfGrowAll* +

gfGrowRel. Sets default *Flags* to (*wfMove* + *wfGrow* + *wfClose* + *wfZoom*). Sets *Title* field 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, the default *ZoomRect* is set to the given *Bounds*.

See also: *TFrame.InitFrame*

Load `constructor Load(var S: TStream);`

Creates and loads a window from the stream *S* by first calling *TGroup.Load* and then reading the additional fields that are 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

Closes and disposes of the window, usually in response to a *cmClose* command event. Corresponds to calling the *Done* destructor.

GetPalette `function GetPalette: PPalette; virtual;`

Override: Sometimes

Returns a pointer to the palette given by the palette index in the *Palette* field.

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 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 the null string if the *Title* field is **nil**.

See also: *TWindow.Title, TFrame.Draw*

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Override: Often

First calls *TGroup.HandleEvent*, and then handles events specific to a *TWindow* as follows:

The following *evCommand* events are handled if the *TWindow.Flags* field permits that operation: *cmResize* (move or resize the window using the *TView.DragView* method), *cmClose* (close the window using the

TWindow.Close method), and *cmZoom* (zoom the window using the *TWindow.Zoom* method).

evKeyDown events with a *KeyCode* value of *kbTab* or *kbShiftTab* are handled by selecting the next or previous selectable subview (if any).

An *evBroadcast* event with a *Command* value of *cmSelectWindowNum* is handled by selecting the window if the *Event.InfoInt* field is equal to *TWindow.Number*.

See also: *TGroup.HandleEvent*, *wfXXXX* constants

InitFrame `procedure InitFrame; virtual;`

Override: Seldom

Creates a *TFrame* object for the window and stores a pointer to the frame in the *TWindow.Frame* field. *InitFrame* is called by *TWindow.Init* but should never be called directly. *InitFrame* can be overridden to instantiate a user defined descendant of *TFrame* instead of the standard *TFrame*.

See also: *TWindow.Init*

SetState `procedure SetState(AState: Word; Enable: Boolean); virtual;`

Override: Seldom

First calls *TGroup.SetState*. Then, if *AState* is equal to *sfSelected*, activates or deactivates the window and all its subviews using a call to *SetState(sfActive, Enable)*, and calls *TView.EnableCommands* or *TView.DisableCommands* for *cmNext*, *cmPrev*, *cmResize*, *cmClose*, and *cmZoom*.

See also: *TGroup.SetState*, *EnableCommands*, *DisableCommands*

SizeLimits `procedure SizeLimits(var Min,Max: TPoint); virtual;`

Override: Seldom

Overrides *TView.SizeLimits*. First calls *TView.SizeLimits* and then changes *Min* to return the value stored in the *MinWinSize* global variable.

See also: *TView.SizeLimits*, *MinWinSize* variable

StandardScrollBar `function StandardScrollBar(AOptions: Word): PScrollBar;`

Creates, 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.

The *AOptions* parameter can 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. Either may be combined with *sbHandleKeyboard* to allow the scroll bar to respond to arrows and page keys from the keyboard in addition to mouse clicks.

See also: sb*XXXX* scroll bar constants.

Store `procedure TWindow.Store(var S: TStream);`

Stores the window on the stream *S* by first calling *TGroup.Store* and then writing the additional fields that are introduced by *TWindow*.

See also: *TGroup.Store*

Zoom `procedure TWindow.Zoom; virtual;`

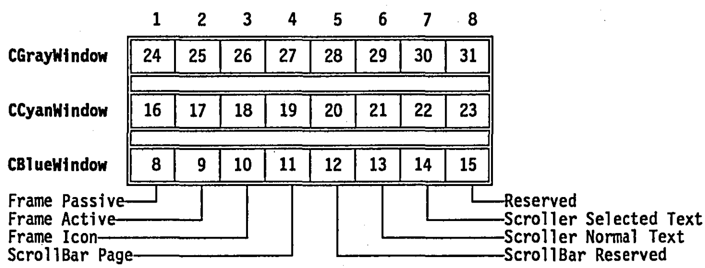
Override: Seldom

Zooms the calling 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).



C H A P T E R

14

Global reference

This chapter describes all the elements of Turbo Vision that are *not* part of the Turbo Vision standard object hierarchy. The standard objects are all described in Chapter 13, "Object reference."

The elements listed in this chapter include types, constants, variables, procedures, and functions defined in the Turbo Vision units. A typical entry looks like this:

Sample procedure — Sample's unit

Declaration `procedure Sample(AParameter);`

Function *Sample* performs some useful function on its parameter, *AParameter*.

See also *Example* function

Abstract procedure — Objects

Declaration `procedure Abstract;`

Function A call to this procedure terminates the program with a run-time error 211. When implementing an *abstract object type*, use calls to *Abstract* in those virtual methods that must be overridden in descendant types. This ensures that any attempt to use instantances of the abstract object type will fail.

See also "Abstract methods" in Chapter 3

Application variable — App

Declaration `Application: PApplication = nil;`

Function The *Application* variable is set to *@Self* at the beginning of *TProgram.Init* (called by *TApplication.Init*) and cleared to **nil** at the end of *TProgram.Done* (called by *TApplication.Done*). Thus, throughout the execution of a Turbo Vision program, *Application* points to the application object.

See also *TProgram.Init*

AppPalette variable — App

Declaration `AppPalette: Integer = apColor;`

Function Selects one of the three available application palettes (*apColor*, *apBlackWhite*, or *apMonochrome*). *AppPalette* is initialized by *TProgram.InitScreen* depending on the current screen mode, and used by *TProgram.GetPalette* to return one of the three available application palettes. You can override *TProgram.InitScreen* to change the default palette selection.

See also *TProgram.InitScreen*, apXXXX constants

apXXXX constants — App

Values The following application palette constants are defined:

Table 14.1 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 Constants beginning with "ap" are used to designate which of three standard color palettes a Turbo Vision application should use. The three palettes are used 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 *TTextDevice*. *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 the text file will read from and write to the *TTextDevice*, using the *StrRead* and *StrWrite* virtual methods. Since *TTextDevice* is an abstract type, the *Screen* parameter typically points to an instance of *TTerminal*, which implements a fully functional TTY-like scrolling view.

See also *TTextDevice*; *TextRec*

bfXXXX constants — Dialogs

Values The following button flags are defined:

Table 14.2 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-justified

Function

A combination of these values is passed to *TButton.Init* to determine the newly created button's style. *bfNormal* indicates a normal, non-default button. *bfDefault* indicates that the button will be the default button. It is the responsibility of the programmer to ensure that there is only one default button in a *TGroup*. The *bfLeftJust* value can be added to *bfNormal* or *bfDefault* and affects the position of the text displayed within the button: If clear, the label is centered; if set, the label is left-justified.

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.

CheckSnow variable — Drivers

Declaration `CheckSnow: Boolean`

Function *CheckSnow* performs the same function as the flag of the same name in the standard Turbo Pascal *Crt* unit. Snow checking is only needed to slow down screen output for some older CGA adapters.

CheckSnow is set *True* by *InitVideo* only if a CGA adapter is detected. The user may 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.

C

ClearScreen procedure — Drivers

Declaration `procedure ClearScreen;`

Function Clears the screen. *ClearScreen* assumes that *InitVideo* has been called first. You seldom need to use this routine, as is explained in the description of *InitVideo*.

See also *InitVideo*

cm*XXXX* constants — Views

Function These constants represent Turbo Vision's predefined *commands*. They are passed in the *TEvent.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 by Turbo Vision and used by standard Turbo Vision objects:

Table 14.3 Standard command codes

Command	Value	Meaning
cmValid	0	Passed to *TView.Valid* to check the validity of a newly instantiated view.
cmQuit	1	Causes *TProgram.HandleEvent* to call *EndModal(cmQuit)*, terminating the application. The status line or one of the menus typically contains an entry that maps *kbAltX* to *cmQuit*.
cmError	2	Never handled by any object. May be used to represent unimplemented or unsupported commands.
cmMenu	3	Causes *TMenuView.HandleEvent* to call *ExecView* on itself to perform a menu selection process, the result of which may generate a new command through *PutEvent*. The status line typically contains an entry that maps *kbF10* to *cmMenu*.
cmClose	4	Handled by *TWindow.HandleEvent* if the *InfoPtr* field of the event record is **nil** or points to the window. If the window is modal (such as a modal dialog), an

Table 14.3: Standard command codes (continued)

		evCommand with a value of *cmCancel* is generated through *PutEvent*. If the window is modeless, the window's *Close* method is called if the window supports closing (see *wfClose* flag). A click in a window's close box generates an *evCommand* event with a *Command* of *cmClose* and an *InfoPtr* that points to the window. The status line or one of the menus typically contains an entry that maps *kbAltF3* to *cmClose*.
cmZoom	5	Causes *TWindow.HandleEvent* to call *TWindow.Zoom* on itself if the window supports zooming (see *wfZoom* flag) and if the *InfoPtr* field of the event record is **nil** or points to the window. A click in a window's zoom box or a double-click on a window's title bar generates an *evCommand* event with a *Command* of *cmZoom* and an *InfoPtr* that points to the window. The status line or one of the menus typically contains an entry that maps *kbF5* to *cmZoom*.
cmResize	6	Causes *TWindow.HandleEvent* to call *TView.DragView* on itself if the window supports resizing (see *wfMove* and *wfGrow* flags). The status line or one of the menus typically contains an entry that maps *kbCtrlF5* to *cmResize*.
cmNext	7	Causes *TDeskTop.HandleEvent* to move the last window on the desktop in front of all other windows. The status line or one of the menus typically contains an entry that maps *kbF6* to *cmNext*.
cmPrev	8	Causes *TDeskTop.HandleEvent* to move the first window on the desktop behind all other windows. The status line or one of the menus typically contains an entry that maps *kbShiftF6* to *cmPrev*.

The following standard commands are used to define default behavior of dialog box objects:

Table 14.4 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's *TDialog.HandleEvent* to terminate the dialog and return that value (by calling *EndModal*). A modal dialog typically contains at least one *TButton* with one of these command values. *TDialog.HandleEvent* will generate a *cmCancel* command event in response to a *kbEsc* keyboard event.

C

The *cmDefault* command causes the *TButton.HandleEvent* of a default button (see *bfDefault* flag) to simulate a button press. *TDialog.HandleEvent* will generate a *cmDefault* command event in response to a *kbEnter* keyboard event.

The following standard commands are defined for use by standard views:

Table 14.5 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 *TView.Owner* whenever *sfFocused* is changed. The *InfoPtr* of the event points to the view ifself. This in effect informs any peer views that the view has received or released focus, and that they should update themselves appropriately. *TLabel* objects, for example, respond to these commands by highlighting or unhighlighting themselves when the peer view they label is focused or unfocused.
cmCommandSetChanged	52	The *TProgram.Idle* method generates an *evBroadcast* event with this value whenever it detects a change in the current command set (as caused by a call to *TView's* *EnableCommands*, *DisableCommands*, or *SetCommands* methods). The *cmCommandSetChanged* broadcast is sent to the *HandleEvent* of *every* view in the physical hierarchy (unless their *TView.EventMask* specifically masks out *evBroadcast* events). If a view's appearance is affected by command set changes, it should react to *cmCommandSetChanged* by redrawing itself. *TButton*, *TMenuView*, and *TStatusLine* objects, for example, react to this command by redrawing themselves.
cmScrollBarChanged	53	A *TScrollBar* uses the *Message* function to send
cmScrollBarClicked	54	an *evBroadcast* event with one of these values to its *TView.Owner* whenever its value changes or whenever the mouse is clicked on the scroll bar. The *InfoPtr* of the event points to the scroll bar ifself. Such broadcasts are reacted upon by any peer views controlled by the scroll bar, such as *TScroller* and *TListViewer* objects.
cmSelectWindowNum	55	Causes *TWindow.HandleEvent* to call *TView.Select* on itself if the *InfoInt* of the event

Table 14.5: Standard view commands (continued)

		record corresponds to *TWindow.Number*. *TProgram.HandleEvent* responds to *Alt-1* through *Alt-9* keyboard events by broadcasting a *cmSelectWindowNum* event with an *InfoInt* of 1 through 9.
cmRecordHistory	60	Causes a *THistory* object to "record" the current contents of the *TInputLine* object it controls. A *TButton* sends a *cmRecordHistory* broadcast to its owner when it is pressed, in effect causing all *THistory* objects in a dialog to "record" at that time.

See also *TView.HandleEvent*, *TCommandSet*

co*XXXX* constants — Objects

Function The *coXXXX* constants are passed as the *Code* parameter to the *TCollection.Error* method when a *TCollection* detects an error during an operation.

Values The following standard error codes are defined for all Turbo Vision collections:

Table 14.6 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*

C

CtrlBreakHit variable — Drivers

Declaration `CtrlBreakHit: Boolean = False;`

Function Set *True* by the Turbo Vision keyboard interrupt driver whenever the *Ctrl-Break* key is pressed. This allows Turbo Vision applications to trap and respond to *Ctrl-Break* as a user control. The flag may be cleared at any time simply 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 14.7, the result is the corresponding *kbXXXX* constant. Otherwise, *KeyCode* is returned unchanged.

Table 14.7 Control-key mappings

Keystroke	Lo(KeyCode)	Result
Ctrl-A	$01	*kbHome*
Ctrl-D	$04	*kbRight*
Ctrl-E	$05	*kbUp*
Ctrl-F	$06	*kbEnd*
Ctrl-G	$07	*kbDel*
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 used 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*

DeskTop variable — App

Declaration `DeskTop: PDeskTop = nil;`

Function Stores a pointer to the application's *TDeskTop*. The *DeskTop* variable is initialized by *TProgram.InitDeskTop*, which is called by *TProgram.Init*. Windows and dialog boxes are normally inserted (*TGroup.Insert*) or executed (*TGroup.ExecView*) on the *DeskTop*.

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

DisposeStr procedure — Objects

Declaration `procedure DisposeStr(P: PString);`

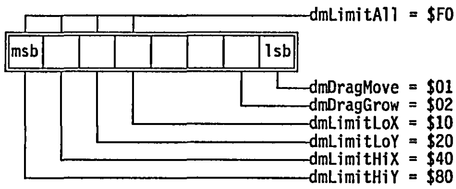
Disposes of a string allocated on the heap by the *NewStr* function.

See also *NewStr*

dm*XXXX* constants — Views

Values The *DragMode* bits are defined as follows:

Figure 14.1
Drag mode bit flags



Function The drag mode constants are used to compose the *Mode* parameter of the *TView.DragView* method. They specify whether the view is allowed to move and/or change size, and how to interpret the *Limits* parameter.

The drag mode constants are defined as follows:

Table 14.8
Drag mode constants

Constant	Meaning
dmDragMove	Allow the view to move.
dmDragGrow	Allow the view to change size.
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*.

The *DragMode* field of a *TView* may contain 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.

DoneEvents procedure — Drivers

Declaration `procedure DoneEvents;`

Function Terminates Turbo Vision's event manager by disabling the mouse interrupt handler and hiding the mouse. Called automatically by *TApplication.Done*.

See also *TApplication.Done, InitEvents*

DoneHistory procedure — HistList

Declaration `procedure DoneHistory;`

Function Frees the history block allocated by *InitHistory*. Called automaticallv 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 automatically 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 automatically 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 automatically 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*

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*

ev*XXXX* constants — Drivers

Function These mnemonics indicate types of events to Turbo Vision event handlers. evXXXX constants are used in several places: In the *What* field of an event record, in the *EventMask* field of a view object, and in the *PositionalEvents* and *FocusedEvents* variables.

Values The following event flag values designate standard event types:

Table 14.9 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

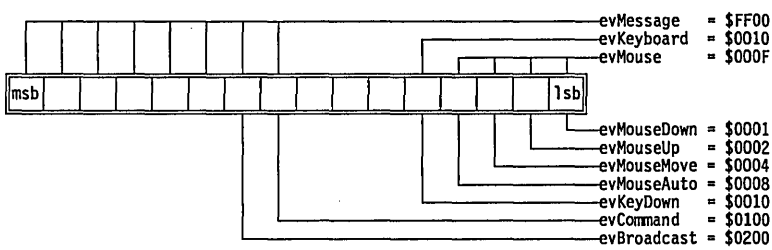
The following constants can be used to mask types of events:

Table 14.10 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 14.2 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;
```

See also *TEvent, TView.EventMask, GetKeyEvent, GetMouseEvent, HandleEvent* methods, *PositionalEvents, FocusedEvents*

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 an event to the group's subviews. If an event class isn't contained in *FocusedEvents* or *PositionalEvents* it is treated as a broadcast event.

See also *PositionalEvents* variable, *TGroup.HandleEvent*, TEvent, evXXXX constants

FormatStr procedure — Drivers

Declaration `procedure FormatStr(var Result: String; 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 is to be used 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-justified (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 14.11
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 *Longints* or pointers or a record containing *Longints* 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 either of two ways, 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

F

FreeBufMem procedure — Memory

Declaration `procedure FreeBufMem(P: Pointer);`

Function Frees the cache buffer referenced by the pointer *P*.

See also *GetBufMem, DoneMemory*

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.* This function 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*

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*. If there is no room for a cache buffer of the requested size, *P* is set to **nil**. Cache buffers differ from normal heap blocks (allocated by *New*, *GetMem*, or *MemAlloc*) in that they can be moved or disposed by the memory manager at any time to satisfy a normal memory allocation request. The pointer passed to *GetBufMem* becomes the cache buffer's *master pointer*, and it (and only it) is updated when the buffer is moved by the memory manager. If the memory manager decides to dispose the buffer, it sets the master pointer to **nil**. A cache buffer can be manually disposed through a call to *FreeBufMem*. Cache buffers will occupy any unallocated heap space between *HeapPtr* and *HeapEnd*, including the area set aside for the application's *safety pool*.

Turbo Vision uses cache buffers to cache the contents of *TGroup* objects (such as windows, dialog boxes, and the desktop) whenever these objects have the *ofBuffered* flag set—this greatly increases performance of redraw operations.

See also *FreeBuffMem*, *InitMemory*, *TGroup.Draw*

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

gf*XXXX* 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 14.3 Grow mode bit mapping

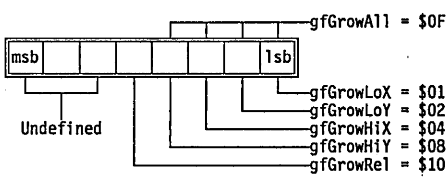


Table 14.12 Grow mode flag definitions

Constant	Meaning
gfGrowLoX	If set, the left-hand side of the view will maintain a constant distance from its owner's right-hand side.
gfGrowLoY	If set, the top of the view will maintain a constant distance from the bottom of its owner.
gfGrowHiX	If set, the right-hand side of the view will maintain a constant distance from its owner's right side.
gfGrowHiY	If set, the bottom of the view will maintain a constant distance from the bottom of its owner's.
gfGrowAll	If set, the view will move with the lower-right corner of its owner.
gfGrowRel	For use with *TWindow* objects that are in the desktop: The view will change size relative to the owner's size. The window will maintain 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*

hc*XXXX* constants — Views

Values The following help context constants are defined:

Table 14.13 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 through 999 for its own use. Programmers may define their own constants in the range 1,000 to 65,535.

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* will decrement this counter, and show 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 *True* by *InitVideo* if the screen supports 43- or 50-line mode (EGA or VGA); otherwise set *False*.

See also *InitVideo*

HistoryAdd procedure — HistList

Declaration `procedure HistoryAdd(Id: Byte; var Str: String);`

Function Adds the string *Str* to the history list indicated by *Id*.

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 corresponding to ID number *Id*.

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 may 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 corresponding to ID number *Id*.

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.

InitEvents procedure — Drivers

Declaration `procedure InitEvents;`

Function Initializes Turbo Vision's event manager by enabling the mouse interrupt handler and showing the mouse. Called automatically 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 automatically 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 automatically 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 automatically by *TApplication.Init*.

See also *DoneVideo, SetVideoMode, smXXXX*

kb*XXXX* constants — Drivers

Function There are two sets of constants beginning with "kb," both dealing 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 $40:$17. For example,

```
var
  ShiftState: Byte absolute $40:$17;
  ...
if ShiftState and kbAltShift <> 0 then AltKeyDown;
```

Table 14.14 Keyboard state and shift masks

Constant	Value	Meaning
kbRightShift	$0001	Set if the Right Shift key is currently down
kbLeftShift	$0002	Set if the Left Shift key is currently down
kbCtrlShift	$0004	Set if the Ctrl key is currently down
kbAltShift	$0008	Set if the Alt key is currently down
kbScrollState	$0010	Set if the keyboard is in the Scroll Lock state
kbNumState	$0020	Set if the keyboard is in the Num Lock state
kbCapsState	$0040	Set if the keyboard is in the Caps Lock state
kbInsState	$0080	Set if the keyboard is in the Ins Lock state

The following values define keyboard scan codes and can be used when examining the *TEvent.KeyCode* field of an *evKeyDown* event record:

Table 14.15
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

K

Table 14.16
Special key codes

Constant	Value	Constant	Value
kbAltEqual	$8300	*kbEnd*	$4F00
kbAltMinus	$8200	*kbEnter*	$1C0D
kbAltSpace	$0200	*kbEsc*	$011B
kbBack	$0E08	*kbGrayMinus*	$4A2D
kbCtrlBack	$0E7F	*kbHome*	$4700
kbCtrlDel	$0600	*kbIns*	$5200
kbCtrlEnd	$7500	*kbLeft*	$4B00
kbCtrlEnter	$1C0A	*kbNoKey*	$0000
kbCtrlHome	$7700	*kbPgDn*	$5100
kbCtrlIns	$0400	*kbPgUp*	$4900
kbCtrlLeft	$7300	*kbrayPlus*	$4E2B
kbCtrlPgDn	$7600	*kbRight*	$4D00
kbCtrlPgUp	$8400	*kbShiftDel*	$0700
kbCtrlPrtSc	$7200	*kbShiftIns*	$0500
kbCtrlRight	$7400	*kbShiftTab*	$0F00
kbDel	$5300	*kbTab*	$0F09
kbDown	$5000	*kbUp*	$4800

Table 14.17
Alt-number kev 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 14.18
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 14.19
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 14.20
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 14.21
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*

LongDiv function — Objects

Declaration

```
function LongDiv(X: Longint; Y: Integer): Integer;
inline($59/$58/$5A/$F7/$F9);
```

Function A fast, inline assembly coded 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.

L

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 6, "Writing safe programs," *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 usual 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 that maximum number of elements that may be contained in a collection, which is essentially the number of pointers that can fit in a 64K memory segment.

MaxViewWidth constant — Views

Declaration `MaxViewWidth = 132;`

Function Sets the maximum width of a view.

See also *TView.Size* field

mb*XXXX* 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 14.22 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. As opposed to the *New* and *GetMem* standard procedures, *MemAlloc* will not allow the allocation to dip into the safety pool. A block allocated by *MemAlloc* can be disposed using the *FreeMem* standard procedure.

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 that the offset part of the resulting pointer value is guaranteed to be 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.

Message function — Views

Declaration

```
function Message(Receiver: PView; What, Command: Word; InfoPtr:
  Pointer): Pointer;
```

Function *Message* sets up a command event with the arguments *What, Command* and *InfoPtr* then, if possible, invokes *Receiver^.HandleEvent* to handle this event. Message returns **nil** if *Receiver* is **nil**, or if the event is not handled successfully. If the event is handled successfully (that is, if *HandleEvent* returns *Event.What* as *evNothing*), the function returns *Event.InfoPtr*. The latter can be used to determine which view actually handled the dispatched event. The *What* argument is usually set to *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

MinWinSize variable — Views

Declaration

```
MinWinSize: TPoint = (X: 16; Y: 6);
```

Function *MinWinSize* defines the minimum size of a *TWindow* or a descendant of *TWindow*. The value is returned in the *Min* parameter on a call to *TWindow.SizeLimits*. Any change to *MinWinSize* will affect all windows, unless a window's *SizeLimits* method is overridden.

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. The *mbXXXX* constants can be used to examine *MouseButtons*.

See also *mbXXX* constants

MouseEvents variable — Drivers

Declaration `MouseEvents: Boolean = False;`

Function Set *True* if a mouse is installed and detected by *InitEvents*; otherwise set *False*. If *False*, all mouse event routines are bypassed.

See also *GetMouseEvent*

MouseIntFlag variable — Drivers

Declaration `MouseIntFlag: Byte;`

Function Used internally by Turbo Vision mouse driver and by views. Set whenever a mouse event occurs.

MouseWhere variable — Drivers

Declaration `MouseWhere: TPoint;`

Function Contains the current position of the mouse in global coordinates. *MouseWhere* is updated by the mouse interrupt handler whenever the mouse is moved. Use the *MakeLocal* routine 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 into a buffer to be used with *TView.WriteBuf* or *TView.WriteLine*. *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*

M

MoveChar procedure — Drivers

Declaration `procedure MoveChar(var Dest; C: Char; Attr: Byte; Count: Word);`

Function Moves characters into a buffer to be used with *TView.WriteBuf* or *TView.WriteLine*. *Dest* must be *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; Str: String; Attrs: Word);`

Function Moves a two-colored string into a buffer to be used with *TView.WriteBuf* or *TView.WriteLine*. *Dest* must be *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 are used to 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; Str: String; Attr: Byte);`

Function Moves a string into a buffer to be used with *TView.WriteBuf* or *TView.WriteLine*. *Dest* must be *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*

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 (*NewStr* is used 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 Pascal statement; for examples of this, refer to Chapter 2, "Writing Turbo Vision applications."

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*

N

NewMenu function — Menus

Declaration `function NewMenu(Items: PMenuItem): PMenu;`

Function Allocates and returns a pointer to a new *TMenu* record. The *Items* and *Default* fields of the record are set to the value given by the *Items* parameter.

See also *TApplication.InitMenuBar, TMenuView* type, *NewLine, NewSubMenu, NewItem*

NewSItem function — Dialogs

Declaration `function NewSItem(Str: String; ANext: PSItem): PSItem;`

Function Allocates and returns a pointer to a new *TSItem* record. The *Value* and *Next* fields of the record are set 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, refer to Chapter 4, "Views."

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. The record is 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, refer to Chapter 2, "Writing Turbo Vision applications."

See also *TApplication.InitStatusLine, TStatusLine, NewStatusKey*

NewStatusKey function — Menus

Declaration `function NewStatusKey(AText: String; AKeyCode: Word; ACommand: Word; ANext: PStatusItem): PStatusItem;`

Function Allocates and returns a pointer to a new *TStatusItem* record. The record is initialized with the given parameter values (*NewStr* is used to allocate the *Text* pointer field). If *AText* is empty (which results in a **nil** *Text* field), the status item is hidden, but will still provide a mapping from the given *KeyCode* to the given *Command*.

See also *TApplication.InitStatusLine, TStatusLine, NewStatusDef*

NewStr function — Objects

Declaration `function NewStr(S: String): PString;`

Function Dynamic string routine. If *S* is nul, *NewStr* returns a **nil** pointer; otherwise, *Length(S)*+1 bytes is allocated containing a copy of *S*, and a pointer to the first byte is returned.

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;`

Function Allocates and returns a pointer to a new *TMenuItem* record, which represents a submenu (*NewStr* is used to allocate the *Name* pointer field).

See also *TApplication.InitMenuBar*, *TMenuView* type, *NewLine*, *NewItem*, *NewItem*

N

of*XXXX* constants — Views

Function These mnemonics are used to refer to the bit positions of the *TView.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 14.23 Option flags

Constant	Meaning
ofSelectable	Set if the view should select itself automatically (see *sfSelected*), for example, by a mouse click in the view, or a *Tab* in a dialog box.
ofTopSelect	Set if the view should move in front of all other peer views when selected. When the *ofTopSelect* bit is set, a call to *TView.Select* corresponds to a call to *TView.MakeFirst*. Windows (*TWindow* and descendants) by default have the

Table 14.23: Option flags (continued)

	ofTopSelect bit set, which causes them to move in front of all other windows on the desktop when selected. See also *TView.Select, TGroup.MakeFirst.*
ofFirstClick	If clear, a mouse click that selects a view will have no further effect. If set, such a mouse click is processed as a normal mouse click after selecting the view. Has no effect unless *ofSelectable* is also set. See also *TView.HandleEvent, sfSelect, ofSelectable.*
ofFramed	Set if the view should have a frame drawn around it. A *TWindow,* and any descendant of *TWindow,* has a *TFrame* as its last subview. When drawing itself, the *TFrame* will also draw a frame around any other subviews that have the *ofFramed* bit set. See also *TFrame, TWindow.*
ofPreProcess	Set if the view should receive focused events before they are sent to the focused view. Otherwise clear. See also *sfFocused, ofPostProcess, TGroup.Phase.*
ofPostProcess	Set if the view should receive focused events in the event that the focused view failed to handle them. Otherwise clear. See also *sfFocused, ofPreProcess, TGroup.Phase.*
ofBuffered	Used for *TGroup* objects only: Set if 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. See also *GetBufMem.*
ofTileable	Set if the desktop can tile (or cascade) this view. Usually used only with *TWindow* objects.
ofCenterX	Set if the view should be centered on the X-axis of its owner when inserted in a group using *TGroup.Insert.*
ofCenterY	Set if the view should be centered on the Y-axis of its owner when inserted in a group using *TGroup.Insert.*
ofCentered	Set if the view should be centered on both axes of its owner when inserted in a group using *TGroup.Insert.*

The *Options* bits are defined as follows:

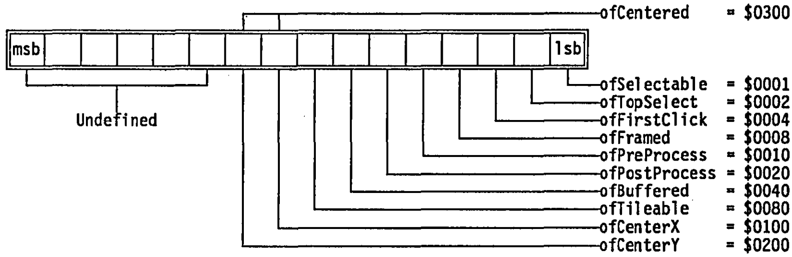


Figure 14.4
Options bit flags

See also *TView.Options*

PChar type — Objects

Declaration `PChar = ^Char;`

Function Defines a pointer to a character.

PositionalEvents variable — Views

Declaration `PositionalEvents: Word = evMouse;`

Function Defines the event classes that are *positional events*. The *FocusedEvents* and *PositionalEvents* variables are used by *TGroup.HandleEvent* to determine how to dispatch an event to the group's subviews. If an event class isn't contained in *FocusedEvents* or *PositionalEvents*, it is treated as a broadcast event.

See also *TGroup.HandleEvent*, *TEvent* type, evXXXX event 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.

O

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.

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*. This allows any of these objects to be used with stream I/O.

See also *TStreamRec, RegisterTypes*

Registertype procedure — Objects

Declaration
```
procedure RegisterType(var S: TStreamRec);
```

Function A Turbo Vision object type must be registered using this method before it can be used in stream I/O. The standard object types are preregistered with *ObjTypes* in the reserved range 0..99. *RegisterType* creates an entry in a linked list of *TStreamRec* records.

See also *TStream.Get, TStream.Put, TStreamRec*

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 starts being generated. The time interval between *evMouseAuto* events is always one clock tick.

See also *DoubleDelay*, *GetMouseEvent*, evXXXX constants

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*

sb*XXXX* constants — Views

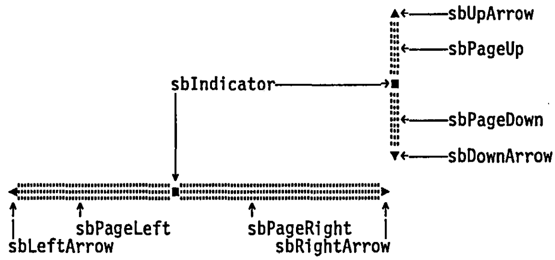
Function These constants define the different areas of a *TScrollBar* in which the mouse can be clicked.

The *TScrollBar.ScrollStep* function serves to convert these constants into actual scroll step values. Although defined, the *sbIndicator* constant is never passed to *TScrollBar.ScrollStep*.

Table 14.24 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 14.5
Scroll bar parts



The following values can be passed to the *TWindow.StandardScrollBar* function:

Table 14.25
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 in 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*

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 *TApplication.SetScreenMode*, which also adjusts the application palette.

See also *InitVideo*, smXXXX constants, *TApplication.SetScreenMode*

sf*XXXX* constants — Views

Function These constants are used to access the corresponding bits in *TView.State* fields. *TView.State* fields must never be modified directly; instead, you should use the *TView.SetState* method.

Values The following state flags are defined:

Table 14.26
State flag constants

Constant	Meaning
sfVisible	Set if the view is visible on its owner, otherwise clear. Views are by default *sfVisible*. The *TView.Show* and *TView.Hide* methods may be used to 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, examine the *sfExposed* bit or call the *TView.Exposed* function.
sfCursorVis	Set if a view's cursor is visible, otherwise clear. The default is clear. The *TView.ShowCursor* and *TView.HideCursor* methods may be used to modify *sfCursorVis*.
sfCursorIns	Set if the view's cursor is a solid block, clear if the view's cursor is an underline. The default is clear. The *TView.BlockCursor* and *TView.NormalCursor* methods can be used to modify *sfCursorIns*.
sfShadow	Set if the view has a shadow, otherwise clear.
sfActive	Set if the view is the active window or a subview in the active window.
sfSelected	Set if the view is the currently selected subview within its owner. Each *TGroup* object has a *Current* field that points to the currently selected subview (or is **nil** if no subview is selected). There can be only one currently selected subview in a *TGroup*.

Table 14.26: State flag constants (continued)

sfFocused	Set if the view is focused. 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 *TGroups* starting at *Application* (the topmost view in the view hierarchy). The last view on the focused chain is the final target of all *focused events*.
sfDragging	Set if the view is being dragged, otherwise clear.
sfDisabled	Set if the view is disabled; clear if the view is enabled. A disabled view will ignore all events sent to it.
sfModal	Set if the view is modal. There is always exactly *one* modal view in a running Turbo Vision application, usually a *TApplication* or *TDialog* object. 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. See also *sfSelected, sfFocused, TView.SetState, TView.HandleEvent, TGroup.ExecView.*
sfExposed	Set if the view is owned directly or indirectly by the *Application* object, and therefore possibly visible on the screen. The *TView.Exposed* method uses this flag in combination with further clipping calculations to determine whether any part of the view is actually visible on the screen. See also *TView.Exposed.*

Values The state flag bits are defined as follows:

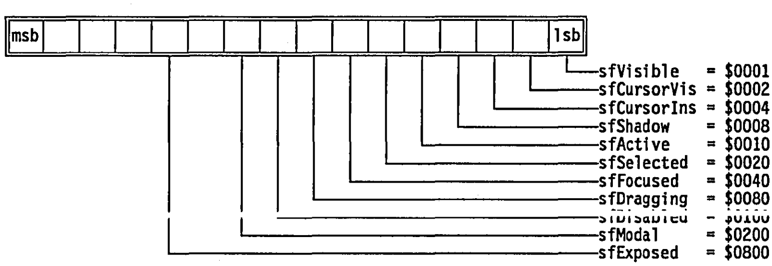


Figure 14.6 State flag bit mapping

See also *TView.State*

S

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 — Drivers

Declaration `ShowMarkers: Boolean;`

Function Used to indicate whether indicators should be placed around focused controls. *TProgram.InitScreen* sets *ShowMarkers* to *True* if the video mode is monochrome, otherwise it is *False*. The value may, however, be set on in color and black and white modes if desired.

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*

sm*XXXX* 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 14.27 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. These characters are displayed if the *ShowMarkers* variable is *True.*

See also *ShowMarkers* variable

S

stXXXX constants — Objects

Function There are two sets of constants beginning with "st" that are 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 14.28 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

The following values are returned by *TStream.Error* in the *TStream.ErrorInfo* field when a stream error occurs:

Table 14.29 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;`

Function The *InitVideo* routine 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 Stores a pointer to the application's status line. The *StatusLine* variable is initialized by *TProgram.InitStatusLine,* which is called by *TProgram.Init.* A value of **nil** indicates that the application has no status line.

See also *InitStatusLine*

StreamError variable — Objects

Declaration `StreamError: Pointer = nil;`

Function In non-**nil**, *StreamError* points to a procedure that will be 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. To change error handling for a particular type of stream you should override that stream type's *Error* method.

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 message 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*

S

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 14.30, 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 14.30 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 14.31 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 message 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 14.32 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*

TByteArray type — Objects

Declaration

```
TByteArray = array[0..32767] of Byte;
```

Function A byte array type for general use in typecasts.

See also *TStringListMaker*

S

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*. A command set can be declared and initialized using the Pascal set syntax:

```
CurCommandSet: TCommandSet = [0..255] - [cmZoom, cmClose, cmResize, cmNext];
```

See also *cmXXXX, TView.DisableCommands, TView.EnableCommands, TViewGetCommands, TView.SetCommands.*

TDrawBuffer type — Views

Declaration `TDrawBuffer = array[0..MaxViewWidth-1] of Word;`

Function The *TDrawBuffer* type is used to declare buffers for a variety of view *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);
  WriteLine(0, 0, Size.X, Size.Y, B);
end;
```

See also *TView.Draw, MoveBuf, MoveChar, MoveCStr, MoveStr*

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*

TItemList type — Objects

Declaration `TItemList = array[0..MaxCollectionSize - 1] of Pointer;`

Function An array of generic pointers used internally by *TCollection* objects.

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 *TMenuItems,* 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)

T

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

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* within 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 cm*XXXX* 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 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 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, *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*

TPalette type — Views

Declaration `TPalette = String;`

Function A string type used to declare Turbo Vision palettes.

See also *GetPalette* methods

TScrollChars type — Views

Declaration `TScrollChars = array[0..4] of Char;`

Function An array representing the characters used to draw a *TScrollBar*.

See also *TScrollBar*

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.

T

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 line. *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 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*

TStreamRec type — Objects

Declaration

```
PStreamRec = ^TStreamRec;
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 14.33
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*.

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*

TTerminalBuffer type — TextView

Declaration `TTerminalBuffer = array[0..65519] of Char;`

Function Used internally by *TTerminal*.

See also *TTerminal*

TTitleStr type — Views

Declaration `TTitleStr = string[80];`

Function This type is used to declare text strings for titled windows.

See also *TWindow.Title*

TVideoBuf type — Views

Declaration `TVideoBuf = array[0..3999] of Word;`

Function This type is used to declare video buffers.

See also *TGroup.Buffer*

TWordArray type — Objects

Declaration `TWordArray = array[0..16383] of Word;`

Function A word array type for general use.

wf*XXXX* constants — Views

Function These mnemonics define bits in the *Flags* field of *TWindow* objects. If the bits are set, the window will have the corresponding attribute: The window can move, grow, close, or zoom.

Values The window flags are defined as follows:

T

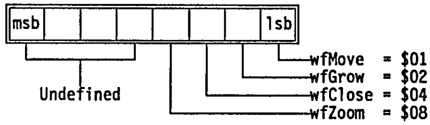


Table 14.34
Window flag constants

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*

wnNoNumber constant — Views

Declaration

```
wnNoNumber = 0;
```

Function If the *TWindow.Number* field holds this constant, it indicates that the window is not to be numbered and cannot be selected via the *Alt*+number key. If the *Number* field is between 1 and 9, the window number is displayed, and *Alt*-number selection is available.

See also *TWindow.Number*

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*

wp*XXXX* constants — Views

Function These constants define the three standard color mapping assignments for windows. By default, a *TWindow* object has a *Palette* of *wpBlueWindow*. The default for *TDialog* objects is *wpGrayWindow*.

Values Three standard window palettes are defined:

Table 14.35 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

A

B

C

D

E

F

G

H

K

L

M

Q

R

T

U

V

W

X

Y

Z

6.0

TURBO VISION GUIDE

TURBO PASCAL®

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300
Offices in: Australia, Denmark, England, France, Germany, Italy, Japan and Sweden ■ Part# 11MN-PAS04-60 ■ BOR 1853