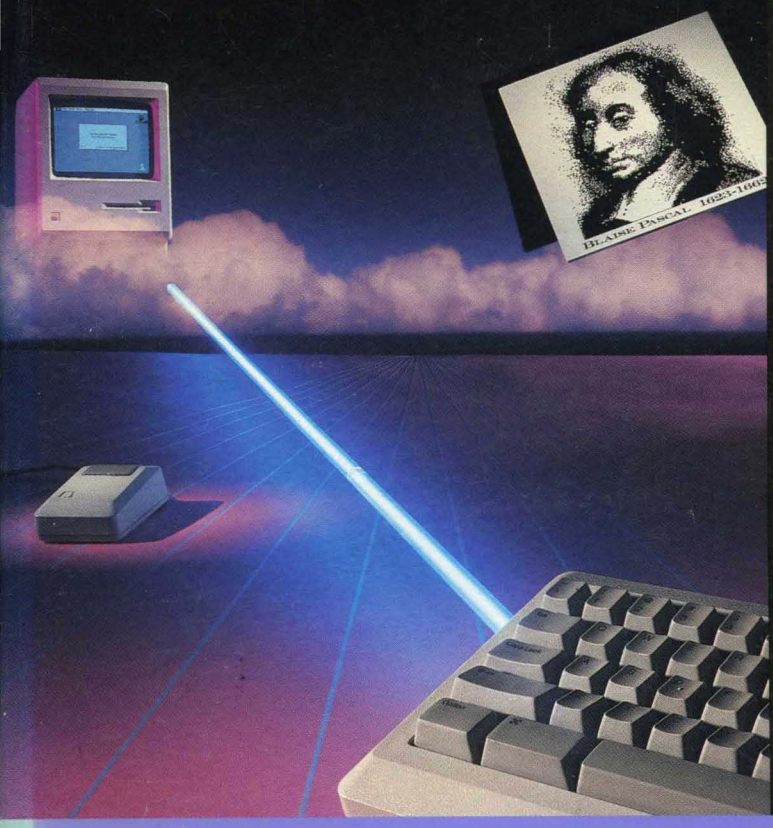
TURBO PASCAL®

The ultimate Pascal development environment

Incredibly fast—compiles more than 12,000 lines a minute!



MACINTOSH™

BORLAND

TURBO PASCAL FOR THE MAC

User's Guide and Reference Manual

BORLAND INTERNATIONAL, INC.
4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066
USA

Table of Contents

Introduction

Welcome to *Turbo Pascal for the Mac*. The programming language Turbo Pascal is designed to meet the needs of all types of Macintosh users: It's a structured, high-level language that can be used to write programs for almost any application.

This manual walks novice programmers through writing, compiling, and saving Turbo Pascal programs. It also teaches you how to take existing Pascal programs and run them under Turbo Pascal.

Sample programs are provided on your distribution disks for you to study and practice on. You can also tailor these sample exercises to your particular needs.

You should be somewhat familiar with the basics of operating a Macintosh before you start this manual. That is, you should know about clicking on icons, using the mouse, opening folders, and other Macintosh features. If you're not comfortable with these terms, spend some time playing with your Mac and using your Macintosh's user's guide. You may also want to skim through the glossary at the end of this manual to get some understanding of the concepts we've used.

The Manual

This manual is divided into three main sections: the *User's Guide* (Part I), the *Reference Section* (Part II), and the *Appendices*. A glossary and index round out the manual.

The *User's Guide* introduces you to Turbo Pascal, shows you how to use it, and includes chapters that focus on such specific features as units, desk accessories, and debugging. Here's a breakdown of the chapters:

Chapter 1: Setting Up shows you how to set up your Mac for Turbo Pascal, describes the files on your distribution disk, and explains how to make backup disks.

Chapter 2: Getting Started with Turbo Pascal leads you directly from loading Turbo Pascal into writing simple programs. It then suggests how you should go about reading the rest of the manual, depending on your familiarity with the Mac and with Pascal.

Chapter 3: Using the Editor explains Turbo Pascal's menus (except for the Compile menu, covered in the next chapter) and shows you how to use the editor to open, edit, change, and save files.

Chapter 4: Using the Compiler describes how to implement the programs you learned to create in Chapter 3, using the compiler. It also shows you common programming errors and how to avoid them.

Chapter 5: Writing Textbook Pascal Programs shows you how to take standard Pascal programs and use them with Turbo Pascal without having to know anything about the Macintosh Toolbox and operating system.

Chapter 6: Harnessing the Full Power of Your Mac is a quick guide to the Macintosh and the tools that exist to help you write more complex programs.

Chapter 7: Units and Other Mysteries tells you what a unit is, how it's used, and what predefined units (libraries) Turbo Pascal provides.

Chapter 8: Writing Your Own Units goes into the general structure of a unit and its interface and implementation portions. It shows you how to initialize and compile a unit.

Chapter 9: Writing Your Own Macintosh Applications shows you how to put together your own Mac-style programs, complete with menus, windows, and cursors.

Chapter 10: Graduation: Writing a Desk Accessory tells you all you need to know to design and write desk accessory programs.

Chapter 11: UNITMOVER, Chapter 12: RMAKER, and **Chapter 13: FONT/DA MOVER** give detailed instructions on these utilities, which come on your Turbo Pascal disks.

Chapter 14: Debugging Your Turbo Pascal Program explains how to use MACSBUG to check for errors in your program.

Chapter 15: The Turbo Pascal Menu Reference is a complete guide to the menu commands in Turbo Pascal.

Now we move on to the *Reference Section* of the manual. The first 11 chapters offer technical information on the following features:

Chapter 16: Tokens and Constants
Chapter 17: Blocks, Locality, and Scope
Chapter 18: Types
Chapter 19: Variables
Chapter 20: Expressions
Chapter 21: Statements
Chapter 22: Procedures and Functions
Chapter 23: Programs and Units
Chapter 24: Input and Output
Chapter 25: Standard Procedures and Functions
Chapter 26: The Standard Apple Numeric Environment (SANE) Library

Chapter 27: Inside Turbo Pascal offers additional technical information for advanced Pascal programmers, including internal data formats, assembly-language interfaces, and user-defined device drivers.

Finally, there are six appendices in the manual:

Appendix A: Comparing Turbo Pascal with Other Pascals
Appendix B: Error Messages and Codes
Appendix C: Compiler Directives
Appendix D: Macintosh Interface Units
Appendix E: Macintosh Character Set
Appendix F: TURTLEGRAPHICS: Mac Graphics Made Easier

Typography

The use of italic and boldface type in this manual follows certain conventions. Reserved words are set in lowercase, boldface type. Constant identifiers, field identifiers, and formal parameter identifiers are italicized when referred to within text. Other identifiers—unit and program names, labels, types, variables, procedures, and functions—begin with an uppercase letter; they also are italicized when referred to within text.

The command key (the cloverleaf key on the Mac keyboard) is represented by the keycap ⌘.

Pascal syntax is illustrated by diagrams. To follow a syntax diagram, start at the top left and follow the arrows through the diagram. Alternative paths are often possible; paths that begin at the left and end with an arrowhead on the right are valid paths. A path traverses boxes that hold the names of elements that are used to construct that portion of the syntax.

The names in rectangular boxes stand for actual constructions. Those in circular boxes—reserved words, operators, and punctuation—are the actual terms that should be used in the program.

Acknowledgments

Apple® is registered to Apple Computer, Inc.
Macintosh™ is a trademark licensed to Apple Computer, Inc.
Inside Macintosh© is a copyright of Apple Computer, Inc.
Lisa® is a registered trademark of Apple Computer, Inc.
Lisa®Pascal™ is a trademark of Aioi Seiki Kabushiki.

How to Contact Borland

If, after reading this manual and using Turbo Pascal, you would like to contact Borland with comments or suggestions, we suggest the following procedures.

The best way is to log on to Borland's forum on CompuServe: Type GO BOR from the main CompuServe menu and follow the menus to section 4. Leave your questions or comments here for the support staff to process.

If you prefer, write a letter describing your comments in detail and send it to the Technical Support Department, Borland International, 4585 Scotts Valley Drive, Scotts Valley, CA 95066, USA.

As a last resort, you can telephone our Technical Support department. If you're calling with a problem, please have the following information handy before you call:

- Product name and version number
- Computer make and model number
- Operating system and version number

PART I

User's Guide

C H A P T E R 1

Setting Up

Before you actually begin using Turbo Pascal, you should make a backup copy of your disks so that you can put your master disks in a safe place. This chapter tells you how to do that. It also describes the files on your Turbo Pascal disks so that you can see what files are provided and which you'll need.

Before you go on, you should have some familiarity with the Mac. You should know how to turn your Mac on and off, how to move the mouse around, how to select commands from a menu, how to manipulate (move, resize, and close) windows with the mouse, and how to select and launch applications. If you have questions about using the Mac, please refer to your Macintosh owner's manual.

Making Working Copies

Borland's philosophy—selling software without copy protection—is based on trust. As it says in the license statement at the beginning of this manual, you are authorized to make working copies of the distribution disks. You can then put the originals in a safe place.

Here's how to copy your Turbo Pascal and Utilities & Sample Programs disks.

First, with the Macintosh off, put your Turbo Pascal disk in the internal disk drive and turn the Mac on. The Mac boots up and displays a window with the contents of your Turbo Pascal disk.

Second, insert a blank disk, or a disk that doesn't contain anything you want to save, in your external disk drive. (If you don't have one, we'll tell you what to do in a few paragraphs.) If you've inserted a new disk or one that's been used with some other computer system, the Mac asks if you want to initialize it. If it is in a double-sided disk drive, you have the option of formatting it as single- or double-sided; it's best to choose whichever corresponds to your internal disk drive. Remember, initializing erases all existing files on the disk. When initialization is done, you'll be asked to name the disk; give it something like "TP Mac."

Third, point to the Turbo Pascal disk icon, press the mouse button, and hold it down. Now drag that icon to the icon of your work disk. That disk's icon should now turn dark. Release the mouse button. You'll now get a dialog box that asks if you really want to replace the contents of the disk in your external drive with the contents of the disk in your internal drive. Point to OK and press the mouse button. All the files on your Turbo Pascal master disk will be copied over to your work disk.

Fourth, eject both disks: Click on the disk icon, then select Eject from the File menu, or press ⌘E. Label your new working copy of Turbo Pascal, and store your Turbo Pascal master disk somewhere safe. Turn your Mac off, place your working disk into the internal disk drive, and turn the Mac back on. You now have a working copy of Turbo Pascal.

Repeat with the Utilities & Sample Programs disk.

What If I Only Have One Disk Drive?

Prepare yourself for some disk swapping. Boot up with your Turbo Pascal master disk, as described above, then eject it.

Insert your blank work disk. If necessary, initialize it as described above. You should now have two disk icons on your desktop (Macintosh screen).

Drag the icon for the Turbo Pascal master disk onto the icon of your work disk. You're asked to reinsert the Turbo Pascal master disk; do so. At the "Replace all this?" prompt, click the OK button.

The actual copying now takes place. You'll be asked to swap disks from time to time, so that your Mac can read from the master disk and write to the destination disk. The actual number of swaps depends upon the size of your disk drive and the amount of memory in your Mac.

Repeat this procedure with the Utilities & Sample Programs disk.

What If I'm Using a Hard Disk?

Copy all the files and folders (except for the SYSTEM FOLDER) from your Turbo Pascal master disks to any volume or subdirectory on your hard disk. Store the masters.

Bypassing the Desktop

If your working copy of Turbo Pascal is a bootable disk (that is, if it is the disk you boot from when you turn your Macintosh on), you can make Turbo Pascal your startup application. This means that when you boot from your Turbo Pascal work disk, instead of having to wait for the desktop to come up and then double-clicking on the Turbo icon, you will automatically go into Turbo Pascal.

To do this, boot up using your Turbo Pascal work disk. Click *once* on the Turbo icon, so that it becomes dark but doesn't start executing. Go to the Special menu and select Set Startup. A dialog box comes up, asking you to verify that you want Turbo to be the startup application. Select the OK button.

From now on, when you boot up using that disk, you'll bypass the desktop and go immediately into Turbo Pascal.

The Files on the Disks

Your Turbo Pascal master disks have quite a few files and folders (which contain more files). Unlike the other Pascal programs, however, Turbo Pascal can run on only the TURBO file. This simplicity makes Turbo Pascal easy to use and conservative of your memory space. However, it doesn't skimp on options.

Here's a quick rundown of the files, with descriptions showing what each file provides. The TURBO and SYSTEM FOLDER files are on the Turbo Pascal disk; all other files are on the Utilities & Sample Programs disk.

Table 1-1 Files on Your Distribution Disk

TURBO	The Turbo Pascal compiler/editor. This file also contains the Pascal run-time and Mac interface units. You definitely need it.
SYSTEM FOLDER	A folder containing the system files:
SYSTEM	The Macintosh operating system. This file also holds system resources, such as fonts and desk accessories. Your disk has only a limited number of these to conserve space. Essential if you are going to boot up using your Turbo Pascal work disk.
FINDER	The Macintosh user interface program. This is what brings up the desktop, allows you to select and run a program, and so on. Also essential if you plan to boot from your work disk.
IMAGEWRITER	The printer driver for the Imagewriter printer. You need this if you're going to print anything, either from within Turbo Pascal or within your own program.
UTILITIES	This folder contains UNITMOVER, RMAKER, FONT/DA MOVER, MACSBUG, MACINTALK, and APPLETALK:
UNITMOVER	Utility for moving units (libraries) in and out of Turbo Pascal. You don't need it unless you write your own units and store them in Turbo Pascal.
RMAKER	The Resource Maker. This converts resource source files (.R) into resource data files (.RSRC), which can then be used by your programs. If you're writing Mac-style programs, you'll need this file.
FONT/DA MOVER	Utility for moving fonts and desk accessories in and out of your SYSTEM file. You need it if you plan to write desk accessories.
MACSBUG	A debugger, that is, a program that helps you to track down and correct errors in your program. This is for more sophisticated users; after reading about it in Chapter 14, you can decide whether to include it.
MACINTALK	A resource file for speech synthesis. You'll need it to use the MacinTalk unit in any of your programs.
ATALK/ABPACKAGE	A resource file for using the APPLETALK network. You'll need it to use the APPLETALK unit in any of your programs.
SAMPLE PROGRAMS	This folder contains sample programs, including:
MYDEMO.PAS	A sample program that shows how to write Mac-style programs. Brings up a window and its own menu bar; allows you to run several different benchmarks (graphics, I/O, etc.); supports desk accessories.
MYDEMO.R	A resource file for MYDEMO.PAS. You must run RMAKER on it (producing MYDEMO.RSRC) before you can compile and run MYDEMO.PAS.

Table 1-1 Files on Your Distribution Disks, continued

MYDA.PAS	A sample desk accessory whose code illustrates all the different events that you might need to handle in a desk accessory.
MYDA.INC	An include file for MYDA.PAS; it contains most of the event-handling routines.
MYDA.R	A resource file for MYDA.PAS. You must run RMAKER on it (to produce MYDA.RSRC) before you can compile MYDA.PAS.

What If I Don't Want to Use Turbo Pascal's System Files?

Format a blank disk and copy onto it the system files you want to use. Boot up using it. Put the Turbo Pascal master disk in your external drive. Copy to your system disk all its files and folders, except for the one labeled SYSTEM FOLDER.

Eject the Turbo Pascal master disk and put it somewhere safe.

What If I Don't Want All the Turbo Pascal Files?

Make copies of the Turbo Pascal master disks, using one of the methods described previously. Throw away (that is, move into the Trash icon) all the files you want to get rid of.

Customizing Turbo Pascal

There are two sets of options that you can change to customize Turbo Pascal. The first set can be examined and changed using the Option command in the Turbo Pascal Edit menu. With it, you can set the tab width, toggle the auto-indent mode, and tell Turbo Pascal whether or not you want it to bring up a new ("Untitled") window each time you go into Turbo Pascal. Chapter 3 has more details on these options.

The second set of options is under the Options command in the Turbo Pascal Compile menu. These options include toggling the auto-save mode, setting the size of the symbol table, and specifying default directories (path names) for units and include, resource, .REL, and output files. See Chapter 4 for more details.

Where to Go from Here

By now, you should be set up to start using Turbo Pascal. Boot up your system (if it isn't on), double-click on the Turbo icon, and go on to Chapter 2. It explains the different menu commands in Turbo Pascal.

You may want to quickly jump to the glossary and scan through the Turbo Pascal icons shown there. That way, you'll be more familiar with the different icons and the types of files they represent before you begin programming.

CHAPTER 2

Getting Started with Turbo Pascal

Now that you're all set up, let's plunge right into writing your first Turbo Pascal program. By the end of this chapter, you'll have written three small programs, saved them, and learned a few basic programming skills. The last section offers a game plan for proceeding through the rest of the manual, depending on your programming experience.

Loading Turbo Pascal

If you're using a floppy-disk drive, first turn off your Macintosh. Put your Turbo Pascal disk into the internal disk drive. Turn the Mac on. The Mac boots up and displays a window with the contents of your Turbo Pascal disk. Near the top of the window, you'll see an icon—labeled "Turbo"—of a hand waving a checkered flag. This is the Turbo Pascal compiler/editor.

To launch it, just point at it with the mouse and click twice, rapidly. The desktop clears. A few moments later, a new menu bar appears, along with an empty window labeled "Untitled" (see Figure 2-1). You're now set up to write your Turbo Pascal program.

A blinking vertical bar is in the left-hand corner of the "Untitled" window. When you enter a program, the text you type appears here. Now type in the following program, pressing the ↵ key at the end of each line:

```
program MyFirst;
begin
  WriteLn('Hello, universe!');
  ReadLn;
end.
```

Note the semicolons (;) at the end of the first, third, and fourth lines, as well as the period (.) after the last line. If you make a mistake while typing, press the ← key to erase what you have typed. (If you're familiar with Mac-style editors, you can use the mouse to select and change text.)

Now go to the Compile menu and select the Run command (or press ⌘R). Turbo Pascal compiles and runs your program. If there is a syntax error (that is, a Pascal language error), Turbo Pascal stops at that place in your program and tells you what the error is. Acknowledge the error by clicking the mouse button or pressing the ↵ key. Correct the error and then select the Run command again.

After all errors are fixed, Turbo Pascal completely compiles your program and executes it. The menu bar and window disappear, a window labeled "MyFirst" opens, and the message `Hello, universe!` appears in the upper left-hand corner of the window.

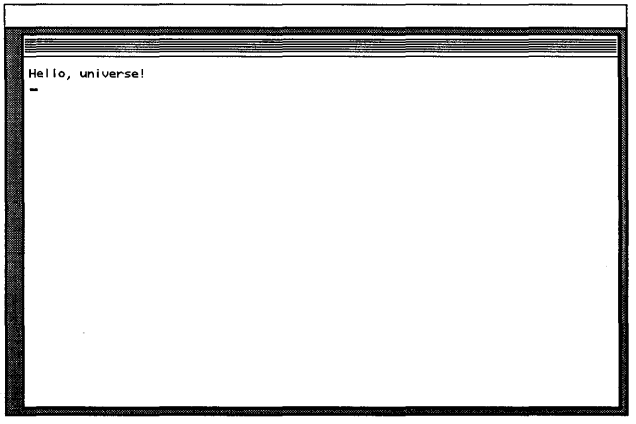


***Figure 2-1** The Compiled MYFIRST Program*

The program then waits for you to press ↵ (this is what the *ReadLn* statement does). When ↵ is pressed, the window disappears, and you're back in Turbo Pascal.

Having written and compiled this masterpiece, you need to save it to disk, so that you can modify it later. Go to the File menu and select the Save option (or press ⌘S). A file-save dialog box comes up. Type in a name for your program, say, "MYFIRST.PAS." Turbo Pascal isn't case-sensitive, so you can use uppercase or lowercase when entering information. Press ↵ and your program is saved to disk.

If you exit Turbo Pascal (select Quit from the File menu), you'll see your program file saved as a document icon with a checkered flag on it. If you want to edit it some more, point the cursor to it and double-click.

If you want to run your program outside of Turbo Pascal, go to the Compile menu and select the To Disk command (or press ⌘K). You don't need to save the file; just exit Turbo Pascal. You'll see your executable program, named MYFIRST, saved under a standard Mac application icon (a hand writing on a piece of paper). If you double-click on this icon, it will execute your `Hello, universe!` program again, then return you to the Mac desktop when you press ↵.

Stepping Up: Your Second Program

Now let's look at a second program that does a bit more. It prompts you for a location and a radius, then draws a black circle of that radius at the specified location.

```
program  MySecond;

uses MemTypes,QuickDraw;

var
  X,Y,Radius    : Integer;
  TRect         : Rect;

begin
  Write('Enter X: ');
  ReadLn(X);
  Write('Enter Y: ');
  ReadLn(Y);
  Write('Enter radius: ');
  ReadLn(Radius);
  SetRect(TRect,X-Radius,Y-Radius,X+Radius,Y+Radius);
  PaintOval(TRect);
  ReadLn;
end.{ of program MySecond }
```

The **uses** statement asks Turbo Pascal to let you use two units (program libraries), *MemTypes* and *QuickDraw*. This gives you access to the graphics data types and routines (*Rect*, *SetRect*, and so on).

You've declared four variables in this program: *X*, *Y*, *Radius*, and *TRect*. *X* and *Y* are integers (numbers); they store the values you type in for the location of the center of the circle. Likewise, *Radius* is an integer that holds the radius (distance from the center to the edge) of the circle. *TRect* is a variable of type *Rect*, a special Macintosh data type that holds a description of a rectangle (top, left, bottom, and right values).

The first six statements of the program consist of three *Write* statements and three *ReadLn* statements. Each *Write* statement writes the string inside of it out to the screen; each *ReadLn* statement waits for you to type a value and press ↵, after which it stores the value in the enclosed variable.

The next two statements call *QuickDraw* routines. *SetRect* gives the variable *TRect* the boundaries indicated (*X-Radius* and so on). The resulting rectangle determines the size of the circle you want to draw. *PaintOval* takes the information in *TRect* and uses it to paint a black circle just within the rectangle's boundaries. The last statement, *ReadLn*, causes the program to wait for you to press ↵ before it exits the program and returns to Turbo Pascal (or, if you've compiled to disk, to the Mac desktop).

Programming Pizazz: Your Third Program

You've now dabbled in graphics, so let's explore a more complex program. It offers more variety and interesting graphics.

```
program MyThird;

uses MemTypes,QuickDraw;

const
  Start   =  50;
  Finish  = 250;
  Step    =   2;

var
  X1,X2,Y1,Y2  : Integer;

begin
  Y1 := Start;
  Y2 := Finish;
  X1 := Start;
    while X1 <= Finish do
```

```
  begin
    X2 := (Start+Finish) - X1;
    MoveTo(X1,Y1);
    LineTo(X2,Y2);
       X1 := X1 + Step
end;
   X1 := Start;
   X2 := Finish;
   Y1 := Start;
while Y1 <= Finish do
  begin
    Y2 := (Start+Finish) - Y1;
    MoveTo(X1,Y1);
    LineTo(X2,Y2);
    Y1 := Y1 + Step
  end;
  ReadLn;
end.{ of program MyThird }
```

This program produces a square with a black center and some interesting patterns (known as Moire patterns) along the edges. The section labeled **const** defines three numeric constants (*Start*, *Finish*, and *Step*) that affect the size, location, and appearance of the square. By changing their values, you can change how the square looks.

WARNING: Don't set *Step* to anything less than 1; if you do, the program will get stuck in what is known as an *infinite loop*. You won't be able to exit except by pressing the interrupt switch or by turning your Mac off.

The variables *X1*, *Y1*, *X2*, and *Y2* hold the values of locations along opposite sides of the square. The square itself is drawn by drawing a straight line from (*X1*,*Y1*) to (*X2*,*Y2*). The coordinates are then changed, and the next line drawn. The coordinates always start out in opposite corners: The very first line drawn goes from (50,50) to (250,250).

The program itself consists primarily of two loops. The first loop, as we mentioned, starts by drawing a line from (50,50) to (250,250). It then moves the *X* (horizontal) coordinates by two, so that the next line goes from (52,50) to (248,250). This continues until it finally draws a line from (250,50) to (50,250).

The program then goes into its second loop, which pursues a similar course, changing the *Y* (vertical) coordinates by two each time. The routines *MoveTo* and *LineTo* are from the *QuickDraw* unit. *MoveTo* moves to the indicated spot on the screen without drawing anything, while *LineTo* draws a line from the current location to the one given.

The final *ReadLn* statement causes the program to wait for you to press ↵ before exiting.

Where to Go from Here

You've now gotten your feet wet and have written three quick programs using Turbo Pascal for the Macintosh. How do you proceed from here?

If you're a complete novice without any Mac or programming experience, read the rest of Part I very carefully, following all the examples shown. Make sure you understand each chapter before moving on to the next. If you're an experienced Mac user but you haven't done any programming, a quick once-through is all you need on Chapter 3. The rest of the chapters will require careful attention, though.

If you're proficient on the Mac and have done a fair amount of programming, but not on the Mac, read Chapters 4 and 5 to familiarize yourself with Turbo Pascal. Then pay special attention to Chapters 6, 9, and 10.

If you've done a lot of Mac programming but not in Pascal, then concentrate on Chapters 4, 5, 7, and 8. Chapters 6, 9, and 10 should then help you to see the differences in programming with Turbo Pascal and whatever language you were using.

If you've already used Pascal on the Mac, Chapters 4 and 7 will require special attention, while you can probably skim the rest.

Finally, there are a few other books you might consider reading after you've finished this manual. If you are planning to do much Mac-style programming with graphics, windows, menus, and so on, we recommend *Inside Macintosh* (Addison-Wesley, 1986). This consists of four softbound volumes (or hardbound and softbound set). It is *the* reference work for information on how to use the Mac Toolbox and operating system routines. If you're not familiar with Pascal, you'll probably want to pick up a good tutorial on the language. Many such books are available, including several that are specific to the Macintosh.

We've tried to make this the best user's guide and reference manual possible. After working through it, you should feel at home with Turbo Pascal. Good luck, and happy programming!

C H A P T E R 3

Using the Editor

In this chapter, you'll learn the basic editing features of Turbo Pascal—how to enter a program, move and format text, undo commands, and save files.

A Quick Review of Clicking

Remember, you should be familiar with the Macintosh—be able to click on icons, open and close folders and disks, and select commands from menus—*before* you go on. If you aren't, read the user's guide that came with your computer and familiarize yourself with those operations first. Let's quickly review the technique of *clicking*, however.

Any movement of the mouse is echoed by the arrow-shaped pointer on the screen. When you place the pointer on, say, an icon and quickly press-and-release the mouse button, that's called clicking. It selects and highlights whatever you just clicked on. You can then go to the menu and choose the command you want performed on the highlighted item.

As a shortcut, you can *double-click* on the item to select and open it—and skip the menu-selection steps.

Shift-clicking is another option. If you hold ⇧ down and move the mouse to a second location, everything between the original mouse location and its current location is selected and highlighted.

Opening Turbo Pascal

Getting into Turbo Pascal is easy. Look for the Turbo Pascal icon, a hand waving a checkered flag, on your disk. Move the cursor to it and click on it twice, rapidly. After a few seconds, the Macintosh desktop is replaced by the Turbo Pascal menu bar, and a window (labeled "Untitled") appears.

You can also get into Turbo Pascal by clicking on its icon once, going to the File menu, and selecting the Open command (or press ⌘O).

Close the "Untitled" file by clicking on the Close box, then select Quit from the File menu. You're back in the Mac desktop. Double-click on the MYFIRST.PAS file you created in the last chapter. A new window appears, called "MYFIRST.PAS," with the program you entered previously.

You can identify programs written with the Turbo Pascal editor; they look like a sheet of paper with the top right corner bent down and a checkered rectangle centered on the sheet. When you open one of these files, you start up Turbo Pascal, which opens a window with that program in it.

Return to the Mac desktop. This time, double-click on the MYFIRST icon (a hand writing on a sheet). Your compiled program appears. Exit it by pressing ↵. The Mac desktop reappears.

Editing a File

An *editor* is a program that allows you to edit text, that is, to enter, delete, or change what you've typed in. Turbo Pascal has a built-in editor that is available at all times. With it, you can write new programs and modify existing ones. You can add, delete, and change code. The Edit menu shows some of these features.

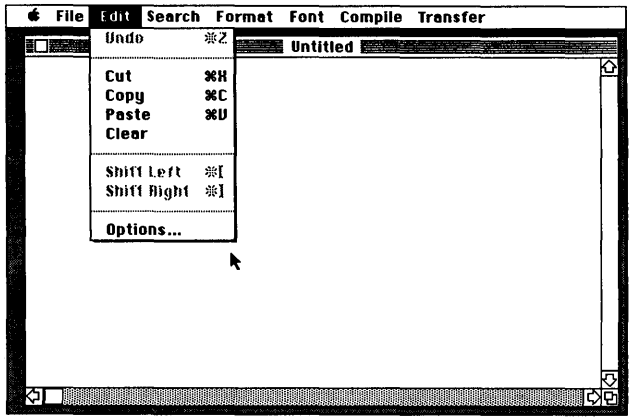


Figure 3-1 The Edit Menu

Let's start by typing in a new program.

Entering a New Program

Double-click the Turbo Pascal icon. An empty window (called "Untitled") covers most the screen. If someone has used this program previously and the window has text in it, close the window by clicking on the Close box in the upper left corner of the window. A blank screen with the desktop and Turbo Pascal menu bar remains; go to the File menu and select the New command (or press ⌘ N). Now you should have an empty window named "Untitled."

There are two different cursors on the screen. One is a vertical blinking bar in the upper left corner of the window. If you type the following line

```
program Quickie;
```

this cursor moves to the right as you type. It indicates where the next letter you type will appear. Press ↵, and the cursor moves to the start of the next line. Now type the following two lines, pressing ↵ after each one:

```
var
begin
```

The bar cursor should now be at the start of the line underneath the word **begin**.

The second cursor on the screen is larger and is shaped like an I-beam. It is "connected" to the mouse; that is, it moves on screen as you move the mouse on your desk top. When you move this cursor outside the window, it turns into an arrow; move it back into the window, and it becomes an I-beam again. Now, move it right after the word **var**, then click once. The bar cursor jumps from the

beginning to the fourth line to where the I-beam cursor was when you clicked the mouse button. Press ↵, indent two spaces, and type

```
A,B,C : Integer;
```

Your window should now have the following text:

```
program Quickie;
var
  A,B,C : Integer;
begin
```

The blinking bar cursor should be just after the semicolon (;) following the word *Integer*. Now move the I-beam cursor to the line below the word **begin** and click once. The bar cursor should jump down there. Add two space indents, type the following line, then press ↵:

```
WriteLn('Hello, world');
```

If you correctly typed two spaces before starting *WriteLn*, the bar cursor should be indented two spaces in: It lines up with the word *WriteLn*. This is known as auto-indenting, and it helps you follow your programming conventions. (You can turn it off, if you like.)

Now, press ← twice (it's located above ↵). The bar cursor should be at the left margin again. Type `end.` (with a period) and press ↵. Your entire program should now look like this:

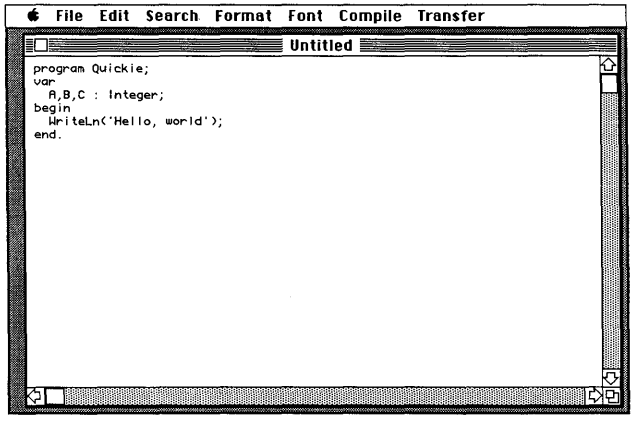


Figure 3-2 *The Quickie Procedure Window*

Changing a Program

There are several ways to change or modify a program. The simplest way is to add new text. Move the I-beam cursor to just after the semicolon following (`Hello, world`) and click once. The bar cursor moves there. Press ⏎ and type:

```
WriteLn('What's your sign?');
```

You've added a new line to your program. Because of the auto-indenting, this statement lines up with the one above it.

The next simplest change is to delete text. Press ⌫ several times. You'll see that you're erasing what you've just typed. If you keep pressing it, or if you hold it down, it continues to erase the characters to its left. When you get to the start of the line, it jumps back up to the end of the previous line, and all the text below (which right now is just the line `end.`) moves up. If you're still holding ⌫ down, you'll find that your entire program will soon be erased, character by character. Stop, and retype all you've erased.

Suppose you wanted to change the string `Hello, world` to `Hi, world`. Use the following steps:

1. Move the I-beam cursor until it's between the O and the comma in `Hello, world`. Click the mouse once to move the bar cursor there.
2. Press ⌫ four times to delete `ELLO`.
3. Type I (comma).

Now, following the steps above, change the word `WORLD` to your own first name.

The Clear key on the numeric keypad (or ⌘⌫ if you don't have a numeric keypad) is also used to delete text. When the cursor is in the middle of a line, pressing Clear deletes all characters to the left of the cursor until the beginning of the line. This is handy in connection with the auto-indent feature when you want to *un-indent* a line, that is, remove blanks that were automatically inserted by pressing ⏎. If you press Clear when the cursor is at the beginning of a line, the line above is deleted.

Selecting, Cutting, and Pasting Text

A powerful feature of the Turbo Pascal editor is that it lets you *cut* portions of text and *paste* them elsewhere. You can use the I-beam cursor and the mouse to *select* portions of text—like setting aside selected pages of a document—while you decide what to do with them.

Let's say that you want to delete the variable declarations in your program. Move the I-beam cursor in front of the word **var**. Now, press the mouse button and *hold it down*. While holding the mouse button down, slowly move the I-beam cursor down the screen. Each line that it passes turns black with the text reversing (called inverse or reverse video). The text is what you are selecting. Now, with the mouse button still pressed, move the I-beam cursor until it's right in front of the word **begin**. The two lines above it,

```
var
  A,B,C : Integer;
```

should be in reverse video. Release the mouse button. The lines remain black because they are selected.

You now have several options. To do nothing, move the I-beam cursor anywhere and click once. The text will be de-selected; that is, it will return to normal. You can do this anytime you accidentally select text that you don't want selected. Try this out, then go back and re-select those two lines.

The next option is to delete the selected text. You can press ⌫, and the text will vanish. You can restore it by selecting Undo from the Edit menu, which is explained in a later section. The same thing happens if you go to the Edit menu and select the Clear command. You can also select the Cut command from the Edit menu (or press ⌘X). That deletes the text but saves it in the *Clipboard*, a holding area for text that's been cut (or will be copied). Practice using these deletion options, then reenter the two lines. Select them again.

The third option is to copy the selected text. Go to the Edit menu and select the Copy command (or press ⌘C). The selected text looks the same on the screen, but a copy of it has just been placed in the Clipboard.

Fourth, you can replace the selected text. Whatever you start typing replaces the selected text. As soon as you press the first key, the entire selected text disappears and your new text replaces it as you type.

If you have cut or copied text, so that you have text in the Clipboard, you can select the Paste command from the Edit menu (or press ⌘V). The text in the Clipboard automatically replaces the selected text.

If you have cut or copied text into the Clipboard, you can insert or *paste* it anywhere in your program. Select a line of text, then cut or copy it using the Edit menu. Now move the I-beam cursor to where you want to insert the text, and click the button once. The blinking bar cursor appears there. Go to the Edit menu and select the Paste command (or press ⌘V). The selected text is now pasted where the cursor is. A copy of that text is still in the Clipboard; you can move somewhere else and paste it in again.

Try out these commands, until you're comfortable with them. Then you can do the following exercises:

1. Change the name of the program to `Mortimer` by selecting the word `Quickie` and then typing `Mortimer`. Practice selecting individual words and letters on a given line.
2. Delete the **var** and `A,B,C : Integer;` lines using three different means. Retype or paste them back in each time.
3. Insert the statement `WriteLn('A is A.');` between the first and second *WriteLn* statements. (Don't forget the semicolon at the end.)

The Undo Command

During the exercise above, you may have accidentally deleted or changed something. You probably went in and retyped the altered text. The Turbo Pascal editor helps protect you from your own mistakes with the Undo command in the Edit menu.

Try the following exercise. Move the I-beam cursor to the start of the program, hold the mouse button down, and move the I-beam cursor to the end of the program. The entire program should now be selected. Now press ⌫. Presto! Your entire program has just disappeared! Don't panic. Instead, select the Undo command in the Edit menu (or press ⌘Z). Your entire program resurfaces.

You can only Undo the last action you did. Select **var**, for example, and press ⌫; **var** disappears. Now move the cursor to the end of the program. If you click on the Edit menu, Undo appears blurred; that is, it cannot be selected. Even if you move the cursor back to the empty line and select Edit, Undo will still be blurred. You have to retype **var**; the selection can't be undone.

Spend some time experimenting with the Undo command, seeing what you can (and can't) undo. This is a really valuable command, so take the time to learn it well.

Formatting Text

Many Pascal programmers format their programs with indentation, aligning **begin** and **end** keywords, nested statements, and so on. Often a level of nesting will change: A set of statements will be moved out of an **if..then** statement, or into a **for** loop. To maintain the correct nesting format, the programmer then has to shift all the code—line by line—left or right, according to the change.

With the Turbo Pascal editor, such formatting changes are easy. Just select the text to be shifted, using the click-and-drag technique: Put the pointer at the beginning of the selected text, hold the button down, and move to the end of the text. Release the button. Then press ⌘[to shift left, or ⌘] to shift right. Each press of the command sequence shifts the entire selected block of text one character left or right. You can also do this by selecting the Shift Left or Shift Right commands in the Edit menu.

Finding a Lost Bar Cursor

The location of the bar cursor—the short, blinking one that indicates where the next character you type will appear—is quite different from the location of the I-beam cursor, the one reflecting mouse movements. In a large file, it is possible to lose the bar cursor, because of scrolling to (that is, viewing) a different part of the program from where the bar cursor is. Two commands in the Turbo Pascal editor help you to deal with that.

First, if you press ↵ (or ⌘↵), the text display will be scrolled upwards or downwards until the first or last line in the window is the line with the bar cursor. No text will be changed. Second, you can use the Home Cursor command in the Search menu (or press ⌘H). This moves the bar cursor to the very top of the file and adjusts the display to show it.

Search and Replace

The Turbo Pascal editor lets you search for a particular string (that is, a delimited group of characters). It also lets you change one string for another.

To find a given string, such as a variable or procedure name, select the Find command in the Search menu (or press ⌘F). A dialog box comes up that asks you to specify what you want found. There are two checkboxes, Words Only and Case Sensitive. The first means that it won't recognize the string if it's embedded in a larger string. For example, if you are looking for *myGlobals* and selected this option, then it wouldn't pick out the string in *myGlobalsH*. Second, specifying Case Sensitive means that uppercase and lowercase letters are not considered to be equivalent. If you are looking for *myGlobals*, then *MyGlobals* doesn't match.

Having typed in your string, start the search by pressing ↵ or by selecting the OK button in the dialog box. The editor starts searching from the current position of the bar cursor until it either finds the string requested or hits the end

of the file. If it finds the string, that section of your program appears on the screen, with the specified string highlighted. If it doesn't find the string, it beeps at you, and the screen stays the same.

Having found the first instance, you can find the next appearance of the string by selecting the Find Next command from the Search menu, or by pressing ⌘D. You can also select Change to replace one string with another.

You can use key equivalents in the Search and Replace dialog box; that is, you can type Y for Yes, N for No, A for All, and C for Cancel.

Saving Your Text

There are several ways of saving the text you have entered in a window:

- Select the Save command in the File menu (or press ⌘S). If your window already has a title, the text is saved on the disk, overwriting the old version of the file. If your window is untitled, the Save-file dialog box comes up. Select the proper drive and directory, type in a name for your text, and click the Save button (or press ↵). After saving the text, your window's title is changed to the file name you just entered. When naming your files, it's advisable to use extensions, for instance .PAS for Pascal programs and .R for RMAKER source files. This enables you to use the same name for different files relating to the same application, such as MYPROG.PAS and MYPROG.R. Furthermore, it makes it easy to determine the type of a textfile without having to actually read it.
- Select the Save As command in the File menu. This corresponds to the Save command, except that it always brings up the Save-file dialog box, thus allowing you to save the text under a new name.
- Select the Close command in the File menu (or press ⌘). This saves the text (corresponding to the Save command) and removes its window from the desktop. The Close command only saves the text if it has been modified since it was last saved, or since the window was opened.
- Click on the window's Close box. This corresponds to selecting the Close command in the File menu.
- Select the Quit command in the File menu. This closes all windows and returns to the Macintosh desktop (the FINDER).

Now that you know how to edit your program, let's move on to Chapter 4. You'll learn how to tell the computer to carry out your program.

C H A P T E R 4

Using the Compiler

You now know how to create a program and save it to disk. Now, let's look at how to tell the computer to carry out the instructions you've typed in. This is done with the commands in the Compile menu.

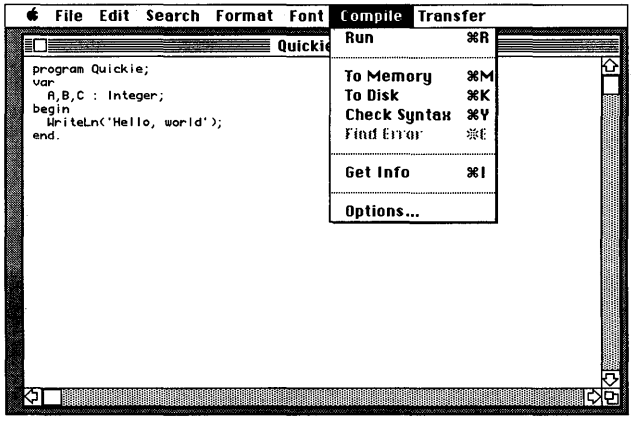


Figure 4-1 *The Compile Menu*

We'll briefly describe the Run command, then examine all the other commands in the Compile menu: Run, To Memory, To Disk, Check Syntax, Find Error, Get Info, and Options. We'll also explain how Turbo Pascal handles syntax and run-time (system) errors.

An Overview of the Run Command

Say you've typed in a program using the Turbo Pascal editor. To make it run, pull down the Compile menu (click on Compile in the menu bar) and select the first option, Run. You can avoid using the mouse by pressing ⌘R. Turbo Pascal then compiles your program, that is, changes it from Pascal (which you can read) to 68000 machine code for the microprocessor (which the Macintosh can execute). You don't see the 68000 machine code; it's stored off in memory somewhere.

While the program is compiling, the cursor is changed to a racing flag and a small box (or window) appears at the top of the screen, saying `Compiling: <file name>`. The box includes a button labeled Cancel. Use it to stop the compiler for any reason—for example, if you've suddenly remembered a change you forgot to make to your program. Just move the cursor over the word Cancel and click the mouse button. Turbo Pascal then returns you to the editor.

Should an error occur during compilation, Turbo Pascal stops compiling and returns to the editor, with the cursor at the error location. A dialog box tells you what the error was. Click on the error box, correct the problem, and select Run again.

Once the translation from Pascal to machine code is complete, Turbo Pascal tells the Mac to execute the code it has generated and your program runs. Your program takes control of the Mac and completely replaces the Turbo Pascal screen and menu bar.

If a run-time error crops up—that is, an error occurs while your program is executing—you'll get the standard Mac system error box. This is a box with a bomb icon in it and two buttons: Restart and Resume. If you select the Resume button, you return to the Turbo Pascal editor. When possible, the cursor is at the section of code where the error took place; for some errors (such as pressing the Interrupt switch on the side of the Mac), there is no way of determining what part of the program was executing when the error occurred, so the cursor is placed at the beginning of the text. Restart reboots your computer.

NOTE: You should not use the Interrupt switch on the Mac Plus unless the debugging program MACSBUG is loaded. Without MACSBUG, the program merely goes into a simple debugger built into the Mac; a bomb box showing you the error *doesn't* appear.

When you press Resume, a box pops up, telling you what the error was (input/output, division by zero, and so on). After you figure out how to fix the program bug, you can recompile and run the program again.

When your program has finished executing, the Macintosh returns control to Turbo Pascal, and you're back where you started. You can now make changes to

improve or cause your program to do something different. If you select the Run command at this point without changing your program, Turbo Pascal immediately executes it, without recompiling.

The Turbo Pascal Compiler

You can now run your programs. As you have seen, Turbo Pascal is very forgiving of errors and does its best to help you track down and fix them. Because of Turbo Pascal's accommodating structure and high speed of compilation, the cycle of entering, testing, and correcting your program wastes little time. Let's look at different aspects of that cycle in more detail.

So, What's a Compiler Anyway?

The Macintosh, like most microcomputers, has a central processing unit (CPU) that does most of the work. On the Mac, the CPU is a single chip: the 68000, a microprocessor designed by Motorola. The 68000 has a set of binary-coded instructions that it can execute. By giving the 68000 the right sets of instructions, you can make it draw objects on the screen, perform math, move text and data around—in short, do all the things that you want it to do. These instructions are known collectively as machine code.

Since machine code consists of pure binary information, it's neither easy to write nor easy to read. You can use a program known as an assembler to write machine-level instructions in a form that you can read. This is known as programming in assembly language. However, you still have to understand just how the 68000 microprocessor works. You'll also find that to perform simple operations—such as printing out a number—often requires a large number of instructions.

If you don't want to deal with machine code or assembly language, you use a high-level language such as Pascal. You can easily read and write programs in Pascal, because it is designed for humans, not for computers. However, the Mac understands only machine code. The Turbo Pascal compiler translates (or compiles) your Pascal program into instructions that the computer can understand. The compiler is just another program that moves data around; in this case, it reads in the text of your program and writes out the corresponding machine code.

What Gets Compiled?

You can edit up to eight different Turbo Pascal programs at the same time, each with its own window. If you have several windows open, which one is affected when you select a command from the Compile menu? As with the editing commands, it's the program in the currently active window, that is, the window whose title bar has horizontal lines and a close box in it. All the other (inactive) windows have nothing in the title bar except for the window's title (either the file name or "Untitled").

As with the editor, to make a window active, you move the cursor into it and click the mouse once. If the windows overlap, this brings the one you just selected to the front, so that you can see the entire window.

You can also use the Window command in the Search menu. Selecting the Window command brings the first window you opened to the front, and so on sequentially.

Where's the Code?

When you use the Run command, Turbo Pascal saves the resulting machine code in memory (RAM). This has several advantages. First, the compiler runs much faster, since it takes less time to write the machine code out to RAM than out to a floppy or hard disk. Second, since your program is already loaded into RAM, Turbo Pascal just tells the Macintosh to execute your code. Third, the Mac more easily returns to Turbo Pascal once your program stops executing, since Turbo Pascal also stays in RAM the whole time. Fourth, Turbo Pascal allows you to open several program windows and compile them to RAM. You can then execute each of them without recompiling.

If compiling to RAM is so wonderful, why wouldn't you want to do it every time? Two reasons. First, you would be able to run your programs only from Turbo Pascal. If you compile only to RAM, the resulting machine code is never saved on the disk, so you have no way of executing your program from the Finder. You also have no way of copying your program.

Though less likely, the second problem is memory: You might not have enough. It could happen if you're using a "thin" (128K) Mac, if your program is very large, if your program uses a lot of memory for dynamic data allocation, or if you have opened several windows and have compiled each of them.

It's easy to produce a code file (application) that you can run from outside Turbo Pascal: Select the To Disk option in the Compile menu (or press ⌘K). This produces a code file that you can run from the Mac desktop by double-

clicking its icon, or from within Turbo Pascal by using the Transfer command in the File menu.

The file produced by a (Compile) To Disk command has the name used in your program header. In other words, if your program has the header

```
program MyOwnProgram;
```

then the resulting code file is named MYOWNPROGRAM. However, you can specify a different file name (and a particular volume or subdirectory) by using the $O compiler option, such as

```
{$O Turbo:code:MyProg}
```

Appendix C, "Compiler Directives," has more details. In either case, the icon used is the standard Mac application icon of a hand writing on a piece of paper. You can create your own icon using a resource file; see Chapter 9 for further details.

Unlike the Run command, the To Disk command does not automatically execute your program once the compilation is done. You can execute it using the Transfer command in the File menu or by leaving Turbo Pascal and clicking on the icon. Or you can recompile it to RAM with the Run command, which then automatically executes it.

You may want to compile a program to RAM without running it. Perhaps you have several programs open, and you want to compile each of them to RAM before running them. In this case, select the To Memory command in the Compile menu (or press ⌘M). It works just like the Run command with two exceptions. First, it does not execute the program once compilation is done; instead, it leaves you in Turbo Pascal. Second, it always compiles the program, while the Run command recompiles only if you've modified the program since the last compilation. If you use the To Memory command, and select Run without making any changes to the program, the Run command won't recompile your program.

Syntax Errors

Just like English, Pascal has rules of grammar that you must follow. However, Pascal's rules are fairly strict, much more so than those of English. You can use poor grammar in speaking and still be understood; if you use poor "grammar" in your Pascal program, however, the compiler won't understand what you want. The result is a *syntax error*, which happens when you don't use the appropriate words or symbols in a statement, or when you organize them incorrectly.

When the compiler detects a syntax error, Turbo Pascal stops the translation and goes back to the editor. Once there, it moves the cursor to the spot in your

program where the error occurred. It then displays a box across the top of the screen, explaining (in brief terms) what the error was. Press ↵ to make the box go away, or move the cursor (via the mouse) into the box and click the mouse button.

What syntax errors are you likely to get? Probably the most common error novice Pascal programmers make is `Unknown identifier`. Pascal requires that you declare all variables, data types, constants, and subroutines—in short, all identifiers—before using them. If you refer to an identifier that you haven't declared, or if you misspell it, you'll get this error. Other common errors are `';' expected`, which means that you need to put a semicolon at the end of the previous statement, and `':=' expected`, which means that you need to use the assignment operator (:=) instead of the equals sign (=). Appendix B, "Error Messages and Codes," lists all the compiler syntax errors.

You can check for syntax errors without compiling the program by using the Check Syntax command in the Compile menu (or pressing ⌘Y). Turbo Pascal then checks your program's syntax, but doesn't produce any machine code. This is faster than compiling to disk, so it's a handy way to clean up syntax errors before producing a code file. On the other hand, it isn't significantly faster than compiling to memory, so consider using the Run or To Memory commands (unless you want to avoid compiling to memory for the reasons previously discussed).

Run-time (System) Errors

In programming, sometimes just following the rules governing correct syntax isn't enough. For example, suppose you write a simple program that prompts you for two integer values, adds them together, then prints out the result. The entire program might look something like this:

```
program AddNums;
var
  A,B,C   : Integer;
begin
  Write('Enter two integer values: ');
  ReadLn(A,B);
  C := A + B;
  WriteLn('The sum is ',C);
end.
```

In response to the prompt `Enter two integer values:`, say you type in real numbers (that is, numbers with decimal points), integer values that are too large, or even character strings instead of numbers. What happens? The Mac system error window appears, with the bomb icon and an error ID code in it. You are given two options, each presented as a button: Restart and Resume.

The Restart button, which you can always use, reboots your Macintosh, just as if you had turned your Mac off and on or you had pressed the Reset switch on the side of your Mac (assuming you have one installed; it's the one closest to the front). This button is best used only when you have no other option.

If you are running from within Turbo Pascal, you can select the Resume button instead. It puts you back into Turbo Pascal, with your windows (and files) still intact. This means that even if you didn't have the Auto Save option selected, the program file you've been editing for the last hour isn't gone. It's still there—unless, of course, your program went totally amok and wrote over large portions of memory (in which case you wouldn't have been able to get back to Turbo Pascal anyway).

For errors within a Turbo Pascal program, such as division by zero, range overflow, and I/O error, the cursor is moved to where the error took place, and a window with the bomb icon and a description of the error type appears. You must acknowledge the error by moving the cursor to the message window and clicking the mouse, or by pressing ↵. The window goes away, and you can figure out what changes (if any) to make to your program. If, after moving around in your program, you want to find the error again, select the Find Error command from the Compile menu (or press ⌘E). Turbo Pascal quickly recompiles your program (without producing code) and places the cursor where the error took place, with the bomb box again explaining the error.

Should the error occur within an include file (the next chapter has more information on include files), Turbo Pascal automatically opens a window for that file, reads it in, and moves the cursor to the error's location. If a window for that file is already open, that window is brought to the front, and the error located.

There's a way to go out on a limb and deliberately trigger a Mac system error: Press the Interrupt switch on the side of the Mac (assuming you have one installed; it's the one closest to the back). You might need to do this if, for example, your program is stuck in some section of code, such as an infinite loop. You won't be able to use the Find Error routine to locate where your program was when you interrupted it, but you can get back to Turbo Pascal without losing your program text and the cursor may be positioned at the point in the program where the execution was.

The Get Info Command

The Get Info command, which you can also invoke by pressing ⌘I, brings up a window that tells you how big the text of your Pascal program is, both in bytes and in lines. If you haven't compiled your program yet, or if you've made changes since your last compilation, it'll tell you that the program is `Not compiled`. Otherwise, it gives you the size of the code (in bytes) as well as the number of bytes that will be allocated for data when the program is run. Finally, it tells you how large the heap is and how much of that space is available. (The heap is where dynamic variables are created using the standard procedures New and Dispose.) Click on the OK button to make it go away, or press ↵.

The Options Command

The last item in the Compile menu is the Options command, which doesn't have a keyboard equivalent. It allows you to set up some default information for use by the Turbo Pascal compiler. First, you can decide how much space (in kilobytes) to allocate for the symbol table. The default is 32K, which is the maximum size. If you're running on a 128K Mac, you might want to make it smaller to get some memory back for compilation.

Second, you can set Auto Save to take effect when Run is selected. Auto Save automatically saves all edited windows to the disk when the Run command is selected from the Compile menu. Turbo Pascal keeps track of whether you've made changes in a given window since the last time you saved it to disk. When you select Run, Turbo Pascal first performs the Save command for all windows that have been modified.

Finally, you can set the default directories for all the compiler directives that reference files: $U (units), $I (include files), $R (resource file), $L (assembly language .REL files), and $O (output file). These compiler directives are discussed in further detail in Appendix C. Having made the changes you want, select either the OK button, which allows you to use these options, or the Cancel button, which ignores whatever changes you've made to the options. In either case, you're returned to the Turbo Pascal editing window.

If you want to make these options your standard settings, select the Save Defaults command in the File menu.

The ability to specify directories is very useful if you're running under Apple's Hierarchical File System (HFS) and want to keep these files in different subdirectories. If you don't specify a directory for a given option, the current directory is assumed. However, if the compiler option itself contains the directory

(such as {$I Turbo:otherstuff:linked.lib}), then the default directory (blank or not) is ignored completely.

Now that you're acquainted with the commands in the Compile menu completely, you're ready to move on to the next chapter. In it, you'll learn how to create a "textbook" Pascal program.

CHAPTER 5

Writing Textbook Pascal Programs

This chapter gives you the information you need to take standard Pascal programs out of textbooks and get them to run under Turbo Pascal. We'll review briefly how to create and save a program, then go into the Pascal run-time environment. We'll also cover compiler directives, input/output error checking, and range checking.

Turbo Pascal makes it easy for you to create a standard or "textbook" Pascal program on the Macintosh. No special knowledge is required; you just type in your program, compile it, and run. Turbo Pascal sets up a window for you and treats it like a plain CRT monitor. You can write to the screen, prompt for (and receive) input, move the cursor around, have the screen automatically scroll, and so on.

In other words, you don't have to know anything at all about the innards of a Macintosh to start writing Pascal programs on it. Most routines you find in textbooks run just fine under Turbo Pascal, with a few exceptions that we'll discuss later in this chapter.

To start with, let's review how to get a new program typed in and running.

Creating a Program: A Quick Review

To write and run a program, you need only follow the steps you've learned so far. Here's the procedure:

1. Move the mouse to the Turbo icon and double-click on it. Turbo Pascal brings up its menu bar and presents you with a blank program window labeled "Untitled". If this window doesn't appear (which could happen if you've disabled the Startup Window option using the Options command in the Edit menu), create a new window with the New command in the File menu (or press ⌘N).

2. Type your program in, using the keyboard, mouse, and menu commands discussed back in Chapter 3. Save it out to disk using the Save command in the File menu. Select the Options command in the Compile menu, and enable the Auto Save option if it's not already enabled.

3. Select the Run command from the Compile menu (or press ⌘R). If an error is found, Turbo Pascal returns you to the editor. Correct the error and select the Run command again.

4. Once you've corrected all syntax errors, your program will execute. If you have run-time errors, the Mac System Error box will appear. If that happens, click on the Resume button. You'll find yourself back in the Turbo Pascal editor, with the cursor placed where the error occurred, if it can be located. Correct the error, and select Run again. If you totally crash the system somehow, reboot the Mac and double-click on your program document icon. You'll be returned to Turbo Pascal, and you can edit your program. (This is why you set the Auto Save command: So that your source code is automatically saved to disk before each Run command.)

5. Once you've corrected all your run-time errors, save your program to disk again (select the Save command from the File menu, or press ⌘S). Now select the To Disk command from the Compile menu (or press ⌘K). When that's done, exit Turbo Pascal by selecting Quit from the File menu (or press ⌘Q).

6. Your program is now an executable file, appearing as the standard Mac application icon (a hand writing on a blank piece of paper). You can run it any time by double-clicking on that icon.

Let's look at some sample programs based on Standard Pascal.

Consider the following program:

```
program Product;
var
  A,B      : Integer;
  C        : Real;
begin
  Write('Enter two integer values:  ');
  ReadLn(A,B);
  C := A * B;
  WriteLn('The product is ',C:8:2);
  ReadLn;
end.
```

This program runs as written. A window (labeled "Ratio") is created. The prompt `Enter two numbers:` is written in the upper left corner of the window, and the blinking cursor sits a few spaces past the end of the prompt. The program then waits for you to type in two integer values. You may separate them with a blank or a carriage return, and you can use ⌫ to delete and retype what you've entered.

After you type the second value, press ↵. The program calculates *A*B* (converting to real) and assigns the resulting value to *C*. It then writes out the message `The product is`, followed by *C*'s value in a field eight characters wide, with two digits appearing after the decimal point.

The program then waits for you to press ↵, at which point it closes the window and returns either to Turbo Pascal (if executed with the Run command) or the Finder (if executed from the desktop or by using the Transfer option in the File menu).

Here's a second, even quicker example:

```
program Table;
var
  I        : Integer;
begin
  for I := 1 to 100 do
    WriteLn(I:3,' ',(I*I):6);
  ReadLn;
end.
```

When you run this program, you'll notice a few things. First, the window is now labeled "Table" to match the name in the program header. Second, Turbo Pascal scrolls the screen, just like a regular monitor, when you get to the bottom of the display. You may, at any time, stop screen output by pressing the mouse button and holding it down. When you release it, output continues. This is handy to keep text from scrolling off the screen before you have read it.

The Pascal Run-time Environment

The key to writing Standard Pascal programs is to simply type in the programs as you see them in your textbook. By default, Turbo Pascal links in a set of routines that implements Standard Pascal I/O on the Mac. These routines perform all the initialization that your program needs to be able to run on the Macintosh. They also create a simple Macintosh window that acts like the standard text screen of a terminal or personal computer. It displays 25 lines of text, with up to 80 characters on each line. The screen-like window disappears when your program ends execution.

Within this environment, the procedures *Read*, *ReadLn*, *Write*, and *WriteLn* function as expected, handling carriage returns and form feeds. Turbo Pascal also scrolls the display when necessary, as the second example program demonstrates. What's more, you can directly position the cursor using *GoToXY* and perform other screen operations using special Turbo Pascal procedures and functions. These are described in Chapter 25.

The Standard Pascal environment is actually implemented as a group of four units: *PasSystem*, *PasInOut*, *PasConsole*, and *PasPrinter*. A unit is a library or collection of useful subroutines and other declarations.

The unit *PasSystem* is *always* used, since it provides certain functions needed by all Turbo Pascal programs. The next two—*PasInOut*, and *PasConsole*—are also automatically used unless you set the {$U-} option. The last one, *PasPrinter*, is used only if you explicitly request it. For more details on using (or not using) units, see Chapter 7.

Compiler Directives

Most Pascal compilers allow some form of compiler directives. These are commands to the compiler, embedded in comment statements within your program. They typically take one of two formats:

```
{$<letter><+ or ->}
```

or

```
{$<letter> <filename>}
```

The first form is used to turn some option on or off. For example, {$R+} tells the compiler to produce range-checking code, while {$R-} tells it not to.

The second form usually directs the compiler to read from or write to some file. For example, the directive {$I MYLIB.PAS} tells the compiler to include the file MYLIB.PAS at this point in the program—in other words, to go to that file and read from it as if the text in that file had been inserted in the current program file.

All the compiler directives are documented in Appendix C, but here are some of the most commonly used directives.

Input/Output Error Checking

An issue often addressed in Pascal textbooks and classes is how to make your code "crashproof"; that is, how to set it up so that users can't cause your program to stop due to input/output (I/O) errors. For example, say you ran the first example program, PRODUCT, and, at the prompt `Enter two integer values:`, typed in a real value (that is, a number with a decimal point). Your program would halt, with a Mac system error box popping up. In a short program like this, such an error isn't a big bother. What if you were entering a long list of numbers, however, and had gotten most of the way through before making this mistake? You'd be forced to start all over again. So, making your program crashproof is important.

Like most compilers, Turbo Pascal allows you to disable automatic I/O error checking and test for it yourself within the program. To turn off I/O error checking at some point in your program, include the compiler directive {$I-}. This instructs the compiler not to produce code that checks for I/O errors and brings up the Mac system error box when one does occur. For example, we could modify the program PRODUCT to look like this:

```
program Product;
var
  A,B  : Integer;
  C    : Real;
begin
  Write('Enter two integer values:  ');
  {$I-}                                  { turn off I/O error checking }
  ReadLn(A,B);
  {$I+}                                              { turn it back on }
  C := A * B;
  WriteLn('The product is ',C:8:2);
  ReadLn;
end.
```

Now, no matter what you enter for *A* and *B*, you won't get a Mac system error box. That doesn't mean that there are no errors nor that *A* and *B* will have the values you think they do. If you make a mistake, the corresponding variable just gets the value zero (0).

With I/O error checking disabled, you can check for an error by calling the standard Turbo Pascal function *IOResult*. *IOResult* returns an integer value corresponding to the appropriate Mac I/O result code (see Appendix C). If the result is 0, then no error has occurred; otherwise, you'll probably want to take some action, even if it's just to ask for the values again. Your code might look like this:

```
program Product;
var
  A,B  : Integer;
  C    : Real;
begin
  {$I-}
  repeat
    Write('Enter two integer values:  ');
    ReadLn(A,B)
  until IOResult = 0;
  {$I+}
  C := A * B;
  WriteLn('The product is ',C:8:2);
  ReadLn;
end.
```

You need to be aware that each call to *IOResult* clears it; that is, sets it to zero. Also, each I/O call (*Write, WriteLn, Read, ReadLn, Assign, Reset, Rewrite*, and so on) sets *IOResult* to an appropriate value. For example, the following code wouldn't work properly:

```
program Product;
var
  A,B  : Integer;
  C    : Real;
begin
  {$I-}
  repeat
    Write('Enter two integer values:  ');
    ReadLn(A,B);
    if IOResult <> 0 then
      WriteLn('Error on input!')
  until IOResult = 0;
  {$I+}
  C := A * B;
  WriteLn('The product is ',C:8:2);
  ReadLn;
end.
```

There are two reasons this wouldn't work. First, the call to *IOResult* in the **if** statement `if IOResult <> 0` clears it, so that the call in the **until** clause doesn't represent what happened with the *ReadLn(A,B)*. Second, the call to *WriteLn* changes *IOResult* anyway. If you did want to print this message out, you'd have to do something like the following program.

```
program Product;
var
  A,B,IOCode  : Integer;
  C           : Real;
begin
  {$I-}
  repeat
    Write('Enter two integer values:  ');
    ReadLn(A,B);
    IOCode := IOResult;
    if IOCode <> 0 then
      WriteLn('Error on input!')
  until IOCode = 0;
  {$I+}
  C := A * B;
  WriteLn('The product is ',C:8:2);
  ReadLn;
end.
```

By saving *IOResult* in *IOCode*, we avoid both problems, since we only reference *IOResult* once (right after the place where we want to check for errors). For more sophisticated applications, you can take some action (for example, print a message) on the actual value of *IOCode*.

Range Checking: The {$R+/-} Directive

Another common compiler directive is {$R+/-}. It controls range checking of array and string indexes, and assignment to scalar data types. By default, range checking is turned off ({$R-}); you can turn it on with {$R+}.

This directive is used to track down errors caused by using array indexes that are out of bounds or by assigning out-of-range values to scalar variables. Suppose you had the following program:

```
program RangeTest;
var
  Indx  : Integer;
  List : array[1..10] of Integer;
begin
  for Indx := 1 to 10 do
    List[Indx] := Indx;
  Indx := 0;
  while (Indx < 11) do
  begin
    Indx := Indx + 1;
    if List[Indx] > 0 then
      List[Indx] := -List[Indx]
  end;
  for Indx := 1 to 10 do WriteLn(List[Indx]);
  ReadLn;
end.
```

If you type in this program, it will compile and run. And run. And run. It will, in fact, get stuck in an infinite loop. Look carefully at this code: The **while** loop executes 11 times, not just 10, and the variable *Indx* has the value 11 the last time through the loop. Since the array *List* only has 10 elements in it, *List[11]* points to some memory location outside of *List*. Because of the way variables are allocated, *List[11]* occupies the same space in memory as the variable *Indx*. This means that the statement

```
List[Indx] := -List[Indx]
```

is equivalent to

```
Indx := -Indx
```

Since *Indx* equals 11, this sets *Indx* to -11, which starts the program through the loop again. That loop now changes additional bytes elsewhere, at the locations corresponding to List[-11..0].

In other words, this program can really mess itself up. And since *Indx* never ends the loop at a value greater than or equal to 11, the loop never ends.

How do you check for things like this? Insert {$R+} at the start of the program. When you run a faulty program, you'll get a Mac system error box. Press the Resume button, and you're back in Turbo Pascal, at the right bracket (]) in the statement `if List[Indx] > 0`. A box appears with the error message `Range check failed`. This tells you that *Indx* has some value outside of *List*'s array bounds (1..10).

You can leave range checking on all the time just by placing {$R+} at the start of each program you write. However, the code generated to do range checking does make the program larger and slower. Also, there are some situations—usually in advanced programming—in which you might want or need to violate range bounds, most notably in working with dynamically allocated arrays, or in using *Succ* and *Pred* with enumerated data types.

You can selectively implement range checking by placing the {$R+} directive at the start of the code that needs it, then placing the {$R-} directive at the end of the code. For example, you could write the loop above as:

```
while Indx < 11 do
begin
  Indx := Indx + 1;
{$R+}
  if List[Indx] > 0 then
    List[Indx] := -List[Indx]
{$R-}
end;
```

Range checking will only be performed in the **if..then** statement and nowhere else in the program. Unless, of course, you have other {$R+} directives elsewhere.

Another commonly used compiler directive, {$I⟨file⟩}, allows you to break one large program file up into several smaller files. (Don't confuse this with the {$I+/-} directive used for I/O error checking.) {$I⟨file⟩} directs Turbo Pascal to include ⟨file⟩ during compilation. Turbo Pascal then opens this file and reads the Pascal code from it, compiling it as if it were part of your program. When it reaches the end of the included file, it closes the file and continues to compile your program.

Most Macintosh-style applications can be organized into chunks, each chunk containing related procedures and functions. If you were writing a bulky program, you could organize it as follows:

```
program BigJob;
{$I BigJob.Def}          { global declarations and definitions }
{$I BigJob.Util}               { utility procedures/functions }
{$I BigJob.Menu}            { menu-driven procedures/functions }
{$I BigJob.Event}        { event-handling procedures/functions }
{$I BigJob.Init}       { initialization and cleanup procedures }
begin
  Initialize;
  repeat
    SystemTask;
    if GetNextEvent(theEvent) then
      HandleEvent(theEvent)
  until Finished;
  Cleanup
end.                                        { of program BigJob }
```

This program text is placed in a file called BIGJOB.PAS. In addition, five other text files (BIGJOB.DEF, and so on) contain the appropriate parts of the program. Since Turbo Pascal allows you to have up to eight windows open at the same time, you can have all the files open for editing. That way, you can quickly switch between them just by clicking inside the different windows, instead of having to jump back and forth within one large file. You can also look at the different portions side by side by arranging the windows on the screen, using the Stack Windows or Tile Windows command in the Format menu.

There is a better way to break up large programs into chunks: units. For example, you could place the definitions and routines in BIGJOB.DEF and BIGJOB.UTIL into a single unit, compile it, and use it with a **uses** statement. Likewise, you could turn BIGJOB.MENU, BIGJOB.EVENT, and BIGJOB.INIT into units. Chapter 8 gives more details on how to do this.

When you compile a Turbo Pascal program to disk, the resulting code file adopts its name from the program header. For example, if your program header is

```
program Banzai;
```

then the code file created on the disk is called BANZAI. However, you can override that default and request a specific code file name using the {$O ⟨file⟩} directive. This defines the name of the output (machine code) file. If you changed your program to read

```
program Banzai;
{$O MyNeatProgram}
```

then a compile to disk produces a code file named MYNEATPROGRAM.

You now know how to get Standard Pascal programs running on the Mac under Turbo Pascal. However, you can refer to Part II and Appendix A as your programs grow more complex. They offer more information on the special features that Turbo Pascal has to offer.

Of course, you don't want to stop with "textbook" Pascal programs. You want to write Mac-style programs, programs that use menus and windows and graphics. The rest of Part I is designed to help you to do just that. Let's lay some groundwork in Chapter 6.

CHAPTER 6

Harnessing the Full Power of Your Mac

The Macintosh has some of the most sophisticated system software ever put on a microcomputer. It gives most larger computers a run for their money. To programmers, however, sophisticated usually means complex, and complex rarely means easy to program. We'll discover whether that is true of the Mac in this chapter. We'll introduce you to the concepts behind the Mac, explore bit-mapped graphics, and explain the Toolbox and operating system tools and resources that are at your disposal.

The Macintosh Philosophy

The designers of the Macintosh had the stated goal of designing a "computer appliance," the microcomputer equivalent of a toaster—that is, a system that people with little computer experience or background could learn to use in a very short time. By this criterion, the Mac is a smashing success: It is, to date, the easiest computer for a naive user to learn. Most Mac software follows a standard user interface, or format, so the typical Mac user can start using new applications almost immediately.

The use of a standard interface is enforced by Toolbox (ROM-based) and operating system (RAM-based) routines that the Mac provides. Simply put, the obstacles to *not* using the Mac's routines are so great that most applications conform to them. However, the routines are so comprehensive and complex that the novice Mac programmer faces a steep learning curve.

The basic Macintosh isn't terribly complex in terms of hardware. It has a 68000 (or 68000-related) processor, a monochrome (black-on-white) bit-mapped screen, RAM memory ranging from 128K in the older Macs to over 4M in upgraded systems, Read Only Memory of either 64K or 128K, and some I/O hardware (serial ports, disk ports, and so forth). Fairly simple and straightforward stuff—until you look at what's in that ROM.

The Mac pioneered four major microcomputer concepts: graphics-only display, visual user interface, event-driven software, and extensive system software.

Graphics-Only Display

Until the Mac appeared, most computers had text-only display or let you choose between text and graphics. In both cases, the text display was a fixed font with predetermined resolution and size (typically 80 columns of text in a 25-line display).

The Mac doesn't have text-only display. Instead, everything is done with bit-mapped graphics, including all text display. Bit mapping simply means that the Mac screen is made up of a grid of bits, which make up the shapes—characters or figures—that appear on your display. It's explained further in the following pages.

Because of bit mapping, writing and editing text on the Mac screen is more complex than on other micros. But it also means that you have tremendous flexibility in how that text is presented, in terms of size, style, and font design, and in mixing text with graphics. In addition, you can change any of these elements and redisplay them on screen countless times.

The Mac almost single-handedly spawned desktop publishing, although this function is rapidly being adapted to other systems. The ability to produce high quality documents with professional layouts used to be limited to companies that could afford to buy or use very expensive typesetting equipment. Now, anyone with a Mac can lay out and prepare slick documents. With access to a laser printer (or even some of the newer typesetting machines), you can produce hard copy that is comparable to copy from a professional printer.

Visual User Interfaces

The second Mac design concept is the *visual user interface* based on menus, icons, windows, and a mouse as the input device. The concept itself isn't new. Neither is the interface unique to the Mac, since other microcomputers now offer similar approaches. However, the Mac represents the first (and still the best) attempt to make such a user interface available at a relatively low price.

Event-Driven Software

The third major concept is *event-driven software*. As with the first two concepts, this concept did not originate with the Mac, but it was the first micro to extensively use it and, in fact, to make it a requirement for just about any application.

At the core of most Mac applications is an *event loop* that polls the Mac operating system for events (mouse clicks, keys pressed, menu selections, window operations, and the like), then calls the appropriate internal routines to handle those events. The goal is what Mac designers call modeless programs, where most functions are available at any point. In modal programs, you have to enter specific modes (insert mode, delete mode, command mode) to be able to perform the corresponding functions.

Extensive System Software

The fourth major concept is *extensive system software* (in ROM and on disk). The software supports the user interface and event-driven programming approach, and makes them standard for all applications. Earlier microcomputers had some software (usually the Basic Input/Output System or BIOS) in ROM, but this was usually on the order of 8K to 16K of ROM and supported rather primitive screen and disk I/O functions. The original Mac came with 64K of ROM (increased in later versions).

The current Mac ROM supports numerous and complex functions. The Mac operating system provides a large set of standard functions and procedures, and it also maintains an event queue that keeps track of events that applications must deal with.

The irony of all this is that the original Macintosh was sorely crippled due to hardware limitations, with no means of adding memory and no expansion slots or hardware bus. However, Apple has learned some lessons since then, and the current Macintosh Plus represents a far better environment for the Mac software concepts. Future Mac products will undoubtedly continue to improve upon that.

Bit-Mapped Graphics

The standard Mac interface is a bit-mapped graphics display consisting of 342 lines, each line containing 512 *pixels* (picture elements, that is, dots). There are 175,104 pixels in all, each of which can be black or white. The display is called bit-mapped because each pixel on the screen corresponds to a single bit (0 or 1)

in memory (RAM). For that reason, this type of display is also referred to as memory-mapped.

Since there are 8 bits in a byte, simple math shows that the screen takes up 21,888 bytes (or about 21K) of memory. By changing the values of those bytes, you change what's on the screen; it's that simple. To draw a line (or other shape), your program simply goes to the appropriate locations in memory and sets the appropriate bits to 0 or 1. What's more, the Mac has an extensive and powerful graphics library (*QuickDraw*) to make using these graphics even easier.

However, manipulating a bit-mapped display can be complicated and tedious. To draw a character on the screen, you can't just poke an ASCII value into a byte somewhere, as you can on most other microcomputers. Instead, the character must be drawn bit by bit. A programmer can simplify matters somewhat by maintaining a character font somewhere, with a bit map for each possible character, but there are still issues of font size and style, of whether the font is letter-spaced proportionally (that is, characters are spaced according to their width), and so on. In addition, adding, deleting, and modifying text can get very elaborate. Fortunately, the Mac comes with a large set of routines for text display, manipulation, and editing. By using these, you don't have to reinvent the wheel.

The real bonus of bit-mapped graphics is the marriage of graphics and text, and the ability to manipulate both on the same display. You can readily mix pictures and words, allowing diagrams to be inserted in documents and explanations in schematics. On the Mac, you can draw a picture, then paste it into a letter or report. And, of course, it is this flexibility that has made the Mac preeminent in the field of desktop publishing.

The Mac User Interface

Three interrelated ideas form the nucleus of the Mac user interface:

- the mouse as an input device
- using icons, menus, windows, and other (mostly) graphic devices for information and command selection
- an orientation toward modeless environments

The mouse may well be the most controversial of the three ideas. Debates continue to rage over whether it aids or hinders user interaction. For that matter, users argue over whether the mouse should have one, two, or three buttons. A graphics-based, visual system does require some sort of pointing device, and the mouse works as well as or better than most.

The second idea, simply put, is that graphics convey more information than

text (or, to abuse an old cliche, a picture is worth 1K words). By presenting files as icons on a desktop, the selection and manipulation of files with the mouse is fairly self-evident, especially for novice users. Likewise, the pull-down menu makes it easy to view and choose available commands and options without having to remember obscure command names or wade through multiple levels of text-based menus. Within applications themselves, there tends to be a heavy orientation towards presenting data in graphic (rather than textual or numeric) form.

The third idea assumes that it is ideal to have as many options as possible available to the user at any given time. Rather than have multiple levels and numerous loads, the Mac attempts to keep all commands on one level, though some commands or options may be disabled when appropriate.

For example, the File menu in Turbo Pascal is always on the menu bar, as are the Edit, Search, Format, Font, Compile, and Transfer menus. However, not all commands in those menus are available at all times. The user is spared the tedium of keeping track of which mode he or she is in and the commands that exist (or don't exist) on that level.

Event-Driven Programming

The basic structure of most Macintosh applications is nearly identical, with a main body that looks something like this:

```
begin
  Initialize;
  repeat
    SystemTask;
    if GetNextEvent(eventMask,theEvent) then
      HandleEvent(theEvent)
  until Done;
  CleanUp
end.
```

The program does its setup with the user-defined routine *Initialize*. It then enters a loop that continues until some condition (such as the user selecting the command Quit in a menu) causes it to set the Boolean flag *Done* to True.

Within that loop, it performs two major tasks. First, it calls *SystemTask* (a Toolbox routine), which allows the Mac operating system to update any desk accessories that might be in use. Second, it calls *GetNextEvent* (another Toolbox routine) to see if any events have occurred. If any have, it calls *HandleEvent*, which is a user-defined routine that handles all the different events that might occur. Such events include key presses; selection of menu items; mouse clicks; windows being opened, closed, uncovered, or resized; and similar occurrences.

When the program is done, it calls the user-defined routine *CleanUp*, which

takes care of any necessary tidying up. (This last task depends on the application itself; usually, it means freeing allocations made in memory by the application back to the system and similar tasks.)

This is quite different from most interactive computer software developed before the Macintosh. In other systems, the program usually sits and waits for the user to type in a specific command, then handles it. The programs tend to be modal, with different levels and modes, each having its own command set. The commands themselves are usually context-sensitive, with the same command (or at least command sequence, that is, a given letter or word) holding different meanings depending upon the current mode or state.

In a Macintosh application, most commands are usually available and applicable. About the only time you can't use a given command is when there is nothing to use it on; for example, if you haven't opened a window to edit text, then most of the editing commands don't make any sense. In the modeless approach, however, those edit commands can work on any text window, whether it be one your application has opened, or one opened by a desk accessory.

Event-driven programming takes some getting used to, but once you understand how it works and have seen a number of examples using it, it becomes straightforward and easy to apply to different situations. In Macintosh applications, the format is so standard that you can move from program to program and see almost identical code in the main procedures and immediate supporting routines (such as *HandleEvent*).

Toolbox and Operating System Routines

To make the Mac user interface standard in most applications, Apple designed it to be easy to follow and difficult to deviate from. This was particularly true of the original Mac, which had 128K of RAM (much of which was consumed by the video display and the operating system) and 64K of ROM (the Toolbox). Since graphic applications tend to be memory intensive (that is, they need lots of RAM), most developers on the Mac just didn't have the extra memory to do things their way. So they were forced to use the extensive libraries of procedures and functions found in the Toolbox ROM and in the operating system itself.

The resulting uniformity in Mac software allows most Mac owners to use a brand-new software package with little or no reference to the manual.

The Toolbox and operating system (OS) routines are organized into related groups, often labeled managers or packages (not unlike units, which you'll learn about in Chapter 7). A list follows with the routine name and a brief description

of what the routines and data types within each allow you to handle. The list is arranged more or less in order of what you need to learn before moving on to the next item, although some concepts are best understood as a group.

Resource Manager: Files can contain *resources*, such as definitions of menus, windows, icons, and text strings, as well as chunks of code. These routines let you access, identify, and manipulate resources within a given file.

QuickDraw: The heart of the Macintosh, this package contains the basic graphics routines used by the other managers and packages.

Font Manager: You can display text on the Mac in different *fonts*, that is, with differently designed character sets. This package helps you (and *QuickDraw*) load or unload specific fonts from the disk for text display.

Toolbox Event Manager: These routines form the foundation for event-driven programming. Besides *GetNextEvent*, this package also allows for direct polling of the mouse, the keyboard, and the system clock.

Window Manager: The Mac allows you to set up multiple windows, each acting like its own screen. This package helps you create, move, modify, update, and delete windows.

Control Manager: Mac software often uses graphic controls — buttons, dials, scroll bars, switches, and check boxes — for selection and display. These routines allow you to select and use predefined controls and to design your own.

Menu Manager: A menu bar across the top of the screen shows you the pull-down menu options; selecting a particular menu allows you to examine commands and pick a specific one. This package lets you create, manipulate, and interrogate your menus (that is, go into the menu commands and find out how they were set up).

TextEdit: TextEdit helps you edit text by providing routines to insert, delete, select, and scroll text within a window, and to transfer text from one location (or window) to another.

Dialog Manager: When a Mac application wants to bring something to the user's attention, it usually does so via a *dialog box*. This is a window with some information (graphics and/or text) in it and with one or more ways to enter a command (buttons, switches, text or numeric entry, and so on). This package helps you design and present dialog boxes, and to correctly interpret a user's response to it.

Desk Manager: Most applications maintain the Apple menu option (the one on the far left with the Apple logo instead of a name), which contains special programs known as desk accessories (covered in Chapter 8). Desk accessories can be open and running even while your program is running, so you need to be able to accept and receive their messages. Desk Manager routines allow you to detect and handle actions required by the desk accessory.

Scrap Manager: This package helps you transfer data (such as text) between applications or between locations in a given application.

Toolbox Utilities: This is a collection of miscellaneous routines, including (but not limited to) fixed-point math; string, byte, and bit, including logical operations on long integers; and miscellaneous graphics-oriented routines.

Package Manager: A package is a set of routines and data structures stored on disk (as resources in the SYSTEM file) and loaded into RAM as needed. Six different packages are available through the Package Manager:

- the Binary-Decimal Conversion Package (conversions between decimal strings and internal binary representations);
- the International Utilities Package (different languages' character sets);
- the Standard File Package (selecting files for I/O);
- the Disk Initialization Package (formatting blank disks);
- the Floating-Point Arithmetic Package (for IEEE-standard floating-point math);
- the Transcendental Functions Package (for floating-point routines, such as trigonometric functions, logs, and financial functions).

Memory Manager: The Mac has a complex way to allocate relocatable blocks of memory so that dynamic garbage collection can occur without disturbing any program currently executing. When used properly, these routines ensure that memory is correctly allocated or recovered.

Segment Loader: Programs that are unwieldy because of size can be divided up into segments; each segment can be up to 32K in size. The Segment Loader governs the execution, segment loading, and termination of an application.

Operating System Event Manager: The Toolbox Event Manager allows you to query the operating system for events; the routines in this manager allow you to work directly with the event queue that the operating system maintains.

File Manager: This manager handles just about everything having to do with files, from high-level volume management to low-level file I/O.

Printing Manager: The Macintosh presents some special challenges in capturing what's on the screen or in a text file out to a printer. In conjunction with the printer drivers found on your system disk, these routines allow you to print the graphic images created by Mac software.

Device Manager: This package is a general version of the File and Printing Managers. It lets you work with custom device drivers and perform I/O with those devices.

Disk Driver, Sound Driver, Serial Drivers: These routines give your software control of the corresponding hardware items (floppy drives, DAC, RS422 ports) on the Mac.

AppleTalk Manager: Apple has defined a simple local-area network (LAN) for Apple products known as AppleTalk. This collection of data structures and routines allows you to communicate over that network.

Vertical Retrace Manager: This allows you to create interrupt-driven tasks that are called every so many ticks, where a tick is one-sixtieth of a second and corresponds to how often a vertical retrace interrupt occurs. A vertical retrace is one cycle of redrawing the screen, and you can use the cycle as a timer to trigger a routine.

System Error Handler: The one routine in this package, *SysError*, brings up the system error dialog box (with the bomb in it). Not for casual use.

Operating System Utilities: Another collection of assorted handy routines, including procedures and functions for pointer and handle manipulation, string comparison, date and time operations, parameter RAM operations, and other utilities.

With the proper use of these routines, your application will fit into the standard Macintosh mold. A regular Mac user will then be able to easily start it up and use it.

Further Reading

If you want to do any serious programming on the Mac, there are a few standard reference works:

- *Inside Macintosh*, written by a team at Apple and published in four volumes (softbound) by Addison-Wesley, documents the hundreds of routines available through the Toolbox and operating system. It is also available hardbound.
- *Macintosh Revealed*, volumes 1 and 2, written by Stephen Chernicoff and published by Hayden (under its Apple Press line of books).
- *MacTutor* is a magazine dedicated to programming on the Mac; each issue usually contains lots of short, working samples of code (call (714) 630-3730).
- *How to Write a Macintosh Application*, written by Scott Knaster and published by Hayden.
- *Macintosh Technical Notes*, published by Apple Computer, Inc.

Now, let's go on to the libraries—units—that the Mac provides.

CHAPTER 7

Units and Other Mysteries

In Chapter 5, you learned how to adapt standard Pascal programs for use by Turbo Pascal on the Mac. What about non-standard programming—more specifically, Mac-style programming? Before anything else, you have to understand the concept of units and of **external** and **inline** procedures and functions.

This chapter explains what a unit is, how you use it, and what predefined units are available for your use. You'll also learn how to set up and use external and inline procedures and functions. Among the other mysteries we delve into here are traps and assembly-language routines.

What's a Unit, Anyway?

Turbo Pascal gives you access to a tremendous number of predefined constants, data types, variables, procedures, and functions. Some are specific to Turbo Pascal; others are specific to the Macintosh. There are literally hundreds of them, but you hardly ever use them all in a given program. Because of their number, they are split up into related groups called *units*. You can then use only the units your program needs.

A unit is a collection of constants, data types, variables, procedures, and functions. Each unit is almost like a separate Pascal program. It can even have a main body that is called before your program starts and is used to do whatever initialization is necessary. In short, a unit is a library of declarations that you can pull into your program and use.

All the declarations within a unit are usually related to one another. For example, the *QuickDraw* unit has all the declarations for *QuickDraw* routines on the Mac.

When a program uses a unit, all its declarations become available, just as if they had been defined within the program itself.

A unit consists of two parts: the *interface* and the *implementation*. The interface is the "public" part of the unit. It contains constants, data types, and variables. It also has a list of procedure and function headers. Any program using the unit can use all these items. In other words, the program uses them as if they had been declared within the program itself.

The implementation is the "private" section of the unit. The bodies of the procedures and functions declared in the interface reside here. Additional constants, data types, and variables can be declared and used within the implementation. Likewise, additional procedures and functions may exist in this section. However, all these items are "invisible" to the program using the unit; the program doesn't know that they exist and can't reference or call them. However, these hidden items can be (and usually are) used by the "visible" procedures and functions, that is, those routines whose headers appear in the interface. Chapter 8 explains more about both these sections.

The units your program uses have already been compiled; that is, they are stored as machine code, not as Pascal source code. They are *not* Include files. Even the interface section is stored in the special binary symbol table format that Turbo Pascal uses. Furthermore, all the standard units (listed in the next paragraph) are stored in the Turbo Pascal compiler/editor file and are loaded into memory along with Turbo Pascal itself.

As a result, using a unit or several units adds very little time (typically less than a second) to your program's compilation time. If the units are being loaded in from a separate disk file, a few additional seconds may be required because of the time it takes to read from the disk.

Turbo Pascal provides 16 standard units for your use. Five of them—*PasSystem*, *PasInOut*, *PasConsole*, *PasPrinter*, and *SANE*—are known as the Pascal Run-time Support units and deal specifically with Turbo Pascal. The other 11 units—*MemTypes*, *QuickDraw*, *OsIntf*, *ToolIntf*, *PackIntf*, *MacPrint*, *FixMath*, *Graf3D*, *AppleTalk*, *SpeechIntf*, and *SCSIIntf*—allow access to the full range of Macintosh Toolbox and operating system routines, including support for *AppleTalk*, *MacinTalk*, and the *SCSI* hard disk port.

NOTE: In Turbo Pascal, each unit is assigned a specific unit number for identification purposes. You don't really need to know these numbers. Just be aware that *negative* numbers are reserved for the standard units and for assignment by the UNITMOVER; *positive* numbers are available for any units you create.

How Are Units Used?

To use a specific unit or collection of units, you place a uses-clause at the start of your program. The uses-clause consists of the keyword **uses**, followed by a list of the unit names you want to use, separated by commas:

```
program MyProg;
uses thisUnit,thatUnit,theOtherUnit;
```

When the compiler sees this uses-clause, it adds the interface information in each unit to the symbol table and links the machine code produced by the implementation to the program itself.

The units are added to the symbol table in the order given; this ordering can be important when one unit uses another unit. For example, if *thisUnit* used *thatUnit*, the uses-clause would have to be:

```
uses thatUnit,thisUnit,theOtherUnit;
```

or

```
uses thatUnit,theOtherUnit,thisUnit;
```

In short, a unit must be listed *after* any units it uses.

If you don't put a uses-clause in your program, Turbo Pascal links in three Pascal Run-time Support units anyway: *PasSystem*, *PasInOut*, and *PasConsole*. These provide some of the standard Pascal routines, a number of Turbo-Pascal specific routines, and also a model Pascal environment (complete with an 80×25 screen, cursor-control and printer routines, and so on).

What if you don't want some or all of these run-time units? You tell Turbo Pascal not to use them by placing the {$U-} option at the start of your program; only the *PasSystem* unit will be linked in. Any additional units you want must be explicitly requested with the uses-clause. For example, if you are writing a Mac-style application and don't want to use the *PasConsole* unit (since you don't need it), your program might look like this:

```
program MyMacProg;
{$U-} { don't automatically use the run-time stuff }
uses PasInOut,MemTypes,QuickDraw,OSIntf,
     ToolIntf,MacPrint,PackIntf;
```

This program does use one of the run-time units: *PasInOut*. However, since you've put the {$U-} option in, you have to explicitly request the unit to use it.

Pascal Run-time Support Units

Turbo Pascal provides a set of routines that make your Macintosh act like a standard terminal, allowing you to read from the keyboard and write to the screen without all the tedious mucking about that the Mac usually requires of you. These routines also let you do conversions, Pascal-style dynamic memory allocation, Pascal-style file I/O, and so on.

Turbo Pascal uses three units—*PasSystem*, *PasInOut*, and *PasConsole*—to do all this. Two other units—*PasPrinter* and *SANE*—are provided for additional support. Here's a brief description of each unit, with its name and a listing of any units it might require. Appendix D lists the interface sections of these units.

PasSystem
Units used: none

PasSystem implements the low-level support routines used by most programs, including *LongInt* math, conversion between *Real* and *Integer* data types, string and set handling, dynamic memory allocation, and byte-oriented procedures. It is linked into every program, even if you use the {$U-} compiler option. (It's the only unit that doesn't have an interface listing in Appendix D.)

PasInOut
Units used: none

PasInOut implements the standard Pascal I/O routines (*Read*, *ReadLn*, *Write*, *WriteLn*, *Reset*, *Rewrite*, and so on), as well as the Turbo-Pascal specific ones (*Close*, *Seek*, *Rename*, *Erase*, and so forth). It also does all I/O and range-error checking. If you look at the interface listing in Appendix D, you'll find that there is very little you can use directly; instead, the compiler makes calls to specific hidden routines in the implementation.

PasConsole
Units used: *PasInOut*

PasConsole is the unit that makes it easy to write textbook Pascal programs. It creates a window that emulates a terminal screen 80 characters wide by 25 lines deep. When this unit is used by a program or unit, any calls to *Read* or *ReadLn* without a file variable are made from the keyboard and automatically echoed to this window; likewise, any calls to *Write* or *WriteLn* without a file variable write to that window. A number of cursor- and screen-control routines are available: *ClearScreen*, *ClearEOL*, *InsertLine*, *DeleteLine*, and *GoToXY*. The functions *KeyPressed* and *ReadChar* are there, too, as are the file variables *Input* and *Output*. This unit also creates a new device ('Console:') that can be assigned to any file of type *Text*. The user can then send output to the screen (instead of to a disk file).

PasPrinter
Units used: *PasInOut*

PasPrinter declares the text-file variable *Printer* and connects it to a device driver that (you guessed it) allows you to send standard Pascal output to the printer using *Write* and *WriteLn*. For example, having included *PasPrinter* in your program, you could do the following:

```
Write(Printer,'The sum of ',A:4,' and ',B:4,' is ');
C := A + B;
WriteLn(Printer,C:8);
```

Like *PasConsole*, this unit creates a new device ('Printer:') which can be assigned to any file of type *Text*. The user can then send output to the printer (instead of to a disk file).

SANE
Units used: none

The SANE unit implements the Standard Apple Numeric Environment (SANE). SANE is the basis for all floating-point mathematical calculations performed by Turbo Pascal. Programmers who are interested in using SANE features not directly supported by Turbo Pascal can access these features through the SANE unit. For detailed instructions about SANE, see Chapter 26 and the *Apple Numerics Manual.*

Macintosh Interface Units

The Macintosh is a complex, sophisticated microcomputer. Some of its power comes from the built-in procedures and functions in the 64K or 128K of ROM and the SYSTEM file on disk. These routines are documented in *Inside Macintosh*, which breaks up the routines into a series of managers or packages: resources, *QuickDraw* (graphics), fonts, events, windows, controls, menus, *TextEdit* (text-editing), dialog boxes, and so on. In fact, you can think of these managers as units built into the Mac itself.

The Macintosh Interface units that Turbo Pascal provides allow you to use these Mac routines. Some of the units encompass several Mac managers or packages. This is to make things more manageable; otherwise, you might need 20 to 30 different units. Appendix D lists the interface sections of these units.

As with the Run-time Support units above, a brief description of each unit is given, along with a list of the units it uses. Also, the *Inside Macintosh* chapters that you can refer to are noted; the first number is the volume number, the second is the chapter number.

MemTypes
Units used: none
Chapters: "Mac Memory Management" (vol. I, chap. 3)

MemTypes defines special Mac data types, such as *SignedByte*, *Ptr*, *Handle*, and *Str255*. That's all it does; there are no constants, variables, or routines defined. It is used by every unit in this list and so must be included in any Mac-style applications.

QuickDraw
Units used: *MemTypes*
Chapters: "QuickDraw" (I-6)

QuickDraw is a Macintosh graphics package that lets you perform complex graphic operations quickly and easily. This unit defines all the constants, types, variables, procedures, and functions needed to use *QuickDraw*. Since *QuickDraw* resides entirely in ROM and uses standard Pascal parameter-passing conventions, the routines are all inline (see next item), and the unit itself contains no actual code.

OSIntf
Units used: *MemTypes*, *QuickDraw*
Chapters: "Memory Manager" (II-1), "Segment Loader" (II-2), "OS Event Manager" (II-3), "File Manager" (II-4), "Device Manager" (II-6), "Disk Driver" (II-7), "Sound Driver" (II-8), "Serial Drivers" (II-9), "Vertical Retrace Manager" (II-11), "System Error Handler" (II-12), "OS Utilities" (II-13)

The Macintosh operating system (Mac OS) is at the lowest level of Macintosh operations. It performs basic tasks such as input/output, memory management, and interrupt handling. Many of the Toolbox procedures and functions call Mac OS routines to support their operations. The *OSIntf* unit declares the Pascal interface to the Mac OS, naming the many constants, data types, variables, and routines. Since few of the Mac OS routines abide by Pascal conventions, inline code can't be used; instead, the unit itself provides the "glue," consisting of various additional external assembly-language routines. *OSIntf* is easily the largest of the interface units.

ToolIntf
Units used: *MemTypes*, *QuickDraw*, *OSIntf*
Chapters: "Resource Manager" (I-5), "Font Manager" (I-7), "Toolbox Event Manager" (I-8), "Window Manager" (I-9), "Control Manager" (I-10), "Menu Manager" (I-11), "TextEdit" (I-12), "Dialog Manager" (I-13), "Desk Manager" (I-14), "Scrap Manager" (I-15), "Toolbox Utilities" (I-16)

The Toolbox implements the Macintosh's user interface features: windows, menus, controls, dialog boxes, text editing commands, and so on. This powerful set of tools helps you create sophisticated applications with comparatively little

effort. A few of these routines need to be linked with routines via the *ToolIntf* unit; most, though, can be taken care of with inline calls.

PackIntf
Units used: *MemTypes*, *QuickDraw*, *OSIntf*, *ToolIntf*
Chapters: "Package Manager" (I-17), "Binary-Decimal Conversion Package" (I-18), "International Utilities Package" (I-19), "Standard File Package" (I-20), "Disk Initialization Package" (II-14)

Packages are sets of data structures and routines that are stored as resources in the SYSTEM file and brought into memory only when needed. They serve as extensions to the Toolbox and Mac OS; the most useful (and most commonly used) is the Standard File Package, which brings up the standard Mac dialog box to open files or select a file name for output. *PackIntf* provides the interface to those packages.

MacPrint
Units used: *MemTypes*, *QuickDraw*, *OSIntf*, *ToolIntf*
Chapters: "Printing Manager" (II-5)

The *MacPrint* unit provides access to the Macintosh Printing Manager. The Printing Manager is a set of RAM-based data types and routines that allow you to use standard *QuickDraw* routines to print text or graphics on a printer. These provide a device-independent interface to printer drivers, which enable you to print on a specific device (ImageWriter, LaserWriter, and so on). One (or more) of these printer drivers—usually found in the SYSTEM folder—must be available in order to use this package.

FixMath
Units used: *MemTypes*
Chapters: none

The *FixMath* unit is a collection of types and functions that implement fixed-point real numbers. This unit is very useful for applications that require real numbers but don't need the accuracy of floating-point math. Fixed-point operations run much faster than regular floating point, so you can choose to sacrifice precision for increased speed.

Graf3D
Units used: *MemTypes*, *QuickDraw*, *FixMath*
Chapters: none

Graf3D is a RAM-based, three-dimensional graphics package that sits on top of *QuickDraw*. It implements 3-D *GrafPorts* and provides a complete set of 3-D operations, including rotation, translation, scaling, and clipping.

AppleTalk
Units used: *MemTypes*, *QuickDraw*, *OSIntf*
Chapters: "AppleTalk Manager" (II-10), (see also *Inside AppleTalk*)

AppleTalk is the Macintosh local-area network — that is, the means by which you connect a group of Macintoshes with printers, disks, other devices, and each other. The AppleTalk Manager is used to communicate with devices connected to an AppleTalk network. *AppleTalk* is implemented as two RAM-based device drivers, .ATP and .MPP, and the *AppleTalk* unit declares the necessary Pascal types and procedures for using them.

The drivers are in the resource branch of the file ABPACKAGE on the distribution disk. If an application will use *AppleTalk*, then these drivers should either be placed in the SYSTEM file or in the application itself. The latter is preferable, since you can then move the application from disk to disk (and system to system) without having to worry about whether or not the drivers are there. To add the drivers to your file, put the following lines in your RMAKER resource file (see Chapter 6 and Appendix C for more details on RMAKER):

```
Type atpl = GNRL
,0 (16)
.R
ABPackage atpl 0
```

The atpl resource type *must* be in lowercase letters.

NOTE: The *AppleTalk* drivers may not be redistributed. They are licensed by Borland International and are for your personal use only.

SpeechIntf
Units used: *MemTypes*
Chapters: none

The *SpeechIntf* unit provides an interface to *MacinTalk*, a speech synthesizer that runs under Mac OS as a driver. In real time, *MacinTalk* converts an ASCII string of phonetic codes into synthetic speech. *MacinTalk* uses a special program, READER, to convert English text into the phonetic codes used by *MacinTalk*. The *MacinTalk* driver *must* be in the SYSTEM folder in order for your program to work.

More information on *SpeechIntf* is contained in the *MacinTalk* Toolkit documentation, in the December 1985 *Mac Software Supplement Document* from Apple.

NOTE: *MacinTalk* may not be redistributed. It is licensed by Borland International and is for your personal use only.

SCSIIntf
Units used: *MemTypes*
Chapters: "The SCSI Manager" (4-31)

The *SCSIIntf* unit provides access to the Small Computer Standard Interface (SCSI) port found on several models of the Macintosh. It allows you to determine what devices are connected to the SCSI port and to communicate with them.

Calling Assembly-Language Routines

Yes, Turbo Pascal allows you to link in external subroutines written in 68000 assembly language. Full details, including how to pass parameters and return function values, can be found in Chapter 27. Following is a quick explanation of how to call assembly-language routines.

Before using an external procedure or function in a program, you must define it. To define it, you write its **procedure** or **function** header, followed by the keyword **external**:

```
procedure LowToUp(var Str : string); external;
function RotLeft(var L : LongInt; D : Integer) : LongInt; external;
```

Note that there is no body to the procedure or function, just the header statement.

The procedure/function headers go wherever a regular procedure or function can go. If they're in a program, you can place them anywhere. If they're in a unit, they can go either in the interface (if you want the user to be able to call them) or in the implementation (if you don't).

Next, write the appropriate routines, using Apple's Macintosh Development System (MDS) or an MDS-compatible assembler. The resulting .REL file must be in MDS format (either version 1 or version 2). Refer to Chapter 27 for details on how Turbo Pascal passes parameters to external routines, and how external functions should pass values back.

Finally, you must tell the compiler what file to link to it, using the {$L} compiler directive. If you had assembled your assembly-language routines into a file called MYSTUFF.REL, then you'd put the following directive somewhere in your program:

```
{$L MyStuff.REL}
```

This directive can appear anywhere before the **begin** of the main body of your program, or the **begin** of the initialization section in your unit (if you're writing your own unit).

When you compile your program, Turbo Pascal goes to MYSTUFF.REL, copies the machine code into your application file, and creates the necessary links.

Inline Code and Traps

In addition to external assembly-language subroutines, Turbo Pascal also allows you to write internal machine-language code for your program. The key phrase here is *machine language*, since the actual inline code is written as numeric constants (preferably, though not necessarily, hexadecimal). The format for defining inline code is

```
‹proc/func declaration›; inline ‹integer constant(s)›;
```

The constants are of type *Integer*, not of type *LongInt*. If more than one constant is used, the constants are separated by commas. So, for example, you could write the following:

```
procedure TextFace(Face : Style);
inline $205F, $1010, $3F00, $A888;
```

This code would disassemble to the following 68000 code:

```
MOVEA.L   (A7)+,A0
MOVE.B    (A0),D0
MOVE.W    D0,(A7)-
DS.W      $A888     ; trap to ROM routine
```

The Mac Toolbox implements most of its routines as *traps*. A trap is a special instruction that causes the CPU to stop what it's doing and attend to the trap. On the 68000, for example, any instruction with the bit pattern $Axxx causes a trap. The 68000 then calls a special trap-handling routine, which decodes the rest of the instruction and decides what to do about it. On the Macintosh, that trap handler looks at the rest of the bits and calls the appropriate ROM or RAM routine before returning to the program where the trap occurred.

Confused? Just think of traps as do-it-yourself machine code instructions. If you'll look through the unit interface listings for, say, *QuickDraw*, you'll see that most of the procedures and functions are **inline** calls to traps. (The example above, in fact, was taken from *QuickDraw*.)

One last bit of information about **inline** calls. An **inline** procedure or function is not set up as a separate subroutine, so there is no JSR (jump to subroutine) to an **inline** routine. Instead, whenever a call to an **inline** routine occurs, the compiler sets everything up as if it were going to make a subroutine call (pushing

parameters and the return address on the stack). It then inserts the actual **inline** code right after that. Say, for example, that you wrote the following code:

```
if BFlag then
   TextFace([bold])  { use bold type }
else TextFace([]);    { use plain type }
```

At each call to *TextFace,* the compiler would generate the code to push the parameter and return address onto the stack, then insert the four words found above ($205F, $1010, $3F00, $A888).

Now that you've been introduced to units, you have two options as to where to go next. If you're interested in writing your own units, go on to Chapter 8. If, instead, you want to start writing Mac-style programs, skip to Chapter 9. However, you should read through Chapter 8 at some time.

CHAPTER 8

Writing Your Own Units

In Chapter 7, you learned how useful units could be. They provide an efficient way to organize groups of data structures and subroutines for use in different programs. In this chapter, you'll learn how to write your own units. You'll be shown the general structure of a unit and its interface and implementation portions, as well as initialization and compilation. Also included are a few program examples.

A Quick Review of Units

A unit is a collection of constants, data types, variables, procedures, and functions. Like a complete Pascal program, it can even have a "main body" that is called before your program starts and does whatever initialization is necessary. In short, it's a library of declarations that you can pull into your program and use. All the program elements in a unit are usually related to one another, so that a unit tends to solve a set of problems or offer a set of capabilities. When a program uses a unit, all its declarations become available, just as if they had been defined within the program itself.

A unit consists of two parts: interface and implementation. The interface is the actual collection of declarations that can be read by the program. This can include constants, data types, variables, and headers for procedures and functions. The implementation is where the bodies (the code) of the procedures and

functions declared in the interface actually reside. Additional constants and the like also can be declared and used within the implementation. These items, however, are not available for viewing by the program using the unit.

Let's talk in more detail about how a unit is laid out and what the different sections do.

A Unit's Structure

As mentioned above, a unit has a structure not unlike that of a program, but with some significant differences. Here's a unit, for example:

```
unit <identifier>(unit #);
interface
  uses <list of units>;  { optional }
  { public declarations }
implementation
  { private declarations }
  { procedures and functions }
begin
  { initialization code }
end.
```

The unit header starts with the reserved word **unit**, followed by the unit's name (an identifier), just as a program has a name. A unit number, in parentheses, appears between the unit name and the semicolon terminating the header. This number—a positive 16-bit integer constant—should be different from any other unit number that your programs might use.

The next item in a unit is the keyword **interface**. This signals the start of the interface section of the unit, that is, the section visible to any other units or programs that use this unit.

A unit can use other units by specifying them in a uses-clause. If present, the uses-clause appears right after the keyword **interface**. Note that the general rule of a uses-clause still applies: If a unit named in a uses-clause uses other units, those units must also be named in the uses-clause, and their names must appear in the list before that of the unit using them.

As with a program, if a unit does not include a {$U-} directive, the *PasInOut* and *PasConsole* units are automatically used by that unit. This further means that a program using that unit would also have to use *PasInOut* and *PasConsole*, even though they may not be required. In general, if a unit does not require any of the functions provided by *PasInOut* and *PasConsole*, you should place a {$U-} directive in the beginning of the unit.

Interface

A unit provides a set of capabilities through procedures and functions—with supporting constants, data types, and variables—but it hides how those capabilities are actually implemented. It does this by breaking the unit into two sections: the interface and the implementation.

The interface portion of a unit starts at the reserved word **interface**, which appears after the unit header, and it ends when the reserved word **implementation** is encountered. The interface determines what is "visible" to any program (or other unit) using that unit. In the unit interface, you can declare constants, data types, variables, procedures, and functions. As with a program, these can be arranged in any order, and sections can repeat themselves (for example, `type ... var ... <procs> ... const ... type ... const ... var`).

The procedures and functions that are visible to any program using the unit are declared here, but their actual bodies—that is, implementations—are found in the **implementation** section. If the procedure (or function) is **external,** that keyword should appear in the interface, and no redeclaration of the procedure need occur in the implementation. If the procedure (or function) is **inline**, the machine code (list of integer constants) should appear in the interface section, and no redeclaration of the procedure should occur in the implementation. **Forward** declarations are neither necessary nor allowed. The bodies of all the regular procedures and functions are held in the implementation section, after all the procedure and function headers have been listed in the interface section.

Implementation

The implementation section starts at the reserved word **implementation.** Everything declared in the interface portion is visible in the implementation: constants, types, variables, procedures, and functions. Furthermore, the implementation can have additional declarations of its own, although these are not visible to any programs using the unit.

If any procedures have been declared **external**, one or more {$L ⟨.REL file⟩} directive(s) should appear before the **begin** marking the initialization section. If there is no initialization section, then it can be anywhere before the final **end** of the unit.

The normal procedures and functions declared in the interface—those that are neither **external** nor **inline**—must reappear in the implementation. The procedure/function header that appears in the interface should not appear in full in the implementation. Instead, just type in the keyword (**procedure** or **function**), fol-

lowed by the routine's name (identifier). The routine should then contain all its local declarations (labels, constants, types, variables, and nested procedures and functions), followed by the main body of the routine itself. Say the following declarations appear in the interface of your unit:

```
procedure Swap(var V1,V2 : Integer);
function Max(V1,V2 : Integer) : Integer;
```

The implementation should look like this:

```
procedure Swap;
var Temp : Integer;
begin
   Temp := V1; V1 := V2; V2 := Temp
end; { of proc Swap }

function Max;
begin
  if V1 > V2 then
    Max := V1
  else Max := V2
end; { of func Max }
```

Routines local to the implementation (that is, not declared in the interface section) should have their complete procedure/function header intact.

Initialization

The entire implementation portion of the unit is normally bracketed within the reserved words **implementation** and **end**. However, if you put the reserved word **begin** before **end** with statements between the two, the resulting compound statement—looking very much like the main body of a program—becomes the initialization section of the unit. When a program using that unit is executed, the unit's initialization section is called before the program's main body is run. If the program uses more than one unit, each unit's initialization section is called (in the order specified in the program's **uses** statement) before the program's main body is executed.

The initialization section is where you initialize any data structures (variables) that the unit uses or makes available (through the interface) to the program using it. You can use it to open files for the program's later use. For example, the runtime unit *PasPrinter* uses its initialization section to make all the calls to open (for output) the text file PRINTER, which you can then use in your program's *Write* and *WriteLn* statements.

Compiling a Unit

You compile a unit using the same commands as when you compile a program. Normally, you'd compile a unit to disk to be able to use it with all your programs. However, if you have windows open for a unit and for a program that uses it, you can compile the unit to memory. The compiler always looks in memory before looking on disk, when searching for a unit named in a uses-clause.

When you compile a unit to disk, the resulting library file adopts its name from the unit header. For example, if your unit header is:

```
unit MyUnit(1);
```

then the library file created on the disk is called MYUNIT. As with a program, you can override that default and request a specific file name using the {*$O ⟨filename⟩*} directive. If you changed your unit to read:

```
unit MyUnit(1);
{$O MyLibrary}
```

then a compile to disk produces a library file named *MyLibrary.*

You may, in fact, compile several units to the same library file. Suppose you have two units and that both include a {$O MyLibrary} directive. Every time you compile one of them, the newly compiled unit replaces the older version in the library file.

The icon used for unit library files is different from the one used for compiled programs; it is an attaché-case (or briefcase), which represents something you can "carry" from program to program. Also, unlike a compiled program, if you double-click on a unit library file, the unit is not "executed." Instead, the UNITMOVER is launched, and the library file is "opened." Chapter 11 contains a complete explanation on using the UNITMOVER.

Using Your Units

Say you've written a unit called MYUNIT.PAS and compiled it to disk; the resulting code file is called MYUNIT. To use it in your program, you need to include two things: a {$U ⟨filename⟩} directive to tell the compiler where to look for the unit and a **uses** statement to tell the compiler that you're using that unit. Your program might look like this:

```
program MyProg;
{$U MyUnit}
uses MyUnit;
```

The unit name and the unit's code file name don't have to be the same. If you compile the unit with the directive {$O MYUNITS.LIB} or change the code file name to that under the FINDER, then the {$U} directive in the program would have to read {$U MYUNITS.LIB}.

Now, suppose you had compiled the units *MyFirst* and *MySecond* with the directive {$O *MyLibrary*}. To use all three units, you would have to use two {$U} directives, both of them appearing before the uses-clause:

```
program MyProg;
{$U MyUnit}
{$U MyLibrary}
uses MyUnit,MyFirst,MySecond;
```

Depending on your Macintosh system, there is a limit to the number of files you can specify with {$U} directives; it is at least ten for all systems, though.

The section at this end of this chapter, "UNITMOVER," explains how you can use this utility to simplify using units. Chapter 11 contains a complete explanation on using UNITMOVER.

An Example

OK, now let's write a small unit. We'll call it *IntLib* and put two simple integer routines, a procedure and a function, in it. Here's the unit:

```
unit IntLib(1);
{$U-}
interface
  procedure Swap(var I,J : Integer);
  function  Max(I,J : Integer) : Integer;
implementation
  procedure Swap;
  var
    Temp  : Integer;
  begin
    Temp := I; I := J; J := Temp
  end;   { of proc Swap }
  function Max;
  begin
    if I > J then
      Max := I
    else Max := J
  end;    { of func Max }
end.   { of unit IntLib }
```

Type this in, save it as the file INTLIB.PAS, then compile it to disk. The resulting unit code file is INTLIB.

Following is a program that uses this unit:

```
program IntTest;
{$U IntLib}    { where to look for IntLib }
uses IntLib;
var
  A,B          : Integer;
begin
  Write('Enter two Integer values:  ');
  ReadLn(A,B);
  Swap(A,B);
  WriteLn('The max is ',Max(A,B));
  ReadLn;
end. { of program IntTest }
```

Units and Large Programs

Up until now, you've probably thought of units only as libraries: collections of useful routines to be shared by several programs. However, units can do something just as important: break a large program up into modules. In fact, units give Turbo Pascal many of the advantages of Modula-2 and other modular languages, with few of their disadvantages.

Two other aspects of Turbo Pascal make this function feasible: its tremendous speed in compiling and linking; and its ability to manage several code files simultaneously, such as a program and several units.

Typically, a large program is divided into units that group procedures by their function. For instance, an editor application could be divided into initialization, printing, reading and writing files, formatting, and so on. Also, there would be a "global" unit—one used by all other units, as well as the main program—that defines global constants, data types, variables, procedures, and functions.

The compiled version of each unit is stored in a unit library file. Each unit references this file twice. First, it should reference it in a {$O ⟨filename⟩} directive, since the compiled version of the unit should be put in that file. Second, that same file name would need to be the first {$U ⟨filename⟩} directive, so that it could get the global declarations from the main unit (as well as any other unit it might happen to use).

The skeleton program might look like this:

```
program Editor;
{$O My Editor}      { output file for application }
{$T APPLMYED}       { application type; creator ID = MYED }
{$R Editor.Rsrc}    { resource file for this application }
{$U Editor.Lib}     { library file with application's units }

{$B+}               { set bundle bit }
{$U-}               { disable automatic use of runtime units }
```

```
uses
  MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint,
  EditGlobals,
  EditInit,
  EditPrint,
  EditRead,EditWrite,
  EditFormat;

  { program's declarations, procedures, and functions }

begin
  { main program }
end. { of program Editor }
```

One of the units—say, *EditPrint*—might look like this:

```
unit EditPrint(3);
{$O Editor.Lib}     { output file = library file }
{$U Editor.Lib}     { but uses units in libarary file as well }
{$U-}               { disable use of runtime units }

interface

uses
  MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint,
  EditGlobals;

  { the rest of the interface }

implementation

  { implementation of the unit }

end.  { of unit EditPrint }
```

A further refinement involves segmentation. Turbo Pascal allows you to break your program up into *segments*, that is, chunks of machine code, each one no larger than 32K bytes. The {$S+} directive instructs Turbo Pascal to create a segmented code file, while the {$S ⟨segname⟩} directive specifies into which segment a unit or a collection of subprograms (procedures and functions) will go.

In the modified example below, the units are grouped into segments by their function. The Mac units, as well as *EditGlobals*, go into the "blank" (or main) segment, so that they will always be resident with the main body of the program. The initialization unit (*EditInit*) resides in a segment by itself, so that it can be disposed of once it has done its job. Likewise, the printing unit (*EditPrint*) is in its own segment, so that it takes up memory only when printing is going on. The modified section of the application looks like this:

```
{$B+}
{$S+}          { enable segmentation }
{$U-}
```

```
uses
  {$S          } MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,
                 MacPrint,EditGlobals,
  {$S Init     } EditInit,
  {$S Print    } EditPrint,
  {$S InOut    } EditRead,EditWrite,
  {$S Format   } EditFormat;
```

Segments are loaded automatically by calling a procedure or function within that segment. To unload a segment, you call the OS routine *UnloadSeg*, passing to it the address of any procedure or function within that segment. For example, if *EditInit* contained the procedure *Initialize*, then you would call

```
UnloadSeg(@Initialize);
```

You need to call this from somewhere **outside** of the segment containing *Initialize*. The main body of the application would probably be the safest place.

UNITMOVER

You don't have to use a {$U⟨filename⟩} directive when using the Pascal Run-time Support units or the Macintosh Interface units. That's because all those units have been moved into the actual Turbo Pascal compiler file. When you compile, those units are always "visible" and able to be used if you want.

Suppose you have a well-designed and thoroughly debugged unit that you want to add to the standard units, so that you don't need a {$U} directive each time you want to use it. Is there any way to move it into the Turbo Pascal application file? Yes, by using the UNITMOVER utility.

You can also use the UNITMOVER to remove units from the Turbo Pascal file, reducing its size and the amount of memory it takes up when loaded.

Summary

As you've seen, it's really quite simple to write your own units. A well-designed, well-implemented unit simplifies program development, since you just solve the problems once and not for each new program. Furthermore, once you've designed a useful unit, you can release it into the public domain without having to disclose your source code. Others can benefit from your expertise, but you won't have to divulge your methods.

C H A P T E R 9

Writing Your Own Macintosh Applications

In previous chapters, you learned how to write a standard Pascal program. Now you can move into the meat of this manual: Macintosh programming. We'll now show you parts of sample programs—contained in the EXAMPLES folder in the Utilities & Sample Programs disk—that illustrate general Macintosh programming techniques.

You're not going to learn everything in this chapter. *Inside Macintosh*, the book on the Toolbox and the operating system, is over 1,200 pages. *Macintosh Revealed*, the two-volume work on how to program the Mac, is some 1,100 pages. Despite this chapter's comparative shortness, however, you should learn the basic skills necessary to program the Macintosh with Turbo Pascal.

The Demo Program

The EXAMPLES folder contains several sample programs. The following sections explain the parts of these programs that exemplify Macintosh programming techniques. One example program, MYDEMO.PAS, uses most of the techniques mentioned in this chapter. It is a fairly simple Macintosh application that uses menus, windows, dialog boxes, and graphics. Let's start by making it run, so that as you learn about it, you'll have in mind what the finished program looks like.

MYDEMO uses a *resource file* to define its menus, windows, and icon. The source text describing the resources is located in the file MYDEMO.R (also in the EXAMPLES folder). You can think of the text in MYDEMO.R as being like an uncompiled Pascal program: It needs to be run through a "compiler" to be useful. That "compiler" is RMAKER (for Resource Maker), so first run RMAKER by double-clicking it (or use the Transfer menu, if you're in Turbo Pascal). (RMAKER is described in detail in Chapter 12.)

Once you're in RMAKER, you'll get the standard file selector box; it only shows files ending with .R. Find MYDEMO.R, select it (point at it with the mouse, then click), and then click on the Open button. RMAKER then builds a resource file (MYDEMO.RSRC) based on the descriptions in MYDEMO.R. When it's done, exit by clicking on the Quit button near the lower left corner of the window. (You can also select the Other command in the Transfer menu, then choose Turbo when the file selector comes up.)

Enter Turbo Pascal and open MYDEMO.PAS: Double-click directly on MYDEMO.PAS or double-click on the Turbo Pascal icon, close the Untitled window that's automatically opened, select the Open command from the File menu, find MYDEMO.PAS in the file selector list, and select it. Now, compile and run MYDEMO: Select the Run command from the Compile menu (or type ⌘R). The compilation takes just a few seconds; MYDEMO runs, and the window and the menu bar (with four menus) appears.

Play around with MYDEMO for a while. Note that you can execute most commands with keyboard equivalents. You can resize the window, drag it around the screen, or close it (which will cause MYDEMO to halt execution and return you to Turbo Pascal). You can run desk accessories using the Apple menu, or bring up the About MYDEMO... dialog box.

When you're done, exit MYDEMO: Either click the Close box in the upper left corner of the window, select the Quit command from the Edit menu, or type ⌘Q. Once you're back in Turbo Pascal, select the To Disk command from the Compile menu (or type ⌘E). Now exit Turbo Pascal. You should see the application icon for MYDEMO. You can double-click the icon to run the program again.

Let's now look at how Mac applications are designed, by studying the parts that make up a Mac program.

Back in Chapter 6, the concept of event-driven programming was introduced. As you saw, the basic structure of most Macintosh applications is nearly identical, with a main body that looks something like this:

```
begin { main body of program }
   Initialize;                                        { set everything up }
   repeat                                           { keep doing the following }
     SystemTask;                                     { update desk accessories }
     CursorAdjust;                                      { update which cursor }
     if GetNextEvent(everyEvent,theEvent) then       { if there's an event... }
       HandleEvent(theEvent)                          {     ...then handle it }
   until Finished;                                     { until user is done }
   Cleanup                                            { clean everything up }
end. { of program }
```

The program sets up once with the user-defined routine *Initialize*. It then enters a loop that continues until some condition (such as the user selecting Quit in a menu) causes it to set the boolean flag Finished to true.

Within that loop, it performs two major tasks. First, it calls *SystemTask* (a Toolbox routine), which allows the Mac operating system to update any desk accessories that might be in use. Second, it calls *GetNextEvent* (another Toolbox routine) to see if any events have occurred. If any have, the highest priority event is returned in the data structure *theEvent*. The program then passes the event to *HandleEvent*, which is a user-defined routine that handles all the different events that might occur. Such events include key presses, selection of menu items, mouse clicks, and windows being opened, closed, uncovered, or resized. When the program is ready to terminate, it calls the user-defined routine *CleanUp*.

Event-driven programming assumes that most of your commands will usually be available, so you need to anticipate how to handle them. That won't be true *all* the time; for example, a program may have editing capabilities, but any editing commands would be active and make sense only when there is a window open for text editing. It is difficult to select, cut, and paste text when there's no text to cut and no window open for the text to be pasted to.

Event-driven programming takes some getting used to, but once you understand how it works and have seen examples of it, it is easy to apply to different situations. With Macintosh applications, the format is so standard that you can move from program to program and see almost identical code in the main bodies and immediate supporting routines.

A Note on Programming Style

Turbo Pascal programs should be written to be as easy to read and modify as possible. In the example programs on the disk, almost every line of code is commented on. This may seem excessive, but the often cryptic nature of Macintosh system calls makes it helpful to understand exactly what each call is doing, and why. Too often, Macintosh sample programs assume too much understanding on the part of the reader.

Emphasis has been placed on organizing programs into small, manageable chunks of code. Mac hackers sometimes enjoy stuffing all the event handling into a few gigantic **case** statements, usually embedded in loops and **if/then/else** statements, with so much indentation and nesting that the program is unreadable.

Program Organization

Although Macintosh applications tend to have the same structure, you might not notice it at first, given the way some of them are coded. Here is a skeleton structure for a sample Macintosh program:

```
program SampleProgram;
global declarations
utility procedures and functions
menu-driven procedures and functions
event-handling procedures and functions
initialization and cleanup procedures
main body of program
```

You've already seen what the main body of the program looks like; let's concentrate on another aspect, event handling.

To understand Macintosh applications, you must understand how to handle events. Following is a sample of the *HandleEvent* procedure in a typical program:

```
procedure HandleEvent(theEvent : EventRecord);
begin
  case theEvent.What of
    mouseDown     : DoMouseDown(theEvent);     { mouse button pushed }
    keyDown       : DoKeyPress(theEvent);        { key pressed down }
    autoKey       : DoKeyPress(theEvent);           { key held down }
    updateEvt     : DoUpdate(theEvent);       { window need updating }
    activateEvt   : DoActivate(theEvent)     { window made act/inact }
  end
end; { of proc HandleEvent }
```

When an event occurs, the operating system creates an event record and sticks it in a queue, ready for you to handle. To see if there's one waiting, you call *GetNextEvent*, a boolean function that returns true if there's an event there for you. You give it a mask of the events you're interested in; you can use the predefined mask *EveryEvent* to look at all events. This event is passed to *HandleEvent*, which takes care of it.

The key to all this is the predefined data type *EventRecord*, which is what *GetNextEvent* passes back to you (through the parameter list). The data structure looks like this:

```
type
  EventRecord =
    record
      What          : Integer;            { event code }
      Message       : LongInt;          { event message }
      When          : LongInt;    { ticks since startup }
      Where         : Point;          { mouse location }
      Modifiers     : Integer         { modifier flags }
    end;
```

Here's what each of the fields mean:

- *What* tells you what type of event has just occurred. There are a total of 16 predefined events, including 4 set aside for application use. Some common events include mouse down, key pressed, key repeated, window activate/deactivate, window update, and disk inserted.
- *Message* contains information specific to the event that has occurred. For keyboard events, it has both the ASCII and keyboard codes in it; for window events, it has a pointer to the window involved; for disk events, it has the drive number and File Manager result code.

- *When* is the time that the event occurred. This is given in the number of ticks (1 tick equals 1/60th of a second) that have elapsed since you booted the Mac.
- *Where* tells you the mouse's location, in global coordinates, when the event happened. The data structure *Point* is a variant record whose components can be accessed either as *V* (Y coordinate) and *H* (X coordinate) or as *VH[0]* and *VH[1]*.
- *Modifiers* offers yet more specific information, when appropriate. Each piece of information is flagged with a single bit, though not all bits are currently in use. Items include the status of the mouse button, Command key, Shift key, Option key, and Caps Lock key, and whether a window is being activated or deactivated.

All this information gets passed to *HandleEvent* via the variable *theEvent* (of type *EventRecord*). *HandleEvent* is just a **case** statement using the *What* field in *theEvent* to determine which of the four procedures to call.

There are additional events you could check for (such as *mouseUp*, *keyUp*, and so on), but these are sufficient for most programs. Here's a brief explanation of the types of events mentioned and what the handling routines will have to do:

- *mouseDown*: The user has moved the mouse to some point and pushed the button. *DoMouseDown* determines where the mouse currently is (in a menu, in a window, and so forth) and takes appropriate action (the following section explains this further).
- *keyDown*: The user has pressed a key. All this program does is check to see if a command-key combination was pressed; if so, it checks if the key is a menu command and takes appropriate action.
- *autoKey*: The user is holding a key down. This program takes the same action as for *keyDown*.
- *updateEvt*: A window has to be updated (redrawn) because of some event (resizing or removing a blocking window, for example).
- *activateEvt*: A window has just been activated (brought to the front and highlighted) or deactivated (another window was activated).

As you can see, there can be a lot of background activity going on while your program is running. Fortunately, your program doesn't have to keep looking all over the place to figure out what to do because the operating system keeps it informed on what has happened, feeding it each event as the event occurs and is processed. Your program then decodes the event and uses a **case** statement to call the procedure best equipped to handle it. As you'll see, further decoding often takes place to pin down exactly what the event was.

The greatest variety of events comes from clicking the mouse's button. The routine *DoMouseDown* determines which window (if any) the mouse was in when the clicking took place and where exactly it happened. Like *HandleEvent*, *DoMouseDown* is mostly a **case** statement:

```
procedure DoMouseDown(theEvent:EventRecord);
var
   Location : Integer;
   theWindow : WindowPtr;
   MLoc : Point; WLoc : Integer;
   begin
      MLoc := theEvent.Where;                          { get mouse position }
      WLoc := FindWindow(MLoc,theWindow);     { get window,window location }

  case WLoc of                                          { handle locations }
    InMenuBar   : HandleMenu(MenuSelect(MLoc));          { in the menu     }
    InContent   : HandleClick(theWindow,MLoc);         { inside the window }
    InGoAway    : HandleGoAway(theWindow,MLoc);       { in the go away box }
    InGrow      : HandleGrow(theWindow,MLoc);           { in the grow box }
    InDrag      : DragWindow(theWindow,MLoc,DragArea); { in the drag bar }
    InSysWindow : SystemClick(theEvent,theWindow)         { in a DA window }
  end
end; { of proc DoMouseDown }
```

DoMouseDown now has a number of sub-events to process, namely:

- *InMenuBar*: The user has selected a command from one of the menus in the menu bar. Decode the command and take appropriate action.
- *InContent*: The user has clicked the mouse down in the text or contents area of the window. If the window is the currently active one, take whatever action is appropriate (if any); otherwise, make it the active window.
- *InGoAway*: The user has clicked the Close box in the window. If this is the active window, then you should close the window and possibly exit the program as well. If it's not the active window, make it the active window.
- *InGrow*: The user has clicked the mouse in the grow box in the lower right corner of the window. Call some standard Toolbox routines (GrowWindow, SizeWindow, InvalRect) to let the user change the size and redraw the now-changed window.
- *InDrag*: The user has clicked the mouse down in the drag bar at the top of the window. Call the Toolbox routine *DragWindow*, which handles everything for you.

- *InSysWindow*: The user has clicked the mouse in a desk accessory window. Call the Toolbox routine *SystemClick*, which passes the event information on to the desk accessory (which then handles things itself).

Two of these events (*InDrag* and *InSysWindow*) are handled by a simple call to a Toolbox routine. The other four (*InMenuBar*, *InContent*, *InGoAway*, and *InGrow*) result in calls to other procedures (*HandleMenu*, *HandleClick*, *HandleGoAway*, and *HandleGrow*, respectively). Let's talk about each of these.

Menu Commands

The procedure *HandleMenu* decodes the mouse position and figures out which menu and which item in that menu were selected. It uses a **case** statement to select the action for the appropriate menu; the menu value is the ID assigned when the menu is created (more on this in the section on initialization near the end of this chapter). The commands in a menu are numbered from the top down, with the first command having a value of 1. The action itself is usually a second **case** statement, based on the menu item (or command) number. Here's a sample *HandleMenu* routine:

```
procedure HandleMenu(MenuInfo : LongInt);
var
  Menu          : Integer;          { menu number that was selected }
  Item          : Integer;         { item in menu that was selected }
  B             : Boolean;         { dummy flag for SystemEdit call }
begin
  if MenuInfo <> 0 then
  begin
    ClearWindow(MainPtr);              { we're clearing the window }
    PenNormal;                        { set the pen back to normal }
    HideCursor;                             { turn off the cursor }
    Menu := HiWord(MenuInfo);   { find which menu the command is in }
    Item := LoWord(MenuInfo);             { get the command number }
    case Menu of                                { and carry it out }
      ApplMenu  : if Item = 1 then
                    DoAbout           { bring up "About..." window }
                    else DoDeskAcc(Item);    { start desk accessory }
      FileMenu  : case Item of
                    1 : NewFile;          { start a new file window }
                    2 : OpenFile;           { open an existing file }
                    3 : SaveFile;               { save file to disk }
                    4 : Quit;                    { quit the program }
                  end;
      EditMenu  : case Item of
                    1 : Undo;                 { undo last operation }
                    3 : Cut;                    { cut to clipboard }
                    4 : Copy;                  { copy to clipboard }
                    5 : Paste;               { paste from clipboard }
                    6 : Clear;                   { clear clipboard }
                    8 : ShowClipBoard;      { show clipboard window }
                  end
      { other application menu case statments go here }
    end; { case of Menu }
```

```
    HiliteMenu(0);                              { reset menu bar }
    if Menu = IOMenu then
     UpdateMenu;                          { make any changes needed }
    ShowCursor                            { turn the cursor back on }
  end
end; { of proc HandleMenu }
```

When an item in a menu is selected, the name of that menu (in the menu bar at the top of the screen) is highlighted, that is, inverted to white-on-black (called inverse or reverse video). When you are done processing the menu command, restore the menu bar to normal: call *HiliteMenu(0)*, which is at the bottom of *HandleMenu*. Another procedure is called before you can leave *HandleMenu*: *UpdateMenu*, a local procedure that tests to see if certain items need to be enabled or disabled.

Notice that in the *EditMenu*, item numbers 2 and 7 are not used. These are used by division lines and are not selectable as menu items.

Enabling and Disabling Menu Items Just as the line separators in menus are disabled, you have the ability to disable (and enable) specific items in specific menus. For example, in the standard Apple edit menu, the paste item is not enabled until something is placed on the Clipboard. To enable and disable menu items, call the standard Macintosh procedures *EnableItem* and *DisableItem*.

Here's a sample *UpdateMenu*, with the *SetItemState* procedure it calls:

```
procedure SetItemState(Mndx,Indx : Integer; Flag : Boolean);
begin
   if Flag then                              { test enable/disable flag }
      EnableItem(MenuList[Mndx],Indx)        { enable menu item         }
    else DisableItem(MenuList[Mndx],Indx)    { disable menu item        }
end; { of proc SetItemState }

procedure UpdateMenu;
begin
   SetItemState(EditMenu,1,True);           { edit Cut is always active  }
   SetItemState(EditMenu,2,True);           { edit Copy is always active }
   SetItemState(EditMenu,3,ClipBoardFull)   { edit Paste depends on flag }
end; { of proc UpdateMenu }
```

UpdateMenu calls the procedure *SetItemState*, passing the menu index, the item number, and a *Boolean* flag. *SetItemState*, in turn, decides whether to enable or disable the item based on the *Boolean* value passed to it. It uses the menu index passed to it to choose the particular menu handle from *MenuList* (an array of type *MenuHandle*), then uses that and the item number to call either *EnableItem* or *DisableItem*—Toolbox routines that do just what their names suggest.

In the *UpdateMenu* procedure, the Edit menu command Paste is enabled or disabled depending on the *Boolean* flag *ClipBoardFull*.

Check Marks You can have a menu item that uses a check mark and a global flag to keep track of a particular option. When this option is selected, a check mark is placed by the item in the menu; when it is de-selected, the check mark disappears.

The following general-purpose routine called *ToggleFlag* provides check mark operations:

```
procedure ToggleFlag(var Flag : Boolean; Mndx,Indx : Integer);
var
  Ch            : Char;
begin
  Flag := not Flag;                          { toggle flag (for you) }
  if Flag then                                 { if flag is true... }
    Ch := Chr(CheckMark)                     { then check item in menu }
  else Ch := Chr(NoMark);                   { else clear any checkmark }
  SetItemMark(MenuList[Mndx],Indx,Ch)       { put char by item in menu }
end; { of proc ToggleFlag }
```

ToggleFlag takes three parameters: a *Boolean* variable (*Flag*), the menu number (*Mndx*), and the item number (*Indx*). *ToggleFlag* then toggles *Flag*; that is, if *Flag* was set to true, it is set to false, and vice versa. Having done that, it sees if *Flag* is now true or false: If *Flag* is true, it places a check mark at item *Indx* in menu *Mndx*; if *Flag* is false, it places a blank there, erasing any existing check marks. The standard Macintosh procedure *SetItemMark* puts the character next to the menu item.

The "About..." Box Most menu commands call other procedures that take the desired action. The first menu (*ApplMenu*) is the Apple menu; it appears at the far left of the menu bar as the Apple logo. When you pull down this menu, the first item you see is `About <the program name>`. If this is selected, the procedure *DoAbout* is called. Here's a sample procedure definition:

```
procedure DoAbout;
var
  theItem       : Integer;
  AboutPtr      : DialogPtr;
  S1,S2,S3,S4   : Str255;
begin
  SetCursor(CursList[myCursor]^^);                    { set to my cursor }
  ShowCursor;                                       { and turn it back on }
  S1 := 'This is a Sample Program';           { set up four strings for the }
  S2 := ' brought to you by the ';                    { About dialog box }
  S3 := '   friendly folks at';
  S4 := ' BORLAND INTERNATIONAL';
  ParamText(S1,S2,S3,S4);                          { set up as parameter text }
  AboutPtr := GetNewDialog(AboutID,NIL,Pointer(-1));     { get a dialog box }
  ModalDialog(NIL,theItem);                    { put dialog box up; get result }
  DisposDialog(AboutPtr);                            { get rid of dialog box }
  SetCursor(Arrow)
end; { of proc DoAbout }
```

DoAbout is a small routine that displays a box with some information about the program, then waits for you to click on the OK button. Once that's done, it removes the box and lets the program continue executing. It uses a predefined dialog template in a resource file to create the display box.

This dialog is known as *modal* dialog, because the entire program stops until you do something. The Toolbox routine *ParamText* provides a means of substituting text strings in parameter items in subsequent dialog or alert boxes. The Toolbox routine *GetNewDialog* uses the *AboutID* to retrieve a predefined dialog box from a resource file. This allows you to have standard dialog boxes whose items have similar types, locations, and contents. The Toolbox routine *ModalDialog* sets up the dialog box for you; the routine *DisposDialog* then gets rid of it.

Handling Desk Accessories The Apple menu also allows you to bring up desk accessories from within your program. The idea behind desk accessories is that you should be able to access them while you're in the middle of an application. Chapter 10 discusses desk accessories and how to write your own; for now, let's see how to make sure your program supports them.

First, you must be sure to set up the Apple menu. This should be the first (leftmost) menu. To define the Apple logo as the title, use the character with hex value 14; in the resource file, you can do this by giving the logo the title \14.

Here's what a resource file would look like:

```
TYPE MENU
   ,1000          ; resource ID number
\14               ; Apple symbol for title
About My Demo...  ; top item--for 'About' box
(-                ; line separator
```

In the initialization procedure, after reading the menu data in (using *GetMenu*), you need to add the desk accessories to the menu. You do this by calling `AddResMenu(<menuhandle>,'DRVR')`, where ⟨*menuhandle*⟩ is the handle of your first menu.

Within your main loop, you need to call *SystemTask* to give the operating system a chance to pass control to any desk accessories that might be executing. If you get the mouse event *inSysWindow*, call *SystemClick* in order to let the desk accessory handle it; likewise, the mouse event *inGoAway* must call *CloseDeskAcc* if the window being closed is a desk accessory.

If you select one of the desk accessories from the Apple menu, you need to start it up. The procedure *DoDeskAcc*—which *HandleMenu* calls for any item from the Apple menu other than the first one—does this by calling *GetItem* to get the name of the desk accessory selected, then calling *OpenDeskAcc* with that name.

You should support the standard Edit commands—Undo, Cut, Copy, Paste, and Clear—since a number of desk accessories rely upon these. Set them up as items 1, 3, 4, 5, and 6, respectively, in a menu. When one is selected, pass its value (less one) to the OS function *SystemEdit*, which will return true if that edit command was for a desk accessory. If *SystemEdit* returns false, then handle the command in the manner appropriate for your application.

Clicking Windows

The following procedure handles clicking the mouse. The code checks to see if its own window is being clicked and if that window is the active window. If it isn't, then it makes its window active:

```
procedure HandleClick(WPtr : WindowPtr; MLoc : Point);
begin
  if WPtr = MainPtr then                    { if this is our window... }
    if WPtr <> FrontWindow                  { and it's not in front... }
      then SelectWindow(WPtr)                 { ...then make it active }
end; { of proc HandleClick }
```

For other applications, though, more can (and needs to) be done. Some programs have a picture, layout, or graph of some kind in the window. In that case, you need to look at exactly where the mouse was clicked and determine what action to take. For example, pointing at a section of the picture could bring up a dialog box telling the user what that section is and allowing the user to modify some data related to it.

For a text-editing application, you click in the window in order to move the cursor to a different spot for text entry. In that case, you need to make the necessary calls to relocate the I-beam (text) cursor and to make sure that any text typed after that is inserted at the proper spot in the text record.

The Close Box

A window can have a Close (or go-away) box in its upper left corner. The user closes windows by pointing at the box with the mouse, then clicking once. On most text editors, including Turbo Pascal, the window closes, but the application continues to execute. In the following procedure, the flag *Finished* is set to true when that happens, leaving a procedure called *CleanUp* to actually close the window:

```
procedure HandleGoAway(WPtr : WindowPtr; MLoc : Point);
var
  WPeek             : WindowPeek;          { for looking at windows }
begin
  if WPtr = FrontWindow then             { if it's the active window }
begin
    WPeek := WindowPeek(WPtr);                 { peek at the window }
    if TrackGoAway(WPtr,MLoc) then         { and the box is clicked }
```

```
begin
      if WPeek^.WindowKind = userKind then     { if it's our window }
        Finished := true                        { then time to stop }
      else CloseDeskAcc(WPeek^.WindowKind)     { else close DeskAcc }
    end
  end
  else SelectWindow(WPtr)                     { else make it active }
end; { of proc HandleGoAway }
```

As you can see, *HandleGoAway* doesn't act immediately upon the Close box being clicked. Instead, it calls the Toolbox function *TrackGoAway*, which returns a value of true or false. *TrackGoAway* allows the user to have a change of mind; by moving the mouse away from the Close box without releasing the mouse button, the user cancels the close request.

This routine also handles closing desk accessories. It does this by checking what kind of window is open. During initialization, the main window was set to type *userKind (=8)*, while the desk accessories use their reference numbers to identify their windows. Indeed, that's how the desk accessory is closed: by using the *WindowKind* field as the reference number to identify which desk accessory to close.

The Grow Box

A window also can have a Grow box in the lower right corner. The user can point the mouse here, hold the button down, and resize the window by dragging the mouse around. More accurately, resizing takes place when the program calls *GrowWindow*, and what actually gets moved around is a dotted outline of the window. *GrowWindow* then returns the new width and height, with the position of the upper left corner being the fixed point of reference. The window is resized by calling *SizeWindow* with the new width and height. *InvalRect* is then called to mark any portions of the window that might no longer be valid because of the resizing. Here's a sample routine:

```
procedure HandleGrow(WPtr : WindowPtr; MLoc : Point);
type
  GrowRec =
    record
      case Integer of
        0 : (Result       : LongInt);
        1 : (Height,Width : Integer)
    end;
var
  GrowInfo : GrowRec;
begin
  if WPtr = MainPtr then                      { if it's our window }
    with GrowInfo do
    begin
```

```
    Result := GrowWindow(WPtr,MLoc,GrowArea);   { get amt of growth }
    SizeWindow(WPtr,Width,Height,true);            { resize window }
    InvalRect(WPtr^.portRect)                   { set up for update }
  end
end; { of proc HandleGrow }
```

This approach is a lazy one; the entire window is marked for updating. In your own applications, you can (and may want to) mark just those sections that have changed and need to be redrawn. In either case, the system issues update events (*updateEvt*) as needed to redraw things.

You may notice that one of the parameters to *GrowWindow* is the rectangle *GrowArea*. This defines the bounds of growth, that is, the minimum and maximum width and height for the window. *GrowArea* is initialized in the procedure *Initialize*. The minimum size is set there (rather arbitrarily) to 50 pixels wide by 20 pixels high. The maximum size is based on the screen dimensions, which are not assumed but instead are copied from *Screenbits.Bounds* and adjusted inwards slightly.

The Drag Bar

A window can have a drag bar across the top. The user can point at the drag bar, hold the mouse down, and move the window around the screen. This is an easy event to handle; just call the Toolbox routine *DragWindow*. If any update events are required—for example, if the window were partially off the screen and you just dragged it all the way back on—the system issues the necessary update events.

Note that *DragWindow*, like *GrowWindow*, gets passed a bounding rectangle called *DragArea*. It determines how far off the screen you can move the window. The idea, of course, is to avoid moving the window off the screen in such a manner as to prevent you from moving it back in. As with *GrowWindow*, *DragWindow* is set using the values in *Screenbits.Bounds*.

Handling Keyboard Events

After this discussion of mouse events, keyboard events will seem relatively simple and straightforward. There are three keyboard events: *keyDown* (a key is pressed); *keyUp* (a key is released); and *autoKey* (a key is held down long enough for it to start automatically repeating). The second one is only of interest in special cases, and you can often handle the third the same as the first. So you only have to worry about one event at this point: A key (or combination of keys) has been pressed.

For a regular application, a key press signals one of three things. First, if text entry is active, it means that a text record is being modified, that is, you're typing text in. Second, your program may interpret certain key combinations as special commands. Third, the key press may be the command-key (⌘) equivalent of a menu selection.

If you pull down most Macintosh application menus (except for the Apple menu), you'll see the command-key equivalents beside the items. To invoke a given command, hold the command (⌘) key down and press the appropriate letter. Here's the routine to handle it:

```
procedure DoKeypress(theEvent : EventRecord);
var
  KeyCh             : Char;
begin
  if (theEvent.modifiers and cmdKey) <> 0 then          { menu key command }
  begin
    KeyCh := Chr(theEvent.Message and charCodeMask);    { decode character }
    HandleMenu(MenuKey(KeyCh))                          { get menu and item }
  end
  else SysBeep(1)                                           { do something }
end; { of proc DoKeypress }
```

The *Modifiers* field in *theEvent* includes a bit that is set to 1 if the command key was held down when the event happened. You can test for that bit with a predefined mask, *cmdKey*. If you pass the character to the Toolbox function *MenuKey*, it will return a *LongInt* value containing the menu and item numbers. Pass these numbers to your menu-handling routine, which will split them up into two integers to separate the menu and item numbers.

Handling Update Events

The Macintosh keeps track of a lot of things for you. For one, it tells you when some portion of a window needs to be redrawn, because of resizing or removing a covering window. This is known as an update event (*updateEvt*), and it requires special handling. For one thing, you need to be able to redraw your entire window (or some portion thereof) at any time. This isn't that difficult for a text-editing window, since the text is stored off in memory and is written to the window as needed. However, it's a little trickier for a window with graphics. You either have a procedure (or set of procedures) that can recreate what you have on the screen, or you need to write to a bitmap that's in memory and copy it to the window as needed.

To handle an update event, follow a given sequence. Here's an example:

```
procedure DoUpdate(theEvent : EventRecord);
var
  SavePort,theWindow    : WindowPtr;
begin
  theWindow := WindowPtr(theEvent.Message);    { find which window }
  if theWindow = MainPtr then                   { only update ours }
  begin
    SetCursor(CursList[watchCursor]^^);       { set cursor to watch }
    GetPort(SavePort);                       { save current grafport }
    SetPort(theWindow);                       { set as current port }
    BeginUpdate(theWindow);                 { signal start of update }
  { and here's the update stuff! }
    ClearWindow(theWindow);                        { do update stuff }
  { now, back to our program...}
    EndUpdate(theWindow);                    { signal end of update }
    SetPort(SavePort);                          { restore grafport }
    SetCursor(Arrow)                             { restore cursor }
  end
end; { of proc DoUpdate }
```

This saves the current *grafport* into *SavePort* and makes *theWindow* the current port so that you can write to it. *BeginUpdate* limits all output to the section of *theWindow* that needs updating. You then do whatever redrawing is needed. When you're done, *EndUpdate* lifts those limits, and *SetPort(SavePort)* restores the old *grafport*.

Handling Activate Events

On the Macintosh desktop, only one window can be active at any one time. This doesn't mean that changes can't occur in other, inactive windows; it just means that, if there is more than one window on the screen, the highlighted front window is considered the active window. The Macintosh interface states that clicking on a window makes it active (and all others inactive); other processes (including a direct call to *SelectWindow*) can also make a window active. Here's an example:

```
procedure DoActivate(theEvent : EventRecord);
var
  I         : Integer;
  AFlag     : Boolean;
  theWindow : WindowPtr;
begin
  with theEvent do
  begin
    theWindow := WindowPtr(Message);              { get the window }
    AFlag := Odd(Modifiers);             { get activate/deactivate }
    if AFlag then
    begin                                  { if it's activated... }
      SetPort(theWindow);                     { make it the port }
      FrontWindow := theWindow;              { know it's in front }
```

```
      DrawGrowIcon(theWindow);                    { set size box }
    end
    else
    begin
      SetPort(ScreenPort);                  { else reassign port }
      if theWindow = FrontWindow              { if it's in front }
        then FrontWindow := NIL             { ...then forget that }
    end;
    if theWindow = MainPtr then
    begin                                    { if it's our window }
      SetItemState(EM,1,not AFlag);            { update edit cmds }
      for I := 3 to 6 do
        SetItemState(EM,I,not AFlag);
      SetItemState(EM,8,AFlag);             { update Quit command }
      for I := PM to IM do                   { update other menus }
        SetItemState(I,0,AFlag);
      DrawMenuBar                               { update menu bar }
    end
  end
end; { of proc DoActivate }
```

You test for activate/deactivate using the lowest bit on the *Modifiers* field of *theEvent*. If the window is being activated, you need to call *SetPort* to make it the current *grafport*; if it's being deactivated, you need to do something else, depending on your application. *SetPort* is called with the variable *ScreenPort*, which earlier (in *Initialize*) had been set to the entire screen with a call to *GetWMgrPort*. Whether the window is being activated or deactivated, a call is made to *DrawGrowIcon*.

Handling Other Events

There are a few other events that can occur. The **disk inserted event** (*diskEvt*) can usually be ignored; the only time it's important is during file selection, and, in that case, the Standard File Package reacts automatically. The **network event** (*networkEvt*) has to do with getting a packet via *AppleTalk*; see the appropriate documentation in *Inside AppleTalk*, as well as Chapter 10 in Volume 2 of *Inside Macintosh*, for more details. Likewise, the nature of a **driver event** (*driverEvt*) is dependent upon the device driver issuing it.

The last four events are application-defined events: *app1Evt*, *app2Evt*, *app3Evt*, and *app4Evt*. Since they're application defined, how do they get into the event queue in the first place? Simple: you put them there with the OS function *PostEvent*. *PostEvent* takes two parameters: the event code (which should be *app1Evt..app4Evt*), and the event message, a *LongInt* value that can be almost anything you want it to be (including a pointer to some data structure). The event code gets assigned to What, and the event message to Message; the other fields (Where, When, Modifiers) are all set for you by *PostEvent*. *PostEvent* then returns a result code (0=Ok, 1=Event code not allowed) telling you how it did.

Data Structures

Until now, we've shown many program parts without talking too much about the data structures involved, other than the type *EventRecord.*

First, you need to understand that Mac software is heavily based on the ideas of *pointers* and *handles.* If you've programmed in Pascal or C before, you are probably familiar with the concept of a pointer: It's a variable that contains, instead of data, the address of where some data is stored. A handle is just a pointer to a pointer. Handles are used heavily in Mac programs, because they allow the Mac to perform memory reorganization—moving data structures from one place in memory to another—without changing any values.

Here's the way it works: A handle points to a pointer, which in turn points to a data structure somewhere in memory. If the Mac needs to move that data structure, the Mac relocates it and then changes the value of the pointer. The handle's value never changes—it's still pointing to the same pointer—and so the movement is invisible to the program.

Handles carry the analogy of pointers directly: To access the data structure a pointer points to, you say *P*^, that is, the pointer variable's name followed by a caret. For a handle, *H*^ refers to the pointer that the handle points to, and *H*^^ refers to the data structure.

Menus are managed using handles. All the menu procedures and functions work with menu handles, such as *GetMenu, SetItemMark, EnableItem,* and *DisableItem.* Since almost all menu manipulation is done via predefined routines, you should never have to directly access any field of the menu records.

Windows are managed in a two-fold way, with both a *WindowPtr* (*MainPtr*) and a *WindowRecord* (*MainRec*). *MainRec* is used in the initial call to *GetNewWindow,* but is never really directly accessed. *MainPtr,* on the other hand, is used in almost all the window-based calls. *MainPtr* actually points to a structure of type *WindowRecord,* but you can't access it through *MainPtr,* since its declared data type is *GrafPtr.* Instead, you need to do the assignment `MainPeek := WindowPeek(MainPtr);`, where *MainPeek* is defined to be of type *WindowPeek* (which is equivalent to ^*WindowRecord*). You can then access the *WindowRecord* fields via *MainPeek*^.

QuickDraw, the graphics package, has some of its own data structures, most notably the types *Point* and *Rect. Point* is a variant record that allows you to refer to its X and Y components either as separate fields (*V, H*) or as elements of an array (*VH[0]* and *VH[1]*). The type *Rect* builds upon that: You can access its four coordinates as either separate fields (Top, Left, Bottom, Right) or as two points (TopLeft, BottomRight), which in turn give you the options (*V, H*) or (*VH[0], VH[1]*).

There are many other predefined data types, as well as numerous predefined constants. Most can be found by looking at the Macintosh Interface units in Appendix D and at *Inside Macintosh*, especially Volume 1.

Resource Files

There are two ways of defining windows, menus, and other data constructs specific to your program. First, you can use calls to *NewWindow*, *NewMenu*, and so on, hardcoding the menu layout or window specs into the initialization section of your program. Or, you can lay it all out in a *resource file*, compile it using RMAKER, and link it into your program. Your initialization section can then load in the menu, window, and other information from the resource section of the application.

A resource file contains items of different resource types: STR (string), ICON (icon), MENU (menu), WIND (window), DLOG (dialog box), and so on. Each item is associated with a resource ID, such as 1000, 1001, and so on. These IDs need to be unique within a type; you can't have, say, two menus with IDs of 1000. But you can have the same numbers as IDs for different types. For example, in the following resource file example, there are several items with ID=1000, but all are of different resource types:

```
EXAMPLE

TYPE MENU              ; menus
   ,1000               ; resource ID
\14                    ; menu title (=Apple logo)
About My Demo...       ; first item
(-                     ; second item: line separator

   ,1001               ; resource ID
File                   ; menu title
New                    ; first item
Open
Save
Quit

   ,1002               ; resource ID (another menu)
Edit                   ; menu title
Undo/Z                 ; first item (with character command equiv)
(-                     ; line separators
Cut/X                  ; and so on...
Copy/C
Paste/V
Clear
(-
Show Clipboard

; more menus go here
```

```
TYPE WIND              ; window
   ,1000               ; resource ID
My Demo Program        ; window title
44 7 335 505           ; coordinates: y1,x2, y2,x2
Visible GoAway         ; window attributes
0                      ; window type = documentProc
0                      ; refCon: user-definable info

TYPE DLOG              ; dialog box
   ,1000               ; resource ID
About My Demo          ; box title
90 50 180 460          ; coordinates
Visible NoGoAway       ; attributes
16                     ; window type = rDocProc
0                      ; refCon
1000                   ; ID for dialog item list
```

and so on.

These IDs are used when you want to read the resource items from within your program.

How do you associate a given resource file with a program? Two steps are necessary. First, the very first line in a resource file names the file that the compiled resources should be written out to; by convention, that file has the extension .RSRC. Second, having run RMAKER on your resource file, you need to pull the resulting data file into your program when it compiles. You do that using the {$R ⟨file⟩} compiler directive (not to be confused with the {$R+/-} range-checking directive). For example,

```
{$R Example.Rsrc}
```

tells Turbo Pascal to copy in the resources from EXAMPLE.RSRC whenever it compiles a program. If you compile to disk, then the resource information is placed in the *resource fork* of the resulting application. That way, you can move and copy the application as much as you'd like, and the resources automatically go with it.

There are two other compiler directives that tie into using resources. The first is the {$B+} directive, which sets the *bundle bit*. Briefly, the bundle bit tells the Mac desktop that the resources in your application should be installed and used as a group. This applies primarily to icons and file types. Using it allows you to define your own icon in your resource file, as well as icons for files produced by your application.

The second directive is the {$T ⟨filetype⟩}, which is used to define the *file type* and *file creator* for your program. The file type should (usually) be APPL, since you are producing an application. The file creator has two uses. First, it associates the application with an icon; second, it allows a document to invoke your application when it's double-clicked, if the document is associated with the creator type of your program. Again, this all ties in to the bundle concept.

Chapter 12 tells you how to use RMAKER and the format needed for your resource file. Additional information about resources can be found in *Inside Macintosh*, Volume I, Chapter 5, and Volume III, Chapter 1.

Initialization

Like the overall program, the initialization procedure for most Macintosh programs follows a certain structure:

- Call *Init* routines.
- Set up menus.
- Set up windows.
- Do other graphics initialization.
- Do program-specific initialization.
- Handle clicked documents.

There is an *Init* routine for most of the major managers. The first, and most important, is *InitGraf(@thePort)*. That sets up *QuickDraw* (which is used by just about everything else) and sets up a *grafport* for the screen. Other *Init* routines you'll probably want to call are

```
InitGraf(@thePort);              { create a grafport for the screen }
InitFonts;                             { start up the font manager }
InitWindows;                       { start up the window manager }
InitMenus;                           { start up the menu manager }
TEInit;                        { start up the text manager for DAs }
InitDialogs(NIL);                  { start up the dialog manager }
FlushEvents(everyEvent,0);       { clear events from previous state }
```

Setting up menus involves four steps. First, you want to define the menus themselves. If you're using a resource file, just do a call to *GetMenu* for each menu handle, or even a single call to *GetNewMBar*. Otherwise, you have to build each menu using an initial call to *NewMenu*, followed by a call or calls to *AppendMenu*. Second, if you're handling desk accessories, call *AddResMenu*, as described earlier in this chapter. Third, add all the menus to the menu bar by making successive calls to *InsertMenu*. Finally, call *DrawMenuBar* to display the menu titles and make them active, as in

```
MenuList[1] := GetMenu(ApplMenu);     { read menus in from resource }
MenuList[2] := GetMenu(FileMenu);
MenuList[3] := GetMenu(EditMenu);
{ get other menus in like fashion }
AddResMenu(MenuList[1],'DRVR');     { pull in all desk acccessories }
for Indx := 1 to MenuCnt do              { place menus in menu bar }
   InsertMenu(MenuList[Indx],0);
DrawMenuBar;                       { draw updated menu bar to screen }
```

In addition to all that, you may want to make calls to *DisableItem* to disable any commands that shouldn't be active when your program starts, such as editing commands (when no editing windows are open).

As with menus, your window initialization takes several steps. If you need a window at startup, create it using either *GetNewWindow* (reading in from resources) or *NewWindow* (building it in place). In either case, you'll now have a pointer to your window, which is what you'll use for most window manager calls. Having created the window, make it the current *grafport* by calling *SetPort*, then make it the active window by calling *SelectWindow*. If it has a drag bar, define dragging limits (*DragArea*). Likewise, if it has a Grow box, define minimum and maximum size (*GrowArea*); you should also call *DrawGrowIcon*. For example:

```
{ set up window stuff }
GetWMgrPort(ScreenPort);                          { get grafport for all windows }
SetPort(ScreenPort);                             { and keep on hand just in case }
MainPtr := GetNewWindow(MainID,@MainRec,Pointer(-1));              { get window }
SetPort(MainPtr);                                { set window to current grafport }
SelectWindow(MainPtr);                                 { and make window active }
FrontWindow := MainPtr;                             { remember that it's in front }
DrawGrowIcon(MainPtr);                            { draw the Grow box in the corner }
MainPeek := WindowPeek(MainPtr);                     { get pointer to window record }
MainPeek^.windowKind := UserKind;              { set window type = user kind (ID=8) }
ScreenArea := screenBits.Bounds;                { get size of screen (don't assume) }
with ScreenArea do
begin
  SetRect(DragArea,5,25,Right-5,Bottom-10);                    { set drag region }
  SetRect(GrowArea,50,20,Right-5,Bottom-10)                    { set grow region }
end;
```

There is a fair amount of graphics initialization you can, but don't have to, do. This includes setting pen size, pattern, and mode; loading the cursor, either from the system or from your resource file; loading or defining patterns; loading icons and bitmaps; and similar tasks.

Your program-specific initialization should probably come here. You've set up all you need to in order to do some work; if your program requires it, you're able to modify menus, windows, and other data structures and display elements.

The last part of initialization happens only if you have an application that can be launched by opening a document associated with your application's creator type. For example, you can get into Turbo Pascal by double-clicking any program file created under it. You can detect this by making the following call:

```
CountAppFiles(Msg,FCount);
```

This returns two values: *FCount*, which tells you how many files were selected; and *Msg*, which tells you if the files were selected for printing or for opening. You can then set up a loop (*Indx := 1 to FCount*) and call *GetAppFiles(Indx,AFRec)*, which will return information on each file to you, one at a time. Once you've handled that file, you can remove it from the list by calling *ClrAppFiles(Indx)*.

As is true with most aspects of Macintosh programming, the best way to learn is to see what others have done. You'll find a diverse variety of approaches in the sample programs on your Turbo Pascal disk(s); there's something to be gleaned from each one.

Cleaning Up

Cleaning up on the Macintosh is actually minor; most details are automatically taken care of when you exit your program. There is a specific procedure, *ExitToShell*, which you can (but don't have to) call. It's useful if you have to abort in the middle of a program, although the Turbo Pascal routine *Halt* performs the same function.

Large Programs and Segmentation

The Mac limits the code size of a program to 32,768 bytes. When you need to write programs that exceed this limit, you must *segment* your program. This means dividing it up into chunks of less than 32K bytes each.

Turbo Pascal makes segmentation simple. At the start of your program, tell the compiler to produce a segmented code file by including the {$S+} compiler directive.

This switch is off by default; you must explicitly turn it on to use segments. When segmentation is off, all subprogram (procedure and function) calls and subprogram address references are coded by the compiler using program counter-relative instructions. When segmentation is on and you're using the {$S+} directive, all calls and address references are routed through the segment loader jump table.

To segment your program, organize your procedures and functions—including those in a unit—into different segments. When you want to specify which segment a procedure, function, or group of procedures and functions should be in, precede the specific item with the directive {$S *segname*}, where *segname* is a string of up to eight characters.

NOTE: The $S directive is case-sensitive. {$S MySeg} and {$S mySeg} refer to two different segments.

If *segname* is less than eight characters long, it will be padded on the right with blanks. All procedures and functions following it will be included in that segment until the next {$S *segname*} directive is encountered.

The default segment, or blank segment, is one whose name consists of eight blanks. This is where any procedures and functions declared before the first {$S *segname*} directive are stored.

You can repeat segment names within a program, collecting procedures and functions in different parts of the program into the same segment. You can also organize your units into segments, as you saw in the last section of Chapter 8.

When you compile a segmented program, the compiler stores the code segments as resources in the code file. They have resource IDs starting at 1, which the compiler generates and uses to replace the segment names you assigned. Segments are mostly invisible to the program. To load in a segment, call any procedure or function located in that segment. If the segment isn't already in memory, it is loaded in.

To remove a segment from memory, call the OS routine *UnloadSeg*, passing to it the address of any procedure or function in that segment. Suppose you had placed your *Initialize* and *Cleanup* routines in their own segment. The main body of your program might look like this:

```
begin
   Initialize;                                  { do initialization code }
   UnloadSeg(@Initialize);                         { unload the segment }
   repeat
      SystemTask;                              { let desk accessories work }
      if GetNextEvent(everyEvent,theEvent)      { see if there is an event }
         then HandleEvent(theEvent)              { if so, handle the event }
   until Finished;
   Cleanup
end.   { of prog MySegProg }
```

After calling *Initialize*, call *UnloadSeg* and pass to it the address of *Initialize*. This frees the memory that *Initialize*'s segment occupies, allowing the Mac to reclaim it if needed. When you call *Cleanup* at the end of your program, that segment is reloaded from the disk, if necessary. If that segment's memory wasn't used, then it has remained in memory and is called without any disk access.

Summary Exercises

Again, within the scope of this manual, much less this chapter, it is impossible to teach you all you need to know about programming on the Macintosh. But this manual gives you a good start, combined with the sample programs on the distribution disk.

Try making changes to the program MYDEMO.PAS. Before doing so, make sure you are *not* working from your master (original) Turbo Pascal disk, but that you're using a backup copy. Each time you successfully modify something, be

sure to make a backup copy of that version. That way, if you later accidentally muck things up beyond recovery, you can always go back and start over again. Some exercises follow that review what you've learned so far.

Editing Resources

Edit the STR resources in MYDEMO.R, inserting your name and the name of your firm in the appropriate places. Run RMAKER on it, recompile MYDEMO.PAS, select the About My Demo command in the Apple menu, and watch your name come up in the middle of the screen.

NOTE: If you start to change file names, make sure the changes are made in all the right spots. MYDEMO.R's output file name (MYDEMO.RSRC) is located at the beginning of the file. If you change that name, you must make exactly the same change to the name in the {$R *MyDemo.RSRC*} directive at the start of MYDEMO.PAS.

Adding Menu Items

Add a new item to one of the menus. This consists of the following steps:

1. Think up a relatively short name for it, say, Yahoo.
2. Go into MYDEMO.R and add that name to the end of one of the menu lists. You can add a command-key equivalent for it, but it can't duplicate any of the other command keys. For example, use Yahoo/Y.
3. Run RMAKER on MYDEMO.R.
4. Go into MYDEMO.PAS. Go to the procedure *HandleMenu*. Find the **case** statement handle for the menu you modified. Add a new statement for the menu handler `n : DoYahoo` (where n is the next available number for the menu).
5. Add the new procedure, *DoYahoo*. It should have roughly the following structure:

```
procedure DoYahoo;
{ declarations }
begin
  { any setup for the procedure }
  {  procedure body }
end;          { of proc DoYahoo }
```

6. Save MYDEMO.PAS to disk. Select the Run command from the Compile menu. Fix any bugs you discover. Once it compiles and runs, go to the menu you modified and select the Yahoo command. Watch it run.

Adding a New Menu

Add a new menu to the program. Call the menu (and its commands) whatever you'd like; for now, we'll call the menu Stooges, with the commands Larry, Curly, and Moe.

1. Go to MYDEMO.R. At the end of the menus, before the line `TYPE WIND`, type in the menu description as follows:

```
,1005
Stooges
Larry
Curly
Moe
```

   Save MYDEMO.R and run RMAKER on it.

2. Get into MYDEMO.PAS. Go to the **const** section at the top of the file. Change *MenuCnt* from 5 to 6. Right under the statement

```
IOMenu = 1004;
```

   add the line

```
StgsMenu = 1005;
```

   Right under the statement

```
IM = 5;
```

   add

```
SM = 6;
```

3. Go to the procedure *Initialize*. Right under the statement

```
MenuList[IM] := GetMenu(IOMenu);
```

   add

```
MenuList[SM] := GetMenu(StgsMenu);
```

4. Go to the procedure *HandleMenu*. Add a semicolon to the **end** at the end of the **case** statement for *IOMenu*. Add the following underneath it:

```
StgsMenu  : case Item of
      1  : DoLarry;
      2  : DoCurly;
      3  : DoMoe
end
```

5. Go to the end of the procedure *DoFileDelete*. Right after its final **end** statement, add the following:

```
procedure DoLarry;
begin
end;                    { of proc DoLarry }

procedure DoCurly;
begin
end;                    { of proc DoCurly }

procedure DoMoe;
begin
end;                      { of proc DoMoe }
```

6. Save MYDEMO.PAS. Compile it until all the syntax errors are gone and it runs OK. Try out the new menu.

7. Be creative: Put something in each of the procedures *DoLarry*, *DoCurly*, and *DoMoe*.

Practice with these exercises until you feel comfortable enough to move on to the next chapter. In it, you'll learn how to write a desk accessory.

CHAPTER 10

Graduation: Writing a Desk Accessory

Desk accessories are analogous to final exams in Mac programming: Once you've written a desk accessory, you've graduated. Of course, there are still device drivers to wrestle with, but you can consider them graduate studies. In the meantime, get your cap and gown ready, because by the time you're done with this chapter, you'll be writing your own desk accessories.

There are a couple of sample desk accessory programs on the Turbo Pascal disk. This chapter discusses writing desk accessories in general. After you're done reading, you can go and look at the samples to see working programs.

Basic Theory and Structure

Let's start off by looking at the overall format of a desk accessory, much as we looked at the overall format of a Mac application back in Chapter 9. Here's how your source file might be laid out:

```
program ThisDeskAcc;
{$D PasDeskAcc}
{$U-}

uses <whatever units>;

procedure Open(var Device: dCtlEntry); forward;
procedure Control(var Device: dCtlEntry; Param: LongInt; Code: Integer);
     forward;
```

```
procedure Close(var Device: dCtlEntry);
forward

            { Global constants and data types }
          { Support procedures and functions }
{ Body of Open, Control, and Close procedures }

begin end.
```

The *Open*, *Control*, and *Close* procedures must be the first three procedures declared in the desk accessory program. They may have other names, but they must be declared in the above order and with the above parameter lists.

- *Open* This procedure is called when the desk accessory is launched. It should do all the initialization: windows, menus, and so on.
- *Control* This procedure is called each time the system wants the desk accessory to do something: respond to an event or handle a timer update, for example.
- *Close* This procedure is called when the desk accessory is shut down. It closes up and disposes of the various items: windows, menus, and so on.

One easy solution to ensuring that these routines are the first three is to declare them all **forward** at the start of your program, then have their bodies located later on. That allows them to call one another, and also allows them to share other procedures and functions located between their declarations and their implementations.

As you can see, the desk accessory structure is not unlike that of a standard Mac application, with procedures for initialization, event-handling, and cleanup. Unlike an application, though, a desk accessory has no main body; instead, the operating system makes all the appropriate calls to these routines. More accurately, the current application—be it the FINDER or some other program—makes the appropriate calls.

Remember the discussion on supporting desk accessories back in Chapter 9? When a desk accessory (DA) is selected from the Apple menu, the application currently running launches it by calling *OpenDeskAcc*, which instructs the operating system (OS) to load that DA into memory and then call its *Open* procedure. When the application calls *SystemTask*—as it should in its main loop and, for that matter, any other persistent loop—the operating system can then check to see if it needs to call the DA's *Control* procedure. Finally, when the application calls *CloseDeskAcc*, the OS issues the call to the DA's *Close* procedure, then purges it from memory. Likewise, when the application itself is finished, the OS issues calls to the *Close* procedure on any DAs still running (though it is possible to write a DA that continues to run after you exit an application).

Like an application, a desk accessory needs to handle most of the usual events—mouse down, key down, auto key, activate, update—and may have to provide support for windows, dialogs, and menus. As you might suspect, however, there are significant differences in event handling between desk accessories and applications. There are three major differences. First, many events do not have to be handled by the desk accessory at all, since the application takes care of them. For example, the mouse-down event in a DA usually just has to do with an in-content event. Other events, such as the drag bar and Close box, are handled by the application.

Second, other events are predigested by the operating system. Instead of handling the in-menu-bar version of the *mouseDown* event, the DA receives a code from the OS telling it that a menu item has been selected, along with the menu and item values.

This is taken one step further with editing commands: When an application calls *SystemEdit*, passing to it a value in the set [1,3,4,5,6] (which correspond to the commands Undo, Cut, Copy, Paste, and Clear, respectively), the operating system calls the *Control* procedure with that specific information, so that the DA already knows which command has been selected.

Third, a desk accessory has to respond to certain events that an application doesn't. For example, a DA can tell the operating system that it needs to be called periodically, such as once per second. The OS then issues those periodic calls—in addition to whatever other events might occur—telling the DA this is its periodic timed event, and the DA should do whatever it needs to. For example, it may have been told to update a clock.

A desk accessory differs from an application in other ways as well. Perhaps the most significant is in the area of global variables: A DA doesn't have any. You can declare global constants, global data types, and global procedures and functions (in addition to *Open*, *Control*, and *Close*), but not global variables. This could be quite a hindrance, if there were no way around it. But, as you might guess, there is a workaround. The operating system passes the same data structure—a record of type *dCtlEntry*—to all three routines. One of the fields in that record, *dCtlStorage*, can be used as a handle (a double pointer, remember?) to a chunk of memory that holds whatever "global" data you need to be passed from *Open* to *Control* to *Close*, or that you need to be preserved between calls to *Control*.

The fact that a desk accessory cannot have global variables also means that a DA may not use any units that depend upon global variables. In particular, this means that the *PasInOut*, *PasConsole*, and *PasPrinter* units cannot be used by a DA, since these units declare and use global variables (such as the *Input* and *Output* files). The *QuickDraw* unit also declares global variables (*thePort* and *arrow*, for example), but as an exception to the rule, you may use *QuickDraw* as long as you don't refer to any of those variables in your DA.

Other differences and limitations exist. A desk accessory has to fit into a single segment, which restricts it to 32K. (*Inside Macintosh* suggests an 8K limit, but this is due to historical reasons, namely the limited memory on the earliest Macs.) In addition, a DA shouldn't have more than one menu and one window for itself, although again exceptions are possible.

A comment on desk accessory design: A desk accessory is, by nature, modeless. It runs concurrently with whatever application happens to be going on at the time, and you can usually switch back and forth between them. By the usual event handling, a DA can tell when it has been activated or deactivated, that is, when its window has been made the active one. Your design needs to accommodate this and to make sure (as far as is possible) that your DA doesn't seriously interfere with whatever application might be running at the time.

Data Structures

Before we dive into a more detailed discussion on how to write *Open*, *Control*, *Close*, and any supporting procedures, we need to talk about the data structures that you can (and have to) work with. Three in particular are critical: the driver header, the device control entry (*dCtlEntry*), and your global data record (whatever you name it). Let's discuss them in that order.

Driver Header

Device drivers—which include desk accessories—have a fixed format for the first 20 bytes of their machine code. This section, documented in *Inside Macintosh* (Vol. II, Chapter 6, "The Device Manager"), contains information about the driver itself, as well as offset pointers to the *Open*, *Control*, and *Close* procedures. Furthermore, device drivers (including desk accessories) must be of a particular file and creator type in order to be installed into your SYSTEM file as a resource. In short, if you want to produce a desk accessory, you have to ensure that the code file is properly set up.

Turbo Pascal makes this easy to accomplish. Put the directive

```
{$D PasDeskAcc}
```

right after your program header. This tells Turbo Pascal to do all the massaging and changing necessary to turn your program into a desk accessory code file. It defines your output file type as DFIL, which makes the output file appear as a suitcase (the icon used for fonts and desk accessories). The output file creator is set to DMOV, which allows you to double-click on the file icon in order to run

FONT/DA MOVER. If necessary or desired, you can override these values using the {$T ttttcccc} compiler option, where *tttt* is the output file type and *cccc* is the output file creator.

The code resource type is DRVR, which identifies the code as a driver. The resource ID is set to 12, which is the lowest possible value for a desk accessory. Since FONT/DA MOVER modifies the resource ID if needed in order to install the desk accessory into the SYSTEM file, you don't need to worry about whether other desk accessories already have this resource ID. However, if you are going to use a resource file for your desk accessory, you do need to know that those resources *must* have ID values in the range (−16000..−15969); more about this later on.

As mentioned, the *PasDeskAcc* driver header initializes the driver information fields. Four of these are copied into the corresponding fields in the device control entry (*dCtlEntry*) record (see next section) when the desk accessory is opened. Here are the *dCtlEntry* fields, the values they get set to, and what the values mean:

dCtlFlags	$0400	DA can respond to Control calls
dCtlDelay	$0000	Delay (in ticks) = 0, if timer update desired
dCtlEMask	$016A	Events mask; enables *activateEvt, updateEvt, autoKey, keyDown, mouseDown* events for DA
dCtlMenu	$0000	Menu ID associated with DA

These are standard settings for most desk accessories. For many, they're the only settings you need, so that you won't need to initialize these fields in your *Open* procedure. However, when in doubt, initialize the fields anyway and don't assume what values have or have not been set.

Device Control Entry

The operating system uses a data structure known as the device control entry (*dCtlEntry*) to keep track of device drivers and desk accessories. Each driver or DA has its own device control entry, which is passed as a parameter to its *Open*, *Control*, and *Close* procedures. The driver or DA can then set different fields to either tell the operating system how to handle it or keep track of information specific to itself between calls to its procedures.

The device control entry has the following declaration:

```
type
  dCtlEntry    =
    record
      dCtlDriver    : Ptr;          { pointer/handle to the driver }
      dCtlFlags     : Integer;         { status and control flags }
      dCtlQHdr      : QHdr;             { driver I/O queue header }
      dCtlPosition  : LongInt;              { used by I/O drivers }
      dCtlStorage   : Handle;          { handle to private storage }
      dCtlRefNum    : Integer;          { the driver's reference # }
      dCtlCurTicks  : LongInt;     { counter for system task calls }
      dCtlWindow    : Ptr;            { pointer to driver's window }
      dCtlDelay     : Integer;      { # of ticks between Run calls }
      dCtlEMask     : Integer;         { desk accessory event mask }
      dCtlMenu      : Integer        { menu ID associated w/driver }
    end;
```

Don't feel overwhelmed if you don't understand what many of those fields are for. Remember, this data structure isn't just used for desk accessories: It's also used for all device drivers, and so many of the fields aren't applicable to desk accessories. Other fields are used only by the operating system and don't need to be dealt with. Here's a brief explanation of the fields that do matter:

- *dCtlFlags*: This two-byte field holds bits called flags, used to keep track of certain information about the desk accessory (or device driver). Seven bits of the upper byte are used to let the operating system know what kind of calls the DA (or driver) can or needs to respond to. Here are the flags and their functions:

dReadEnable	0100	Set if driver can respond to *Read* calls
dWriteEnable	0200	Set if driver can respond to *Write* calls
dCtlEnable	0400	Set if driver can respond to *Control* calls
dStatEnable	0800	Set if driver can respond to *Status* calls
dNeedGoodBye	1000	Set if driver needs to be called before application heap is re-initialized
dNeedTime	2000	Set if driver needs to be called on a periodic (=*dCtlDelay*) basis
dNeedLock	4000	Set if driver needs to be locked in memory

  Two are of particular interest to the DA: bit 2 (*dCtlEnable*), which tells the OS if the DA can respond to *Control* calls, and bit 5 (*dNeedTime*), which is set if you want the DA to be called on a periodic basis (such as once per second). The lower byte has flags that the OS uses to keep track of the DA's (or driver's) status; they aren't of much concern here.

- *dCtlStorage*: This handle (a pointer to a pointer) is used to keep track of a chunk of memory in which the DA can store any "global" variables it needs. The standard approach is to define a type of record whose fields comprise the variables desired. A call to *NewHandle* is used to do the allocation; after that,

type casting can be used to reference the record and its fields. More on this a little later.

- *dCtlRefNum*: Each DA (or driver) has a reference number used by the operating system (and application) to identify it. It is stored here, where it can be used by the DA to calculate (among other things) the appropriate ID for any resource (such as a menu) associated with the DA. This value is in the range −13 to −32 and is related to the DA's resource ID (12 to 31) by the following equation:

```
ID = -1 * (dCtlRefNum + 1)
```

  Even though the header *PasDeskAcc* sets the resource ID to 12, FONT/DAMOVER might change that (to avoid duplicate IDs) when it installs your desk accessory into the SYSTEM file. You cannot and should not assume that your DA has a *dCtlRefNum* of −13.

- *dCtlWindow*: This field points to the window (if any) opened by the desk accessory. The operating system needs this information to pass certain window events to the current application, which is then responsible for handling them. The DA itself uses *dCtlWindow* to set the window as the current *grafport* before doing anything in it. Also, if the DA does open a window, this field can be checked at the start of the *Open* procedure to screen out any attempt to open a DA that is already open.
- *dCtlDelay*: This field tells the operating system how often to issue a run event. It's only effective if you've set the *dNeedTime* bit in the *dCtlFlags* field. The value stored is in ticks (remember, 1 tick equals 1/60th of a second), so that you would set this field to 60 if you wanted the DA to be called once a second.
- *dCtlEMask*: This field tells the operating system what events the DA wants to respond to, just like the mask that an application passes as the first parameter in the procedure *GetNextEvent*. The event values should be set in *Open*, so that the operating system knows when (and when not) to call *Control*. The *PasDeskAcc* header initializes this field so that the desk accessory handles (just) the following five events: *mouseDown*, *keyDown*, *autoKey*, *updateEvt*, and *activateEvt*.
- *dCtlMenu*: If the DA has a menu associated with it, *dCtlMenu* should be set to that menu's ID. Resources associated with a desk accessory must have IDs that fall within a specific range of values. More on this in "Setting Up the Resources" several pages ahead.

The same device control entry is passed to all three procedures (*Open*, *Control*, and *Close*), so information set in one procedure can be read or modified in another. For example, the DA might use *dCtlWindow* to point to the window opened in *Open*, set that window as the current *grafport* in *Control*, and close and dispose of that window in *Close*.

Now you need to learn about the other major data structure in your desk accessory: your global variables-equivalent record. Since you can't actually declare any global variables, this record is designed to hold all the information you want to preserve between calls to your desk accessory. Your first step is to define this record as a data type, in the following manner:

```
type

  DAGlobals    =
    record
      theMenu        : MenuHandle;
      theString      : String;
      thePlace       : Point;
      CHandle        : CursHandle
    end;

  DAGlobalsP   = ^DAGlobals;
  DAGlobalsH   = ^DAGlobalsP;
```

This record (*DAGlobals*) assumes that you have four "global" variables that you want to maintain: *theMenu*, *theString*, *thePlace*, and *CHandle*. (**NOTE:** This is just an example. You do not have to have fields with those names or of those data types; instead, your global record should reflect what the DA you're writing needs.) You've also declared a data type that's a pointer (*DAGlobalsP*) to this type of record and another that's a handle (*DAGlobalsH*) to the same type of record. The pointer is actually just an intermediary step, since you can't directly declare a handle (such as *DAGlobalsH* = ^^*DAGlobals*).

The example shows you how to define the proper data type, but not what to do with it. You need to set aside a block of memory large enough to contain a record of this type, then save a handle to that block in the device control entry. The next section shows you exactly how to do that.

Initialization

When a desk accessory is launched and its *Open* procedure is called, the procedure's only parameter is the device control entry that the operating system has now associated with that DA. The *Open* procedure then takes care of three major tasks: setting the fields in the device control entry; allocating and initializing any global variables; and setting up any resources, such as windows, dialog boxes, menus, and so on. The first task is unique to the desk accessory (or driver), but the next two are similar to those found in the initialization procedure of any Mac application. These tasks overlap, but they're worth discussing individually.

This record is passed to *Open* as a parameter of type *dCtlEntry*; we'll assume that the parameter is named Device. You need to set certain fields in Device, either for the use of the operating system or for your own use later on. Following are the fields you may need to initialize, with some guidelines on how to do so.

The field *dCtlFlags* lets the operating system know what type of calls it can make to your desk accessory. The flag *dCtlEnable* (bit 2 in the upper byte) is already set, thanks to *PasDeskAcc*. However, if your desk accessory needs to update something periodically, set the flag *dNeedTime* (bit 5 in the upper byte). The sample desk accessory (*MYDA*) moves a string of text a little bit down the window every half a second, so it sets this flag. The constant *dNeedTime* is defined in the program as $2000, again corresponding to the proper bit; a logical OR can be used to set this bit:

```
dCtlFlags := dCtlFlags or dNeedTime;
```

If this bit is set, then the operating system makes calls of the type *accRun* to *Control* on a regular basis. Note that these calls are made in addition to any control event calls made as a result of setting *dCtlEnable*.

If your DA has requested periodic calls to *Control*, you must also specify how often those calls occur. You do this by assigning a value to the field *dCtlDelay*. This field is initialized to 0 by *PasDeskAcc*. This means that the operating system will make calls to *Control* as often as it can if you have set the *dNeedTime* bit in *dCtlFlags*. Any value greater than 0 defines how long (in ticks) the operating system will wait before making another *accRun* call to *Control*; however, this delay will not stop the OS from making other control calls to *Control*. Since a tick is 1/60th of a second, setting *dCtlDelay* to 60 asks the operating system to call your DA once a second.

You may need to modify *dCtlEMask*, which serves as an event mask for your desk accessory. The event types include mouse down, key down, auto key, update event, and activate event. You can use the predefined constants (*mouseDown*, *keyDown*, *autoKey*, *updateEvt*, and *activateEvt*) and the logical OR instruction to set up the mask, as in:

```
Device.dCtlEMask := mouseDown or updateEvt or activateEvt;
```

By the way, *Inside Macintosh* states that you should never request *mouseUp* events if your desk accessory has a window, though it fails to explain exactly why.

There are three more fields that you may want or need to initialize: *dCtlStorage*, *dCtlWindow*, and *dCtlMenu*. The first is used to keep track of any private (global) data storage your desk accessory needs; the second is a pointer to the window that your DA opens (if it opens one); and the third contains the ID

number of the menu that your DA creates (if it creates one). The next two sections explain more about these fields.

Setting Up the Global Variables

Back in the section on global variables, you learned about how to define the data types you need for your global variables. You defined a record (*DAGlobals*) whose fields corresponded to the variables you needed, and a handle (*DAGlobalsH*) to that type of record. However, if you remember, these are all just data types; no actual variables have been declared, and no storage has been set aside. Instead, in the *Open* procedure of your desk accessory, you should have the following statement:

```
Device.dCtlStorage := NewHandle(SizeOf(DAGlobals));
```

The OS function *NewHandle* returns a handle to a chunk of memory; the parameter passed specifies the size in bytes. The built-in function *SizeOf* returns the size in bytes of the data type or variable passed to it. This statement allocates memory equal in size to a variable of type *DAGlobals* and assigns a handle to that memory to the field *dCtlStorage*.

Fine, the memory is set aside, and *dCtlStorage* points (indirectly) to it. Now, how do you use this memory as a global variable? Use the following statements:

```
HLock(Device.dCtlStorage);
with DAGlobalsH(Device.dCtlStorage)^^ do
begin
  { use theMenu, etc., as desired }
end;
HUnLock(Device.dCtlStorage);
```

The OS procedure *HLock* ensures that the operating system (which has the ability to move chunks of memory around) doesn't move your chunk until you're done using it. The **with** statement uses a form of *type casting*, that is, converting one data type into another. The expression *DAGlobalsH(Device.dCtlStorage)* takes the generic handle *dCtlStorage* and converts it to the data type *DAGlobalsH*, which is a handle to a record of type *DAGlobals*. The two pointer symbols or carets (^^) mean that you are now directly referencing a record of type *DAGlobals*, so that you can freely refer to the fields (*theMenu*, *theString*, and so on) by their names. When you're finished, the OS procedure *HUnLock* tells the OS that it is once again free to move that chunk of memory around—until the next time you want to use it.

Almost all desk accessories open a window or dialog box of some sort, since they need an area on the screen in which to perform their function. In addition, other resources—menus, strings, cursors, icons—may be associated with the desk accessory. As with an application, resources can be handled two ways: either by defining them in a resource file, or by building them right within the initialization section. For example, you can open a window by using *GetNewWindow* to read in the values from the resource fork (assuming that you've defined it there), or by using *NewWindow* and passing all the specifics as parameters.

Resource IDs

Like an application, a DA may have resources associated with it. Since the codes of all DAs and all their resources are stored in the SYSTEM file, the operating system needs a mechanism to identify which resources are related to a particular DA. Therefore, the operating system requires all DA-related resources to have IDs that follow this binary pattern:

```
11 000 rrrrrr xxxxx
```

where *rrrrrr* is the DA's resource ID (12 to 31), and *xxxxx* is a number determined by you. This gives you only 32 different resource ID values for a given desk accessory. However, since resources of different types (such as MENU and WIND) can have the same ID, this actually means that you can have up to 32 instances of each resource type in your resource file.

The *PasDeskAcc* header fixes your DA resource ID at 12. This means that the resource ID values in your resource file must be in the range $C180 to C$19F, which corresponds to −16000 to −15969.

This numbering system is fine for compiling your desk accessory, but a complication occurs when you move it into the SYSTEM file, using FONT/DA MOVER. FONT/DA MOVER changes the resource ID of your DA if an existing DA in your SYSTEM file has the same ID. Similarly, it adjusts the IDs of all related resources by changing the ⟨rrrrrr⟩ field to the DA's new resource ID value. This means that you can't use hard-coded constant values for resource IDs in a DA. Instead, you must calculate resource IDs at run-time.

You can calculate the resource IDs using the field *dCtlRefNum*. Remember that *dCtlRefNum* and the desk accessory's resource ID are related by the expression

```
resID = -1 * (dCtlRefNum+1)
```

Therefore, the equation for calculating the starting resource ID within your desk accessory is

```
startID = $C000 - 32*(dCtlRefNum+1)
```

A convenient place for storing this value is the *dCtlMenu* field, if you have a menu; this should be its resource ID anyway. If you don't have a menu, then this field should be set to 0, and you'll have to calculate the value as needed or store it as a field in your globals record.

Opening the Window

Most desk accessories open a window of some kind, which requests or displays information. The system must tell the current application, which needs to handle many window events for the desk accessories, about this window. This is done via a pointer to the DA's window, which is stored in the field *dCtlWindow*. Furthermore, the window's *windowKind* field must be set to the DA's reference number (*dCtlRefNum*). All this involves some type casting (the programming equivalent of hand signaling), so that the compiler thinks *dCtlWindow* is first of type *WindowPtr*, then of type *WindowPeek*. The following statements open a DA's window:

```
WindowPtr(dCtlWindow) := GetNewWindow(dCtlMenu,NIL,Pointer(-1));
WindowPeek(dCtlWindow)^.windowKind := dCtlRefNum;
```

The first reads the window data in from the resource fork and creates it; the second sets the *windowKind* field in the corresponding window record.

Later on in *Open*, the window's contents should be set up as needed. The sequence of code should look something like this (assuming that *SavePort* is a variable of type *GrafPtr*):

```
GetPort(SavePort);                              { save cur port }
SetPort(GrafPtr(dCtlWindow));                   { get our window }
{  do your window text or drawing  }
SetPort(SavePort)                               { restore grfprt }
```

This code saves the current *grafport*, then sets up the DA's window so that it can be written to. After you've set up the window, the current *grafport* is restored and continued.

Setting Up a Menu

A desk accessory can have a menu associated with it. You've already seen how to calculate a DA's resource ID; having done that, you can read the DA in and set it up with the following statements (assuming *theMenu* is a variable of type *MenuHandle*):

```
theMenu := GetMenu(dCtlMenu);                    { get menu hndl }
theMenu^^.menuID := dCtlMenu;                    { verify ID val }
InsertMenu(theMenu,0);                             { add to bar }
DrawMenuBar;                                       { redraw bar }
```

The first statement reads the menu in from the resource fork, while the second verifies that the menu record has the correct ID. This is necessary because some resource-moving utilities don't properly name the resource ID within the menu itself. The last two statements put the menu at the end of the current menu bar, then redraw the bar so that the menu shows up.

It is possible for a desk accessory to set up more than one menu. Should you choose to do so, however, you should create an entirely new menu bar and swap menu bars as the desk accessory is activated or deactivated (see *Inside Macintosh*, Volume 1, Chapter 14, for more details).

Opening Other Resources

You can read in other resources in a similar fashion. One example would be to read in a new cursor from the resource file for use when the desk accessory is active and the cursor is over its window. Other resources, such as strings and patterns, can be read in the same way.

Handling Multiple Calls to Open

A desk accessory can be launched more than once. That is easily illustrated by starting one up, then pulling down the Apple menu. The desk accessory remains on the menu, just waiting to be selected again. You can prevent a new window (or whatever else) from being created each time the desk accessory is selected from the Apple menu.

The key is to realize that the same *Open* routine is called with the same device control entry record (*Device*). By checking the fields in *Device*, you can tell if *Open* is being called again. For example, a test is performed to see if *dCtlWindow* is **nil**:

```
procedure Open;
var
  SavePort      : GrafPtr;
  TempVal       : Integer;
```

```
begin
  with Device do
  if dCtlWindow = nil then          { make sure not already open }
  begin
  { do all the initialization }
  end
end;                                              { of procedure Open }
```

If *dCtlWindow* equals is **nil**, then the initialization occurs; otherwise, *Open* knows that the desk accessory already has a window open (and, presumably, has everything else set up as well), and so skips all the initialization steps.

Event Handling

Desk accessories, like Mac applications, are event-driven programs. In fact, they are more event-driven than applications are, since they are only called when the operating system has some event it thinks the desk accessory should handle. When that happens, the DA calls the procedure *Control*, passing it three parameters: *Device*, *Code*, and *Param*. *Device* is the same old device control entry record, which you'll need to access its fields, as well as any global variables. *Code* is an *Integer* value that tells *Control* what type event the OS wants it to handle, while *Param* is a *LongInt* value that contains (or points to) any parameters associated with the particular event.

The Control Procedure

Here's an example of what the *Control* procedure could look like:

```
procedure Control(var Device: DCtlEntry; Param: LongInt; Code: Integer);
forward;
 ...
procedure Control;
var
  SavePort      : GrafPtr;
begin                                    { main body of proc Control }
  with Device do
begin                                              { access DCtlEntry }
    HLock(dCtlStorage);                          { freeze reloc block }
    with DAGlobalsH(dCtlStorage)^^ do
  begin                                              { access globals }
      GetPort(SavePort);                        { save current port }
      SetPort(GrafPtr(dCtlWindow));                { switch to ours }
      case Code of
        accEvent    : HandleEvent(EventRecP(Param)); { deal with event }
        accRun      : HandleTick;                    { deal with timer }
        accCursor   : CursorAdjust;                   { change cursor }
        accMenu     : HandleMenu(Param);             { item from menu }
```

```
        accUndo..AccClear
                    : HandleEdit(Code);    { standard edit commands }
        goodBye     : Close(Device)           { about to shut down }
      end;
      SetPort(SavePort)                        { restore old port }
    end;
    HUnLock(dCtlStorage)                      { unfreeze rel block }
  end
end;                                           { of proc Control }
```

Control first locks down the block that *dCtlStorage* points to, to prevent the operating system from moving it while it's in use. It then sets up its window as the current *grafport* and handles the event code (*Code*) that the operating system has passed to it. Here's a description of the event codes, which should have been declared as constant values at the start of your desk accessory:

- *accEvent* (64): Some system event (*mouseDown*, for example) has occurred, and the desk accessory must handle it. *Param* points to a record of type *EventRecord*; type casting is used to pass that record as a parameter to *HandleEvent*.
- *accRun* (65): It's time for the periodic update requested by the DA via the *dCtlFlags* and *dCltDelay* fields. You should then do whatever it was you wanted the DA to do on a periodic basis.
- *accCursor* (66): It's time to change your cursor, if you want a different cursor when your desk accessory is active and the cursor is over its window. This parallels putting a call to a *CursorAdjust* procedure in the main event loop of a Mac application.
- *accMenu* (67): The user has just selected an item from the DA menu(s). *Param* now contains the menu and item values in its high and low words, respectively. You should pass it to a menu-handling routine.
- *accUndo* (68), *accCut* (70), *accCopy* (71), *accPaste* (72), *accClear* (73): The user has passed a value in the set [1,3..6] to the *SystemEdit* function, signifying that the Undo, Cut, Copy, Paste, or Clear command has been selected from the Edit menu. You should take appropriate action, if there is any. The gap between *accUndo* and *accCut* accounts for the line separator.
- *goodBye* (−1): The system, for whatever reason, is about to shut the desk accessory down and free up the memory it uses. You should call *Close* to do all your cleaning up.

This version of *Control* handles all possible messages from the operating system, but your desk accessory may not need to handle that many. If it doesn't have a menu, you can drop the check for *accMenu*; likewise, you can ignore any or all of the editing commands if they don't apply to your desk accessory. In addition, if you don't care about the cursor, you can remove the check for *accCursor*. If you haven't requested a periodic update, you don't need to worry about *accRun*. In short, the only two messages you must at least handle are *accEvent* and *goodBye*.

The *HandleEvent* procedure looks very much like its counterpart in a Mac application:

```
procedure HandleEvent(EPtr : EventRecP);
begin
  case EPtr^.What of
    mouseDown   : DoMouseDown(EPtr^);          { mouse button pushed }
    keyDown     : DoKeyPress(EPtr^);             { key pressed down }
    autoKey     : DoKeyPress(EPtr^);                { key held down }
    updateEvt   : DoUpdate;                    { window need updating }
    activateEvt : DoActivate(EPtr^)             { activate/deactivate }
  end
end;                                          { of proc HandleEvent }
```

It should handle the five basic events: *mouseDown*, *keyDown*, *autoKey*, *updateEvt*, and *activateEvt*. However, it probably won't have much to do for most events, since a lot of the DA's event handling is done by the current application or by the operating system itself.

A *mouseDown* event is particularly simple to handle, since all the varieties except for *inContent* are dealt with by the application or the OS. Furthermore, you don't need to check to see if the event happened within the DA's window, since the event wouldn't have been passed to the DA otherwise. Here's a sample procedure:

```
procedure DoMouseDown(theEvent:EventRecord);
var
  MLoc          :       Point;
begin
  with Device, DAGlobalsH(dCtlStorage)^^ do
begin
    thePlace := theEvent.Where;              { find where click was }
    GlobalToLocal(thePlace);                { convert to local coord }
    UpdateWindow                               { and update window }
  end
end;                                         { of proc DoMouseDown }
```

The location of the event, in global coordinates, is found in the field *Where* of the event record; you can convert it to local (window-based) coordinates by passing it *GlobalToLocal*. *UpdateWindow*, a user-defined procedure, is called to allow the desk accessory to respond to the mouse click.

The *keyDown* and *autoKey* events should be handled as you would in a Mac application. You should check (as usual) to see if the event is a menu command and, if so, handle it accordingly; otherwise, do whatever the DA should do if someone just presses a key. Be aware, though, that the *MenuKey* function doesn't work properly within a desk accessory, so you'll have to do the conversion by hand, probably by using a **case** statement.

Here's an example:

```
procedure DoKeyPress(theEvent : EventRecord);
var
  Ch : Char;
begin
  with theEvent, Device, myGlobalsH(dCtlStorage)^^ do
  begin
    Ch := Chr(Message and charCodeMask);    { get character pressed }
    if (modifiers and cmdKey) <> 0 then { check for command key }
    begin
      if Ch in ['a'..'z'] then
        Ch := Chr(Ord(Ch)-32);              { convert to uppercase }
      case Ch of
        'Z' : HandleEdit(accUnDo);
        'X' : HandleEdit(accCut);
        'C' : HandleEdit(accCopy);
        'V' : HandleEdit(accPaste);
                { handle other menu command, if the DA has a menu }
      end
    end
    else SysBeep(1)          { do something in response to the key }
  end
end;                                        { of procedure DoKeyPress }
```

HandleEdit, you may remember, is a user-defined procedure that takes care of the standard editing commands: Undo, Cut, Copy, Paste, and Clear. Its structure should be something like this:

```
procedure HandleEdit (ECode : Integer);
begin
  case ECode of
    accUnDo  : { handle Undo command  };
    accCut   : { handle Cut command   };
    accCopy  : { handle Copy command  };
    accPaste : { handle Paste command };
    accClear : { handle Clear command }
  end
end;      { of procedure HandleEdit }
```

The comment sections should be replaced by whatever code is needed to handle that particular command, even if it's just a beep (SysBeep(1)).

The *updateEvt* event should also be handled as in a Mac application. It means that the window has just been moved or uncovered and needs to be redrawn, with the updating code bracketed between calls to *BeginUpdate* and *EndUpdate*. For example:

```
procedure DoUpdate;
begin
  with Device, myGlobalsH(dCtlStorage)^^ do
  begin
    BeginUpdate(GrafPtr(dCtlWindow));
    { perform DA window redraw }
    EndUpdate(GrafPtr(dCtlWindow))
  end
end;              { of procedure DoUpdate }
```

Unlike the *DoUpdate* routine in Chapter 9, this one has no calls to *GetPort* and *SetPort*. That's because those calls have already been made back in the main body of the *Control* procedure.

The *DoActivate* routine doesn't need to do window selection or similar tasks; that's handled by the current application. It does, however, need to take care of any functionality associated with whether the DA is active or inactive. The following procedure, *DoActivate*, enables or disables a DA's menu, depending on whether the DA is being activated or deactivated:

```
procedure DoActivate(theEvent : EventRecord);
begin
  with theEvent, Device, myGlobalsH(dCtlStorage)^^ do
  begin
    if Odd(modifiers) then
      EnableItem(theMenu,0)              { enable entire menu }
    else DisableItem(theMenu,0);      { disable entire menu }
    DrawMenuBar
  end
end;                                 { of procedure DoActivate }
```

HandleTicks does pretty much as described: It takes whatever periodic action the user wants it to. In the following example, it moves the desired location of a string of text down, then calls *UpdateWindow* to draw the string in the new location:

```
procedure HandleTicks;
begin
  with Device, myGlobalsH(dCtlStorage)^^ do
  begin
    thePlace.V := (thePlace.V + 5) mod 200;  { do wraparound at bottom }
    UpdateWindow
  end
end;                                          { of procedure HandleTicks }
```

Likewise, *CursorAdjust* (if present) tests for the current location of the mouse and sets the cursor accordingly. In the procedure below, a cursor defined in the resource file is read into a global variable and is used whenever the mouse is over the DA's window:

```
procedure CursorAdjust;
var
  MLoc : Point;
begin
  with Device, DAGlobalsH(dCtlStorage)^^, WindowPeek(dCtlWindow)^ do
  begin
    GetMouse(MLoc);
    if PtInRect(MLoc,Port.portRect) then { if mouse over DA window }
      SetCursor(CHandle^^)                  { then use DA's cursor }
      else InitCursor                     { else use default cursor }
  end
end;                                     { of procedure CursorAdjust }
```

Type casting the window pointer (*dCtlWindow*) is done in order to get the window's rectangle (*Port.portRect*).

Menu Handling

If your desk accessory has its own menu, you need to be able to handle the commands as they come in. They'll be signalled by the *Code* parameter (*Code = accMenu*) in the *Control* procedure, instead of by a *mouseDown* event, and the menu information is passed in the *Param* parameter. Your menu-handling routine (*HandleMenu*) is called directly from *Control* and might look like this:

```
procedure HandleMenu(MenuInfo : LongInt);
var
  Menu,Item : Integer;
begin
  if MenuInfo <> 0 then
  with Device,DAGlobalsH(dCtlStorage)^^ do
  begin
    Item := LoWord(MenuInfo);
    case Item of
      1 :  { handle first item in menu };
      2 : { handle second item in menu };
      { and so on }
    end
  end;
  HiliteMenu(0)
end;                                  { of procedure HandleMenu }
```

It is possible to implement more than one menu in a desk accessory, by saving the current menu bar and creating an entirely new one. In that case, you should make *HandleMenu* look like the version in a regular Mac application: have a **case** statement based on the menu ID, then internal **case** statements for the items of each menu.

Support Routines

The purpose of your desk accessory is, of course, to do something, which the support routines help you accomplish. These are the procedures and functions that your event-handling routines call, just as in a regular Mac application. Most look as they would in a regular Mac application, the major difference being that many probably have a main body that looks something like this:

```
procedure Whatever({any parms});
{ any local declarations }
begin
  with Device,DAGlobalsH(dCtlStorage)^^ do
  begin
  { do whatever needs to be done }
  end
end;
```

These support routines are probably located in one of two places. First, they may be declared within one or more of the main procedures, such as *Control*. If they are included inside the procedure *Control*, this makes all those routines local to (that is, only able to be called by) *Control*. This approach allows all those routines access to *Device*, which is one of *Control*'s parameters and therefore is visible to all local routines.

The other approach is to declare some (or all) of the support routines to be global; that is, to declare them outside of and before the procedures *Open*, *Control*, and *Close*. That way, these routines may be called by any or all of those three major procedures. The disadvantage is that *Device* must be explicitly passed to the support (and event-handling) procedures by the calling procedure.

Closing Down

Several things can bring about the termination of a desk accessory. An application can explicitly terminate the DA by calling *CloseDeskAcc*. It should do this, for example, when handling the *inGoAway* version of the *mouseDown* event in a desk-accessory window. Second, the desk accessory may want to terminate itself, perhaps due to insufficient or missing resources. Third, the operating system may need (or want) to shut down the DA. In any case, the DA should be able to make a graceful exit, cleaning up after itself as best it can.

To do all this, the desk accessory uses its *Close* procedure. This is the procedure the operating system calls when it (or the application it's running) wants to close down the DA. The close routine should get rid of the DA's window (if it has one), delete its menu (if it has one), and free up any space pointed to by *Device.dCtlStorage*. The calls to do all this are standard Toolbox and OS calls. Here's an example:

```
procedure Close;
var
  MHandle           : MenuHandle;
begin
  with Device, DAGlobalsH(dCtlStorage)^^ do
  begin
    DisposeWindow(GrafPtr(dCtlWindow));          { get rid of window }
    DeleteMenu(dCtlMenu);                          { remove from bar }
    DisposeMenu(theMenu);                          { get rid of menu }
    DrawMenuBar;                                 { and update menubar }
    DisposHandle(dCtlStorage);                   { get rid of globals }
    dCtlWindow := NIL                              { clear window ptr }
  end
end;                                             { of procedure Close }
```

If your DA has any data (such as text) that has been entered by the user, you may want to ask the user whether or not to save it to disk. At this point, there's no way to avoid having the desk accessory terminated, but you can at least minimize the effects.

In some cases, the procedure *Control* gets the notification that the application heap is about to be re-initialized. This happens if two conditions are met: First, the desk accessory has not been locked in RAM; second, the DA has specifically requested a *goodBye* event. In such a case, you should have a *goodBye* entry in the main **case** statement in the procedure *Control*, and that entry should call *Close*.

Compiling and Installing a Desk Accessory

Writing a desk accessory doesn't do much good unless you know how to correctly compile and install it. In this section, you'll learn how to compile MYDA, a sample desk accessory on your Turbo Pascal disk, and install it into your SYSTEM file.

MYDA: A Desk-Accessory Template

MYDA consists of three source files:

- *MYDA.PAS* contains the global declarations, as well as the *Open*, *Control*, and *Close* procedures.
- *MYDA.INC* contains the event-handling and support routines. The include {$I} directive for this file is just inside the *Control* procedure.
- *MyDA.R* contains the resources—window, menu, and cursor—used by MYDA.

MYDA opens a window, creates a menu, and sets up its own cursor. It then writes a text string into its window and starts moving the string down. When it reaches the bottom of the window, it starts over at the top. By moving the cursor to any part of the window and clicking, you can move the string to that point; it continues to move down from there.

The three items in the DA's menu all change the string being moved. Likewise, the editing commands don't edit the string at all; they just set it equal to the corresponding command (Cut, for example), so that you can see where you would need to put the code to handle that command.

Finally, you can change the string by typing on the keyboard, though you first may want to use the backspace key to delete whatever text is already there.

Compiling MYDA

Compiling MYDA is a two-step process. First, just as with a Mac application, you need to "compile" the resource file (MYDA.R) using RMAKER. Start up RMAKER by double-clicking on its icon from the desktop (or by selecting it from the Transfer menu). Open the file MYDA.R, using the file selection box that RMAKER brings up. When it is done, the file MYDA.RSRC is created. You can now go back to Turbo Pascal; either push the Quit button (to get back to the desktop first) or use the Transfer menu to go directly back into Turbo Pascal.

From Turbo Pascal, open up MYDA.PAS. If you didn't transfer over from RMAKER, you can do this either by double-clicking on MYDA.PAS, or by double-clicking on Turbo Pascal and using the Open item of the File menu to find and open the file. Having done that, go over to the Compile menu and select the To Disk option (or type ⌘K). The result is the file MYDA, which contains the machine code of the desk accessory in the proper format.

Installing MYDA

Having successfully compiled MYDA, you need to install MYDA into your system file (named SYSTEM). Which brings up an **important warning**: Moving desk accessories in and out of system files can, in some cases, corrupt them (the system files, that is). Likewise, a desk accessory with significant bugs can also trash your system file. So, before going any farther—before doing anything else at all—make a copy of the system (boot) disk onto which you plan to install MYDA and use that copy, putting the original away in a safe place somewhere.

If you are using a hard disk, be even more careful. Back up the entire hard disk once, just so you can restore it should anything terribly catastrophic occur. (You back up your hard disk periodically, anyway, right?) Then back up your SYSTEM folder and/or volume on the hard disk, so that you can restore just that, if necessary. This may seem like a lot of unnecessary work, but should anything really bad happen, you'll be glad you took the time.

Do all that before you go any farther. Everything backed up? Now launch the program FONT/DA MOVER, which has a little truck icon, by double-clicking it (or by selecting it from the Transfer menu within Turbo Pascal). A dialog box containing two windows appears. The window on the left automatically opens the current SYSTEM file; the window on the right is empty. The box comes up in

Font mode, so click on the Desk Accessory button under the Font button at the top of the screen. The left window now displays all the desk accessories currently installed in the SYSTEM file. If MYDA (a previous version) is already there, remove it: point at it with the cursor, click the mouse button, then click on the Remove button located between the two windows.

You now need to open the file MYDA to copy your newly-compiled version into the SYSTEM file. Click on the Open button below the right window. This brings up a file selection dialog box. Find the file MYDA (you might have to click on the Drive button to look on different disks or volumes) and select it by double-clicking on it. The file selection box goes away, and MYDA appears in the right window. Click on it once to select it, then click on the ⟨⟨Copy⟨⟨ button located between the two windows to copy MYDA into the SYSTEM file. (This may take a few minutes, especially if you're working on floppy disks.) When the transfer is done and the cursor changes back from a watch to an arrow, select the Close button under each of the two windows. Then select the Quit button to leave FONT/DA MOVER and go back to the FINDER.

If you installed MYDA in the current SYSTEM file, MYDA should appear in the Apple menu. Pull it down and, if it's there, select it. A window (labeled `My DA`) appears on the right side of the screen. The phrase `Hello, world` displays near the top of the window and starts moving down. Also, the DA menu is added to the menu bar. If you select any of the items in it, the corresponding string replaces the string in the window.

Likewise, if you select any of the top five commands in the Edit menu, that command name replaces the string in the window. When you move the cursor over the DA's window, it turns into a "smiley face." Click the mouse and the string moves to where the cursor is, then starts moving down again. If you click on the title bar of another window, de-selecting the DA window, the DA menu is disabled, but the string keeps moving down. You can reactivate the DA window by clicking on it again; you can move it around using the drag bar at the top; and you can make it go away by clicking on the Close box.

If you want to remove MYDA from your SYSTEM file, run FONT/DA MOVER again. Select the Desk Accessory button at the top. Look through the list of DAs until you find MYDA. Click on it to select it, then click on the Remove button to get rid of it. Click the Close button under the left window to close the SYSTEM file, then click the Quit button to get out of FONT/DA MOVER. MYDA should now be gone from the Apple menu.

MYDA was created to help you write your own desk accessories. It's a very basic template that contains most of the event-handling code you'll need; in fact, it probably contains more than you need, since it attempts to handle all situations. If you print it out and study the code, you may get a better understanding of desk accessories than you would reading the sparse and sometimes cryptic comments in *Inside Macintosh.*

The first step in writing your own desk accessory should be to make some modification to MYDA (which you know works). Before doing this, back up the original MYDA files so that you don't modify or destroy the originals. Or make copies of them and rename the copies to something like NEWDA. If you do this, however, be sure to make corresponding changes to the {$R} and {$I} directives inside the .PAS file. You'll probably want to change the name in the program header at the top of the .PAS file, as well. And you'll need to change the name of the resource output file (at the top of the .R file).

To make the DA smaller, you can eliminate the smiley cursor by doing the following steps. Make a deletion in each of the three files (which, for now, we'll assume are called NEWDA.PAS, NEWDA.INC, and NEWDA.R). In NEWDA.R, delete the cursor definition, which is the last resource in the file and starts with the line `Type CURS = GNRL`. In NEWDA.INC, delete the procedure *CursorUpdate.* In NEWDA.PAS, make changes to both the *Open* and *Control* procedures: In *Open,* delete the line `CHandle := GetCursor(dCtlMenu);`. In *Control,* delete the line in the **case** statement that says `accCursor : CursorUpdate;`. You could also delete the field *CHandle* from the data type *DAGlobals.* Now, run RMAKER, Turbo Pascal, and FONT/DA MOVER as described in the previous section. When you run NEWDA, you'll notice that the cursor no longer changes to a smiley face when it is moved over the DA window.

In a similar fashion, you can delete other portions of MYDA. For example, you could stop handling the Edit commands: Delete the *HandleEdit* procedure, move the calls to *HandleEdit* in *DoKeypress,* remove the call to *HandleEdit* in *HandleMenu,* and remove the line `accUndo..accClear : HandleEdit(Code);` in *Control.* Likewise, you could remove the DA menu by deleting the menu from the resource file and eliminating the code to handle it from the source code files.

Another sample desk accessory, after which MYDA was patterned, is CLOCK.PAS. This DA brings up an analog clock—with second, minute, and hour hands—and updates it as you watch. It is well worth studying, since it shows a working DA that does only as much event handling as it has to. Between CLOCK.PAS and MYDA, you should be able to design and implement the desk accessory you want to write.

More References

There is more to writing desk accessories than is discussed here. The primary source, as always, is *Inside Macintosh*, especially Vol. I, Chapter 14, "Desk Manager," and Vol. II, Chapter 6, "Device Manager." However, the information in *Inside Macintosh* isn't comprehensive, and it assumes that your desk accessory is written in assembly language. An excellent supplement is *MacTutor*, a magazine devoted to Macintosh programming. Back issues of *MacTutor* contain a number of articles on desk accessories, and while many of the DAs discussed are written in C or assembly language, almost all of the articles contain information worth knowing.

CHAPTER 11

Using UNITMOVER

When you write units, you want to make them easily available to any programs that you develop. You do this by moving the units into the TURBO file. You can then pull the TURBO file units back out to fix code errors or to make the TURBO file smaller by using the application UNITMOVER.

Chapter 7 explains what a unit is; Chapter 8 tells how to create your own units. This chapter shows you how to use UNITMOVER to move units between two files.

Moving Units

Normally, when you write your own unit, it gets saved out to a file; to use that unit, you have to specify the file name in a {$U ⟨filename⟩} compiler directive. You may have noticed, though, that you can use the standard Turbo Pascal units without giving a file name. That's because these units are stored in the Turbo Pascal application file, that is, in the actual editor/compiler file named TURBO that shows up on your desktop with the icon of a hand waving a checkered flag. Since the units are in that file, any program can use them without a {$U} directive.

UNITMOVER is used to move your units in the TURBO file, so that they can be used as easily as the standard units.

UNITMOVER is a stand-alone application; its icon appears as a standard Macintosh application icon (a hand writing on a piece of paper). To use it, point the mouse at it and double-click. Or, if you're inside Turbo Pascal, exit Turbo Pascal and get into UNITMOVER by selecting it from the Transfer menu at the far right end of the menu bar. A third option is to double-click on any unit code file, that is, the file produced by compiling a unit to disk.

However you get into it, UNITMOVER brings up a display like that shown in Figure 11-1.

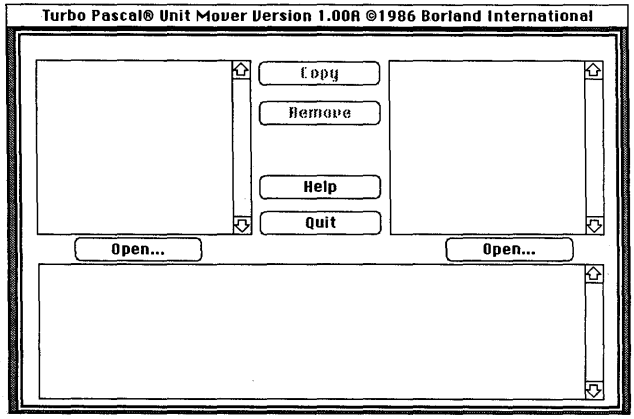


Figure 11-1 The UNITMOVER Screen

If you've ever used the FONT/DA MOVER utility, you'll feel at home here, for the display and functions are very similar.

Since UNITMOVER's purpose is to move units between two files, first select and open those two files. Click on the Open button beneath the left-hand window. A standard file selector display appears; you then select the file.

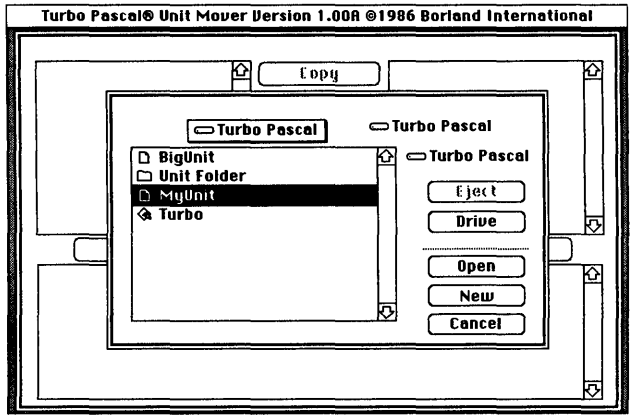


Figure 11-2 The UNITMOVER File Selector Box

The resulting display, with the file open on the left-hand side, appears in Figure 11-3.

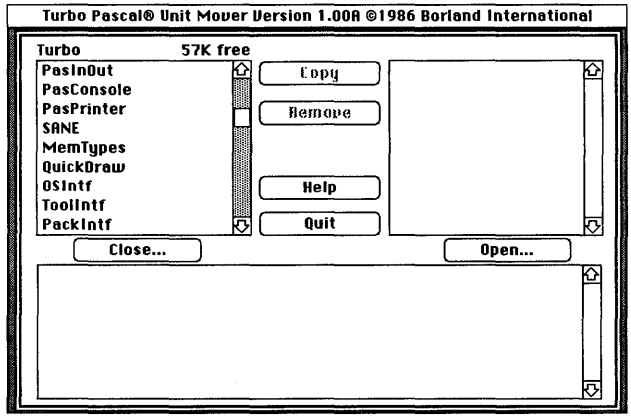


Figure 11-3 UNITMOVER with a File Open

You should do the same thing on the other side (that is, click on the right-hand window's Open button), so that your source file (the one you're copying from) is open on one side, and your destination file (the one you're copying to) is open on the other. If you want to install a unit into Turbo Pascal, then one of the two files has to be TURBO.

Select the unit to be copied by finding it in the appropriate display (using the scroll bar if necessary) and clicking on its name. You can select more than one: Hold ⇧ down and click on the name of each unit to be copied.

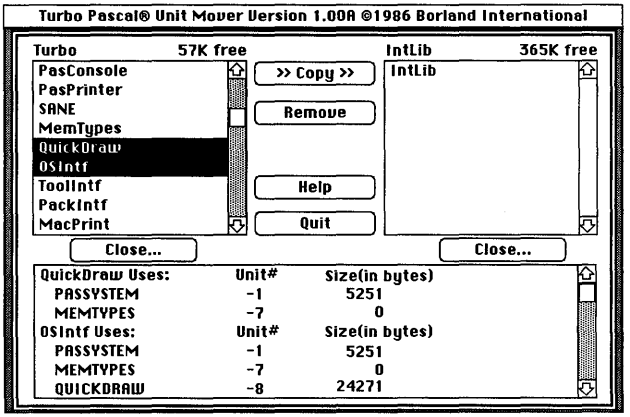


Figure 11-4 UNITMOVER with Units Selected

As you select each unit, information concerning it appears in the area at the bottom of the screen. The information box shows its size, the other units it uses (so that you can copy them as well, if necessary), and if the unit is not in the destination file.

To copy the unit(s), just click on the Copy button located in the top center portion of the screen. A copy of the selected unit(s) is placed in the other file. Make sure you have enough room to receive them.

Deleting Units

You can remove units from a file using the same method. Open the file, select the units, and click on the Remove button. The units are deleted from the file. Be aware that they are gone forever; if you want to save them, copy them to another file first.

CHAPTER 12

Using RMAKER

As mentioned in Chapter 9, there are two ways to define windows, menus, cursors, icons, and other data structures specific to your program. One is to call routines such as *NewWindow*, *NewMenu*, *AppendItem*, and so on, embedding the information about these items in your program. The other is to create the structures as resources, store them in the *resource fork* of your program code file, and then pull them in using the appropriate Toolbox calls.

The method is more flexible for a number of reasons. First, you specify the resources independent of the program itself, so it's easier to make changes to the resources without altering the program. Second, in applications such as *ResEdit*, you can edit the resource fork of a program without changing the machine code, which means that you can modify the resources without having to recompile the program. Third, you can more easily develop international software: You can store all text (menus, window titles, strings) as resources and simply have a different resource file for each language (such as English or Spanish).

This chapter gives you enough information to get started on using resources. Also, Chapter 5 contains detailed information on resources. In addition, check the appropriate chapters of *Inside Macintosh* for the given data types (menus, and so on).

A Quick Guide to Using Resources

There are five steps to follow to use resources in your programs. They are listed here and then explained in the subsequent sections.

1. Create the resource text file. You can do this with the Turbo Pascal editor and save it with a file name ending in .R, such as MYPROG.R.
2. "Compile" this resource text file into a resource "code" file by running RMAKER. The resulting file can be named just about anything you'd like, but the Turbo Pascal convention is to give it the same name as the resource text file with the file extension .RSRC, such as MYPROG.RSRC.
3. Make the appropriate changes to your program so that it reads in the resources as needed. This chapter gives a list of the most commonly used routines. All other routines (for menus, windows, dialogs) are in the corresponding sections of *Inside Macintosh*.
4. At the beginning of your program, put a {$R ⟨filename⟩} compiler directive to tell the compiler which file the resources are located in. For example, if you are working on the file MYPROG.RSRC, then the compiler directive would be *{$R MYPROG.RSRC}*.
5. Compile your program as usual. When running the program in memory, Turbo Pascal automatically opens the resource file before transferring control to the program. When it compiles the program to disk, Turbo Pascal copies all resources in the resource file into the resulting code file.

That's all there is to it. If you're already familiar with each of those steps, you can skim through the rest of this chapter. Otherwise, the following sections discuss each step in detail.

Creating a Resource Text File

The first step is to define your resources in a resource text file. It's almost like writing a program; however, instead of defining actions, you define menus, windows, and other similar items. These are known as resources and can be stored for use in a number of places: within a document, an application, the SYSTEM file, or in a separate resource file. In all cases, they are placed in what is known as the *resource fork* of that file (as opposed to the *data fork*, which holds any data associated with that file). Your program can then load and use the resources as

needed. To get them there, however, you must first specify them in a resource text file, then use RMAKER to convert them to the proper resource formats.

You must abide by certain rules when creating a resource text file.

- All resource definitions must be separated by blank lines.
- All resource types must be four characters long. Take particular care when defining a resource of type *STR* ; don't forget to put a blank after the *R*.
- Comments are allowed. Any line beginning with an asterisk (*) is ignored. You can place comments on the same line as the information by preceding the comments with two semicolons (;;); the rest of the line (semicolons included) is then ignored.
- Resource templates are fairly rigid; you can't rearrange them to suit your own preferences.
- Trailing and leading blanks are ignored, except in strings and in separating numeric values on the same line.
- If data (such as a string) can't fit on one line, end the data with two plus signs (++) and continue on the next line. You can do this for several lines; remember, though, that strings are limited to 255 characters.
- If you want to include resources from the resource forks of other files, just type `Include ‹filename›` such as

  ```
  Include MYOTHERDEMO.RSRC
  ```

  Note that these are "compiled" resources, that is, resources that have already been converted from text to binary format (by using RMAKER, for example).
- All numbers are in decimal (base 10), unless preceded by the .H directive. In that case, they are assumed to be hexadecimal (base 16) for the duration of that resource definition, or until an .I (decimal *Integer*) or .L (decimal *LongInt*) directive is encountered. You will probably use these directives ONLY in a GNRL resource.
- Special characters (ASCII codes less than 32 or greater than 126) can be inserted using a backslash, followed by two hexadecimal digits. For example, the Apple character has an ASCII code of 20; to use it as, say, a menu title, you would enter \14, since 14 in hexadecimal is equal to 20 in decimal.

Resource File Header

Every resource text file should start with the output file name. This should be the first non-comment, non-blank line of the file. As mentioned, the convention is to give the output file the same name as the text file, but ending with .RSRC instead of .R. The line following the output file name should either specify the file type and file creator, or be blank. For example, the two lines

```
YPROG.RSRC
APPLDM01
```

tell RMAKER to create the output file MYPROG.RSRC with the file type APPL and the file creator DM01.

The file type and creator bytes are normally left blank in the resource text file. To specify the file type and creator of the final application, you should place a {$T ttttcccc} compiler directive at the beginning of your program. For example,

```
{$T APPLDM01}
```

If you wish to add the resources defined in your resource text file to those in an existing resource file, place an exclamation point in front of the output file name. For example,

```
!MYPROG.RSRC
```

It's a common practice (for applications) to define a dummy string resource, whose type is the same as the application's signature (file creator), and whose ID is zero. This string, called the *version data resource*, specifies the program name and version, for instance:

```
type DM01 = STR          ; note: blank space after 'STR'
,0                       ; zero ID
MY PROGRAM, Version 1.0, 21 Apr 1987
```

This forms the start of your resource file. The balance comprises resource definitions, which can then be read in by your program.

Defining Resources

Each resource or object to be defined has the following basic structure:

```
type <resource type>
<name>,<id> (<attributes>)
<specifications>
<blank line>
```

Groups of resources of the same type can be defined by repeating the ⟨id⟩, ⟨specifications⟩, and ⟨blank line⟩ sections. Remember, each instance of a resource must be separated from others by blank lines.

The Macintosh supports over 30 standard resource types; however, only 12 of them are recognized by RMAKER:

ALRT	Alert box for error messages, warnings
BNDL	Bundling information (for icons, file types, etc.)
CNTL	Controls scroll bars
DITL	Dialog item — objects in dialog or alert boxes
DLOG	Dialog box — user input, selection
FREF	File reference — linking file types, icons
GNRL	General — for defining other resource types
MENU	Menus — menu title with list of commands
PROC	Procedure — machine code
STR	String — text strings used within the program
STR#	String list — a list of text strings
WIND	Window — window title, size, attributes, type

Some of the additional types recognized by the Toolbox and OS routines but not directly supported by RMAKER include ICON (icon), CURS (cursor), and others. To define them, you must use the GNRL type and start the definition with

```
type ‹resource type› = GNRL
```

The same method is used to define your own resource types. This is discussed in further detail at the end of the chapter.

Each resource may be given a name. It can then be referenced by that name. The name can be up to 255 characters long, though you'll want to keep it fairly short to avoid eating up too much memory or disk space. If you choose not to give the resource a name (a common practice), you must still place a comma before the resource ID.

Each resource must have an ID, an integer value that uniquely identifies it from all other resources of the same type stored in your program's resource fork.

- The values from −32768 to −16385 are reserved for system use.
- The values from −16384 to −1 are reserved for system resources owned by other system resources (the private resources of a desk accessory for example).
- The values from 0 to 127 are also reserved for system use.
- You can use any value from 128 to 32767.

Note that resources of different types — say, MENU and WIND — can have the same ID, but all resources of a given type (such as MENU) must have different IDs.

Each resource has a set of attributes associated with it. Each attribute is represented by a single bit. To enable that attribute, you must set the bit to one. Here's a partial list of those attributes, their bit positions and values, and what happens if the attribute is set:

Attribute	Bit	Val	Effect on resource of setting bit
resPreload	2	4	load after opening resource file
resProtected	3	8	can't be changed by Resource Manager
resLocked	4	16	can't be relocated or purged
resPurgeable	5	32	can be purged
resSysHeap	6	64	load in system heap

To set one or more of these attributes for a given resource, just add the corresponding values together and place the sum (in parentheses) right after the resource ID. For example, a resource with the name of *MyRes*, an ID of 1000, and the attributes *Preload* (4) and *Locked* (16) would look like this:

```
MyRes,1000 (20)
```

Setting attributes is optional. If you do not specify any attributes, a value of 0 (all attributes disabled) is assumed. Also note that you should not try to set any other attributes; all other bits are reserved for system use.

Resource Specifications

The resource specification depends entirely upon the resource type. The predefined types (including those not directly supported by RMAKER) have fixed formats; these must be adhered to when defining the resource. It is possible, though, to define your own resource and load it from your program. In that case, the data structure you use must match the resource as defined in the resource text file.

Following is a list of commonly used resource types, including all those recognized by RMAKER and some that are not. Each section tells where the resource is mentioned in *Inside Macintosh*, gives a very brief description of what the resource does, shows an example of one in a resource file, and states how you would load the resource from within your program. In the last case, the variable to which the result of a function call is being assigned is always assumed to be a handle to that resource type.

ALRT—Alert template

Chapters: "The Dialog Manager" (I-13)

An alert box is a special type of dialog box used to caution, warn, or even stop the user; it uses a list of dialog items (see DITL).

```
type ALRT
,128 (4)                ; preloaded to avoid disk access
100 50 150 250          ; top left bottom right
1000                    ; ID of item list (DITL)
F721                    ; stages (4th 3rd 2nd 1st)
```

BNDL—Bundling information

Chapters: "The Finder Interface" (III-1)

A bundle defines an application's signature (the same as its file creator) and the resource ID of its version data. It specifies (through a list of FREF resources) all file types related to the application and the icons (through a list of ICN resources) to be displayed by the FINDER. The local IDs in the list are needed only by the FINDER. For a bundle to be processed by the FINDER, set the bundle bit when compiling the application: Place a {$B+} directive in the beginning of the program. The DM01 signature must have its own resource, such as `TYPE DM01=STR ,0 TEXT`.

```
type BNDL
,128                    ; resource ID
DM01 0                  ; signature and version data ID
ICN#                    ; ICN# map
0 128 1 129             ; local ID 0 is resource ID 128; 1 is 129
FREF                    ; FREF map
0 128 1 129             ; local ID 0 is resource ID 128; 1 is 129
```

CNTL—Control template

Chapters: "The Control Manager" (I-10)

A control is a graphical object on the screen that the user can modify (using the mouse) as a form of setting or data input. Controls may be specified as *Visible* or *Invisible*.

```
type CNTL
,256 (4)                ; preloaded
Throttle                ; control title
100 50 250 66           ; top left bottom right
Visible                 ; attribute
16                      ; control definition ID (ProcID)
0                       ; reference value (RefCon)
0 999 0                 ; minimum, maxium, initial value
```

An example of accessing a control template follows:

```
theCntl := GetNewControl(256,theWindow);
```

CURS—Cursor

Chapters: "QuickDraw" (I-6), "Toolbox Utilities" (I-16)

The cursor is the shape that moves around the screen as you move the mouse. Besides the default arrow, there are four standard cursors that you can load, or you can define your own.

```
type CURS = GNRL
,128 (4)                    ; preloaded
.H                          ; hexadecimal data follows
* cursor data
07E0 1818 300C 6006         ; 16x16 bitmap, top to bottom
4422 8421 8421 8001
8001 9819 8C31 47E2
6006 300C 1818 07E0
* cursor mask
0000 0000 0000 0000         ; 16x16 bitmap, top to bottom
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 0000
* hot spot
0008 0008                   ; (y,x) = (8,8)
```

An example of accessing a cursor resource follows:

```
myCurs := GetCursor(128);
```

DITL—Dialog item list

Chapters: "The Dialog Manager" (I-13)

Specifies the items — texts, buttons, icons — used in dialog and alert boxes. Eight different item types are available: *StatText*, *EditText*, *Button*, *RadioButton*, *CheckBox*, *IconItem*, *PicItem*, and *UserItem*. Items may be specified as *Enabled* or *Disabled*; they are assumed to be enabled if you don't specify anything. All coordinates are relative to the window of the dialog or alert box using the items. Be sure to have blank lines between all item specifications.

```
type DITL
,1000                           ; resource ID
8                               ; number of items in list

StatText Disabled               ; static text
20 60 34 160                    ; top left bottom right
Print the document              ; text

EditText Enabled                ; edit box
120 45 140 195                  ; top left bottom right
Annual Report                   ; initial text

Button                          ; rounded button
15 170 34 220                   ; top left bottom right
Cancel                          ; text (in button)
```

```
RadioButton                    ; radio-type button
50 15 64 140                   ; top left bottom right
8 1/2" x 11" paper             ; text (to right of button)

CheckBox                       ; check box
96 15 110 200                  ; top left bottom right
Stop after each page           ; text (to right of check box)

IconItem Disabled              ; icon
10 20 42 52                    ; top left bottom right
128                            ; ICON resource ID

PicItem Disabled               ; QuickDraw picture
0 100 0 100                    ; top left bottom right
128                            ; PICT resource ID

UserItem                       ; user-defined item
150 150 200 200                ; top left bottom right
```

DLOG—Dialog template

Chapters: "The Dialog Manager" (I-13)

A dialog box displays information and allows user input or modification. Like an alert box, it uses a list of items (DITL) to determine its layout. A dialog may be specified as *Visible* or *Invisible*, followed by *GoAway* or *NoGoAway* to determine whether it has a Close box.

```
type DLOG
,1000 (4)                ; preloaded
A Dialog Box             ; title
50 100 300 450           ; top left bottom right
Visible NoGoAway         ; attributes
0                        ; window definition ID (ProcID)
0                        ; reference value (RefCon)
1000                     ; item list ID (DITL)
```

An example of accessing a dialog template is

```
theDialog := GetNewDialog(1000,nil,Pointer(-1));
```

FREF—File reference

Chapters: "The Finder Interface" (III-1)

Used in conjunction with a bundle resource (BNDL). An FREF associates a file type with a local ID used in the bundle. This local ID in turn determines the resource ID of the icon (ICN#) to be displayed for files of that type created by the application.

```
type FREF
,128                     ; resource ID as found in BNDL
APPL 0                   ; file type and local ID
```

```
,129                    ; resource ID as found in BNDL
DMTX 1                  ; file type and local ID
```

ICN—Icon list

Chapters: "The Finder Inteface" (3-1)

An icon followed by a mask, used primarily for associating icons with files in the FINDER.

```
type ICN   = GNRL
,128                    ; resource ID
.H                      ; hex values
* icon data
FFFFFFFF 80000001      ; 32x32 bitmap
80000001 80000001
      ...
80000001 FFFFFFFF      ; 32 longwords in total
* icon mask
FFFFFFFF FFFFFFFF      ; 32x32 bitmap
FFFFFFFF FFFFFFFF
      ...
FFFFFFFF FFFFFFFF      ; 32 longwords in total
```

ICON—Icon

Chapters: "Toolbox Utilities" (1-16)

A single icon that can be associated with other resources (such as item lists for dialogs) or can be read in by your program.

```
type ICON = GNRL
,128                    ; resource ID
.H                      ; hex values
FFFFFFFF 7FFFFFFF      ; 32x32 bitmap
3FFFFFFF 1FFFFFFF
      ...
00000003 00000001      ; 32 longwords in total

,129                    ; resource ID
.H                      ; hex values
80000000 C0000000      ; 32x32 bitmap
E0000000 F0000000
      ...
FFFFFFFE FFFFFFFF      ; 32 longwords in total
```

An example of accessing an icon resource is

```
theIcon := GetIcon(129);
```

MBAR—Menu bar

Chapters: "The Menu Manager" (I-11)

Defines an entire menu bar through a list resource IDs for individual menus (MENU).

```
type MBAR = GNRL
,1000                    ; resource ID
.I                       ; integer values
5                        ; number of menus in bar
1000                     ; resource ID of 1st menu
1001                     ; resource ID of 2nd menu
1002                     ; resource ID of 3rd menu
2000                     ; resource ID of 4th menu
2001                     ; resource ID of 5th menu
```

An example of accessing a menu bar resource is

```
theMBar := GetNewMBar(2000);
```

MENU—Menu

Chapters: "The Menu Manager" (I-11)

Defines a menu, that is, the menu title that appears on the menu bar and the listed commands, including any command-key equivalents. You can, as with other resources, list mutiple resources under one **type** statement.

```
type MENU
,1000                    ; resource ID
\14                      ; title (Apple character)
About MyProgram...       ; first command
(-                       ; disabled dotted line

,1001                    ; resource ID
file                     ; title
New/N                    ; New command (⌘N)
Open/O                   ; Open command (⌘O)
Save/S                   ; Save command (⌘S)
Save As...               ; Save As command
Close                    ; Close command
(-                       ; separator line
Quit/Q                   ; Quit command (⌘Q)

,1002                    ; resource ID
Edit                     ; title
Undo/Z                   ; Undo command (⌘Z)
(-                       ; separator line
Cut/X                    ; Cut command (⌘X)
Copy/C                   ; Copy command (⌘C)
Paste/V                  ; Paste command (⌘V)
Clear                    ; Clear command
(-                       ; separator line
(Options...              ; Options command (disabled)
```

Here's an example of accessing a menu resource:

```
theMenu := GetMenu(1000);
```

PAT—Pattern

Chapters: "QuickDraw" (I-6), "Toolbox Utilities" (I-16)

Represents an 8×8 pattern that is used for drawing and filling. Be sure to leave the trailing space after PAT

```
type PAT  = GNRL
,500                          ; resource ID
.H                            ; hex values
FF00 FF00                     ; alternating black/white lines
FF00 FF00                     ; total of eight bytes
```

Example of accessing a pattern resource:

```
thePat := GetPattern(500);
```

PAT#—Pattern list

Chapters: "QuickDraw" (I-6), "Toolbox Utilities" (I-16)

List of patterns, all under one resource ID. The *GetIndPattern* procedure lets you specify which one (1,2,..,N) you want.

```
type PAT#  = GNRL
,600                          ; resource ID
.I                            ; integer value
5                             ; number of patterns
.H                            ; hex values
FF00 FF00 FF00 FF00           ; 1st pattern
8844 2211 8844 2211           ; 2nd pattern
3F8A 9BC9 380A AA03           ; 3rd pattern
0000 0180 0180 0000           ; 4th pattern
AA55 AA55 AA55 AA55           ; 5th pattern
```

Example of accessing a pattern in a pattern list (*myPat* is a variable of type *Pattern*):

```
GetIndPattern(myPat,600,4);
```

PROC—Procedure

The PROC type is used to create resources that contain machine code. It reads the first code segment from an application file (the CODE resource with ID = 1), strips the first 4 bytes off of it (these are used by the Segment Loader), and saves it as a resource of type PROC (unless retyped as shown below). It is mainly useful for defining other code resource types, such as CDEF, DRVR, FKEY, INIT, MDEF, PACK, PDEF, and WDEF.

```
type PROC                    ; create PROC resource
,128                           ; resource ID
MyProcedure                    ; filename

type CDEF = PROC             ; create CDEF resource
,3                             ; resource ID
MyControl                      ; filename
```

STR—String

Chapters: "Toolbox Utilities" (I-16)

Represents a text string; note the trailing blank after STR . If you need to continue a string onto another line, use the ++ continuation mark. All leading and trailing blanks are significant.

```
type STR
,1000 (4)                    ; resource ID (preloaded)
Turbo Pascal for the Macintosh

,1001 (4)                    ; and another string
Copyright (C) 1986 by Borland International

,1002 (4)                    ; and another
All rights reserved.
```

Example of accessing a string resource:

```
theStr := GetString(1001);
```

STR#—String list

Chapters: "Toolbox Utilities" (I-16)

Represents a list of strings. Same rules apply as for STR #.

```
type STR#
,128 (4)                     ; resource ID (preloaded)
3                            ; number of strings
Turbo Pascal for the Macintosh
Copyright (C) 1986 by Borland International
All rights reserved.
```

Example of accessing a string in a string list (*myStr* is a variable of type *Str255*):

```
GetIndString(myStr,128,2);
```

WIND—Window

Chapters: "The Window Manager" (I-9)

Defines a window. A window may be specified as *Visible* or *Invisible*, followed by *GoAway* or *NoGoAway* to determine whether it has a Close box.

```
type WIND
,1000                          ; resource ID
This Program                   ; window title
44 7 335 505                   ; top left bottom right
Visible GoAway                 ; attributes
0                              ; window definition ID (ProcID)
0                              ; reference value (RefCon)

,1001                          ; resource ID
About This Program             ; title
90 50 180 460                  ; boundaries
Invisible NoGoAway             ; attributes
16                             ; window definition ID (ProcID)
0                              ; reference value (RefCon)
```

An example of accessing a window resource follows:

```
theWindow := GetNewWindow(1000,nil,Pointer(-1));
```

Defining Your Own Resources

You've already learned much of what you need to know about defining your own resources. Since RMAKER only supports twelve of the standard resource types, we've had to define the others in terms of the general resource type, GNRL. But what if you want to define a resource that isn't found among any of the standard types? Here's what you would do:

- Define the resource in terms of a Pascal data structure — most likely an array or a record.
- Determine the byte-by-byte mapping of the data structure, that is, the size and starting offset of each element or field.
- Design a skeleton resource type based on the byte-by-byte mapping, and define resources in your file using this skeleton.
- Read the resources into your program using the *GetResource* function.

Let's look at an example. Suppose you wanted to define a resource type called CUBE. Your data structure might look something like this:

```
type
  Cube     = record
               vertex: array[1..8,1..3] of Integer;
               color:  Pattern;
               title:  Str255;
             end;
  CubePtr  = ^Cube;
  CubeHndl = ^CubePtr;

var
  MyCube: CubeHndl;
```

The mapping of the *Cube* record is as follows: 48 bytes for the vertex list (2 bytes each), followed by 8 bytes for the color, followed by between 1 and 256 bytes for the title (the length is stored in the first byte). A resource based on this could be

```
type CUBE = GNRL
,1000                           ; resource ID
.I                              ; decimal integers follow
0      0     0                  ; 1st vertex
0      0   100                  ; 2nd vertex
0    100     0                  ; 3rd vertex
100    0     0                  ; 4th vertex
0    100   100                  ; 5th vertex
100    0   100                  ; 6th vertex
100  100     0                  ; 7th vertex
100  100   100                  ; 8th vertex
.H                              ; hexadecimal values follow
AA55 AA55 AA55 AA55             ; pattern
.P                              ; Pascal string follows
This is my cube                 ; title
```

Having defined such a cube in your resource file, you could read it into your program and process it with the following statements:

```
MyCube := CubeHndl(GetResource('CUBE',1000));
with MyCube^^ do
begin
  { do whatever you want to }
end;
```

You can use Turbo Pascal's retyping mechanism to convert from the generic handle that *GetResource* returns to the specific data type *CubeHndl*.

If your resource includes a string, it should be the last item in the resource. When a string is declared in Pascal, say with a maximum length of 80, the compiler sets aside 81 bytes for that string, even though its dynamic length may be less. However, when RMAKER processes a string, it only stores as many characters as you write (plus the preceding length byte).

Now, if the *title* field in the *Cube* record was the first field, you would have to write exactly 255 characters for the following fields to line up properly; by placing

it at the end, you can write a string of any length. Be aware, though, that the handle allocated by *GetResource* only has room for the string you write, not for the full 255 characters.

Here's a list of the data type commands that you can use, with a brief explanation of each:

- **.H** Hexadecimal values follow. RMAKER accepts the values as integers (four digits) or long integers (eight digits).

```
.H
00F00 1234 00FF 4321
00FFFF00 FF0000FF
```

- **.I** Decimal integers follow. Values must be in the range −32768 to 32767, separated by blanks.

```
.I
32 -532 10043 15
```

- **.L** Decimal long integers follow. Values must be in the range −2147483648 to 2147483647, separated by blanks.

```
.L
642393 -100101 0 3
```

- **.P** Pascal string follows. A length is calculated and stored as the first bytes, followed by the text itself. Leading blanks are significant. Use the continuation symbol to extend to the next line. The maximum length is 255 characters.

```
.P
And still I persist in wondering if folly ++
must always be our nemesis.
```

- **.S** String follows. Only the text is stored; no length is calculated. No maximum length.

```
.S
Four score and seven years ago, ++
our forefathers brought forth upon ++
<and so on>
```

- **.R** Resource follows. The next line gives a file name, a resource type, and a resource ID; the corresponding resource is copied from the given file.

```
.R
MyStuff MENU 1000
```

RMAKER itself is a very easy application to use. You can open it from the desktop by double-clicking on it, or you can transfer to it using the Transfer menu from within Turbo Pascal.

Once you've started it, RMAKER brings up a file selector dialog box as shown in Figure 12-1.

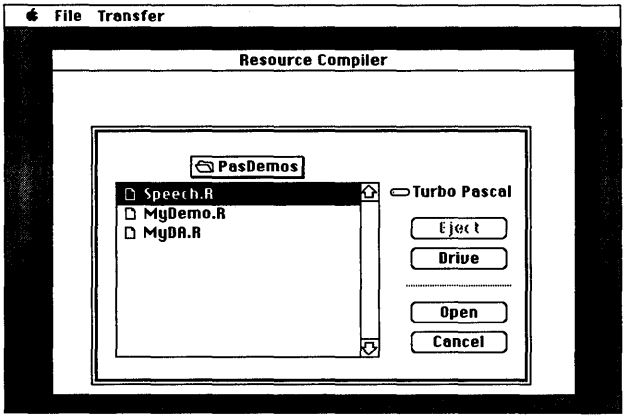


Figure 12-1 *The RMAKER File Selector Box*

It lists only those files ending with .R; to choose one click on it, then select the Compile button, or double-click on the file name. The file selector then disappears, and the "compilation" starts. The left side of the display shows the contents of the resource text file as it is read in, while the right side shows the size of the resulting resource code file.

You can abort the process at any time by clicking the Stop button in the lower left corner of the display.

When it's done and you want to exit RMAKER, click on the Quit button (same as the Stop button) or select the Quit command from the File menu. You may process another resource file by selecting the Compile command from the File menu. And you can transfer back to Turbo Pascal (or any other program): Select the Other... command in the Transfer menu, then select the appropriate program from the file selector that appears.

To let your program use a given resource file, just place the name of the resource code file in a {$R} directive at the beginning of your program, such as

```
{$R MyProg.Rsrc}
```

When you run your program in memory, Turbo Pascal automatically opens the resource file. When you compile your program to disk, Turbo Pascal copies all resources from the resource file into the final application code file.

To actually use those resources within your program, you must read each one in as desired. Most often, this is done as part of an initialization procedure, which reads in the menus, windows, and other resources, setting up the desired display. Since you usually refer to these resources by their IDs, you must know the ID of each one that you are reading in.

Again, a common practice is to declare those IDs as constants. For example, suppose you had defined five menus, with resource IDs 1001 through 1005, respectively. You might then write the following code to read them in:

```
const
  menuID    = 1000;
  menuCount = 5;
type
  MenuList: array[1..menuCount] of MenuHandle;

procedure Initialize;
begin
  ...
  for I := 1 to menuCount do
    MenuList[I] := GetMenu(menuID+I);
  ...
end;
```

GetMenu reads in each of the menus from the resource file, points a handle to it, and then returns that handle, which is assigned to an element of *MenuList*.

Similar techniques are used for other resources, though most other resources are assigned to individual handles, rather than to arrays. Again, *Inside Macintosh* is the best reference for more details on using resources.

CHAPTER 13

Using FONT/DA MOVER

This chapter tells you how to use the FONT/DA MOVER to install the desk accessories (DAs) that you write in Turbo Pascal. You can then apply these instructions on moving DAs to moving fonts.

Fonts and DAs normally reside in the SYSTEM file of a Macintosh startup disk, but can also be kept in "suitcase" files (files with the suitcase icon) for later inclusion in a SYSTEM file. The FONT/DA MOVER utility is used to copy or remove fonts and DAs on Mac disks. Apple's FONT/DA MOVER program is included on your Turbo Pascal distribution disk.

Text fonts are used by the system, the Turbo Pascal editor, and your application programs. These fonts are usually provided as part of your Mac computer. *Using the Macintosh*, the handbook that came with your Mac, explains how to use FONT/DA MOVER.

Starting Up FONT/DA MOVER

There are three ways to start up the FONT/DA MOVER. First, if you are in Turbo Pascal and have included the FONT/DA MOVER in the Transfer menu, select that menu option, exit Turbo Pascal, and enter the FONT/DA MOVER. **When adding the FONT/DA MOVER to the Turbo Pascal Transfer menu, remember to rename the file to change the slash (/) to another character.** (We

suggest using the plus sign, +.) You must make this change both to the Transfer menu and to the FONT/DA MOVER file itself.

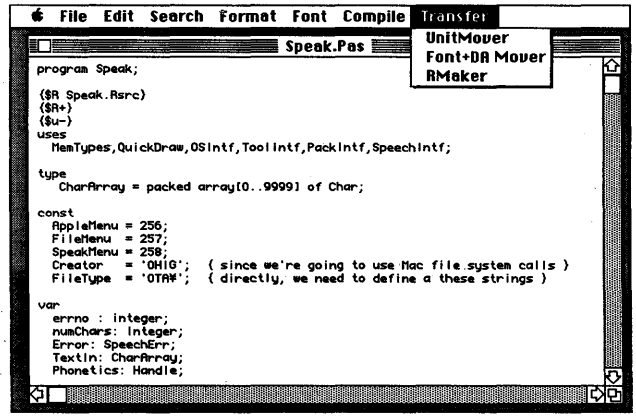


Figure 13-1 The Transfer Menu

You can also launch the FONT/DA MOVER from within Turbo Pascal by selecting the Transfer command from the File menu. This brings up a standard file selector window; scroll through until you find the FONT/DA MOVER, click on its name, and then click on the Transfer button.

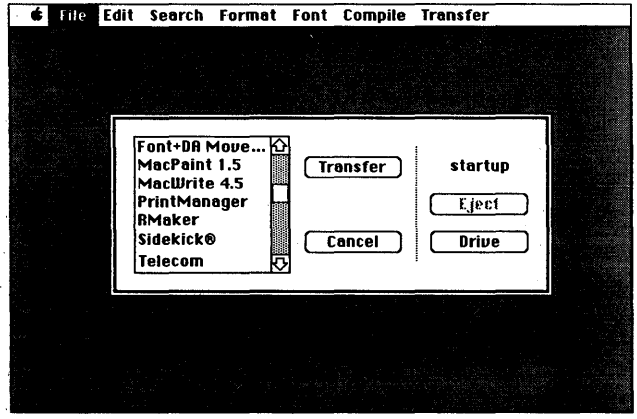


Figure 13-2 The Transfer Command File Selector Window

If you are in the FINDER, you can double-click on the FONT/DA MOVER icon or on a suitcase icon. The DA suitcase in the file results from compiling a Turbo Pascal desk accessory program to disk.

When the FONT/DA MOVER starts up, a dialog box is displayed.

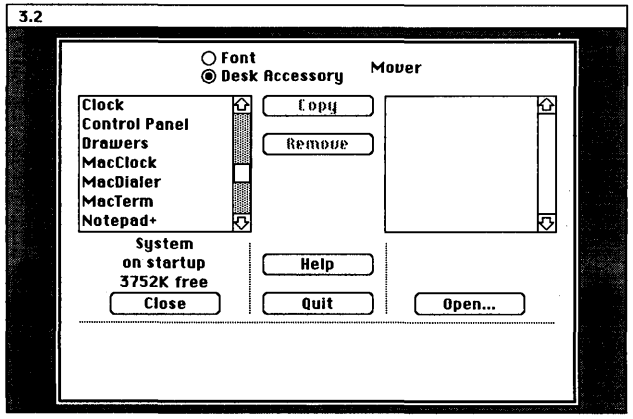


Figure 13-3 FONT/DA MOVER Dialog Box

You'll notice two things. First, FONT/DA MOVER automatically opens the current SYSTEM file or, if you clicked on a DA suitcase icon, the DA file you selected. Second, the top of the dialog box has two buttons. If you launched FONT/DA MOVER directly by double-clicking on it or transferring to it, the Font button is selected. Click on the Desk Accessory button right below it. You're now ready to go on.

Installing Desk Accessories

To use a desk accessory, you must copy it into your SYSTEM file. At this point, either the SYSTEM file or your DA file is open, probably in the left-hand display. Both need to be open, so click the Open button under the other empty display. A file selector box comes up; scroll through it and open the appropriate file. You should now have a list of desk accessories in each display, one for the SYSTEM file and one for your DA file (which probably has only one desk accessory in it).

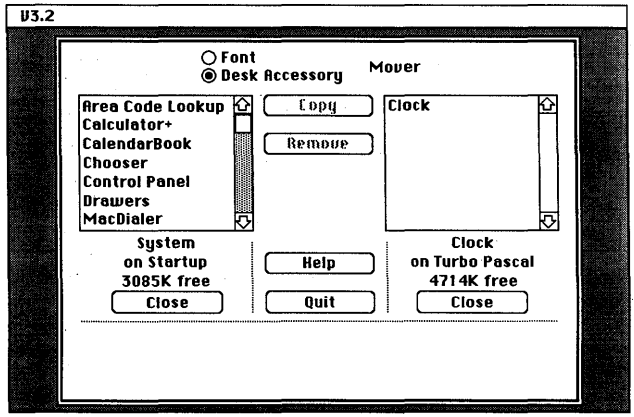


Figure 13-4 Selector Box with SYSTEM and DA Files Open

Now you can copy your DA into the SYSTEM file. Click on the DA file name in the display. At this point, the Copy button is enabled, with arrows showing the direction in which the file will be copied. It also shows how large the desk accessory being copied is.

To copy it, select the Copy button. The process may take a while, as the SYSTEM file is usually quite large and updating it may require some shuffling. When it's done, your DA's name appears in the SYSTEM file display.

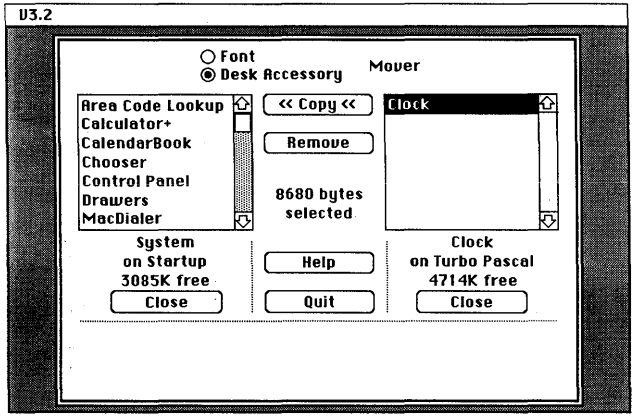


Figure 13-5 FONT/DA MOVER with DA Selected for Copying

Close both files by clicking on their respective Close buttons, then exit FONT/DA MOVER by clicking on the Quit button.

You can copy DAs out of the SYSTEM file into other files (including other SYSTEM files). The same technique is used to move DA files into other DA files, so you can build libraries of desk accessories. Click the Open button and select the files desired, or create a new one.

You can also remove DAs from files, including SYSTEM files. Select the DA as if you were going to copy it, but click on the Remove button instead. The DA is deleted from the file. It is lost for good, so if you wish to save it, copy it into another file first.

You can select more than one DA to copy or remove at a time. Click on a name, then drag the mouse up or down while holding the button down to select other names. Or you can hold ⇧ down, then click on individual names to select them. Then copy or remove the whole group with a single operation.

A Few Warnings

You need to be careful of a few things when using the FONT/DA MOVER. First, be sure that you have enough free space (memory) on the disk to make a copy of the desk accessories selected. Under each file display, FONT/DA MOVER shows the amount of free space on the disk. That number must be somewhat greater than the size of the desk accessory(ies) being copied. If you cut it too close, the copy operation fails.

Before removing desk accessories, you should have backup copies. Be very careful about removing some of the standard DAs (such as Scrapbook and Control Panel) that are crucial to using the Macintosh.

CHAPTER 14

Debugging Your Turbo Pascal Program

The term "debugging" comes from the early days of computers, when actual bugs (moths and the like) sometimes clogged up the machinery. Nowadays, it means correcting errors in a program.

You'll undoubtedly have bugs to contend with — errors of syntax, semantics, and logic within your program — and you'll have to fix them by trial and error. However, there are tools and methods to make it less of a trial and to cut down on the errors. In this chapter, we'll look at common errors and at different methods of debugging these errors.

Compiler Errors

The first type of error is a syntax or compiler error: You forget to declare a variable, you pass the wrong number of parameters to a procedure, you assign a *Real* value to an *Integer* variable. In other words, you are writing Pascal statements that don't follow the rules of Pascal. Pascal has strict rules, especially compared to other languages, so once you've cleaned up your syntax errors, most of your debugging is done.

Turbo Pascal won't compile your program, that is, generate machine code, until all your syntax errors are gone. If Turbo Pascal finds a syntax error while it is in the process of compiling your program, Turbo Pascal stops the compilation,

goes into your program, locates the error, positions the cursor there, and brings up a window at the top of the screen telling you what the error was. Once you've corrected it, you can start compiling again.

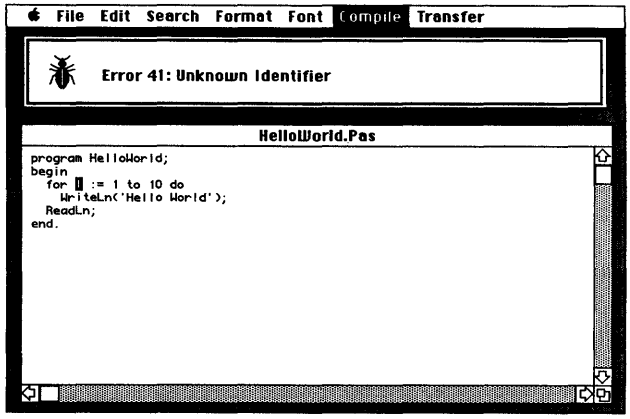


Figure 14-1 *Turbo Pascal Error Box*

You can even check for syntax errors without generating machine code at all, by using the Check Syntax command in the Compile menu. This goes through the same process as if you were compiling your program; however, no machine code is produced, even if there are no errors.

Run-time Errors

Another type of error that can occur is a run-time (or semantic) error. This happens when you compile a legal program, but then try to do something illegal while executing it, such as open a nonexistent file for input or divide an integer by 0. When this happens, Turbo Pascal brings up a Macintosh system error box that tells you an error has occurred and gives you the choice of restarting the system (essentially the same as turning your computer off, then on again) or resuming to find where in your program the run-time error occurred.

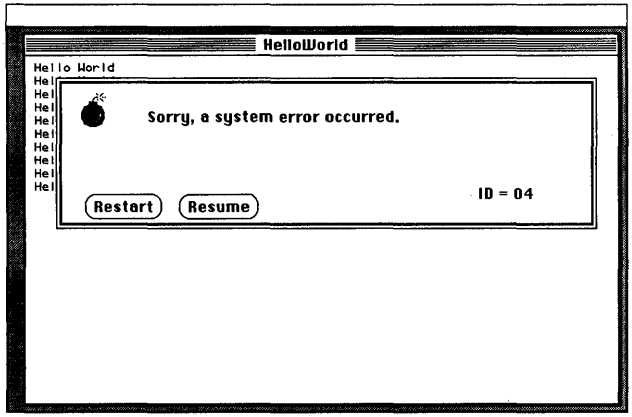


Figure 14-2 Mac System Error Box

The second choice, when available, is preferable. Turbo Pascal again takes you inside your program, shows you where the error occurred, and tells you what the error was. As with a syntax error, you can then correct the error, recompile, and run the program again.

Input/Output Error Checking

This type of error is covered in Chapter 5 (pages 41–43), but it is briefly described here.

Suppose you run a program that prompted for and read in two integer values. Instead of entering integer values, however, you type in a *Real* value (that is, a number with a decimal point). Your program then halts with a Mac system error box.

Turbo Pascal allows you to disable automatic I/O error checking and test for it yourself within the program. To turn off I/O error checking at some point in your program, include the compiler directive {$I-}. This instructs the compiler not to produce code that checks for I/O errors (and that brings up the Mac system error box when one does occur).

Range Checking

Range checking is also covered in Chapter 5 (pages 43–44).

Another common class of semantic errors involves out-of-range or out-of-bounds values. Examples include assigning too large a value to an integer variable, or trying to index an array beyond its bounds. If you want it to, Turbo Pascal will generate code to check for range errors. It makes your program somewhat larger and slower, but it can be invaluable in tracking down any range errors in your program.

You turn range checking on using a compiler directive (see Appendix C for more on compiler directives). Insert {$R+} at the start of your program.

You can leave range checking on all the time by placing {$R+} at the start of each program you write. Or, you can selectively implement range checking by placing the {$R+} directive at the start of the code that needs it, then placing the {$R-} directive at the end of the code.

Invoking Your Own Run-time Errors

Suppose your program is getting stuck somewhere, or the system error box doesn't let you select the Resume option, or (worse yet) the system error box never comes up. What if enabling range checking doesn't track down the error? How do you find it?

You can create your own run-time errors if you want, by calling the Mac routine *SysError*. To do so, you must use the units *MemTypes, QuickDraw,* and *OSIntf* — all of which you'll probably use if you're writing a Mac-style application. You then just call *SysError* with an integer value, such as

```
...
if <some condition> then SysError(32);
...
```

When *SysError* is called, a system error box appears, just as it does for regular run-time errors. You can then select the Resume button, which puts you back into your program in Turbo Pascal.

Theoretically, *SysError* accepts any integer value. In practice, it acts funny with negative values or with positive values above 99. The value 32 is a good one, since it doesn't conflict with any currently defined system error (see Volume II, Chapter 12, of *Inside Macintosh*).

Tracing Errors

A tried-and-true debugging practice is to insert trace statements within your program. A trace statement is usually just a statement that writes something to the screen, telling you where you are and listing some current values. Often it's set up to execute only if a global boolean variable has been set to *True*, so that you can turn tracing on or off.

Suppose you have a large program in which some variables are being set to wrong (but not necessarily illegal) values. The program consists of several procedures, but you haven't been able to figure out so far which one has been causing the problem. You might do something like this for each procedure:

```
procedure ThisOne{any parameters});
{ any declarations }
begin
  if Trace
     then WriteLn('start of ThisOne:  A = ',A,'  B = ',B);
  { rest of procedure ThisOne }
  if Trace
     then WriteLn('end of ThisOne:    A = ',A,'  B = ',B)
end; { of proc ThisOne }
```

This code assumes that *Trace* is a global variable of type *Boolean*, and that you somehow set it to *True* or *False* at the start of the program. It also assumes that *A* and *B* are parameters to *ThisOne* or global variables of some sort.

If *Trace* is *True*, then each time *ThisOne* is called, it writes out the values of *A* and *B* just after it is called and again just before it returns to where it was called from. By putting similar statements in other procedures, you can trace the values of *A* and *B* and find out where and when they change to the undesired values.

Once the wrong values of *A* and *B* come out in a trace statement, you know that the changes occurred somewhere before that statement but after the previously executed one. You can then start moving those two trace statements closer together, or you can insert additional trace statements between the two. By doing this, you can eventually pinpoint where the error is happening and take appropriate steps.

Using a Debugger (MACSBUG)

Sometimes, none of these approaches work. The nature of the problem(s) is such that either you can't track down where the errors are, or having located them, you can't figure out why they're occurring or what's causing them. It's time to call in the heavy artillery: a debugger.

A debugger is a program designed to allow you to trace the execution of your program step by step, one instruction at a time. There are many varieties of debuggers, but most require that you be familiar with assembly-language (machine code) instructions and with the architecture (registers, memory map, and so on) of your computer's microprocessor.

Turbo Pascal comes with such a debugger, known as MACSBUG. To use MACSBUG, you just copy it into the SYSTEM folder on your boot-up disk. Then, when you boot up your Macintosh, MACSBUG will automatically be loaded into memory. The statement `MACSBUG loaded` appears right beneath the `Welcome to Macintosh` greeting that appears when you start your system.

If it doesn't load, make sure that the file's name is MACSBUG. The operating system looks for that name specifically. This means, of course, that you can control whether or not it is loaded by renaming the file; a common "don't load this" name is MAXBUG. If you want to load or unload MACSBUG, then you have to reboot (which you can do by selecting the Shut Down option in the desktop's Special menu, or by turning the Mac off, then on again).

Invoking MACSBUG

Once MACSBUG is loaded, you have three ways of invoking it, which you typically want to do from within your program. First, while your program is running, press the Interrupt switch on the left side of your Macintosh, assuming that you have such a switch installed. If you do have that switch, note carefully that it actually has two buttons, one closer to the front and the other behind it. The one in front is the Reset switch, which causes your Macintosh to act as though you had turned the power off and then on again: It is a switch of last resort. The switch in back is the Interrupt switch. Press that one to get into MACSBUG.

The second method is automatic: If MACSBUG is installed, then it is invoked whenever you have a run-time (system) error. This is true only if the error actually occurred (such as a division by zero). If you produce the error yourself with a call to *SysError*, then the system error box appears as it normally does.

The third means of invoking MACSBUG is to call it directly from within your program. Put the following statement somewhere in the declaration portion of your program, that is, after the program statement but before start of the main body of the program itself:

```
procedure MACSBUG; inline $A9FF;
```

You can now get into MACSBUG by calling it:

```
...
if <some condition> then MACSBUG;
...
```

This causes the MACSBUG display to come up, and you can go on from there.

To help you debug your programs, Turbo Pascal can include the names of your procedures and functions in the resulting machine code. It doesn't normally do this, because of the extra space required, but it will if you ask it to, using the {$D+} compiler directive. Just place the directive at the start of your program:

```
{$D+}
program Whatever;
...
```

The information is compiled and saved for MACSBUG's use.

The MACSBUG Display

When you invoke MACSBUG, the screen goes blank, and you'll get a display like this:

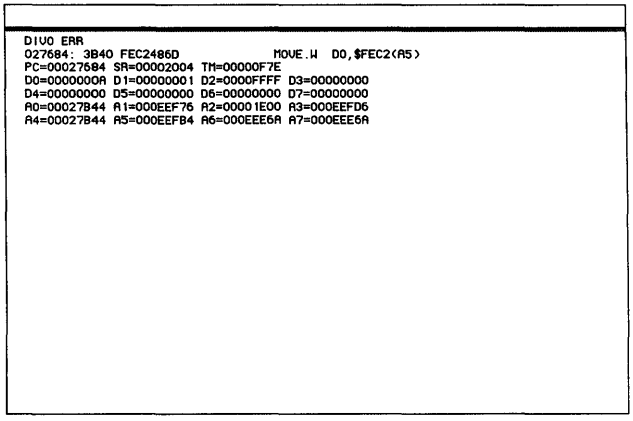


Figure 14-3 *The MACSBUG Screen*

Don't be intimidated by the display; it's actually rather easy to understand. The first line indicates what (if any) made MACSBUG come up, such as `DIVO ERR` (divide-by-zero error) or `USERBRK` (called directly from within your program). If you've brought MACSBUG up by pressing the Interrupt switch, then no message is given.

The second line shows where your program was when MACSBUG was invoked. It gives the address in base-16 (hexadecimal), then shows you the machine code instruction in disassembled form. This means that it has converted the machine code from binary format to something you can read, such as `ORI.B #$02,(A0)`. (You might not find that particularly legible, but it beats trying to

interpret 00103202, the machine-code equivalent.) This instruction, by the way, is the next to be executed, which is why its address is the same as the contents of the program counter (PC).

The next line shows the contents of two important registers (special storage locations) of the 68000 microprocessor. The PC is where the 68000 keeps the address of the next instruction it wants to execute. The Status Register (SR) is where the 68000 sets and clears indivdual bits (0s and 1s) to keep track of certain information. The third value, TM, keeps track of the number of instructions executed.

The final four lines show the contents of the 68000's data and address registers. There are eight of each, D0..D7 and A0..A7. A7 serves a special purpose as the stack register.

If you're completely lost at this point, don't worry. What you need to do is to find a good reference manual on the 68000 processor, so that you can become familiar with what all these things mean. There are several available; one you might consider is *The 68000, 68010, 68020 Primer* by Stan Kelly-Bootle and Bob Fowler, published by Howard W. Sams & Co.

MACSBUG Commands

We'll assume that you have become reasonably familiar with the 68000 processor, at least to the extent that you want to push ahead. We'll also assume that you have put MACSBUG onto your boot disk and rebooted, so that it has been loaded into memory. Type in the following program:

```
{$D+}
program Test;
var
  I,Sum   : Integer;

procedure MACSBUG; inline $A9FF;
begin { main body of program Test }
  Write('press return to invoke MACSBUG');
  ReadLn;
  MACSBUG;
  Sum := 0;
  for I := 1 to 10 do
     Sum := Sum + I;
  WriteLn('The sum is ',Sum)
end. { of prog Test }
```

Now, compile and run it. You'll get the prompt `Press return to invoke MACSBUG`. Press ↵. The MACSBUG display has now come up, and you're ready to start debugging.

Table 14-1 follows showing all the MACSBUG commands, after which the commands are discussed in some detail.

Table 14-1 Summary of MACSBUG Commands

G	continue execution from PC
EA	relaunch application
ES	relaunch FINDER
RB	reboot
DM ⟨adr⟩ ⟨#bytes⟩	display memory
IL ⟨adr⟩ ⟨#lines⟩	disassemble memory
SM ⟨adr⟩ ⟨v1⟩ ⟨v2⟩ ⟨v2⟩	change memory
F ⟨adr⟩ ⟨#bytes⟩ ⟨val⟩ ⟨mask⟩	find value
TD	display all registers
PC [val]	display (or change) program counter
SR [val]	display (or change) status register
D0 [val] (ditto for D1..D7)	display (or change) data register
A0 [val] (ditto for A0..A7)	display (or change) address register
S	single-step executing
T	single step; execute traps completely
MR	single step; execute subroutines
GT ⟨adr⟩	execute until PC = ⟨adr⟩
ST ⟨adr⟩	like GT, but ⟨adr⟩ can be in ROM
BR ⟨adr⟩	set breakpoint
CL	clear all breakpoints
CS ⟨adr1⟩ ⟨adr2⟩	set up checksum area
CS	test checksum area
SS ⟨adr1⟩ ⟨adr2⟩	interrupt on change to checksum area
WH ⟨trap⟩	locate trap
WH ⟨adr⟩	find trap near ⟨adr⟩
AT ⟨t1⟩ ⟨t2⟩ ⟨a1⟩ ⟨a2⟩ ⟨d1⟩ ⟨d2⟩	log calls to traps in trap range
AR ⟨t1⟩ ⟨t2⟩ ⟨a1⟩ ⟨a2⟩ ⟨d1⟩ ⟨d2⟩	log last call to trap in trap range
AB ⟨t1⟩ ⟨t2⟩ ⟨a1⟩ ⟨a2⟩ ⟨d1⟩ ⟨d2⟩	break on any call to trap in range
AS ⟨t1⟩ ⟨t2⟩ ⟨a1⟩ ⟨a2⟩	break on memory change from trap call
AX	clear all trap commands
HD	dump heap
HT	show heap totals
HS ⟨t1⟩ ⟨t2⟩ ⟨a1⟩ ⟨a2⟩	scramble heap on trap call

The first MACSBUG command you need to learn is G for *Go*, which causes your program to continue execution. It restores the Mac's display first, so that it appears as though you had never stopped. However, be aware that G means "Go from where I am right now." For example, if you've changed the PC so that it points to some other area of memory, that's where execution starts. Likewise, if

you've changed register values or memory locations, the program continues with those changed values.

Press G now and watch yourself go back into your program. Once your program is done, run it again and get back into MACSBUG.

There are similar commands to exit MACSBUG. The EA command restarts your program, so that you'll exit MACSBUG and begin to execute your program again. The ES command gets you back to the FINDER (desktop) or to Turbo Pascal if you executed your program in memory. The RB command reboots the entire system, that is, it acts as if you had turned your Mac off and then on again.

The next few commands have to do with displaying the contents of your computer memory. The values stored in memory can be interpreted as either data (information to be acted upon) or instructions (what to do with data). Part of a program's job is to keep track of which values are which, so that it doesn't try to execute data or manipulate instructions (although both those actions can be done).

Similarly, when you display memory, you have to decide if you want to see it as data or as instructions. The Dump Memory (DM) command shows memory as data. Specifically, it gives you a list of 4-digit hexadecimal (base-16) values, each value representing a 16-bit word. The format of this command is

```
>DM  addr  #bytes
```

The command `DM 1E3F0 80` displays 128 bytes (64 words), starting at location 1E3F0. Why 128 bytes? Because MACSBUG assumes all values are in hexadecimal, unless you specifically request otherwise by putting an ampersand (&) in front of the number. If you actually want only 80 bytes, you have to put either `DM 1E3F0 50` (since 50 hex equals 80 decimal) or `DM 1E3F0 &80` (to specify a decimal value).

The DM command displays the contents of memory in two forms: first as hex values, and then at the end of the line as ASCII characters. If you are dumping memory that has text in it, you can easily detect and read that text. The command displays eight words (16 bytes) on each line.

For the address, MACSBUG accepts a variety of expressions. You can, of course, put an absolute address there, as shown above. You can also use values in registers. For example, the command `DM PC 30` displays three lines (48 words altogether), starting at the address found in the program counter. Likewise, `DM TEST+30 100` displays 10 lines of data, beginning 30 bytes after the start of your program TEST, while `DM RA0 20` shows data at the address contained in register A0.

What if the memory you want to examine has instructions instead of data in it? Do you have to disassemble (decode) those instructions yourself? No, you can use the I L command, which decodes the instructions for you. The format is a little different from that of the D M command:

```
>IL addr #lines
```

The command `IL 1E3F0 20` disassembles and displays 32 (20 is in hex, remember?) instructions, starting at location 1E3F0. You don't have to specify the number of lines; the default is 16. For that matter, you don't have to specify the starting address, either. MACSBUG keeps track of the memory location you're looking at, so that if you just type `IL`, it disassembles the next 16 instructions, and so on. In fact, you can just press ↵, and MACSBUG executes the I L command again. This is also true for D M and several other MACSBUG commands. As with D M, you can use registers (PC, A0, and so on) and symbols (TEST), along with offsets (+50, −100), to express the address.

What if you want to change memory? The SM command allows you to set the contents of specific memory locations as data. The format is

```
SM addr val1 val2 val3 val4...
```

This stores the indicated values at the indicated address. The values can be bytes, words (2 bytes), or longwords (4 bytes); MACSBUG stores them appropriately. If you want to enter instructions instead of data, you have to "hand assemble" them yourself; MACSBUG can't convert them from assembly-language instructions to machine code for you, though other commercial debuggers (such as TMON) can.

What if you're looking for a given value in memory somewhere, but you don't know where it is? You can then use the Find (F) command. The format is

```
>F  addr #bytes value mask
```

The *addr* is the address you want to start at, while *#bytes* indicates how many bytes to examine. The *value* is what you're looking for; as with the S M command, it can be a byte, a word, or a longword. The *mask* allows you even more precision by specifying which bits in each location being examined to ignore. For example, you might search for a particular command, such as L E A (Load Effective Address). That instruction has the bit pattern 0100xxx111xxxxxx, where the x's represent information about which registers are involved, and so on. You might then use the following instruction:

```
>F  TEST 100 41C0 F1C0
```

With this command, MACSBUG searches the range from TEST to TEST+256 for the L E A instruction, ignoring all the x bits in the pattern shown above.

Having looked at memory, you might want to go back and look at the registers again. The T D command brings up the same display you had when you first entered MACSBUG.

You can also display the contents of a given register by typing its name: PC, SR, A0 through A7, and D0 through D7. If you type a value after its name (such as "D3 10FA"), then that value is stored in that register.

To debug a program, you not only want to be able to examine and change memory, but to execute instructions. The most useful technique is to execute one instruction at a time and see what changes with each step. This is known as single-stepping or tracing.

The basic single-step command is S. If you enter this command, MACSBUG executes the next instruction that the program counter (PC) is pointing at, then shows you the next instruction and the register contents. You'll note that the screen flashes when you do execute this command; that's because MACSBUG momentarily switches back to your program's regular screen display. As with D M and I L, you can just press ↵ each time to repeat the S command.

There is one potential problem with using the S command. What if the instruction you execute calls a Toolbox or operating system routine? You then have to single-step through that entire routine before you get back to your program. Obviously, this could be very tedious. The answer is to use the T command. When MACSBUG encounters a call (known as a "trap") to a Toolbox or OS routine, it just executes the routine without single-stepping. It then stops when it gets back to your program, so that the routine ends up looking like a single instruction.

A similar case can occur within your own program. What if you're stepping through some instructions and come across a call to one of Turbo Pascal's subroutines, or even one of your own? You may not want to single-step all the way through it. The solution is to use (for that instruction only) the M R command. This lets MACSBUG go to that subroutine, execute it, return to where you were, and put you back into single-step mode.

Let's take this one step farther. What if you are stepping through your program and want to quickly get through some code? You can then use the Go Until (G T) command. The syntax is

```
>GT addr
```

Your program then executes without stopping until the desired address is reached, at which point MACSBUG is again invoked. Notice that your program's regular display comes up again until MACSBUG is re-invoked. A variation of this instruction, Step Until (S T), allows the address to be in ROM, that is, within the Toolbox and OS routines.

Let's make things even more complicated. What if there are several points in your program where you want to stop and reenter MACSBUG? You can set breakpoints, which are (as you might guess) locations in your program at which MACSBUG is automatically invoked. (The call to MACSBUG in the sample program can be thought of as a type of breakpoint.) To set a breakpoint, type

```
>BR addr
```

where *addr* is once again a standard address expression (absolute, register, procedure name, with or without offset). Once you've set breakpoints, you can see where they are by pressing B R without an address. To clear a breakpoint, you use the CL Addr (C L) command to clear an individual breakpoint, or just press C L to clear them all.

Just as you can have MACSBUG stop or be re-invoked if and when a given instruction is executed, you can also have MACSBUG check if any memory location within a given range is changed. This is done using a *checksum*, which is simply the sum of all the locations in that range. If the checksum changes, then some location has been changed. Be aware, though, that if two locations are changed in certain ways — for example, if their values are swapped — the checksum remains the same, and MACSBUG won't be able to detect the modification.

The command C S is for use during single-step debugging. To set things up, use this format:

```
>CS  addr1 addr2
```

MACSBUG then computes the checksum in the memory locations from *addr1* to *addr2* and remembers it. As you trace through your program, you can recheck that range by pressing C S. If any locations have been changed (that is, the checksum is different), then MACSBUG prints `CHKSUM F`; otherwise, it prints `CHKSUM T`.

The second command, S S, is for use during multiple-step debugging. It uses the same syntax, namely:

```
>SS addr1 addr2
```

You can now press G, G T, S T, or whatever you want. The effect of this is that a C S command is done after every instruction is executed. As you might imagine, this makes your program run very, very, very slowly; it should be used only in desperate circumstances or if you're really bored.

As mentioned before, calls to the Macintosh Toolbox and OS routines are known as traps. This is because they're implemented using a special 68000 instruction called a trap. These instructions always have a hex value of $Axxx and (theoretically) range from $A000 to $AFFF. In reality, most of the Mac traps have values of $A0xx, $A1xx, $A8xx, and $A9xx. You might have noticed that

MACSBUG is smart enough to detect those traps with the [I][L] command and print out the name of the corresponding routine on the same line.

First, if you're not sure which trap is which, use the [W][H] command. It has a varied syntax. The command

```
>WH  trap
```

where *trap* is a trap's name or number, returns the trap's vector, address, and name. If you type

```
>WH addr
```

then MACSBUG looks for the trap nearest to and preceding that address. This last version doesn't refer to calls to that trap, but to where the routine itself is actually located.

There are several MACSBUG commands to help you determine what traps your program is calling. The most useful one is [A][T], which has this syntax:

```
>AT trap1 trap2 addr1 addr2 d01 d02
```

The parameters are as follow:

trap1	low end of trap range
trap2	high end of trap range
addr1	low end of address range
addr2	high end of address range
D01	low end of D0 value
D02	high end of D0 value

The [A][T] command checks for a range of traps; Appendix C in Volume 3 of *Inside Macintosh* contains a complete list. If you only want to check for one trap, then you can repeat its name for *trap2*. Note that *trap1* ⟨= *trap2*. Also, you can use the trap names if you want, instead of the hex values. *Trap2* may be left off if you aren't specifying an address or D0 range.

Addr1 and *addr2* specify a range in which to check for the trap values. Note that *addr1* ⟨= *addr2*. These follow the usual address conventions and may be left off if you aren't specifying a D0 value range. Likewise, *D01* and *D02* specify a range of values in the D0 register and are optional.

Suppose you compile and run TEST, get into MACSBUG, then enter the command

```
>AT SETPORT GETPORT TEST TEST+300
```

Now press [G] to rerun your program. The screen flashes as MACSBUG reappears each time a trap in the range is called. When you finally get back into MACSBUG, you'll see a display that looks something like this:

```
A874 GETPORT PC:00413270 A0:00019B3A D0:00000000 TM:000001EB
A873 SETPORT PC:00413274 A0:000D0F72 D0:00000000 TM:000001EC
```

The first value is the trap number; the second, the trap name; the third, the memory location where the trap was called; the fourth and fifth, the contents of registers A0 and D0; the sixth, the time (in ticks).

What if you don't care about all the calls to those traps except for the last one before you get back into MACSBUG? Press [A][R], which has the same syntax. When you run your program, there'll be no flashing or other breaking in. Instead, when you get back into MACSBUG, press [A][R]. MACSBUG then gives you the information shown earlier on the last trap call, as well as dumping part of the stack.

What if you want to stop and drop immediately into MACSBUG whenever a given trap (or range of traps) is called? Use the [A][B] command. As with [A][T] and [A][R], you can specify memory range and D0 value range. Now, as soon as one of the specified traps is called (within the desired memory range and so on), MACSBUG is immediately invoked and stops at that location.

The [A][S] command works like a trap-oriented version of the [S][S] command, which checks for changes within a given memory range. Its syntax is

```
>AS  trap1 trap2 addr1 addr2
```

However, *addr1* and *addr2* do not refer to where the traps are; instead, they refer to the area to be checked for modifications. Each time a trap (located anywhere) in the range *trap1..trap2* is called, the memory area *addr1..addr2* is checked. If the checksum has changed, MACSBUG is invoked, and you get the same trap display as with previous commands.

You can only have one trap command ([A][T],[A][R],[A][B],[A][S]) active at any one time. To clear the existing command, type [A][X].

There are also a few commands for examining the heap. A discussion of these commands and the heap itself is beyond the scope of this chapter, but here are the actual commands:

```
>HD                          dumps the heap to the screen
>HT                          shows only heap totals
>HS trap1 trap2 addr1 addr2  scrambles heap if traps called
```

There are quite a few reference books on debugging Macintosh programs. One is *How to Write Macintosh Software*, written by Scott Knaster and published by the Hayden Book Company's Apple Press. It has additional information on MACSBUG, TMON, and writing Macintosh programs in general.

CHAPTER 15

The Turbo Pascal Menu Reference

This chapter is designed to help you quickly review all the menu commands available in Turbo Pascal. You'll first learn how to select menu commands, then review all the menus and what each one does. Finally, you'll go through each menu in detail.

Selecting a Menu Command

Menu commands can be selected two ways. First, you can use the "point-press-drag-release" method. Point to the menu by moving the cursor with the mouse. Press the mouse button down and hold it down; the menu's commands appear below the menu name. Drag the cursor down to the command you want; that command now appears as white text on a black bar. Release the mouse button; the command blinks a few times, the menu commands disappear, and the command is executed. Any menu command can be selected using this method.

The second approach, available for most commands, is the command-key method. The command key, with the ⌘ (cloverleaf) symbol on it, sits just to the left of the space bar on the bottom row of your keyboard. If you pull a menu, you'll notice that some commands have a command-key equivalent listed: the ⌘ symbol, followed by some letter or character. You can select those commands

by holding ⌘ down while you type the specified letter or symbol. For example, to select the New command from the File menu, hold down ⌘ and type the letter *N*.

The Menu Bar

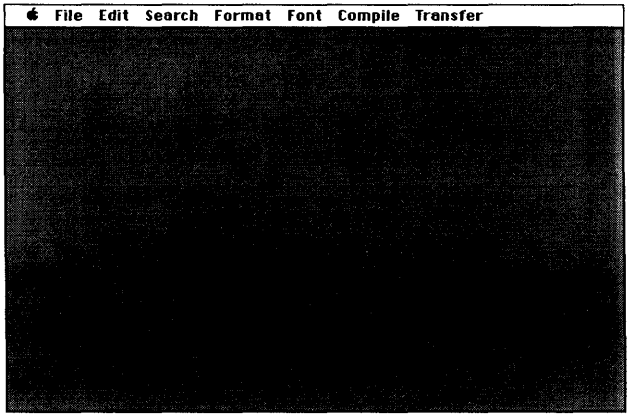


Figure 15-1 *The Menu Bar*

The menu bar in Figure 15-1 shows you Turbo Pascal's eight menus:

- **Apple**—Use this menu to bring up Macintosh desk accessories while working in Turbo Pascal.
- **File**—Use this menu to open, save, and close programs and other text files that you edit with Turbo Pascal. You can also print files, save options, transfer to other programs, and exit Turbo Pascal with this menu.
- **Edit**—Use this menu to perform editing and formatting commands, and to set certain options.
- **Search**—Use this menu to search for given strings and, if desired, to replace them with others. It also allows you to bring the cursor home and to cycle through your editing windows.
- **Format**—Use this menu to organize your editing windows (if you have more than one open) and to select the font size within a given window.
- **Font**—Use this menu to choose the font to be used in each editing window.
- **Compile**—Use this menu to compile and run your programs, to get information about a given program, and to set certain options.

- **Transfer**—Use this menu to exit Turbo Pascal by transferring directly to one of a specific list of programs; you can edit this list.

Having reviewed the basic functions of each menu, let's discuss each menu's commands in depth.

The Apple Menu

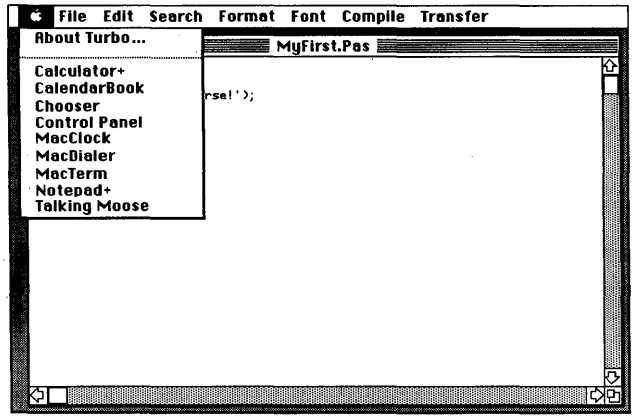


***Figure 15-2** The Apple Menu*

This is a standard menu found in most Mac applications, and it is always the first menu on the menu bar. It has two parts: the About Turbo... command and the current list of desk accessories. Since that list varies according to your system file, the example shown in Figure 15-2 may not match what you see when you pull your menu down.

About Turbo...

You can't select any of the commands in the Apple menu using the ⌘. All must be selected with the mouse.

The About Turbo... command brings up a window in the middle of the screen, giving the Turbo Pascal version number and copyright notice. Press ↵ or click the mouse button to make it go away.

Desk Accessories

The desk accessory commands activate the different desk accessories (DAs). Once activated, most DAs continue to run until you get rid of them (usually by clicking on the Close box in the upper left corner of the DA's window). It's a good idea to close any DAs that you are no longer using.

The File Menu

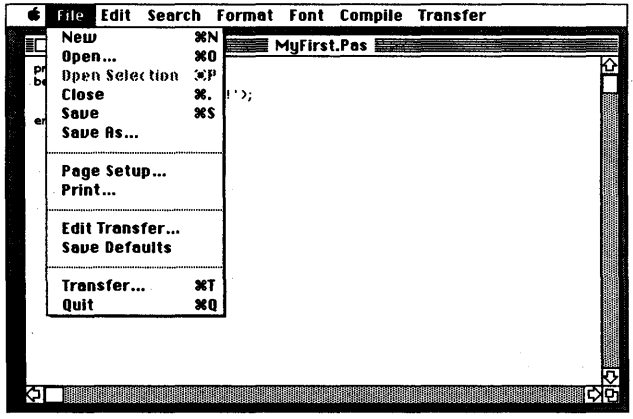


Figure 15-3 The File Menu

The File menu is concerned primarily with reading and writing programs (and other data) from and to the disk. The commands fall into three major groups: accessing files, printing files, and exiting Turbo Pascal. For more details on these commands, review Chapter 3.

New ⌘ N

Opens a new ("Untitled") window. That window becomes the current editing window. If eight editing windows (the maximum) are already open, this command is disabled.

Open ⌘O

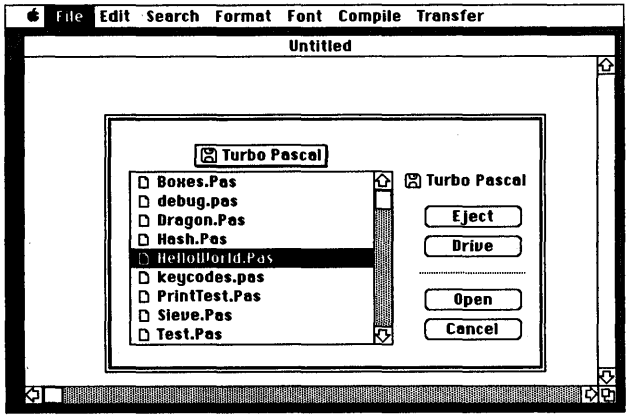


Figure 15-4 *The Mac File Selector*

Brings up the standard Mac file selector that allows you to select a file for editing. Disabled if eight windows are already open.

Open Selection ⌘P

Attempts to open a file whose name matches the currently selected text in the current editing window. Primarily used for opening include files. Disabled if no text is selected or if eight windows are already open.

Close ⌘.

Closes the current editing window. If the contents of that window have been changed since it was last saved to disk, a dialog box lets you choose to save or not save your changes before closing, or to cancel the command. Disabled if no windows are open.

Save ⌘S

Saves the contents of the current editing window out to disk. If that window isn't associated with a file (that is, it is "Untitled"), it brings up the standard file-name selector, allowing you to enter the file name. Disabled if no windows are open.

Save as... (no command equivalent)

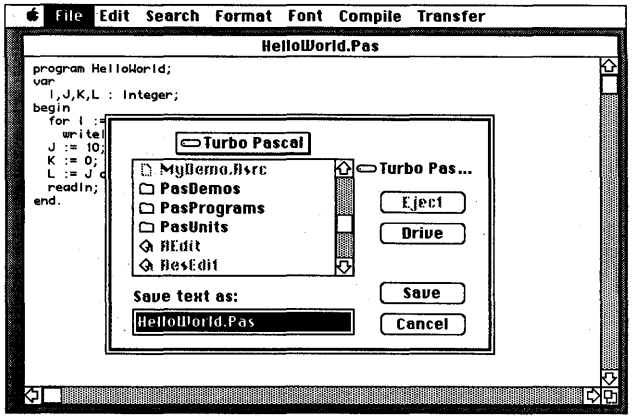


Figure 15-5 *The File-Name Selector*

Brings up the standard file-name selector, whether or not the current editing window is associated with a file. Saves the window's contents out to that file and associates the window with that file. Disabled if no windows are open.

Page Setup... (no command equivalent)

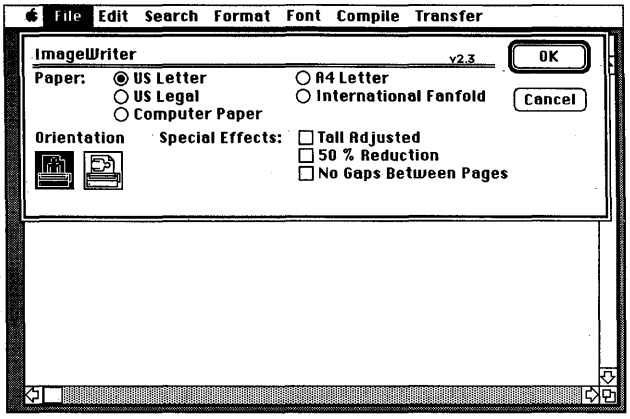


Figure 15-6 *The Page-Setup Dialog Box*

Brings up the standard page-setup dialog box. Any changes made are erased when you exit Turbo Pascal. Disabled if no windows are open.

Print... (no command equivalent)

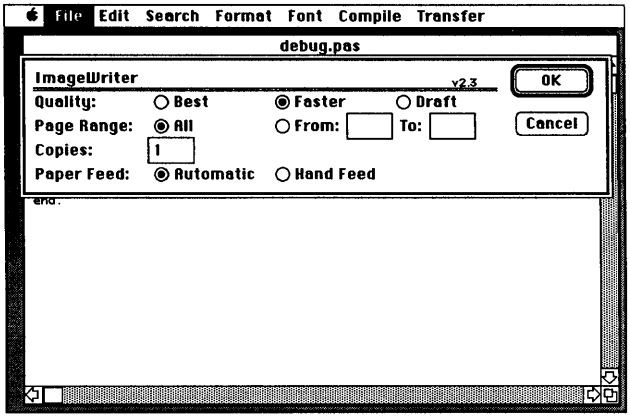


Figure 15-7 *The Printing Dialog Box*

Brings up the standard printing dialog box. Disabled if no windows are open.

Edit Transfer... (no command equivalent)

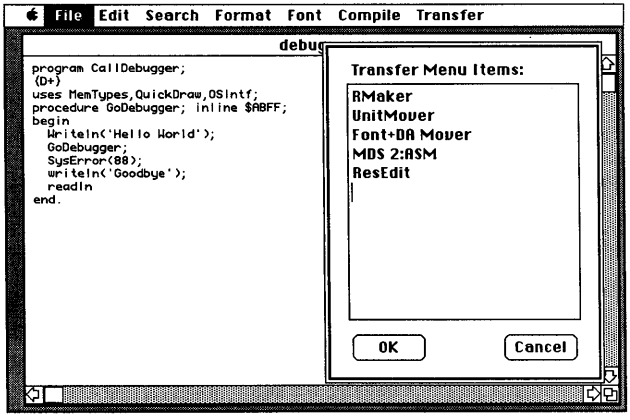


Figure 15-8 *The Edit Transfer Dialog Box*

Allows you to edit the list of programs found in the Transfer menu. (Not to be confused with the Transfer command in this menu; see next page.)

Save Defaults (no command equivalent)

Saves to disk any changes made using the Options commands in the Edit and Compile menus. Otherwise, any changes made are erased once you exit Turbo Pascal.

Transfer ⌘T

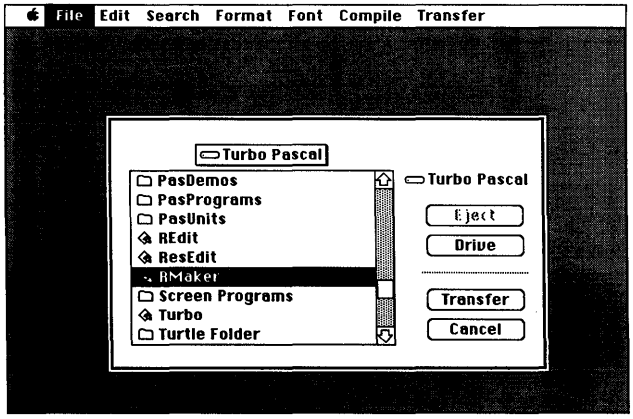


Figure 15-9 *The Transfer File Selector*

Closes all open editing windows, allowing you to verify whether changes made to each should be saved. Brings up the standard file selector, but only lists applications (executable programs). If you select one, it exits Turbo Pascal and executes that program without going back to the desktop first.

Quit ⌘Q

Closes all open editing windows, allowing you to verify whether changes made to each should be saved. Exits back to the desktop.

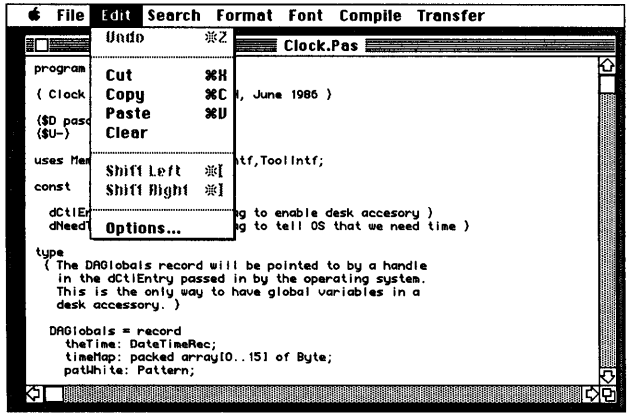


Figure 15-10 The Edit Menu

The Edit menu contains editing and formatting commands. The commands are discussed in greater detail in Chapter 3.

Undo ⌘ Z

Attempts to undo the last editing command or action you performed. You can undo the last undo performed, that is, restore the window contents to what they were before you did the first undo command. Disabled when not applicable.

Cut ⌘ X

Cuts the currently selected text, deleting it from the editing window and saving it onto the Clipboard (a temporary storage area). Disabled when no text is selected.

Copy ⌘ C

Copies the currently selected text onto the Clipboard, but does not delete it from the editing window. Disabled when no text is selected.

Paste ⌘V

Copies the contents of the Clipboard into the current editing window at the cursor's location. If text in the window has been selected, replaces that text with the Clipboard's contents. Disabled if the Clipboard is empty and no window is open.

Clear (no command equivalent)

Deletes the currently selected text from the editing window without saving it to the Clipboard. Disabled when no text is selected.

Shift Left ⌘[

Shifts the currently selected text one space to the left. Ignored if any part of the selected text is already at the left margin of the editing window. Disabled when no text is selected and when the selection starts or ends with a partial line.

Shift Right ⌘]

Shifts the currently selected text one space to the right. Disabled when no text is selected and when the selection starts or ends with a partial line.

Options (no command equivalent)

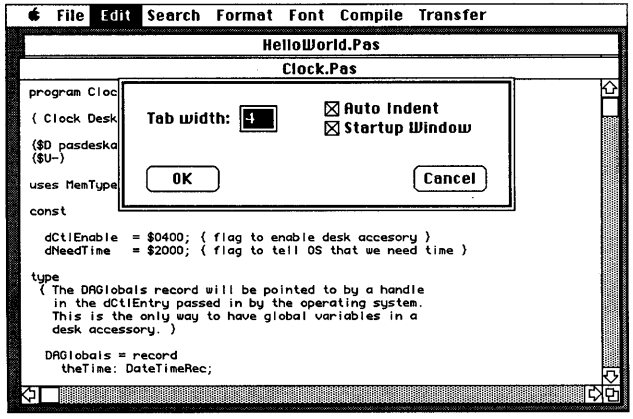


Figure 15-11 *The Options Dialog Box*

Allows you to set the tabulator width and to enable or disable the auto-indent and Startup Window options. The tabulator width must be between 1 and 8. When auto-indent is enabled, each new line (created by pressing ↵) is automatically indented the same number of spaces as the line above. When Startup Window is enabled, Turbo Pascal automatically creates an "Untitled" window when started. The changes only affect the current session unless you use the Save Defaults command in the File menu to save them as the new defaults.

The Search Menu

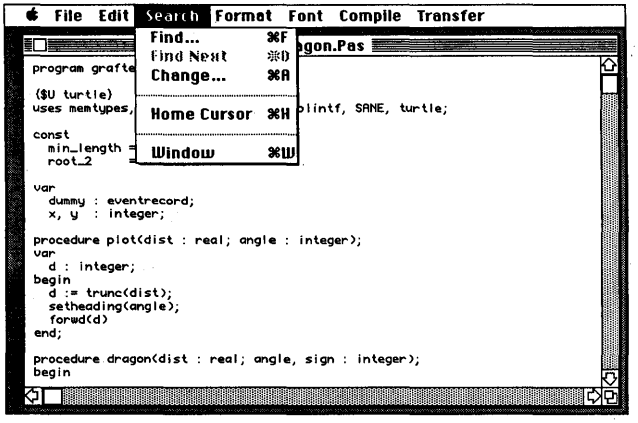


Figure 15-12 *The Search Menu*

The Search menu allows you to search for strings (and, if desired, replace them), to move the cursor home, and to select the current editing window. These commands are discussed in more detail in Chapter 3.

Find... ⌘F

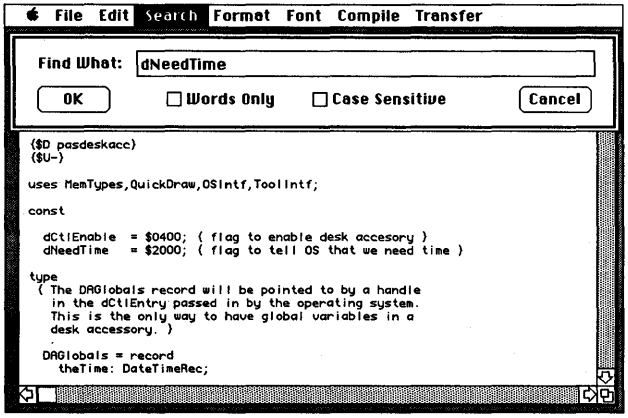


***Figure 15-13** The Find Dialog Box*

Presents a dialog box that lets you enter the string to find and to select options for Words Only and Case Sensitive. The default search string is the currently selected text (if any). The search starts from the current location of the cursor. Disabled if no windows are open.

Find Next ⌘D

Attempts to locate the next occurrence of the search string entered via the Find or Change command, starting at the current location of the cursor. Disabled if no windows are open or if no search string has been entered.

Change... ⌘A

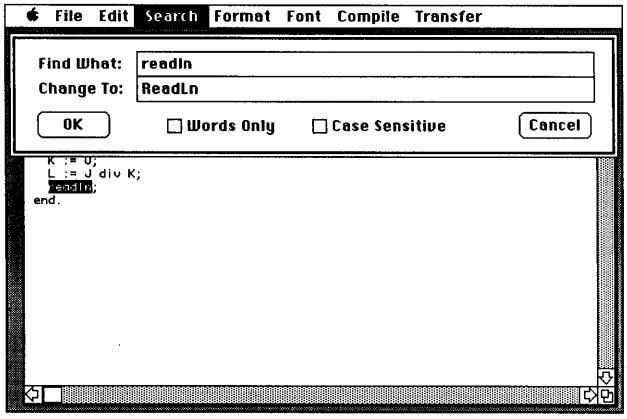


Figure 15-14 *The Change Dialog Box*

Presents a dialog box that lets you enter both the string to find and the string with which to replace it, and to select options for Words Only and Case Sensitive.

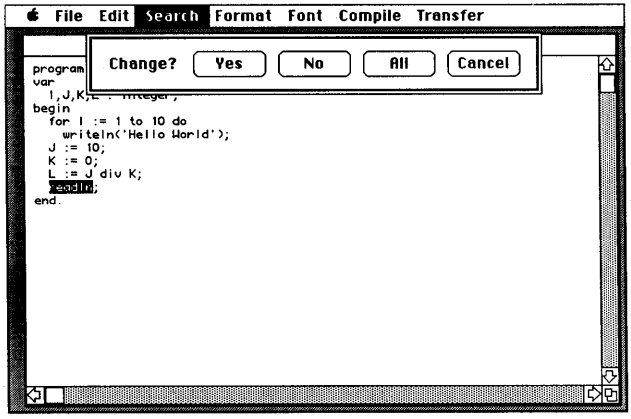


Figure 15-15 *The Verification Box*

Each replacement is verified, with Yes, No, All, and Cancel options. Command-key equivalents are Y, N, A, and C. Disabled if no windows are open.

Home Cursor ⌘H

Moves cursor to the top of the currently active editing window and displays that portion of the window. Disabled if no windows are open.

Window ⌘W

Cycles through all opened editing windows, making each successive one the current editing window each time this command is selected. Disabled when no windows are open.

The Format Menu

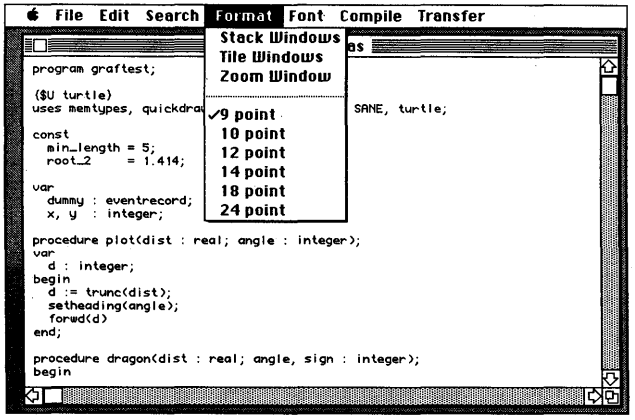


Figure 15-16 *The Format Menu*

The Format menu has two major functions: to organize windows and to select text size. Its commands are discussed in Chapter 3. None of the commands in this menu have any command-key equivalents.

Stack Windows

Organizes the editing windows into a stack, that is, with the current editing window in the front and all other windows stacked behind it with only their title bars showing. Disabled when no windows are open.

Tile Windows

Organizes the editing windows into tiles, that is, shrinks the windows so that all can fit onto the screen at the same time. Disabled when no windows are open.

Zoom Window

Expands the current editing window so that it takes up most of the screen. If the current window is already at full size, shrinks it back down. Corresponds to double-clicking on the window's title bar. Best used with the Tile Windows command. Disabled when no windows are open.

Character Sizes
9, 10, 12, 14, 18, and 24 points

Selects the character size to use in the current editing window. If no windows are open, selects the default character size for new windows. Each window may have its own font size.

The Font Menu

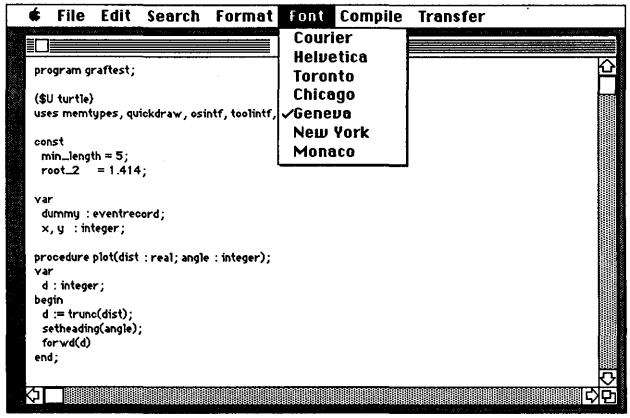


Figure 15-17 *A Typical Font Menu*

The Font menu allows you to select the character font for the text in the current editing window. If no windows are open, you may select the default character font for new windows. The list of fonts available depends upon your system file; Figure 15-17 shows a typical Font menu.. Each window may have its

own font type, that is, not all windows have to use the same font. These commands are never disabled, and none of the commands have command-key equivalents.

The Compile Menu

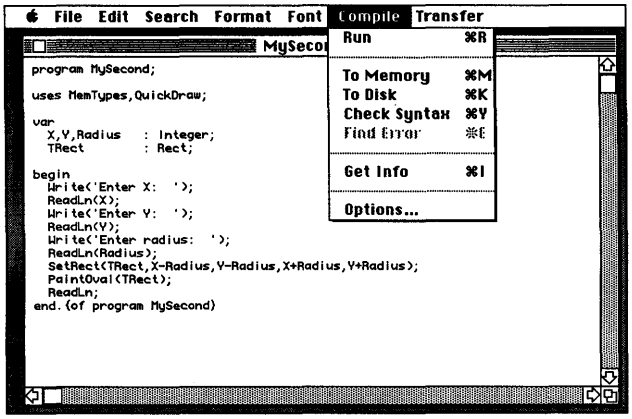


Figure 15-18 *The Compile Menu*

The Compile menu allows you to compile and execute your Turbo Pascal programs. The Compile menu commands are discussed in more detail in Chapter 4.

Run ⌘ R

Executes the program in the current editing window. If needed, compiles the program (to memory) first. Disabled when no windows are open.

To Memory ⌘ M

Compiles to memory the program in the current editing window. The code file is disposed of if you in any way edit the text or exit Turbo Pascal. Disabled when no windows are open.

To Disk ⌘K

Compiles to disk the program in the current editing window. The code file is saved as a clickable application. Disabled when no windows are open.

Check Syntax ⌘Y

Compiles the program in the current editing window without producing 68000 machine code; checks for any syntax errors. Disabled if no windows are open.

Find Error ⌘E

Positions the cursor at the statement that caused the last run-time error or, if that statement can't be found, at the beginning of the text. Disabled if no error has occurred or no windows are open.

Get Info ⌘I

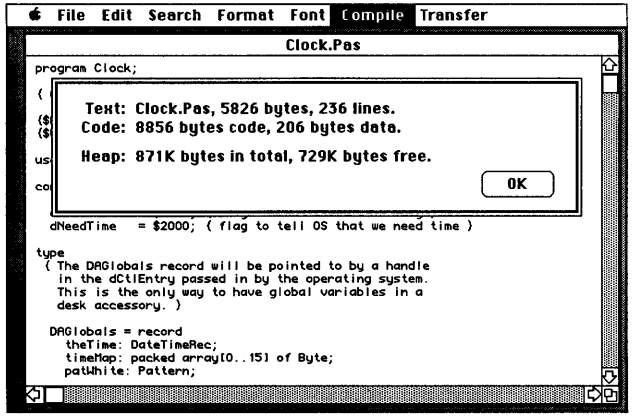


Figure 15-19 *The Get Info Box*

Displays information about the program in the current editing window: text size, number of lines, code size, and data size. Disabled if no windows are open.

Options... (no command equivalent)

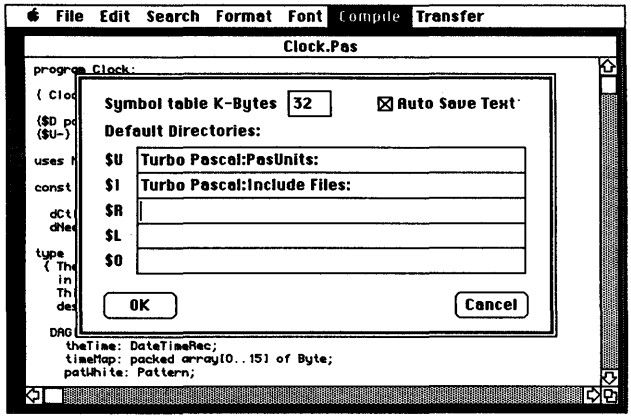


Figure 15-20 *The Compile/Options Dialog Box*

Brings up the compiler options dialog box. Allows you to enable or disable the auto-save option, set symbol table size, and specify default directories for the $U, $I, $R, $L, and $O compiler directives.

The Transfer Menu

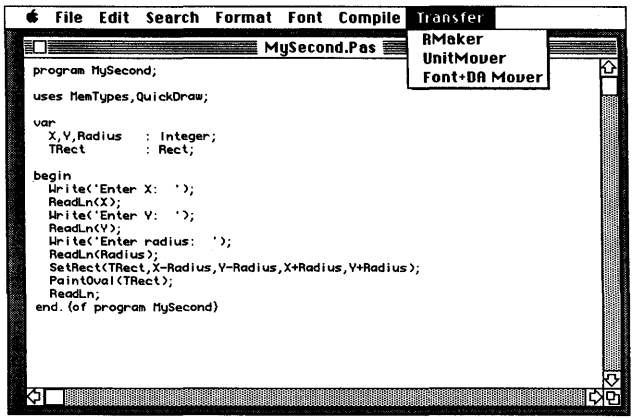


Figure 15-21 *The Transfer Menu Default Setup*

The Transfer menu contains a list of applications to which you can directly transfer without having to go to the FINDER. Figure 15-21 shows the default

setup for the Transfer menu: RMAKER, UNITMOVER, and FONT/DA MOVER. The last application is normally called FONT/DA MOVER, but the slash has a special meaning within a menu item, and so the name here, as well as the actual file name on the disk, must be changed to accommodate that.

You can edit the list of applications by selecting the Edit Transfer command in the File menu. This brings up a dialog box with the current Transfer Menu list; you then modify the list using standard editing methods.

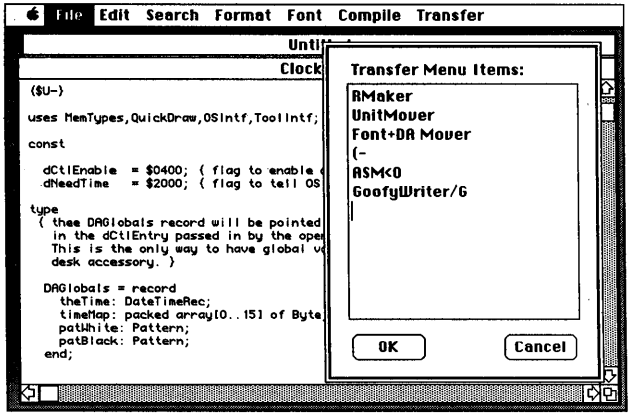


Figure 15-22 *Transfer Menu Dialog Box*

When you edit the Transfer menu, a number of meta-characters are available:

";"	Separates multiple items on a single line.
"/"	Item has a command-key equivalent.
"<"	Item has a special character style.
"("	Item is disabled.

A slash (/) followed by a character associates that character with the application, allowing the application to be invoked from the keyboard with that command key. Remember to specify the character in uppercase if it's a letter, and not to specify other shifted characters or numbers. A less-than symbol (<) followed by a character specifies a special character style for the item. Five stylistic variations are available: B (bold), I (italic), U (underline), O (outline), and S (shadow). The text

```
GoofyWriter>B/G
```

defines an application called *GoofyWriter*, which is displayed in boldface and can be invoked by ⌘G. For a dividing line between the applications, use (-, which specifies a disabled dotted line.

Once you are done and have saved the list, the Transfer menu displays the new list of applications. Take care, however, that any name you add to the list is that of an actual application. If it isn't, selecting it will cause an error box to appear at the top of the screen with a `File not found` message. If you define any new command-key equivalents, make sure they don't conflict with command keys Turbo Pascal has already defined. Also, remember that any changes you make to the Transfer menu will be lost when you exit Turbo Pascal unless you select the Save Defaults command after making those changes.

PART II

Reference Section

CHAPTER 16

Tokens and Constants

Tokens are the smallest meaningful units of text in a Pascal program, and they are categorized as special symbols, identifiers, labels, numbers, and character strings.

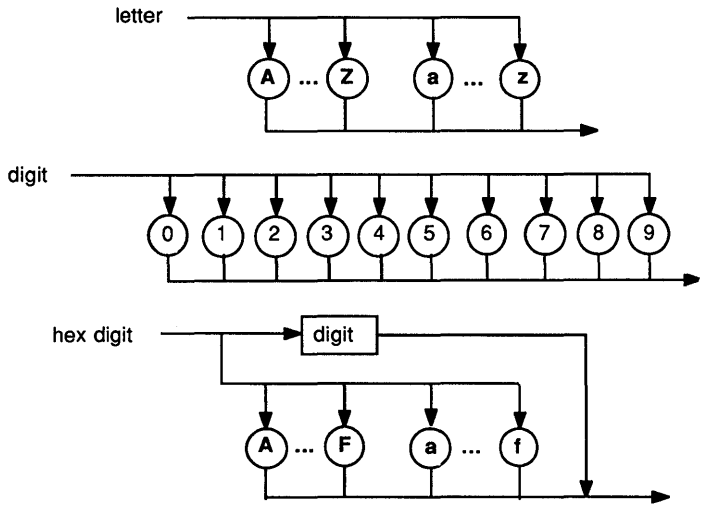
A Pascal program is made up of tokens and separators, where a separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators if each token is a reserved word, an identifer, a label, or a number.

Separators cannot be part of tokens, except in character strings.

Special Symbols and Reserved Words

Turbo Pascal uses the following subsets of the ASCII character set:

- *Letters*—the English alphabet, A through Z and a through z.
- *Digits*—the Arabic numerals 0 through 9.
- *Hex digits*—the Arabic numerals 0 through 9, the letters A through F, and the letters a through f.
- *Blanks*—the space character (ASCII 32), and all ASCII control characters (ASCII 0 to 31), including the end-of-line or return character (ASCII 13).



Special symbols and reserved words are symbols that have one or more fixed meanings. These single characters are special symbols:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { } $ #
```

These character pairs are also special symbols:

```
<>  <=  >=  :=  ..  (*  *)  (.  .)
```

Some special symbols are also operators. A single bracket,[, is equivalent to the character pair (.; similarly,] is equivalent to the character pair .).

Following are Turbo Pascal's reserved words:

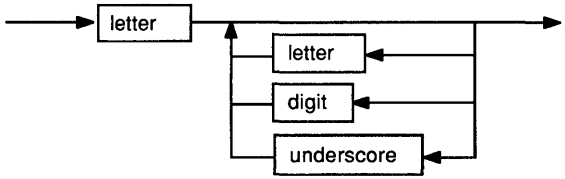
and	else	inline	procedure	type
array	external	interface	program	unit
begin	file	label	record	until
case	for	mod	repeat	uses
const	forward	nil	set	var
div	function	not	shl	while
do	goto	of	shr	with
downto	if	or	string	xor
implementation	otherwise	then		
in	packed	to		

Reserved words appear in lowercase **boldface** throughout this manual. Turbo Pascal isn't case sensitive, however, so you can use either uppercase or lowercase letters in your programs.

Identifiers

Identifiers denote constants, types, variables, procedures, functions, units, programs, and fields in records. An identifier can be of any length, but only the first 63 characters are significant.

label identifier,
constant identifier,
type identifier,
field identifier,
variable identifier,
procedure identifier,
function identifier,
program identifier,
unit identifier



underscore

An identifier must begin with a letter and may not contain spaces. Letters, digits, and underscore characters (ASCII $5F) are allowed after the first character. Like reserved words, identifiers are not case-sensitive.

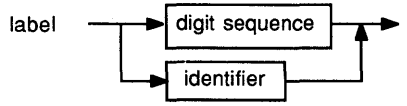
Here are some Turbo Pascal standard identifiers:

ClearEOL
GotoXY
Exit
StringTOReal
WriteLn

In this manual, standard identifiers are italicized when they are referred to in text.

Labels

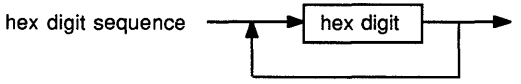
A label is a digit sequence whose value ranges from 0 to 9999. Leading zeros are not significant. Labels are used with **goto** statements.

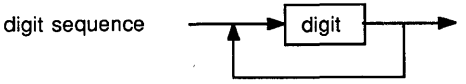


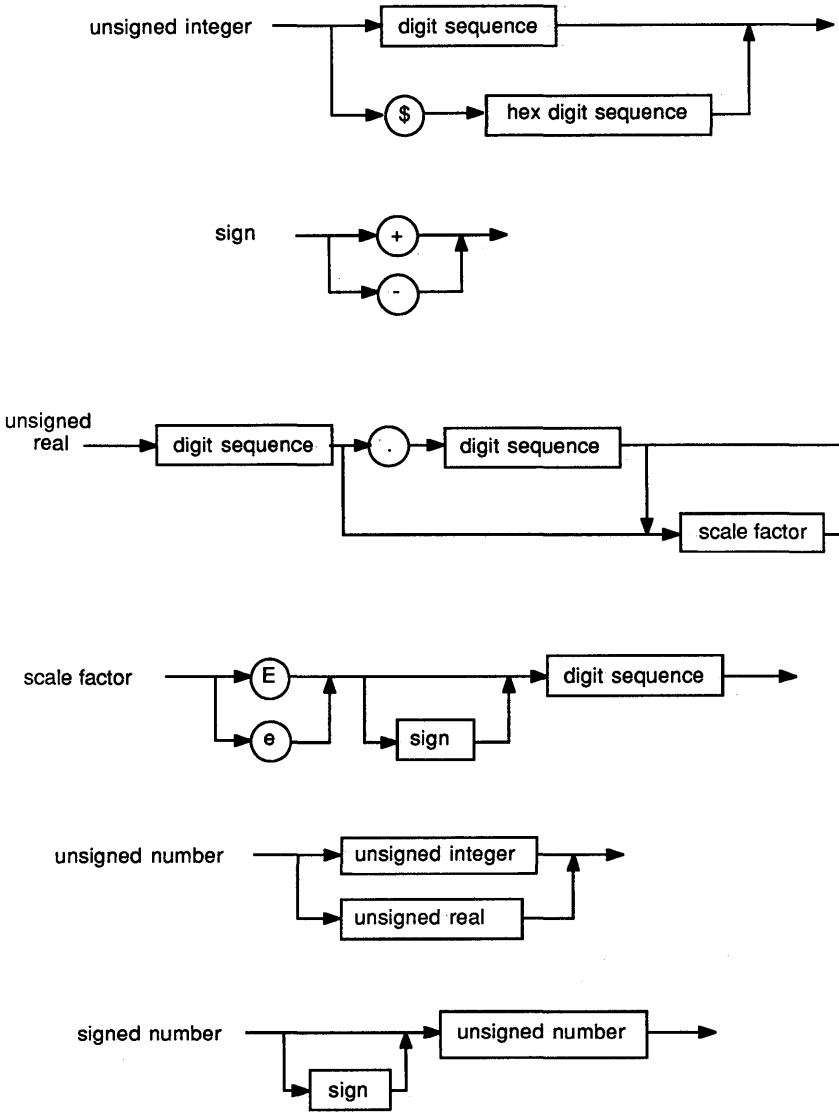
As an extension to Standard Pascal, Turbo Pascal also allows identifiers to function as labels.

Numbers

Ordinary decimal notation is used for numbers that are constants of the data types *Integer*, *LongInt*, *Real*, *Single*, *Double*, *Extended*, and *Comp*. A hexadecimal integer constant uses *$* as a prefix. Engineering notation (*E* or *e* followed by an exponent) is read as "times ten to the power of " in real-types. For example, 7E−2 means 7×10^{-2}; 12.25e+6 or 12.25e6 both mean $12.25 \times 10^{+6}$. Syntax diagrams for writing numbers follow.







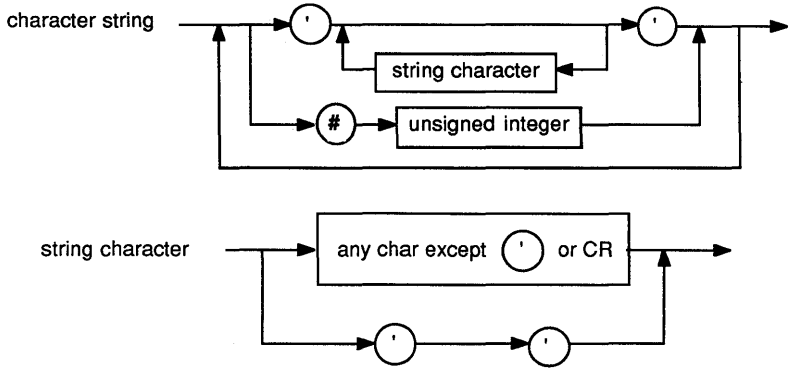
Numbers with decimals or exponents are stored as type *Extended* (unless explicitly assigned to a variable of another real-type). Other decimal numbers are stored as type *Integer* or *LongInt* as needed for that value.

A one-to-four digit hexadecimal constant is stored as an *Integer* (2 bytes). A five-to-eight digit constant is stored as a *LongInt* (4 bytes). An integral hexadecimal value with over eight significant digits produces an overflow error. The resulting value's sign is implied by the hexadecimal notation.

Character Strings

A character string is a sequence of zero or more characters from the Macintosh character set (Appendix E) written on one line in the program and enclosed by apostrophes. A character string with nothing between the apostrophes is a null string. Two sequential apostrophes in a character string denote a single character, an apostrophe. The length attribute of a character string is the actual number of characters within the apostrophes.

As an extension to Standard Pascal, Turbo Pascal allows control characters to be embedded in character strings. The # character followed by an unsigned integer constant in the range 0 to 255 denotes a character of the corresponding ASCII value. There must be no separators between the # character and the integer constant. Likewise, if several control characters are part of a character string, there must be no separators between them.



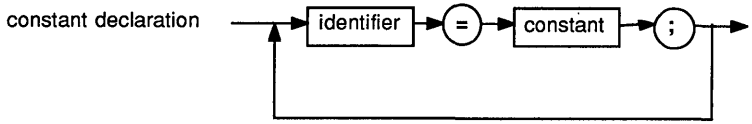
A character string of length zero (the null string) is compatible only with string-types. A character string of length one is compatible with any char-type and any string-type. A character string of length *n*, where *n* is greater than or equal to 2, is compatible with any string-type and with packed-string-types of *n* characters.

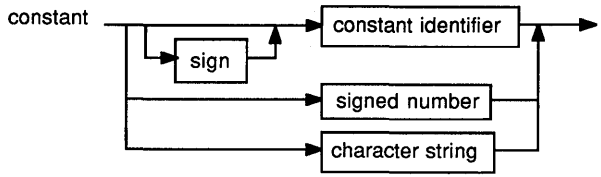
These are examples of character strings:

```
'TURBO'       'You''ll see'      ''''        ';'      ' '
#13#10        'Line 1'#13'Line2'           #7#7'Wake up!'#7#7
```

Constant Declarations

A constant declaration declares an identifier to denote a constant within the block that contains the declaration. A constant identifier may not be included in its own declaration.





A constant identifier following a sign must denote a value of type *Integer*, *LongInt*, *Real*, *Double*, *Extended*, or *Comp*. Real-type constants are stored in *Extended* precision.

Comments

The constructs

```
{ any text not containing right-brace }
 (* any text not containing star-right-paren *)
```

are comments. The compiler ignores them.

A comment that contains $ immediately after the opening { or (* is a compiler directive. A mnemonic of the compiler command follows the $ character. The compiler directives are summarized in Appendix C.

Program Lines

Turbo Pascal program lines have a maximum length of 128 characters.

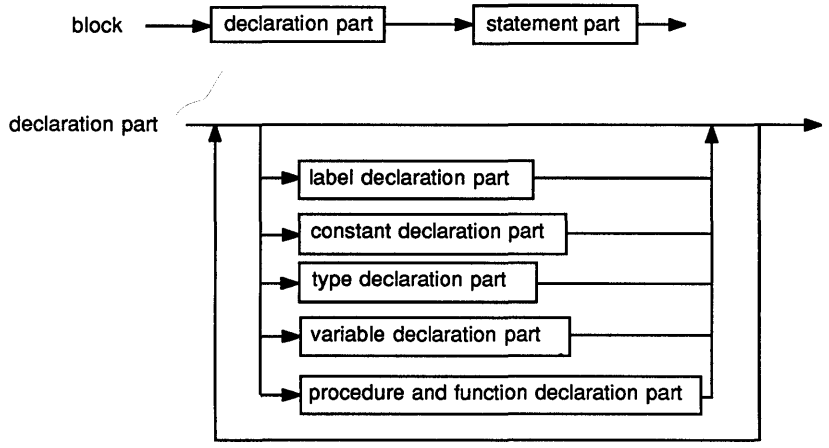
CHAPTER 17

Blocks, Locality, and Scope

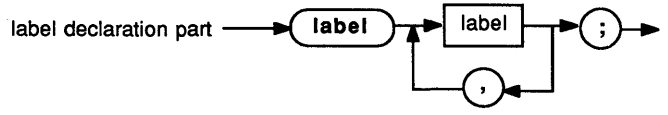
A block is made up of declarations, which are written and combined in any order, and statements. Each block is part of a procedure declaration, a function declaration, or a program or unit. All identifiers and labels declared in the declaration part are local to the block.

Syntax

The overall syntax of any block follows this format:



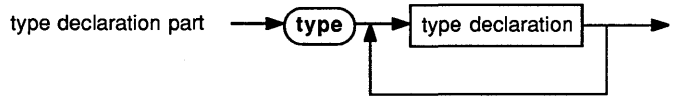
The *label declaration part* is where all labels that mark statements in the corresponding statement part are declared. Each label must mark only one statement.



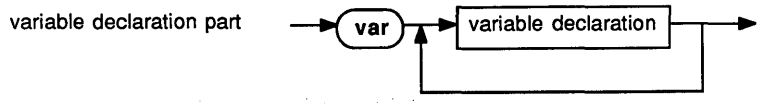
The digit sequence used for a label must be in the range 0 to 9,999.

The *constant declaration part* consists of all constant declarations local to the block.

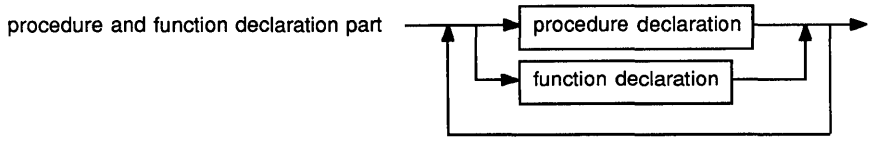
The *type declaration part* includes all type declarations local to the block.



The *variable declaration part* is composed of all variable declarations local to the block.



The *procedure and function declaration part* comprises all procedure and function declarations local to the block.



The *statement part* defines the statements or algorithmic actions to be executed by the block when an *activation* occurs.

Rules of Scope

The presence of an identifier or label in a declaration defines the identifier or label. Each time that the identifier or label occurs subsequently, it must be within the *scope* of this declaration. The scope of an identifier or label basically encompasses its declaration to the end of the current block, including all blocks enclosed by the current block. Exceptions follow.

Redeclaration in an Enclosed Block

Suppose that Exterior is a block that encloses another block, Interior. Any identifier declared in Exterior with a further declaration in Interior excludes Interior and all its blocks from Exterior's scope of declaration.

Position of Declaration Within Its Block

Identifiers and labels cannot be used until after they are declared. An identifier or label's declaration must come before any occurrence of that identifier or label in the program text, with one exception.

The base type of a pointer type can be an identifier that has not yet been declared. However, the identifier must eventually be declared in the same type declaration part that the pointer type occurs in.

Redeclaration Within a Block

An identifier or label can only be declared *once* in the outer level of a given block except if it is declared within a contained block or is in a record's field-list.

A record field identifier is declared within a record type and is significant only in combination with a reference to a variable of that record type. So, you can redeclare a field identifier (with the same spelling) within the same block, but not at the same level within the same record type. However, an identifier that has been declared can be redeclared as a field identifier in the same block.

Identifiers of Standard Objects

Turbo Pascal equips you with a set of predefined constants, types, procedures, and functions whose identifiers function as if they were declared in a block enclosing the whole program. Their scope is the entire program.

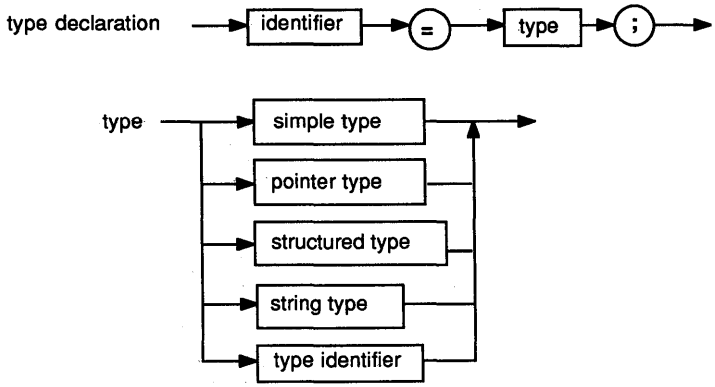
Scope of Interface Identifiers

Programs or interface-parts containing **uses** clauses are provided the identifiers belonging to the units in the **uses** clauses. These identifiers act as if they were declared in a block enclosing the whole program.

CHAPTER 18

Types

When you declare a variable, you must state its *type*. A variable's type circumscribes the set of values that it can have and the operations that can be performed upon it. A type declaration specifies the identifier that denotes a type.



When an identifier occurs on the left side of a type declaration, it is declared as a type identifier for the block in which the type declaration occurs. A type identifier's scope does not include itself, except for pointer types.

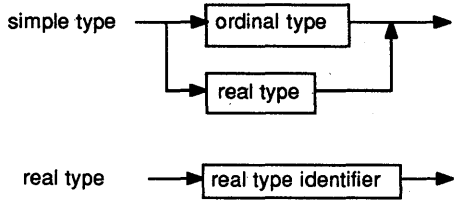
The following list of terms distinguish the seven types of identifiers according to what they denote:

- simple-type
- structured-type
- pointer-type
- ordinal-type
- integer-type
- real-type
- string-type

A simple-type identifier, for example, is declared to denote a simple-type variable, and so on.

Simple-Types

Simple-types define ordered sets of values.



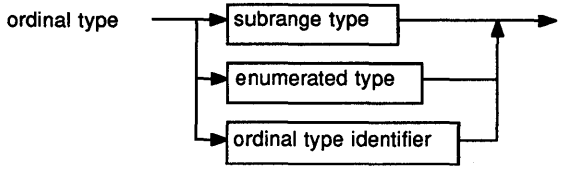
An integer-type identifier is one of the standard identifiers *Integer* or *LongInt*. A real-type identifier is one of the standard identifiers: *Real*, *Single*, *Double*, *Extended*, or *Comp*. See "Numbers" and "Character Strings" in Chapter 16 for how to denote constant integer-type and real-type values.

Ordinal-Types

Ordinal-types are a subset of simple-types. All simple-types other than real-types are ordinal-types, which are set off by the following four characteristics.

- All possible values of a given ordinal-type are an ordered set, and each possible value is associated with an *ordinality*, which is an integral value. Except for integer-type values, the first value of every ordinal-type has ordinality 0, the next has ordinality 1, and so on for each value in that ordinal-type. An integer-type value's ordinality is the value itself. In any ordinal-type, each value other than the first has a predecessor, and each value other than the last has a successor based on the ordering of the type.
- The standard function *Ord* can be applied to any ordinal-type value to return the ordinality of the value.
- The standard function *Pred* can be applied to any ordinal-type value to return the predecessor of the value. If applied to the first value in the ordinal-type, *Pred* produces an error.
- The standard function *Succ* can be applied to any ordinal-type value to return the successor of the value. If applied to the last value in the ordinal-type, *Succ* produces an error.

The syntax of an ordinal-type follows.



Turbo Pascal has four predefined ordinal-types: *Integer*, *LongInt*, *Boolean*, and *Char*. In addition, there are two other classes of user-defined ordinal-types: enumerated-types and subrange-types.

The Integer-Type

Integer-type values are a subset of the whole numbers. An integer-type variable can have a value within the *-maxint-1* to *maxint* range, that is, −32,768 to 32,767. The standard *Integer* constant *maxint* is defined as 32,767. The range encompasses 16-bit, two's-complement integers.

The LongInt-Type

Longint-type values are also a subset of the whole numbers, a larger subset. A longint-type variable can have a value within the *-maxlongint-1* to *maxlongint* range. The standard *LongInt* constant *maxlongint* is defined as +2,147,483,647. The range encompasses 32-bit, two's-complement integers.

Arithmetic operations with integer-type operands uses *Integer* (16-bit) or *LongInt* (32-bit) precision according to the following rules:

- Integer constants in the range of type *Integer* are considered to be of type *Integer*. Other integer constants are considered to be of type *LongInt*.
- When both operands of an operator (or the single operand of a unary operator) are of type *Integer*, 16-bit precision is used, and the result is of type *Integer* (truncated to 16-bits if necessary). Similarly, if both operands are of type *LongInt*, 32-bit precision is used, and the result is of type *LongInt*.
- When one operand is of type *LongInt*, and the other is of type *Integer*, the *Integer* operand is converted to *LongInt*, 32-bit precision is used, and the result is of type *LongInt*.
- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left.

An *Integer* value may be explicitly converted to *LongInt* (and vice versa) through type casting. Type casting is described in Chapers 19 and 20.

The Boolean-Type

Boolean-type values are denoted by the predefined constant identifiers *False* and *True*. Because *Boolean* is an enumerated-type, these relationships hold: *False* < *True*; *Ord(False)* = *0*; *Ord(True)* = *1*; *Succ(False)* = *True*; and *Pred(True)* = *False*.

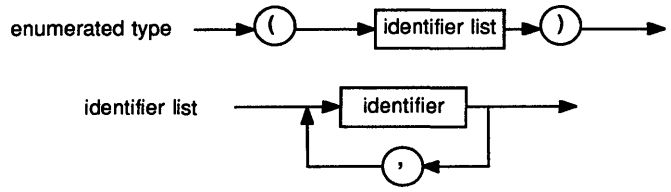
The Char-Type

This type's set of values are characters, ordered according to the ordering of the Macintosh character set (Appendix F). The function call *Ord(Ch)*, where *Ch* is a *Char* value, returns *Ch*'s ordinality.

A string constant of length 1 can denote a constant *Char* value. Any value of type *Char* can be generated with the standard function *Chr*.

The Enumerated-Type

Enumerated-types define ordered sets of values by enumerating the identifiers that denote these values. Their ordering follows the sequence in which the identifiers are enumerated.



When an identifier occurs within the identifier list of an enumerated-type, it is declared as a constant for the block in which the enumerated-type is declared. This constant's type is the enumerated-type being declared.

An enumerated constant's ordinality is determined by its position in the identifier list in which it is declared. The enumerated-type in which it is declared becomes the constant's type. The first enumerated constant in a list has an ordinality of 0 (zero).

An example of enumerated-type is

```
suit = (club,diamond,heart,spade)
```

Given these declarations, *diamond* is a constant of type *suit*.

When the *Ord* function is applied to an enumerated-type's value, *Ord* returns an integer that shows where the value falls with respect to the other values of the enumerated-type. Given the declarations above, for example, *Ord(club)* returns 0, *Ord(diamond)* returns 1, and so on.

The Subrange-Type

A subrange-type is a range of values from an ordinal-type called the host-type. The definition of a subrange-type specifies the least and the largest value in the subrange. Its syntax is

Both constants must be of the same ordinal-type. Subrange-types of the form *a..b* require that *a* is less than or equal to *b*.

Examples of subrange-types:

```
0..99
-128..127
club..heart
```

A variable of a subrange-type has all the properties of variables of the host-type, but its run-time value must be in the specified interval.

The Real-Type

A real-type has a set of values that is a subset of real numbers, which you can represent in floating-point notation with a fixed number of digits. A value's floating-point notation normally comprises three values—*m*, *b*, and *e*—such that $m \times be = n$, where *b* is always 2 and both *m* and *e* are integral values within the real-type's range. These *m* and *e* values further prescribe the real-type's range and precision.

Arithmetic with real-type values includes results that floating-point notation can't handle, such as dividing 0 by 0. Chapter 26 discusses the methods you can use for such specialized calculations.

There are four kinds of real-types: *Real*, *Double*, *Extended*, and *Comp*. In addition, the type *Single* is identical to the type *Real*. The real-types differ in the range and precision of values they hold:

Table 18-1 Range and Decimal Digits for Real Types

Type	Range	Decimal_Digits
Real	1.5×10^{-45} to 3.4×10^{38}	7 to 8
Double	5.0×10^{-324} to 1.7×10^{308}	15 to 16
Extended	1.9×10^{-4951} to 1.1×10^{4932}	19 to 20

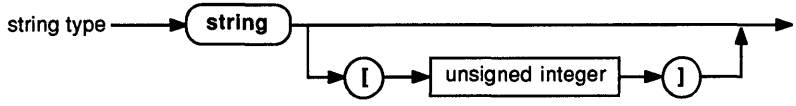
The *Comp* type holds only integral values within the range $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18}.

All real-type values are converted to *Extended* before any operations are performed on them, and the results of such operations are always of type *Extended*. An *Extended* value may always be used where a *Real*, *Double*, or *Comp* value is required, provided that the value (rounded to an integral value in the case of *Comp*) falls within the required range.

Note: Calculations on *Extended* type variables are faster and more compact than other real-type calculations, since the automatic conversion to *Extended* is not required. You may want to declare all real-type temporary variables, formal value parameters, and function results as *Extended* in order to improve execution time and code size.

String-Types

A string-type value is a sequence of characters with a dynamic length attribute (depending on the actual character count during program execution) and a constant size attribute from 1 to 255. A string-type declared without a size attribute is given the default size attribute 255. The length attribute's current value is returned by the standard function *Length*.

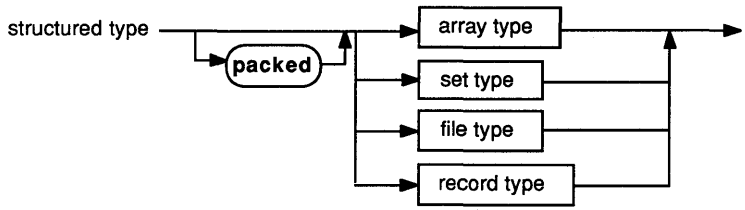


Ordering between any two string values is set by the ordering relationship of the character values in corresponding positions. In two strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a higher or greater-than value; for example, 'Xs' is greater than 'X'. Null strings can only be equal to other null strings, and they hold the least string values.

Characters in a string can be accessed as components of an array as described in "Arrays, Strings, and Indexes" in Chapter 19. String-type operators are described in "String Operators" and "Relational Operators" in Chapter 20. String-type standard procedures and functions are described in Chapter 25.

Structured-Types

A structured-type, characterized by its structuring method and by its component-type(s), holds more than one value. If a component-type is structured, the resulting structured-type has more than one level of structuring. A structured-type can have unlimited levels of structuring.



The word **packed** in a structured-type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a

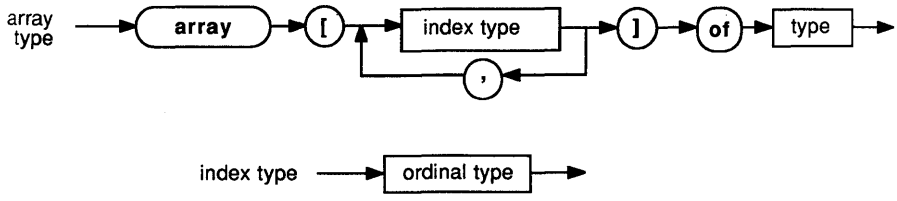
variable of this type. Only the storage of record-types and array-types can be packed. (String-types are always packed.)

Packed only affects the representation of one level of the structured-type it occurs in. If a component is also a structured-type, then for it to be packed, its declaration must also include the word **packed**.

You cannot use components of packed variables as actual variable parameters to procedures or functions.

Array-Types

Arrays have a fixed number of components of one type, the component-type. In the following syntax diagram, the component-type follows the word **of**.



The index-types, one for each (unlimited) dimension of the array, specify the number of elements. The array can be indexed in each dimension by all values of the corresponding index-type; the number of elements is therefore the number of values in each index-type. Arrays may not occupy more than 32,767 bytes in total, and index-types may not be *LongInt* or subranges of *LongInt*.

An example of an array-type is:

```
array[1..100] of Real
```

If an array-type's component-type is also an array, you can treat the result as an array of arrays or as a single multi-dimensional array. For instance,

```
array[Boolean] of array[1..10] of array[Size] of Real
```

is interpreted the same way by the compiler as

```
array[Boolean,1..10,Size] of Real
```

You can also express

```
packed array[1..10] of packed array[1..8] of Boolean
```

as

```
packed array[1..10,1..8] of Boolean
```

You access an array's components by supplying the array's identifier with one or more indexes in brackets (see "Arrays, Strings, and Indexes" in Chapter 19).

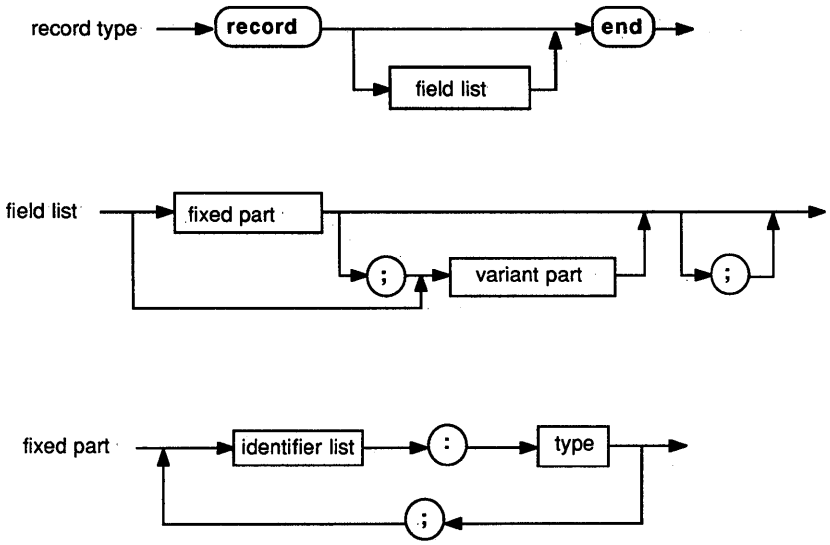
An array-type of the form

```
packed array[1..n] of Char
```

is called a packed-string-type. A packed-string-type has certain properties not shared by other array-types (see "Identical and Compatible Types" later in this chapter).

Record-Types

A record-type comprises a set number of components, or fields, which can be of different types. The record-type declaration specifies the type of each field and the identifier that names the field.

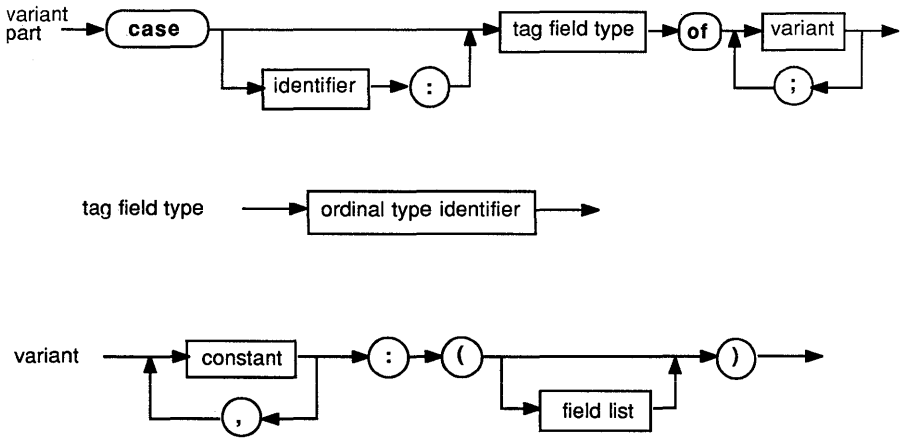


The fixed-part of a record-type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

The following is an example of a record-type:

```
record
  year:  Integer;
  month: 1..12;
  day:   1..31;
end
```

The variant-part shown in the previous syntax diagram distributes memory space for more than one list of fields, so the information can be accessed in more ways than one. Each list of fields is a *variant*. The variants overlay the same space in memory, and all fields of all variants can be accessed at all times.



As you can see from the diagram, each variant is identified by at least one constant. All constants must be distinct and of an ordinal-type that is compatible with the tag-field-type. Variant and fixed fields are accessed the same way.

An optional identifier, the *tag-field identifier*, can be placed in the variant-part. If a tag-field identifier is present, it becomes the identifier of an additional fixed field, the *tag-field*, of the record. The program can use the tag-field's value to show which variant is active at a given time. Without a tag-field, the program selects a variant by another criterion.

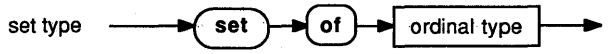
Some record-types with variants follow.

```
record
  firstName,lastName: string[40];
  birthDate: Date;
  case citizen: Boolean of
    true:  (birthPlace: string[40]);
    false: (country: string[20];
            entryPort: string[20];
            entryDate: Date;
            exitDate: Date);
end

record
  x,y: Real;
  case kind: Figure of
    rectangle: (height,width: Real);
    triangle:  (size1,side2,angle: Real);
    circle:    (radius: Real);
end
```

Set-Types

A set-type's range of values is the powerset of a particular ordinal-type (the base-type). Each possible value of a set-type is a subset of the possible values of the base-type.

A variable of a set-type can hold from none to all values of the set.



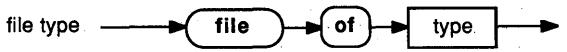
The base-type must not have more than 256 possible values. For that reason, the base-type of a set cannot be *Integer* or *LongInt*. If the base-type of a set is an integer-type subrange, the upper and lower bounds of the subrange must be within the range 0 to 255.

Set-type operators are described in "Set Operators" in Chapter 20. "Set Constructors" in the same chapter shows how to construct set values.

Every set-type can hold the value *[ ]*, called the *empty set*.

File-Types

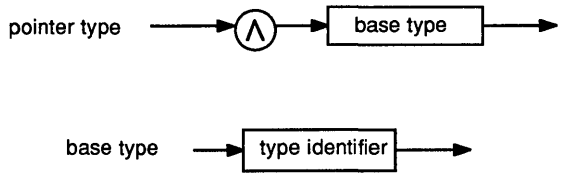
A file-type consists of a linear sequence of components of one type, the component-type, which may be of any type except a file-type or any structured-type with a file-type component. The number of components is not set by the file-type declaration.



The standard file-type *Text* signifies a file containing characters organized into lines. Textfiles use special input/output procedures, discussed in Chapter 24.

Pointer-Types

A pointer-type defines a set of values that point to *dynamic variables* of a specified type called the *base-type*. A pointer-type variable contains the memory address of a dynamic variable.



If the base-type is an undeclared identifier, it must be declared in the same type declaration part as the pointer-type.

You can assign a value to a pointer variable with the *New* procedure, the @ operator, or the *Pointer* function. The *New* procedure allocates a new memory area in the application heap for a dynamic variable and stores the address of that area in the pointer variable. The @ operator directs the pointer variable to the memory area containing any existing variable, including variables that already have identifiers. The *Pointer* function points the pointer variable to a specific memory address.

The reserved word **nil** denotes a pointer-valued constant that does not point to anything.

See Chapter 19 for the syntax of referencing the dynamic variable pointed to by a pointer variable.

Identical and Compatible Types

Two types may be the same, and this sameness (identity) is mandatory in some contexts. At other times, the two types need only be compatible or merely assignment compatible. They are identical when they are declared with, or their definitions stem from, the same type identifier.

Type Identity

Type identity is required only between actual and formal variable parameters in procedure and function calls.

Two types—say, T_1 and T_2—are identical if one of the following is true: T_1 and T_2 are the same type identifier; T_1 is declared to be equivalent to a type identical to T_2.

The second condition connotes that T_1 does not have to be declared directly to be equivalent to T_2. The type declarations

```
T1 = Integer;
T2 = T1;
T3 = Integer;
T4 = T2;
```

result in T_1, T_2, T_3, T_4, and *Integer* as identical types. The type declarations

```
T5 = set of Integer;
T6 = set of Integer;
```

don't make T_5 and T_6 identical, since **set of** *Integer* is not a type identifier. Two variables declared in the same declaration, for example,

```
V1, V2:  set of Integer;
```

are of identical types—unless the declarations are separate. The declarations

```
V1: set of Integer;
V2: set of Integer;
V3 Integer;
V4 Integer;
```

mean V_3 and V_4 are of identical type, but not V_1 and V_2.

Compatibility of Types

Compatibility between two types is sometimes required, such as in expressions or in relational operations. Type compatibility is important, however, as a pre-condition of assignment compatibility. Types compatibility exists when at least one of the following conditions is true:

- Both types are the same.
- Both types are real-types.
- Both types are integer-types.
- One type is a subrange of the other.
- Both types are subranges of the same host-type.
- Both types are set-types with compatible base-types.
- Both types are packed-string-types with an identical number of components.
- One type is a string-type and the other is a string-type, packed-string-type, or char-type.

Assignment Compatibility

Assignment compatibility is necessary when a value is assigned to something, such as in an assignment statement or in passing value parameters.

A value of type T_2 is assignment compatible with a type T_1 (that is, $T_1 := T_2$ is allowed) if any of the following are true:

- T_1 and T_2 are identical types and neither is a file-type or a structured-type that contain a file-type component at any level of structuring.
- T_1 and T_2 are compatible ordinal-types, and the values of type T_2 falls within the range of possible values of T_1.
- T_1 and T_2 are real-types, and the value of type T_2 falls within the range of possible values of T_1.
- T_1 is a real-type, and T_2 is an integer-type.
- T_1 and T_2 are string-types.
- T_1 is a string-type, and T_2 is a char-type.
- T_1 is a string-type, and T_2 is a packed-string-type.
- T_1 and T_2 are compatible packed-string-types.
- T_1 and T_2 are compatible set-types, and all the members of the value of type T_2 fall within the range of possible values of T_1.

A compile or run-time error occurs when assignment compatibility is necessary and none of the above is true.

The Type Declaration Part

Programs, procedures, and functions that declare types have a type declaration part. An example of this part follows:

```
type
  Range       = Integer;
  Number      = Integer;
  Color       = (red,green,blue);
  TestIndex   = 1..100;
  TestValue   = -99..99;
  TestList    = array[TestIndex] of TestValue;
  TestListPtr = ^TestList;
  Date        = record
                  year: Integer;
                  month: 1..12;
                  day: 1..31;
                end;
  MeasureData = record
                  when: Date;
                  count: TestIndex;
                  data: TestListPtr;
                end;
  MeasureList = array[1..50] of MeasureData;
  Name        = string[80];
  Sex         = (male,female);
  Person      = ^PersonDetails;
  PersonData  = record
                    name,firstName: Name;
                    age: Integer;
                    married: Boolean;
                    father,child,sibling: Person;
                    case s: Sex of
                      male:   (bearded: Boolean);
                      female: (pregnant: Boolean);
                  end;
  People  = file of PersonData;
  IntFile = file of Integer
```

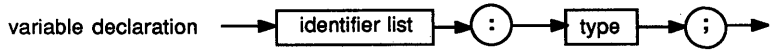
In the example *Range*, *Number*, and *Integer* are identical types. *TestIndex* is compatible and assignment compatible with, but not identical to, the types *Number*, *Range*, and *Integer*.

C H A P T E R 19

Variables

Variable Declarations

A variable declaration embodies a list of identifiers that designate new variables and their type.



The type given for the variable(s) can be a type identifier previously declared in a type declaration part in the same block, in an enclosing block, or in a unit, or it can be a new type definition.

When an identifier is specified within the identifier list of a variable declaration, that identifier is a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the block, unless the identifier is redeclared in an enclosed block. Redeclaration causes a new variable using the same identifier, without affecting the value of the original variable.

An example of a variable declaration part follows:

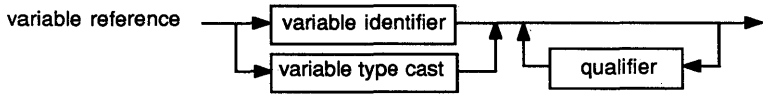
```
var
   X,Y,Z: Real;
   I,J,K: Integer;
   Digit: 0..9;
   C: Color;
   Done,Error: Boolean;
   Operator: (plus, minus, times);
   Hue1,Hue2: set of Color;
   Today: Date;
   Results: MeasureList;
   P1,P2: Person;
   Matrix: array[1..10,1..10] of Real;
```

Variable References

A variable reference signifies one of the following:

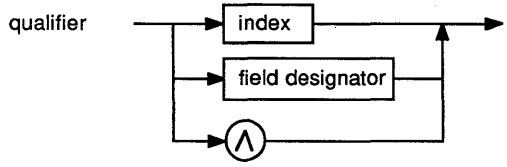
- a variable
- a component of a structured-type or string-type variable
- a dynamic variable pointed to by a pointer-type variable

The syntax for a variable reference is



Qualifiers

A variable reference is a variable identifier with zero or more qualifiers, which modify the meaning of the variable reference.



An array identifier with no qualifier, for example, references the entire array:

```
Results
```

An array identifier followed by an index denotes a specific component of the array—in this case a structured variable:

```
Results[Current+1]
```

With a component that is a record, the index may be followed by a field designator; here the variable access signifies a specific field within a specific array component.

```
Results[Current+1].data
```

The field designator in a pointer field may be followed by the pointer symbol, ^ (a caret), to differentiate between the pointer field and the dynamic variable it points to.

```
Results[Current+1].data^
```

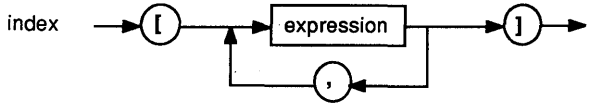
If the variable being pointed to is an array, indexes can be added to denote components of this array.

```
Results[Current+1].data^[J]
```

Arrays, Strings, and Indexes

A specific component of an array variable is denoted by a variable reference that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable is denoted by a variable reference that refers to the string variable, followed by an index that specifies the character position.



The index expressions select components in each corresponding dimension of the array. The number of expressions can't exceed the number of index-types in the array declaration. Furthermore, each expression's type must be assignment compatible with the corresponding index-type.

When indexing a multi-dimensional array, multiple indexes or multiple expressions within an index can be used interchangeably. For example,

```
Matrix[I][J]
```

is the same as

```
Matrix[I,J]
```

You can index a string variable by a single index expression, whose value must be in the range *0..n*, where *n* is the declared size of the string. This accesses one character of the string value, with the type *Char* given to that character value.

The first character of a string variable (at index 0) contains the dynamic length of the string; that is, *Length(S)* is the same as *Ord(S[0])*. If a value is assigned to the length attribute, the compiler does not check that this value is less than the declared size of the string. It is possible to index a string beyond its current dynamic length. The characters thus read are random, and assignments beyond the current length will not affect the actual value of the string variable.

Records and Field Designators

A specific field of a record variable is denoted by a variable reference that refers to the record variable, followed by a field designator specifying the field.

Some examples of field designators:

```
Today.year
Results[1].count
Results[1].when.month
```

In a statement within a **with** statement, a field designator does not have to be preceded by a variable reference to its containing record.

Pointers and Dynamic Variables

The value of a pointer variable is either **nil**, or a value that points to a dynamic variable.

The dynamic variable pointed to by a pointer variable is referenced by writing the pointer symbol ^ after the pointer variable.

You create dynamic variables and their pointer values with the standard procedure *New*. The @ operator and standard procedure *Pointer* can be employed to create pointer values that are treated as pointers to dynamic variables.

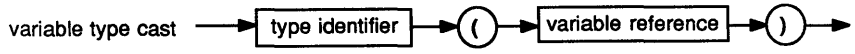
nil does not point to any variable. It is an error if you access a dynamic variable when the pointer's value is **nil** or undefined.

Some examples of references to dynamic variables:

```
P1^
P1^.sibling^
Results[1].data^
```

Variable-Type-Casts

A variable reference of one type can be changed into a variable reference of another type through a variable-type-cast.



When a variable-type-cast is applied to a variable reference, the variable reference is treated as an instance of the type specified by the type identifier. The size of the variable (the number of bytes occupied by the variable) must be the same as the size of the type denoted by the type identifier. A variable-type-cast may be followed by one or more qualifiers as allowed by the specified type.

Some examples of variable-type-casts:

```
type
  Point = record
            x,y: Integer;
          end;
  List  = array[1..2] of Integer;
var
  P: Point;
  L: LongInt;
  N: Integer;
begin
  P := Point(L);
  N := Point(L).x;
  LongInt(P) := Longint(P) + $00080008;
  List(P)[N] := 32;
end.
```

Turbo Pascal also allows you to type cast the value of an expression. This is described in Chapter 20.

CHAPTER 20

Expressions

Expressions are made up of *operators* and *operands*. Most Pascal operators are *binary*, that is, they take two operands; the rest are *unary* and take only one operand. Binary operators use the usual algebraic form, for example, $a+b$. A unary operator always precedes its operand, for example, $-b$.

In more complex expressions, rules of precedence clarify the order in which operations are performed. Table 20-1 shows the precedence of operators. However, there are three basic rules of precedence. First, an operand between two operators of different precedence is bound to the operator with higher precedence. Second, an operand between two equal operators is bound to the one on its left. Third, expressions within parentheses are evaluated prior to being treated as a single operand.

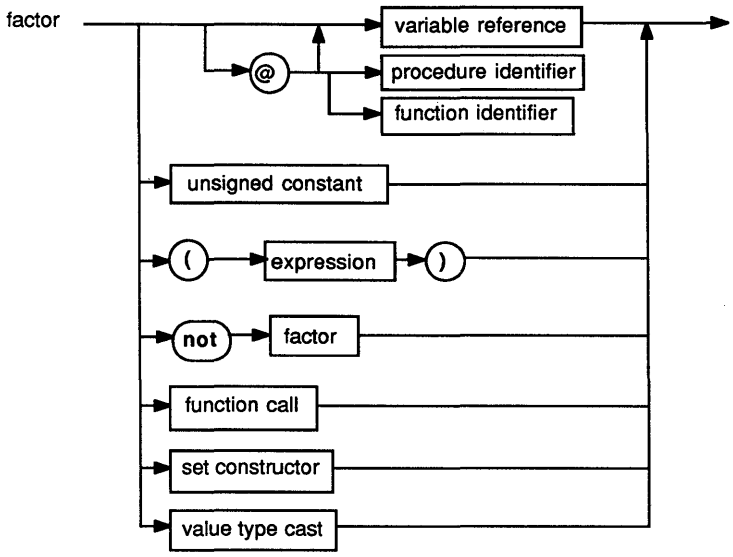
Table 20-1 Precedence of Operators

Operators	Precedence	Categories
@, **not**	highest	unary operators
*,/,**div**,**mod**,**and**,**shl**,**shr**	second	multiplying operators
+,-, **or**, **xor**	third	adding operators
=, ⟨⟩, ⟨, ⟩, ⟨=, ⟩=, **in**	lowest	relational operators

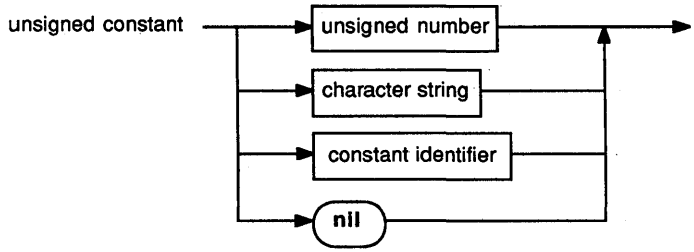
Operations with equal precedence are normally performed from left to right.

The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple-expressions.

The syntax of a factor is



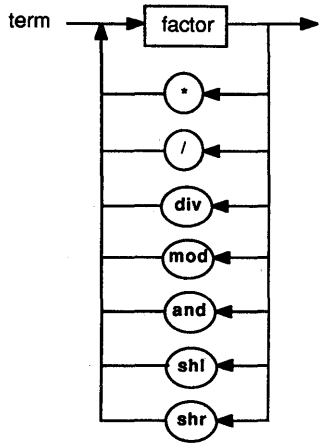
A function call activates a function, and denotes the value returned by the function (see "Function Calls" later in this chapter). A set constructor denotes a value of a set type (see "Set Constructors"). A value-type-cast changes the type of a value (see "Value-Type-Casts"). An unsigned constant has the following syntax:



Some examples of factors:

```
X                       {variable reference}
@X                      {pointer to a variable}
15                      {unsigned constant}
(X+Y+Z)                 {subexpression}
Sin(X/2)                {function call}
['0'..'9','A'..'Z']     {set constructor}
not Done                {negation of a boolean}
Char(Digit+48)          {value-type-cast}
```

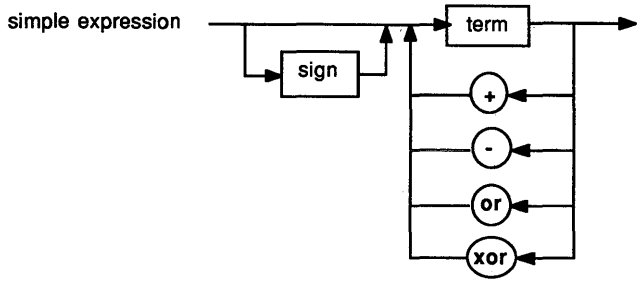
Terms apply the multiplying operators to factors:



Some examples of terms:

```
X*Y
Z/(1-Z)
Done or Error
(X <= Y) and (Y < Z)
```

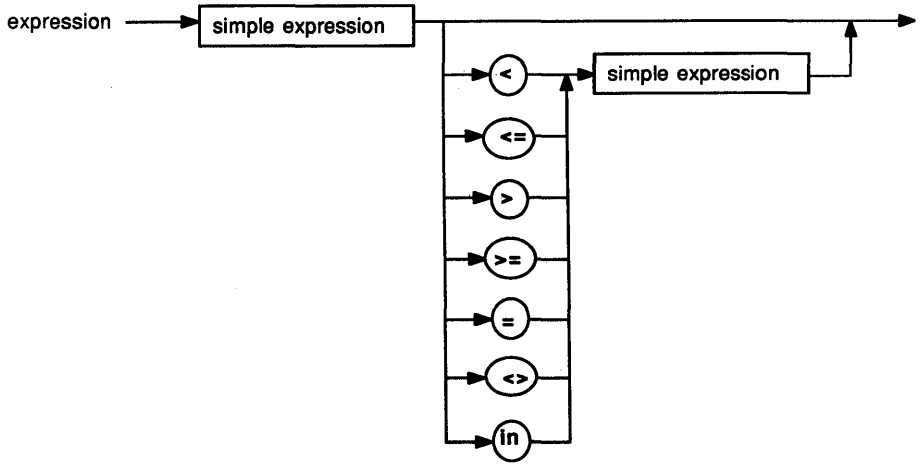
Simple expressions apply adding operators and signs to terms:



Some examples of simple expressions:

```
X+Y
-X
Hue1 + Hue2
I*J + 1
```

An expression applies the relational operators to simple expressions:



Some examples of expressions:

```
X = 1.5
Done <> Error
(I < J) = (J < K)
C in Hue1
```

Operators

The operators are classified as arithmetic operators, logical operators, string operators, set operators, relational operators, and the @ operator.

Arithmetic Operators

The following tables show the types of operands and results for binary and unary arithmetic operations.

Table 20-2 Binary Arithmetic Operations

Operator	Operation	Operand Types	Type of Result
+	addition	integer-type real-type	integer-type Extended
−	subtraction	integer-type real-type	integer-type Extended
*	multiplication	integer-type real-type	integer-type Extended
/	division	integer-type real-type	Extended Extended
div	integer division	integer-type	integer-type
mod	remainder	integer-type	integer-type

*NOTE: The + operator is also used as a string or set operator, and the − and * operators are also used as set operators.*

Table 20-3 Unary Arithmetic Operations

Operator	Operation	Operand Types	Type of Result
+	sign identity	integer-type real-type	integer-type *Extended*
−	sign negation	integer-type real-type	integer-type *Extended*

Any operand whose type is a subrange of an ordinal-type is treated as if it were of the ordinal-type.

If both operands of a +, −, *, **div**, or **mod** operator are of type *Integer*, the result is of type *Integer*. If one or both operands are of type *LongInt*, the result is of type *LongInt*.

If one or both operands of a +, −, or * operator are of a real-type, the type of the result is *Extended*.

If the operand of the sign identity or sign negation operator is of an integer-type, the result is of the same integer-type. If the operator is of a real-type, the type of the result is *Extended*.

The value of *x*/*y* is always of type *Extended*, regardless of the operand types. An error occurs if *y* is zero.

The value of *i* **div** *j* is the mathematical quotient of *i*/*j*, rounded in the direction of zero to an integer-type value. An error occurs if *j* is zero.

The **mod** operator returns the remainder obtained by dividing its two operands, that is,

```
i mod j = i - (i div j) * j
```

The sign of the result of **mod** is the same as the sign of *i*. An error occurs if *j* is zero.

Logical Operators

The types of operands and results for logical operations are shown inTable 20-4.

Table 20-4 *Logical Operations*

Operator	Operation	Operand Types	Type of Result
not	negation	*Boolean*	*Boolean*
	bitwise negation	integer-type	integer-type
and	logical **and**	*Boolean*	*Boolean*
	bitwise **and**	integer-type	integer-type
or	logical **or**	*Boolean*	*Boolean*
	bitwise **or**	integer-type	integer-type
xor	logical **xor**	*Boolean*	*Boolean*
	bitwise **xor**	integer-type	integer-type
shl	shift left	integer-type	integer-type
shr	shift right	integer-type	integer-type

Note: The **not** operator is a unary operator.

For operands of type *Boolean*, normal boolean logic governs the results of these operations. For instance, *a* **and** *b* is *True* only if both *a* and *b* are true.

If the operand of the **not** operator is of an integer-type, the result is of the same integer-type.

If both operands of an **and, or,** or **xor** operator are of type *Integer*, the result is of type *Integer*. If one or both operands are of type *LongInt*, the result is of type *LongInt*.

The operations *i* **shl** *j* and *i* **shr** *j* shifts the value of *i* to the left or to the right by *j* bits. The type of the result is the same as the type of *i*.

String Operators

The types of operands and results for string operations are shown in Table 20-5.

Table 20-5 *String Operations*

Operator	Operation	Operand Types	Type of Result
+	concatenation	string-type, char-type, or packed-string-type	string-type

Turbo Pascal allows the + operator to be used to concatenate two string operands. The result of the operation $s+t$, where s and t are of a string-type, a char-type, or a packed-string-type, is the concatenation of s and t. The result is compatible with any string-type (but not with char-types and packed-string-types). If the resulting string is longer than 255 characters, it is truncated after the 255th character.

Set Operators

The types of operands for set operations are shown in Table 20-6.

Table 20-6 *Set Operations*

Operator	Operation	Operand Types
+	union	compatible set-types
−	difference	compatible set-types
*	intersection	compatible set-types

The results of set operations conform to the rules of set logic:

- An ordinal value c is in $a+b$ only if c is in a or b.
- An ordinal value c is in $a-b$ only if c is in a and not in b.
- An ordinal value c is in $a*b$ only if c is in both a and b.

If the smallest ordinal value that is a member of the result of a set operation is a and the largest is b, then the type of the result is **set of** $a..b$.

The types of operands and results for relational operations are shown in Table 20-7.

Table 20-7 Relational Operations

Operator	Operation	Operand Types	Type of Result
=	equal	compatible simple, pointer, set, string, or packed-string types	*Boolean*
⟨⟩	not equal	compatible simple, pointer, set, string, or packed-string types	*Boolean*
⟨	less than	compatible simple, string, or packed-string types	*Boolean*
⟩	greater than	compatible simple, string, or packed-string types	*Boolean*
⟨=	less or equal	compatible simple, string, or packed-string types	*Boolean*
⟩=	greater or equal	compatible simple, string, or packed-string types	*Boolean*
⟨=	subset of	compatible set-types	*Boolean*
⟩=	superset of	compatible set-types	*Boolean*
in	member of	left operand: any ordinal-type t right operand: **set of** type t	*Boolean*

Comparing Simple-Types

When the operands of =, ⟨⟩, ⟨, ⟩, ⟩=, or ⟨= are of simple-types, they must be compatible types, except that if one operand is of a real-type the other may be of an integer-type.

Because real-type values are approximations, the results of comparing real-type values are not always as expected. For instance, if *X* is a variable of type *Real* and *Y* is a variable of type *Double*, and if the assignments

```
X := 1/3;
Y := 1/3;
```

have been made, then *X*=*Y* will return *False*. The reason is that *X* is accurate only to 7 to 8 digits, whereas *Y* is accurate to 15 to 16 digits, and when both are converted to *Extended*, they will differ after 7 to 8 digits.

See Chapter 26, for extensions that affect the ordering of comparisons involving NaNs.

Comparing Strings

The relational operators =, ⟨⟩, ⟨, ⟩, ⟩=, and ⟨= compare strings according to the ordering of the Macintosh character set. Any two string values can be compared, since all string values are compatible.

A char-type value is compatible with a string-type value, and when the two are compared, the char-type value is treated as a string-type value with length 1. When a packed-string-type value with *n* components is compared with a string-type value, it is treated as a string-type value with length *n*.

Comparing Packed Strings

The relational operators =, ⟨⟩, ⟨, ⟩, ⟩=, and ⟨= may also be used to compare two packed-string-type values if both have the same number of components. If the number of components is *n*, then the operation corresponds to comparing two string each of length *n*.

Comparing Pointers

The operators = and ⟨⟩ can be used on compatible pointer-type operands. Two pointers are equal only if they point to the same object.

Comparing Sets

If *a* and *b* are set operands, their comparisons produce these results:

- *a* = *b* is true only if *a* and *b* contain exactly the same members; otherwise, *a* ⟨⟩ *b*.
- *a* ⟨= *b* is true only if every member of *a* is also a member of *b*.
- *a* ⟩= *b* is true only if every member of *b* is also a member of *a*.

Testing Set Membership

The **in** operator returns *True* when the value of the ordinal-type operand is a member of the set-typed operand; otherwise, it returns *False*.

The @ Operator

A pointer to a variable can be created with the @ operator. Table 20-8 shows the operand and result types.

Table 20-8 Pointer Operations

Operator	Operation	Operand Types	Type of Result
@	pointer formation	variable reference or procedure or function identifier	pointer (same as **nil**)

@ is a unary operator that takes a variable reference or a procedure or function identifier as its operand returns a pointer to the operand. The type of the value is the same as the type of **nil**, therefore it can be assigned to any pointer variable.

@ with a Variable

The use of @ with an ordinary variable (not a parameter) is uncomplicated. Given the declarations:

```
type
  TwoChar = packed array[0..1] of Char;
var
  Int: integer;
  TwoCharPtr: ^TwoChar;
```

then the statement:

```
TwoCharPtr := @Int;
```

causes *TwoCharPtr* to point to *Int*. *TwoCharPtr*^ becomes a reinterpretation of the value of *int*, as though it were a ***packed array[0..1] of Char***.

@ with a Value Parameter

Applying @ to a formal value parameter results in a pointer to the stack location containing the actual value. Say *Foo* is a formal value parameter in a procedure and *FooPtr* is a pointer variable. If the procedure executes the statement:

```
FooPtr := @Foo;
```

then *FooPtr*^ references *Foo*'s value. However, *FooPtr*^ does not reference *Foo* itself, but rather it references the value that was taken from *Foo* and stored on the stack.

@ *with a Variable Parameter*

Applying @ to a formal variable parameter results in a pointer to the actual parameter (the pointer is taken from the stack). Say *One* is a formal variable parameter of a procedure, *Two* is a variable passed to the procedure as *One*'s actual parameter, and *OnePtr* is a pointer variable. If the procedure executes the statement

```
OnePtr := @One;
```

then *OnePtr* is a pointer to *Two* and *OnePtr*^ is a reference to *Two* itself.

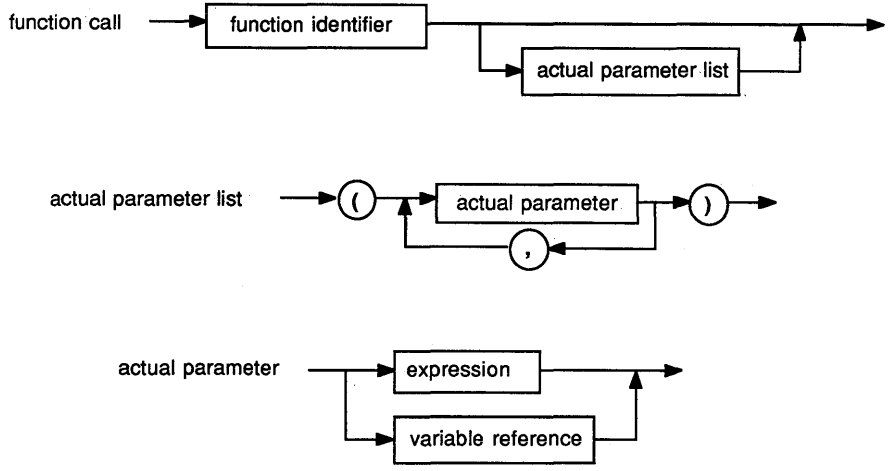
@ *with a Procedure or Function*

You can apply @ to a procedure or a function to produce a pointer to its entry point. Turbo Pascal does not give you a mechanism for using such a pointer. The only use for a procedure pointer is to pass it to an assembly-language routine.

Function Calls

A function call activates the function specified by the function identifier. Any identifier declared to denote a function is a function identifier.

The function call must have a list of actual parameters if the corresponding function declaration contains a list of formal parameters. Each parameter takes the place of the corresponding formal parameter according to parameter rules set forth in Chapter 22.

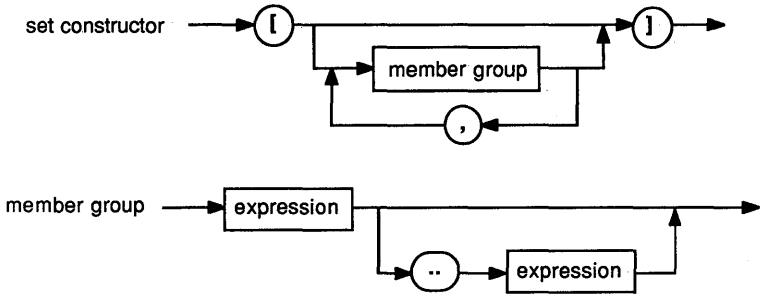


Some examples of function calls follow:

```
Sum(A,63)
Maximum(147,J)
Sin(X+Y)
Eof(F)
Volume(Radius,Height)
```

Set Constructors

A set constructor denotes a set-type value, and is formed by writing expressions within brackets ([]). Each expression denotes a value of the set.



The notation *[]* denotes the empty set, which is assignment compatible with every set-type. Any member group *x..y* denotes as set members all values in the range *x..y*. If *x* is greater than *y*, then *x..y* does not denote any members and *[x..y]* denotes the empty set.

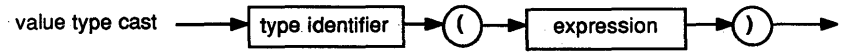
All expression values in member groups in a particular set constructor must be of the same ordinal-type.

Some examples of set constructors follow:

```
[red, C, green]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit+48)]
```

Value-Type-Casts

The type of an expression can be changed to another type through a value-type-cast.



The expression argument must be of an ordinal-type or a pointer-type. The result is of the specified type, and its ordinal value is obtained by converting the expression. This conversion may involve truncation or extension of the original value if the size of the specified type is different from that of the expression. In cases where the value is extended, the sign of the value is always preserved, that is, the value is sign-extended.

The syntax of a value-type-cast is almost identical to that of a variable-type-cast see "Variable-Type-Casts" in Chapter 19. However, value-type-casts operate on values, not on variables, and can therefore not participate in variable references; that is, a value-type-cast may not be followed by qualifiers. In particular, value-type-casts may not appear on the left-hand side of an assignment statement.

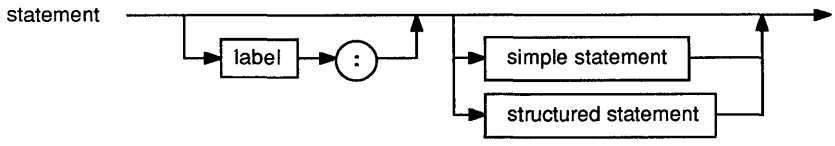
Some examples of value-type-casts:

```
Integer('A')       Char(48)      Boolean(0)    Color(2)
Longint(@Buffer)   IntPtr(-1)    IntPtr(LongInt(P)+2)
```

C H A P T E R **21**

Statements

Statements describe algorithmic actions that can be executed. Labels can prefix statements, and these labels can be referenced by **goto** statements.

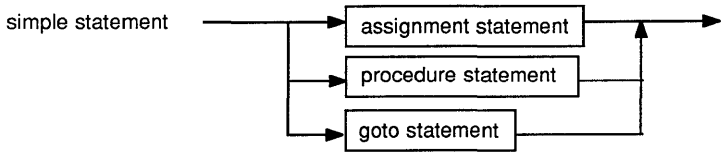


As you saw in Chapter 16, a label is either a digit sequence in the range 0 to 9999 or an identifier.

There are two main types of statements: simple statements and structured statements.

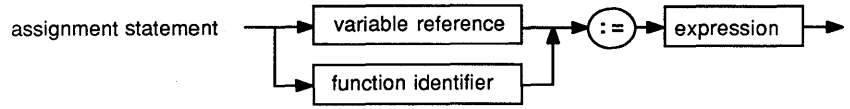
Simple Statements

A simple statement is a statement that does not contain any other statements.



Assignment Statements

Assignment statements either replace the current value of a variable with a new value specified by an expression or specify an expression whose value is to be returned by a function.



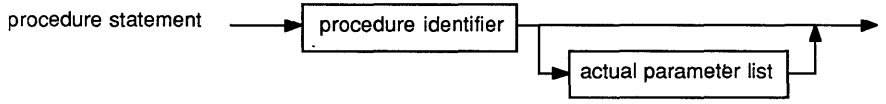
The expression must be assignment compatible with the type of the variable or the result type of the function (see Chapter 18, "Compatibility of Types").

Some examples of assignment statements:

```
X    := Y+Z;
Done := (I>=1) and (I<100);
Hue1 := [blue,Succ(C)];
I    := Sqr(J) - I*K;
```

Procedure Statements

A **procedure** statement specifies the activation of the procedure denoted by the procedure identifier. If the corresponding procedure declaration contains a list of formal parameters, then the procedure statement must have a matching list of actual parameters (parameters listed in definitions are *formal* parameters; in the calling statement, they are *actual* parameters). The actual parameters are passed to the formal parameters as part of the call.



Some examples of procedure statements:

```
PrintHeading;
Transpose(A,N,M);
Find(Name,Address);
```

Goto Statements

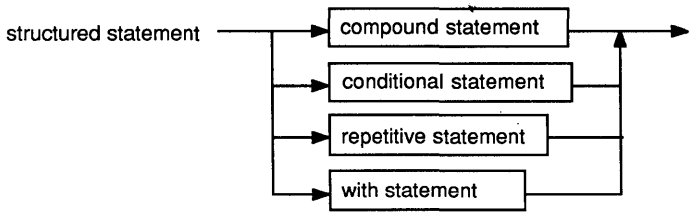
A **goto** statement transfers program execution to the statement prefixed by the label that is referenced in the **goto** statement. Following is the syntax diagram of a **goto** statement:

The following rules should be observed when using **goto** statements:

- The label referenced by a **goto** statement must be in the same block as the **goto** statement. In other words, it is not possible to jump into or out of a procedure or function.
- Jumping into a structured statement from outside that structured statement (that is, jumping to a "deeper" level of nesting) can have undefined effects, although the compiler will not indicate an error.

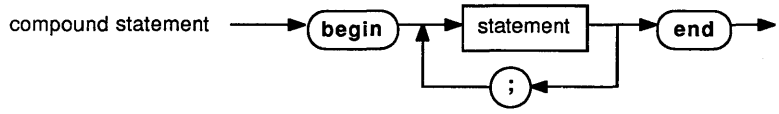
Structured Statements

Structured statements are constructs composed of other statements that are to be executed in sequence (compound statements and **with** statements), conditionally (conditional statements), or repeatedly (repetitive statements).



Compound Statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The component statements are treated as one statement, crucial in contexts where the Pascal syntax only allows one statement. **Begin** and **end** bracket the statements, which are separated by semicolons.

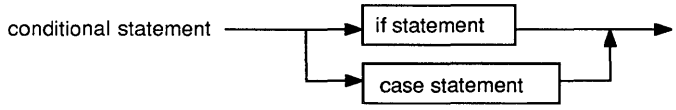


An example of a compound statement is

```
begin
   Z := X;
   X := Y;
   Y := Z;
end;
```

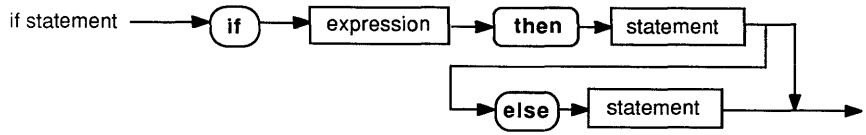
Conditional Statements

A conditional statement selects for execution a single one (or none) of its component statements.



If Statements

The syntax for **if** statements is



The expression must yield a result of the standard type *Boolean*. If the expression produces the value *True*, then the statement following **then** is executed.

If the expression produces *False* and the **else** part is present, the statement following **else** is executed; if the **else** part is not present, nothing is executed.

The syntactic ambiguity arising from the construct

```
if e1 then if e2 then s1 else s2
```

is resolved by interpreting the construct as follows:

```
if e1 then
begin
  if e2 then
    s1
  else
    s2
end
```

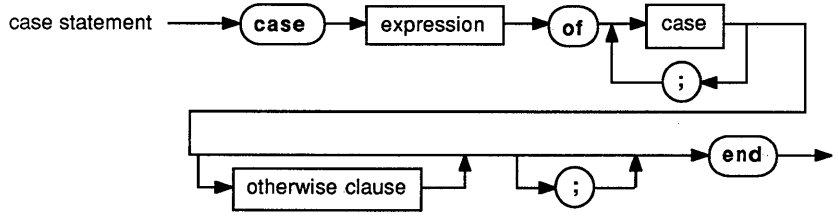
In general, an **else** is associated with the closest **if** not already associated with an **else**.

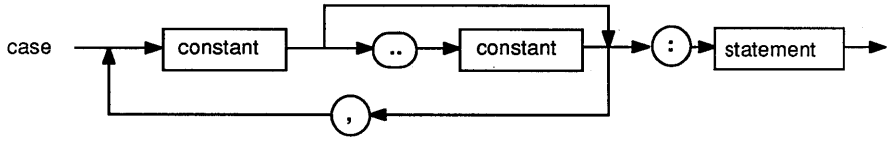
Two examples of **if** statements follow:

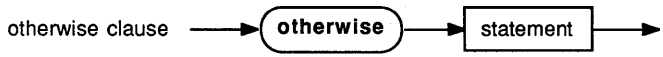
```
if X < 1.5 then
  Z := X+Y
else
  Z := 1.5;

if P1 <> nil then
  P1 := P1.father;
```

Case Statements

The **case** statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called **case** constants) or with the word **otherwise**. The selector must be of an ordinal-type, and all the **case** constants must be unique and of an ordinal-type that is compatible with the type of the selector.







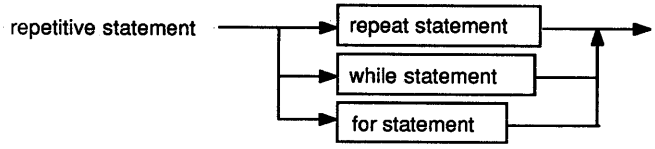
The **case** statement executes the statement prefixed by a **case** constant that equals the value of the selector or a **case** range that contains the value of the selector. If no such **case** constant of the **case** range exists and an **otherwise** part is present, the statement following **otherwise** is executed. If there is no **otherwise** part, nothing is executed.

Examples of **case** statements follow:

```
case Operator of
   plus:   X := X+Y;
   minus:  X := X-Y;
   times:  X := X*Y;
end;

case I of
  0,2,4,6,8: WriteLn('Even digit');
  1,3,5,7,9: WriteLn('Odd digit');
  10..100:   WriteLn('Between 10 and 100');
otherwise
  WriteLn('Negative or greater than 100');
end;
```

Repetitive Statements

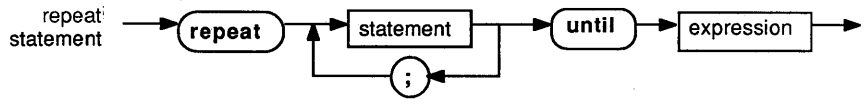
Repetitive statements specify that certain statements are to be executed repeatedly.



If the number of repetitions is known beforehand, the **for** statement is the appropriate construct. Otherwise, the **while** or **repeat** statement should be used.

Repeat Statements

A **repeat** statement contains an expression that controls the repeated execution of a statement sequence within the **repeat** statement.



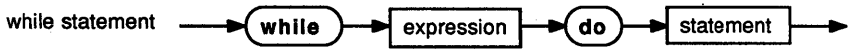
The expression must produce a result of type *Boolean*. The statements between the symbols **repeat** and **until** are executed in sequence until, at the end of a sequence, the expression yields True. The sequence is executed at least once, because the expression is evaluated *after* the execution of each sequence.

Examples of **repeat** statements:

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter value (0..9): ');
  ReadLn(I);
until (I >= 0) and (I <= 9);
```

While Statements

A **while** statement contains an expression that controls the repeated execution of a statement (which may be a compound statement).



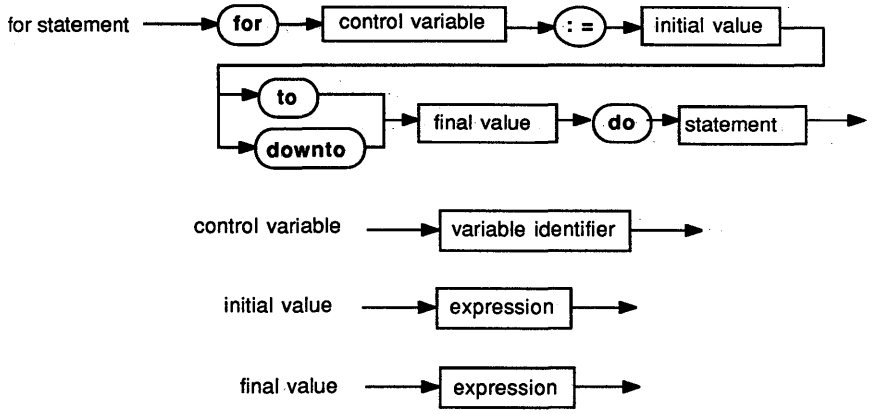
The expression controlling the repetition must be of type *Boolean*. It is evaluated *before* the contained statement is executed. The contained statement is executed repeatedly as long as the expression is *True*. If the expression is *False* at the beginning, the statement is not executed at all.

Examples of **while** statements:

```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InFile) do
begin
  ReadLn(InFile,Line);
  Process(Line);
end;
```

For Statements

The **for** statement causes a statement (which may be a compound statement) to be repeatedly executed, while a progression of values is assigned to a variable called the control variable.



The control variable must be a variable identifier (without any qualifier) that signifies a variable declared to be local to the block containing the **for** statement. The control variable must be of an ordinal-type. The initial and final values must be of a type that is assignment compatible with the ordinal-type.

When a **for** statement is entered, the initial and final values are determined once for the remainder of the execution of the **for** statement.

The statement contained by the **for** statement is executed once for every value in the range *initial-value* to *final-value*. The control variable always starts off at *initial-value*. With a **for** statement using **to**, the value of the control variable is incremented by one for each repetition. If *initial-value* is greater than *final-value*, the contained statement is not executed. With a **for** statement using **downto**, the value of the control variable is decremented by one for each repetition. If *initial-value* value is less than *final-value*, the contained statement is not executed.

It is an error if the contained statement alters the value of the control variable. After a **for** statement is executed, the value of the control variable value is undefined, unless execution of the **for** statement was interrupted by a **goto** out of the **for** statement.

With these restrictions in mind, the **for** statement

```
for V := Expr1 to Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 <= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Succ(V);
      Body;
    end;
  end;
end;
```

and the **for** statement

```
for V := Expr1 downto Expr2 do Body;
```

is equivalent to

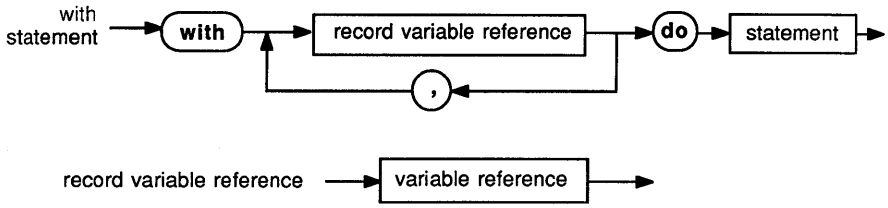
```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 >= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Pred(V);
      Body;
    end;
  end;
end;
```

where *Temp1* and *Temp2* are auxiliary variables of the host-type of the variable *V* that do not occur elsewhere in the program.

Examples of **for** statements follow:

```
for I := 2 to 63 do
  if Data[I] > Max then Max := Data[I]

for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I,K] * Mat2[K,J];
    Mat[I,J] := X;
  end;

for C := red to blue do Check(C);
```

The **with** statement is a shorthand method for referencing the fields of a record. Within a **with** statement, the fields of one or more specific record variables can be referenced using their field identifiers only. The syntax of a **with** statement is



Following is an example of a **with** statement:

```
with Date do
  if month = 12 then
  begin
    month := 1;
    year  := year + 1
  end else
    month := month + 1;
```

This is equivalent to

```
if Date.month = 12 then
begin
  Date.month := 1;
  Date.year  := Date.year + 1
end else
  Date.month := Date.month + 1;
```

Within a **with** statement, each variable reference is first checked as to whether it can be interpreted as a field of the record. If so, it is always interpreted as such, even if a variable with the same name is also accessible. Suppose the following declarations have been made:

```
type
  Point = record
            x,y: Integer;
          end;
var
  x: Point;
  y: Integer;
```

In this case, both *x* and *y* can refer to a variable or to a field of the record. In the statement

```
with x do
begin
  x := 10;
  y := 25;
end;
```

the *x* between **with** and **do** refers to the variable of type *Point*, but in the compound statement, *x* and *y* refer to *x.x* and *x.y*.

The statement

```
with V1,V2, ... Vn do s;
```

is equivalent to:

```
with V1 do
  with V2 do
    ....
      with Vn do
        S;
```

In both cases, if V_n is a field of both V_1 and V_2, it is interpreted as $V_2.V_n$, not $V_1.V_n$.

If the selection of a record variable involves indexing an array or dereferencing a pointer, these actions are executed once before the component statement is executed.

CHAPTER 22

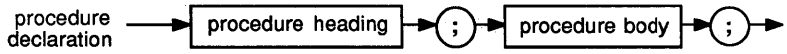
Procedures and Functions

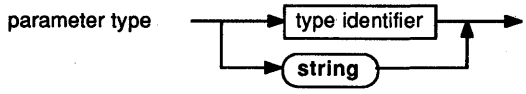
Procedures and functions allow you to nest additional blocks in the main program block. Each procedure or function declaration has a heading followed by a block. A procedure is activated by a procedure statement; a function is activated by the evaluation of an expression that contains its call and returns a value to that expression.

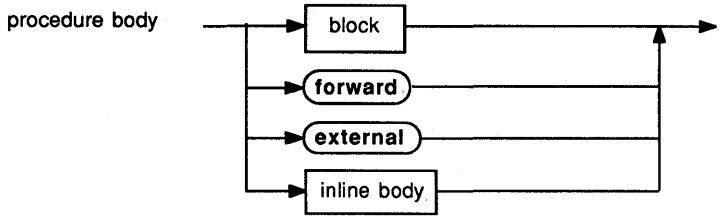
This chapter discusses the different types of procedure and function declarations and their parameters.

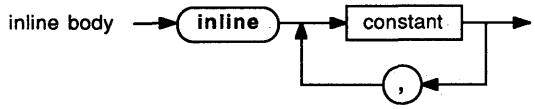
Procedure Declarations

A **procedure** declaration associates an identifier with a block as a procedure; that procedure then can be activated by a procedure statement.

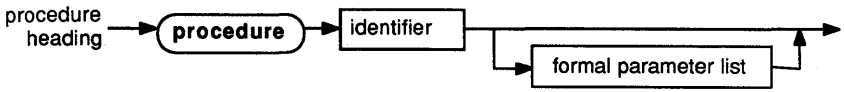








The procedure heading names the procedure's identifier and specifies the formal parameters (if any).



The syntax for a formal parameter list is shown under "Parameters" later in this chapter.

A procedure is activated by a **procedure** statement, which states the procedure's identifier and any actual parameters required. The statements to be executed upon activation are noted in the statement part of the procedure's block. If the procedure's identifier is used in a **procedure** statement within the procedure's block, the procedure is executed recursively (that is, it calls itself while executing).

Here's an example of a **procedure** declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(N mod 10 + Ord('0')) + S;
    N := N div 10;
  until N = 0;
  if N < 0 then S := '-' + S;
end;
```

Instead of the block in a procedure or function declaration, you can write a **forward**, **external**, or **inline** declaration.

Forward Declarations

A **procedure** declaration that specifies the directive **forward** instead of a block is a *forward declaration*. Somewhere after this declaration, the procedure must be defined by a *defining declaration*—a **procedure** declaration that uses the same procedure identifier, but omits the formal parameter list and includes a block. The **forward** declaration and the defining declaration must appear in the same procedure and function declaration part. Other procedures and functions can be declared between them, and they can call the forwardly declared procedure. Mutual recursion is thus possible.

The **forward** declaration and the defining declaration constitute a complete declaration of the procedure. The procedure is considered declared at the **forward** declaration.

An example of a **forward** declaration follows.

```
procedure Walter(m,n: Integer); forward;

procedure Clara(x,y: Real);
begin
  :
  Walter(4,5);
  :
end;

procedure Walter;
begin
  :
  Clara(8.3,2.4);
  :
end;
```

Forward declarations are not allowed in the interface part of a unit.

External Declarations

External declarations allow you to interface with separately compiled procedures and functions written in assembly language. The **external** code must be linked with the Pascal program or unit through {$L FileName} directives. For further details on linking with assembly language, refer to Chapter 27.

Examples of **external** procedure declarations follow:

```
{$L BlockStuff.Rel}

procedure MoveWord(var source,dest; count: LongInt); external;
procedure MoveLong(var source,dest; count: LongInt); external;

procedure FillWord(var dest; data: Integer; count: LongInt); external;
procedure FillLong(var dest; data: Longint; count: LongInt); external;
```

You should use **external** procedures when you need to incorporate substantial amounts of assembly code. If you only require small amounts of code, use **inline** procedures instead.

Inline Declarations

The **inline** directive permits you to write machine code instructions instead of the block. The code consists of constants, typically written in hexadecimal notation.

When a normal procedure is called, the compiler generates code that pushes the procedure's arguments on the stack, and then generates a JSR (Jump to SubRoutine) instruction to call the procedure. When you "call" an **inline** procedure, the compiler generates code from the constants following **inline** instead of the JSR. Each constant represents exactly one word in the code generated by the compiler. The code is generated in the order of the constants.

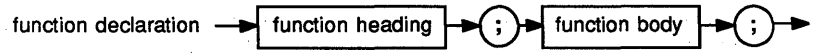
Use **inline** procedures, rather than **external** procedures, for writing small routines.

Example of an **inline** procedure:

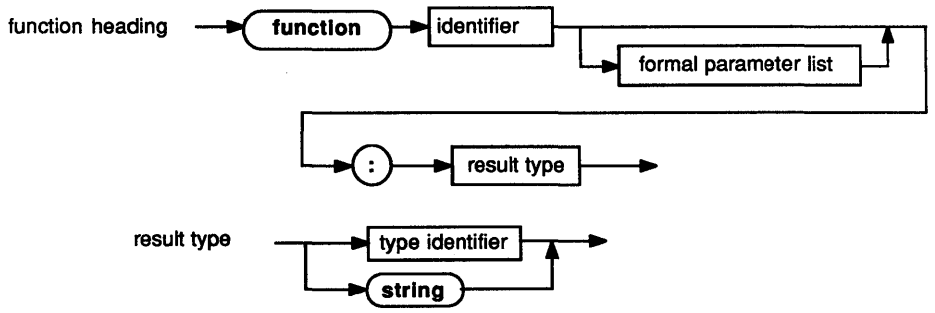
```
procedure Trap(Tos: LongInt); inline $A9ED;
```

Function Declarations

A **function** declaration defines a part of the program that computes and returns a value.



The **function** heading specifies the identifier for the **function**, the formal parameters (if any), and the **function** result type.



A **function** is activated by the evaluation of a **function** call. The **function** call gives the **function**'s identifier and any actual parameters required by the **function**. A **function** call appears as an operand in an expression. When the expression is evaluated, the **function** is executed, and the value of the operand becomes the value returned by the **function**.

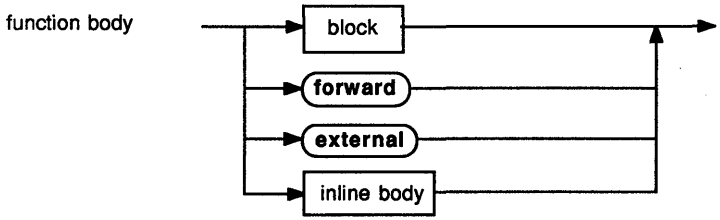
The statement part of the **function**'s block specifies the statements to be executed upon activation of the **function**. The block should contain at least one assignment statement that assigns a value to the **function** identifier. The result of the **function** is the last value assigned. If no such assignment statement exists, or if it is not executed, the value returned by the **function** is unspecified.

If the **function**'s identifier is used in a **function** call within the **function**'s block, the **function** is executed recursively.

Following are examples of **function** declarations:

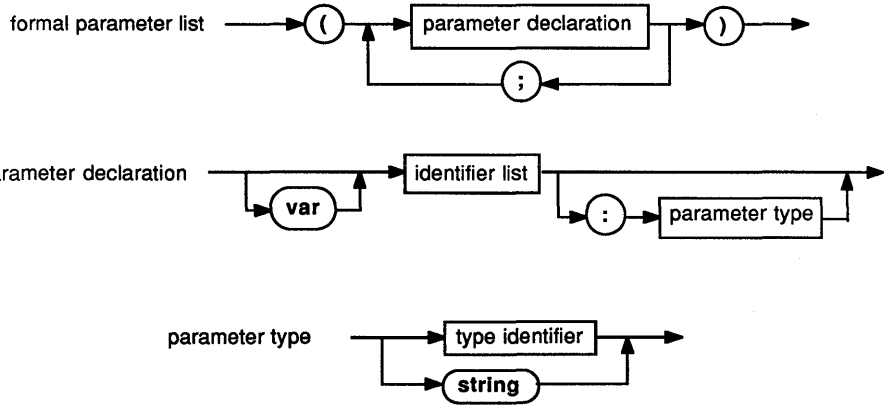
```
function Max(a: Vector; n: Integer): Extended;
var
  x: Extended;
  i: Integer;
begin
  x := a[1];
  for i := 2 to n do if x < a[i] then x := a[i];
  Max := x;
end;

function Power(x: Extended; y: Integer): Extended;
var
  z: Extended;
  i: Integer;
begin
  z := 1.0; i := y;
  while i > 0 do
  begin
    if Odd(i) then z := z * x;
    i := i div 2;
    x := Sqr(x);
  end;
  Power := z;
end;
```

Functions may be declared as **forward**, **external**, or **inline** in the same way as procedures are.



Parameters

The declaration of a procedure or function specifies a formal parameter list. Each parameter declared in a formal parameter list is local to the procedure or function being declared, and can be referred to by its identifier in the block associated with the procedure or function.



There are three kinds of parameters: *value parameters*, *variable parameters*, and *untyped variable parameters*. They are characterized as follows:

- A parameter group without a preceding **var** and followed by a type is a list of value parameters.
- A parameter group preceded by **var** and followed by a type is a list of variable parameters.
- A parameter group preceded by **var** and not followed by a type is a list of untyped variable parameters.

Value Parameters

A formal value parameter acts like a variable local to the procedure or function, except that it gets its initial value from the corresponding actual parameter upon activation of the procedure or function. Changes made to a formal value parameter do not affect the value of the actual parameter.

A value parameter's corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of file-type or of any structured-type that contains a file-type.

The actual parameter must be assignment compatible with the type of the formal value parameter. If the parameter type is **string**, then the formal parameter is given a size attribute of 255.

Variable Parameters

A variable parameter is employed when a value must be passed from a procedure or function to the caller. The corresponding actual parameter in a procedure statement or function call must be a variable reference. The formal variable parameter represents the actual variable during the activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected in the actual parameter.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (you can bypass this restriction through untyped variable parameters). If the formal parameter type is **string**, it is given the length attribute 255, and the actual variable parameter must be a string-type with a length attribute of 255.

File-types can only be passed as variable parameters.

If referencing an actual variable parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Components of variables of any packed structured type (including components of string-type variables) cannot be used as actual variable parameters.

Untyped Variable Parameters

When a formal parameter is an untyped variable parameter, the corresponding actual parameter may be any variable reference, regardless of its type.

Within the procedure or function, the untyped variable parameter is typeless; that is, it is incompatible with variables of all other types, unless it is given a specific type through a variable-type-cast.

An example of untyped variable parameters:

```
function Equal(var source,dest; size: Integer): Boolean;
type
  Bytes = array[0..MaxInt] of -128..127;
var
  N: Integer;
begin
  Equal := true;
  for N := 0 to size - 1 do
  if Bytes(source)[N] <> Bytes(dest)[N] then Equal := false;
end;
```

The above function may be used to compare any two variables of any size. For instance, given the declarations

```
type
  Vector = array[1..10] of Integer;
  Point  = record
             x,y: Integer;
           end;
var
  Vec1,Vec2: Vector;
  N: Integer;
  P: Point;
```

then the function calls

```
Equal(Vec1,Vec2,SizeOf(Vector))
Equal(Vec1,Vec2,SizeOf(Integer)*N)
Equal(Vec[1],Vec1[6],SizeOf(Integer)*5)
Equal(Vec1[1],P,4)
```

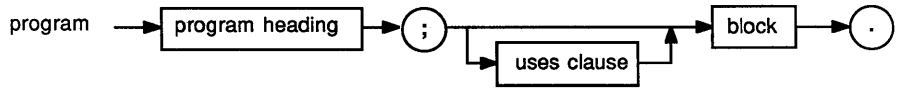
compare *Vec1* to *Vec2*, compare the first *N* components of *Vec1* to the first *N* components of *Vec2*, compare the first five components of *Vec1* to the last five components of *Vec1*, and compare *Vec1[1]* to *P.x* and *Vec1[2]* to *P.y*.

CHAPTER 23

Programs and Units

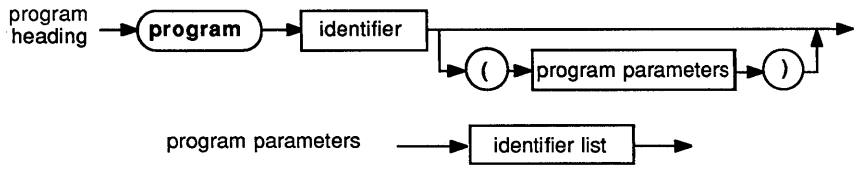
Program Syntax

A Turbo Pascal program has the form of a procedure declaration except for its heading and an optional *uses-clause*.



The Program-Heading

The program heading specifies the program's name and its program parameters.

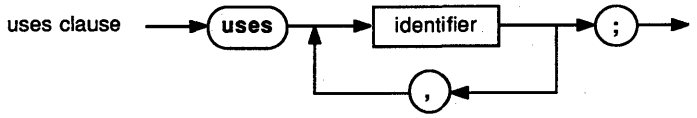


The program name is used to determine the default name of the code file in which to store the program's code when it is compiled to disk. You may override the default code file name with a {$O ⟨filename⟩} compiler directive (see Appendix C, "Compiler Directives").

The program-parameters, if present, are purely decorative and are ignored by the compiler.

The Uses-Clause

The uses-clause identifies all units used by the program, including units that it uses directly and units that are used by those units.



The *PasSystem* unit is always used automatically. *PasSystem* implements all low-level run-time support routines, such as string handling, set handling, dynamic memory allocation, and range checking.

The *PasInOut* and *PasConsole* units are also used automatically (and in that order), unless a {$U−} compiler directive appears before the uses-clause. *PasInOut* implements the Standard Input and Output procedures and functions, and *PasConsole* implements Console device. These two units are required when writing a textbook Pascal program, but, when writing a Macintosh Application, none or only some of them may be required; in that case, a {$U−} compiler directive should appear before the uses- clause.

The {$U ⟨filename⟩} directive is used to specify the name of a unit library file to search in addition to the resident units. Multiple unit library files may be specified through multiple {$U ⟨filename⟩} directives. All {$U ⟨filename⟩} directives must appear before the uses-clause. See Appendix C for further details.

Segmentation

A Macintosh application consists of one or more *code segments*. Small programs are usually contained in a single code segment, but larger programs are divided into several segments for two reasons. First, the Macintosh restricts the size of a single segment to 32K bytes, which means that a program cannot be larger than

32K bytes if it is not segmented. Second, parts of a program that are not executed often (such as initialization and printing) don't have to be kept in memory when they aren't being used; instead, they can be in a separate segment that is swapped in when needed.

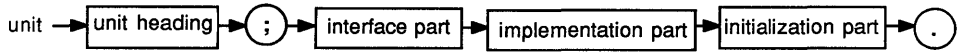
By default, segmentation is disabled in Turbo Pascal, which means that the size of a program cannot exceed 32K bytes. To enable segmentation, place a {$S+} compiler directive in the beginning of the program.

The code of a program's main statement-part is always placed in a segment whose name is a string of blanks (the *blank segment*). When segmentation is enabled, the code of any unit, procedure, or function may be placed in a different segment by using the {$S ⟨SegName⟩} compiler directive. When a {$S ⟨SegName⟩} directive appears in the uses-clause, the code of the following units are placed in the named segment. When a {$S ⟨SegName⟩} directive appears in the declaration-part, the code of the following procedures and functions are placed in the named segment. If no {$S ⟨SegName⟩} directives appear in a program, or if segmentation is not enabled through a {$S+} directive, the code of all units, procedures, and functions are placed in the *blank segment*.

For further information about the {$S+}, {$S−}, and {$S ⟨SegName⟩} compiler directives, see Appendix C.

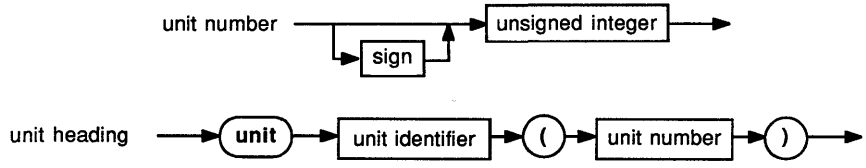
Unit Syntax

Units are the basis of modular programming with Turbo Pascal. They are used to create libraries that you can include in various programs without making the source code available, and to divide large programs into logically related modules.



The Unit-Heading

The unit heading specifies the unit's name and its unit number.



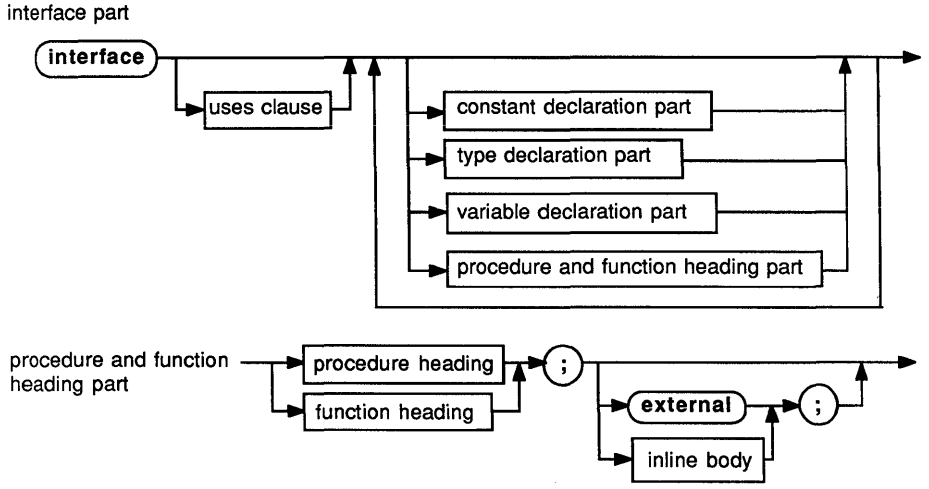
The unit name is used when referring to the unit in a uses-clause. Furthermore, it is used to determine the default name of the unit library file in which to store the unit when it is compiled to disk. You may override the default library file name with a {$O ⟨filename⟩} compiler directive (see Appendix C).

The unit number is a non-zero integer within the range −32768 to 32767. It is used internally by the Turbo Pascal compiler to identify symbols from different units, and for that reason it must be unique; that is, no two units may have the same unit number if they are to be used in the same compilation.

Unit numbers −1 to −32 are reserved by Turbo Pascal for the Pascal run-time support units and the Macintosh interface units. In general, you should not use negative unit numbers, although the compiler will not issue an error message if you do.

The Interface-Part

The interface-part declares constants, types, variables, procedures, and functions that are *public*, that is, available to the host (the program or unit that uses the unit). The host can access these entities as if they were declared in a block that encloses the host.

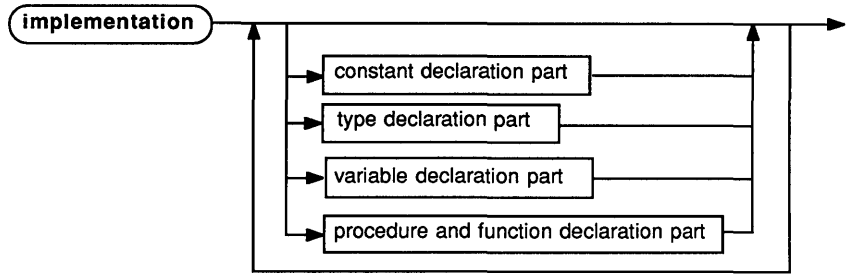


Unless a procedure or function is **inline** or **external**, the interface-part only lists the procedure or function heading. The block of the procedure or function follows in the implementation-part.

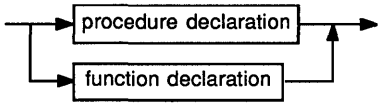
The Implementation-Part

The implementation-part defines the block of all public procedures and functions (unless they are **inline** or **external**). In addition, it declares constants, types, variables, procedures, and functions that are *private*, that is, not available to the host.

implementation part



procedure and function declaration part

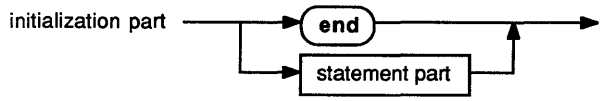


In effect, the procedure and function declarations in the interface-part are like **forward** declarations, although the **forward** directive is not specified. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation-part. While you repeat the procedure and function identifiers, you don't specify the formal parameter list for procedures and functions in the implementation-part.

The size of a unit's code cannot exceed 32K bytes. Segmentation directives are ignored when compiling a unit, but when a unit is used by a program, its entire code may be placed in any segment with a {$S ⟨SegName⟩} compiler directive (see "Segmentation" a few pages back and Appendix C).

The Initialization-Part

The initialization-part is the last part of a unit. It consists either of the reserved word **end** (in which case the unit has no initialization code) or of a statement-part to be executed in order to initialize the unit.



The initialization-parts of units used by a program are executed in the same order as the units appear in the uses-clause.

Units that Use Other Units

The uses-clause in the host must name all units that are used by the host, whether they are used directly or indirectly. Consider the following example:

```
unit Unit1(1);          unit Unit2(2);          program Host;
interface               interface               uses Unit1, Unit2;
const c = 1;            uses Unit1;             const a = b;
implementation          const b = c;            begin
const d = 2;            implementation          end.
end.                    end.
```

Unit2 uses *Unit1*, so for *Host* to use *Unit2* it must first name *Unit1* in its uses-clause. Even though *Host* does not directly reference any identifiers in *Unit1*, it must still name *Unit1*.

When changes are made in the interface-part of a unit, other units that use the unit must be recompiled. However, if changes are only made to the implementation-part or the initialization-part, other units that use the unit *need not* be recompiled. Referring to the example above, if the interface-part of *Unit1* is changed (for example, "c = 2") *Unit2* must be recompiled; changing the implementation-part (for example, "d = 1") doesn't require a recompilation of *Unit2*.

When a unit is compiled, Turbo Pascal computes a *unit version number*, which is basically a checksum of the interface-part. Referring to the example above, when *Unit2* is compiled, the current version number of *Unit1* is saved in the compiled version of *Unit2*. When *Host* is compiled, the version number of *Unit1* is checked against the version number stored in *Unit2*. If the version numbers do not match, indicating that a change was made in the interface-part of *Unit1* since *Unit2* was compiled, the compiler shows an error.

CHAPTER 24

Input and Output

This chapter describes the standard (or built-in) input and output (I/O) procedures and functions of Turbo Pascal.

The code for all I/O procedures and functions reside in the *PasInOut* unit. If you are compiling in the {$U-} state and want to use standard I/O, your program must name the *PasInOut* unit in its uses-clause.

An Introduction to I/O

A Pascal file variable is any variable whose type is a file-type. There are two classes of files: *textfiles* and *typed-files*. A file variable declared to be a type *identical* to the standard type *text* is a textfile. A file variable declared to be a type that was defined using the **file of** construct is a typed-file.

Before a file variable is used, it must be *opened*. When a file is opened, the file variable is associated with an external file that stores the information written to the file or supplies the information read from the file. An external file is typically a named disk file, but it may also be a device, such as the keyboard or the display.

An existing file may be opened via the *Reset* procedure, and a new file may be created and opened via the *Rewrite* procedure. Textfiles opened with *Reset* are *read-only* and textfiles opened with *Rewrite* are *write-only*. Typed-files always

allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

The standard file variables *Input* and *Output* are opened automatically when program execution begins. *Input* is a read-only file associated with the keyboard and *Output* is a write-only file associated with the display. Note that *Input* and *Output* are defined by the *PasConsole* unit. If you are compiling in the {$U−} state and want to use the *Input* and *Output* files, your program must name the *PasConsole* unit in its uses-clause.

A file is a linear sequence of *components*, each of which has the component-type of the file. Each component has a *component number*. The first component of a file is considered to be component zero.

Files are normally accessed *sequentially*. That is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write*, the current file position moves to the file component that is numerically next. However, typed-files may also be accessed *randomly* via the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* may be used to determine the current file position and the current file size of a typed-file.

When a program completes processing a file, the file must be closed using the standard procedure *Close*. Closing a file completely updates the external file it was associated with and breaks the link between the file variable and the external file. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors: If an error occurs, the program terminates displaying the System Error dialog box. This automatic checking may be turned on and off using {$I+} and {$I−} compiler directives. When I/O checking is off, i.e., when a procedure or function call is compiled in the {$I−} state, an I/O error does not cause the program to halt. To check the result of an I/O operation, you must call the standard function *IOResult*.

Standard Procedures and Functions for All Files

The Reset Procedure

Syntax: Reset (f [, title [, bufsize]])

Opens an existing file or rewinds an open file. *f* is a file variable of any file-type. *title* is an optional string-type expression. *bufsize* is an optional expression of type *Integer*.

If *title* is specified, *Reset* opens the existing external file with the name *title* and associates *f* with this external file. It is an error if there is no existing external file of the given name. If *f* is a textfile, the call to *Reset* may optionally specify the size of the buffer to be used when reading from the external file. The default buffer size is 512 bytes.

If *title* is not specified, *f* must already be open. *Reset(f)* causes *f* to be "rewound", that is, the current file position is reset to the beginning of the file.

If *f* is a textfile, *f* becomes *read-only.* After a call to *Reset*, *Eof(f)* is true if the file is empty. Otherwise, *Eof(f)* is false.

The Rewrite Procedure

Syntax: Rewrite (f [, title [, bufsize]])

Creates and opens a new file or rewinds and erases an open file. *f* is a file variable of any file-type. *title* is an optional string-type expression. *bufsize* is an optional expression of type *Integer.*

If *title* is specified, *Rewrite* creates a new external file with the name *title* and associates *f* with this external file. If an external file with the same name already exists, it is deleted and a new empty file is created in its place. If *f* is a textfile, the call to *Rewrite* may optionally specify the size of the buffer to be used when writing to the external file. The default buffer size is 512 bytes.

If *title* is not specified, *f* must already be open. *Rewrite(f)* causes *f* to be "rewound", that is, the current file position is reset to the beginning of the file and any prior contents of *f* are deleted.

If *f* is a textfile, *f* becomes *write-only.* After a call to *Rewrite*, *Eof(f)* is always true.

The Close Procedure

Syntax: Close (f)

Closes an open file. *f* is a file variable of any file-type. The association between *f* and its external file is broken, and the external file is completely updated and then closed.

The Rename Procedure

Syntax: Rename (oldtitle , newtitle)

Renames an external file. *oldtitle* and *newtitle* are string-type expressions. The external file with the name *oldtitle* is renamed to *newtitle.*

The Erase Procedure

Syntax: Erase (title)

Erases an external file. *title* is a string-type expression. The external file with the name *title* is erased.

The IOResult Function

Syntax: IOResult
Result type: *Integer*

IOResult returns an integer value that is the status of the last I/O operation performed. The codes returned are summarized in Appendix B. A value of zero reflects a successful I/O operation.

Standard Procedures and Functions for Typed-Files

The procedures and functions described in this section may only be applied to typed-files. Furthermore, a file passed to one of these procedures or functions must have been opened using *Reset* or *Rewrite*.

The Read Procedure

Syntax: Read (f , v1 [, v2 ,..., vn])

Reads a file component into a variable. *f* is a file variable, and each *v* is a variable of the same type as the component type of *f*. For each variable read, the current file position is advanced to the next component. It is an error to attempt to read from a file when the current file position is at the end of the file, that is, when *Eof(f)* is true.

The Write Procedure

Syntax: Write (f , v1 [, v2 ,..., vn])

Writes a variable into a file component. *f* is a file variable, and each *v* is a variable of the same type as the component type of *f*. For each variable written, the current file position is advanced to the next component. If the current file position is at the end of the file, that is, if *Eof(f)* is true, the file is expanded.

The Seek Procedure

Syntax: Seek (f , n)

Changes the current file position to a specified component. *f* is a file variable, and *n* is an expression of type *LongInt*. The current file position of *f* is moved to component number *n*. The number of the first component of a file is zero. If *n* is greater than the number of the last component in *f*, the current file position is moved to the end of the file, and *Eof(f)* becomes true.

The Eof Function

Syntax: Eof (f)
Result type: *Boolean*

Returns the end-of-file status of a file. *f* is a file variable. *Eof(f)* returns *true* if the current file position is beyond the last component of the file, or if the file contains no components. Otherwise, *Eof(f)* returns *false*.

The FilePos Function

Syntax: FilePos (f)
Result type: *LongInt*

Returns the current file position of a file. *f* is a file variable. If the current file position is at the beginning of the file, *FilePos(f)* returns zero. If the current file position is at the end of the file, that is, if *Eof(f)* is true, *FilePos(f)* is equal to *FileSize(f)*.

The FileSize Function

Syntax: FileSize (f)
Result type: *LongInt*

Returns the current size of a file. *f* is a file variable. *FileSize(f)* returns the number of components in *f*. If the file is empty, *FileSize(f)* returns zero.

Standard Procedures and Functions for Textfiles

This section describes input and output using file variables of the standard type *text*. Note that in Turbo Pascal, the type *text* is distinct from the type **file of** *Char*.

When a textfile is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into *lines*, where each line is terminated by an *end-of-line* character (CR character, ASCII value 13).

For textfiles, there are special forms of *Read* and *Write* that allow you to read and write values that are not of type *Char*. Such values are automatically translated to and from their character representation. For example, *Read(f,i)* where *i* is an integer-type variable will read a sequence of digits, interpret that sequence as a decimal integer, and store it in *i*.

As noted previously there are two standard textfile variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the keyboard, and the standard file variable *Output* is a write-only file associated with the display. *Input* and *Output* are automatically opened when a program begins execution.

All of the standard procedures and functions described in this section need not have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented. For instance, *Read(x)* corresponds to *Read(Input,x)* and *Write(x)* corresponds to *Write(Output,x)*.

If you do specify a file when calling one of the procedures or functions in this section, the file must have been opened using *Reset* or *Rewrite*. It is an error to pass a file that was opened with *Reset* to an output-oriented procedure or function. Likewise, it is an error to pass a file that was opened with *Rewrite* to an input-oriented procedure or function.

The Read Procedure

Syntax: Read ([f ,] v1 [, v2 ,..., vn])

Reads one or more values from a textfile into one or more variables. *f*, if specified, is a textfile variable. If *f* is omitted, the standard file variable *Input* is assumed. Each *v* is a variable of a char-type, an integer-type, a real-type, or a string-type.

Read with a Char-Type Variable With a char-type variable, *Read* reads one character from the file and assigns that character to the variable. If *Eof(f)* was true before *Read* was executed, the value *Chr(0)* is assigned to the variable. If *Eoln(f)* was true, the value *Chr(13)* is assigned to the variable. The next *Read* will start with the next character in the file.

Read with an Integer-Type Variable With an integer-type variable, *Read* expects a sequence of characters that form a signed whole number according to the syntax shown (except that hexadecimal notation is allowed). Any blanks, tabs, or end-of-line characters preceding the numeric string are skipped. Reading ceases at the first blank, tab, or end-of-line character following the numeric string or if *Eof(f)* becomes true. If the numeric string does not conform to the expected format, an I/O error occurs. Otherwise, the value is assigned to the variable. If *Eof(f)* was true before *Read* was executed, or if *Eof(f)* becomes true while skipping blanks, tabs, and end-of-line characters, the value zero is assigned to the variable. The next *Read* will start with the blank, tab, or end-of-line character that terminated the numeric string.

Read with a Real-Type Variable With a real-type variable, *Read* expects a sequence of characters that form a signed number according to the syntax shown (except that hexadecimal notation is allowed). Any blanks, tabs, or end-of-line characters preceding the numeric string are skipped. Reading ceases at the first blank, tab, or end-of-line character following the numeric string or if *Eof(f)* becomes true. If the numeric string does not conform to the expected format, an I/O error occurs. Otherwise, the value is assigned to the variable. If *Eof(f)* was true before *Read* was executed, or if *Eof(f)* becomes true while skipping blanks, tabs, and end-of-line characters, the value zero is assigned to the variable. The next *Read* will start with the blank, tab, or end-of-line character that terminated the numeric string.

Read with a String-Type Variable With a string-type variable, *Read* reads all characters up to *but not including* the next end-of-line character, or until *Eof(f)* becomes true. The resulting character string is assigned to the variable. If the resulting string is longer than the maximum length of the string variable, it is truncated. The next *Read* will start with the end-of-line character that terminated the string.

NOTE: *Read* with a string-type variable does not skip to the next line after reading. For this reason, you cannot use successive *Read* calls to read a sequence of strings, as you will never get past the first line — after the first *Read*, each subsequent *Read* will see the end-of-line character and return a zero-length string. Instead, use multiple *ReadLn* calls to read successive string values.

The ReadLn Procedure

Syntax: ReadLn ([f ,] v1 [, v2 ,..., vn]) or ReadLn [(f)]

The *ReadLn* procedure is an extension to the *Read* procedure. After doing the same as *Read* for the parameter list, it skips to the beginning of the next line of the file. *ReadLn(f)* with no parameters causes the current file position to advance to the beginning of the next line (if there is one, else to the end of the file). *ReadLn* with no parameter list altogether corresponds to *ReadLn(Input)*.

The Write Procedure

Syntax: Write ([f ,] p1 [, p2 ,..., pn])

Writes one or more values to a textfile. *f*, if specified, is a textfile variable. If *f* is omitted, the standard file variable *Output* is assumed. Each *p* is a *write-parameter*. Each write-parameter includes an *output-expression*, whose value is to be written to the file. As explained below, a write-parameter may also contain the specifications of a field-width and a number of decimal places. Each output-expression must be of a char-type, an integer-type, a real-type, a string-type, a packed-string-type, or a boolean-type.

Write Parameters A write-parameter has the form

```
OutExpr [ : MinWidth [ : DecPlaces ] ]
```

where *OutExpr* is an output-expression. *MinWidth* and *DecPlaces* are integer-type expressions.

MinWidth specifies the *minimum* field width. *MinWidth* must be greater than zero. Exactly *MinWidth* characters are written (using leading blanks if necessary), except when *OutExpr* has a value that must be represented in more than *MinWidth* characters; in that case, enough characters are written to represent the value of *OutExpr*. Likewise, if *MinWidth* is omitted, then enough characters as necessary are written to represent the value of *OutExpr*.

DecPlaces specifies the number of decimal places in a fixed-point representation of a real-type value. It can be specified only if *OutExpr* is of a real-type, and if *MinWidth* is also specified. If specified, it must be greater than or equal to zero.

Write with a Char-Type Value If *MinWidth* is omitted, the character value of *OutExpr* is written to the file. Otherwise, *MinWidth*-1 blanks followed by the character value of *OutExpr* is written.

Write with an Integer-Type Value If *MinWidth* is omitted, the decimal representation of *OutExpr* is written to the file with no preceding blanks. If *MinWidth* is specified, and its value is larger than the length of the decimal string, enough blanks are written before the decimal string to make the field width *MinWidth*.

Write with a Real-Type Value If *OutExpr* has a real-type value, its decimal representation is written to the file. The format of the representation depends on the presence or absence of *DecPlaces*.

If *DecPlaces* is omitted, a *floating-point* decimal string is written. If *MinWidth* is also omitted, a default *MinWidth* of 10 is assumed; otherwise, if *MinWidth* is outside the range 10..80, it is truncated to be within that range. The format of the decimal string is

```
[   ¦ - ] <digit> . <decimals> e [ + ¦ - ] <exponent>
```

These are the components of the output string:

[| −]	"" or "−" according to the sign of *OutExpr*.
⟨digit⟩	Single digit, "0" only if *OutExpr* is 0.
⟨decimals⟩	Digit string of *MinWidth*-9 digits.
e	Lowercase "e" character.
[+ | −]	"+" or "−" according to sign of exponent.
⟨exponent⟩	Exponent with trailing blanks to make its width 4.

If *DecPlaces* is present, a *fixed-point* decimal string is written. The format of the string is

```
[ ‹blanks› ] [ - ] ‹digits› [ . ‹decimals› ]
```

These are the components of the string:

[⟨blanks⟩]	Blanks to satisfy *MinWidth*.
[-]	"-" if *OutExpr* is negative.
⟨digits⟩	At least one digit, but no leading zeros.
[. ⟨decimals⟩]	Decimals if *DecPlaces* ⟩ 0.

If *DecPlaces* is present, and *OutExpr* is greater than or equal to 10 ^ (19 − *DecPlaces*), the number is formatted in floating-point style with 19 significant digits, and written with enough leading blanks to make the field width *MinWidth*.

Write with a String-Type Value If *MinWidth* is omitted, the string value of *OutExpr* is written to the file with no leading blanks. If *MinWidth* is specified, and its value is larger than the length of *OutExpr*, enough blanks are written before the decimal string to make the field width *MinWidth*.

Write with a Packed-String-Type Value If *OutExpr* is of a packed-string-type, the effect is the same as writing a string whose length is the number of elements in the packed-string-type.

Write with a Boolean Value If *OutExpr* is of type *Boolean*, the effect is the same as writing the strings 'TRUE' or 'FALSE' depending on the value of *OutExpr*.

The WriteLn Procedure

Syntax: WriteLn ([f ,] p1 [p2 ,..., pn]) or WriteLn [(f)]

The *WriteLn* procedure is an extension to the *Write* procedure. After doing the same as *Write* for the parameter list, it writes an end-of-line character to the file. *WriteLn(f)* with no parameters writes an end-of-line character to the file. *WriteLn* with no parameter list altogether corresponds to *WriteLn(Output)*.

The Eof Function

Syntax: Eof [(f)]
Result type: *Boolean*

Returns the end-of-file status of a file. *f*, if specified, is a textfile variable. If *f* is omitted, the standard file variable *Input* is assumed. *Eof(f)* returns *True* if the current file position is beyond the last character of the file, or if the file contains no components. Otherwise, *Eof(f)* returns *False*.

The Eoln Function

Syntax: Eoln [(f)]
Result type: *Boolean*

Returns the end-of-line status of a file. *f*, if specified, is a textfile variable. If *f* is omitted, the standard file variable *Input* is assumed. *Eoln(f)* returns *True* if the character at the current file position is an end-of-line character or if *Eof(f)* is true. Otherwise, *Eoln(f)* returns *False*.

The SeekEof Function

Syntax: SeekEof [(f)]

Returns the end-of-file status of a file. *SeekEof* corresponds to *Eof*, except that it skips all blanks, tabs, and end-of-line characters before returning the end-of-file status. This is useful when reading numeric values from a textfile.

The SeekEoln Function

Syntax: SeekEoln [(f)]

Returns the end-of-line status of a file. *SeekEoln* corresponds to *Eoln*, except that it skips all blanks and tabs before returning the end-of-line status. This is useful when reading numeric values from a textfile.

Disk Files

A file variable is associated with a disk file by specifying the *pathname* of the disk file in a call to *Reset* or *Rewrite*. A file variable *must* be associated with an external file before it can be used; otherwise Turbo Pascal wouldn't know where to store the information written to the file or where to retrieve the information read from the file.

Pathnames

The Macintosh File Manager uses a *pathname* to identify a specific file on a specific volume. A pathname consists of a file name optionally preceded by a volume name and a colon, for instance:

```
MyVolume:MyFile
```

On systems with the Hierarchical File Manager, a pathname may optionally specify one or more directory names, for instance:

```
MyVolume:MyDir1:...:MyDirn:MyFile
```

If a pathname passed to one of the standard I/O procedures does not include a volume and directory specification, the file is assumed to reside in the default directory on the default volume.

The Macintosh File Manager also allows a file to be specified through a *volume reference number* and a filename. Basically, a volume reference number identifies a specific volume or a specific directory on a specific volume. To make a standard I/O procedure operate on a file with a given volume reference number, call the *SetVol* routine in the *OSIntf* unit to set the default volume before calling the standard I/O procedure.

More information on volumes, directories and files can be found in the "File Manager" chapter of *Inside Macintosh*.

File Types and Creators

The Macintosh File System requires that you assign a *file type* and *file creator* to each new file. The file type determines the type of the file, such as text or application The file creator identifies the application that created the file; this is used in determining which application to launch when the file's icon is double-clicked.

Turbo Pascal determines the file type and file creator of each new file created by the *Rewrite* procedure using four standard variables, which are declared as:

```
var
  FileType, FileCreator,
  TextType, TextCreator: packed array[1..4] of Char;
```

The *FileType* and *FileCreator* variables determine the type and creator of typed-files. The *TextType* and *TextCreator* variables determine the type and creator of textfiles. The default values are:

```
FileType    := 'BINA';
FileCreator := 'TPAS';
TextType    := 'TEXT';
TextCreator := 'TPAS';
```

You may change these defaults by assigning new values to the standard variables before calling *Rewrite*.

More information on file types and creators can be found in the "Finder Interface" chapter of *Inside Macintosh*.

Devices in Turbo Pascal

In Turbo Pascal, external hardware, such as the keyboard, the display, and the printer, are regarded as a *devices*. A device is treated as a textfile, and it is identified through a *device name*. To associate a device with a textfile variable, the device name is passed as the *title* parameter in a call to *Reset* or *Rewrite*.

Turbo Pascal defines two standard devices: A Console device, which is used to communicate with the keyboard and the display, and a Printer device, which is used to communicate with the printer. Their device names are

```
Console device      'Console:'
Printer device      'Printer:'
```

Turbo Pascal also allows you to define your own devices.

The Console Device

The Console device is used to communicate with the keyboard and the display. Its device name is 'Console:'.

The Console device is implemented by the *PasConsole* unit. The *PasConsole* unit furthermore defines the standard file variables *Input* and *Output* and associates these files with the Console device. If you are compiling in the {$U-} state and want to use the Console device, your program must name the *PasConsole* unit in its uses-clause.

Using the *PasConsole* unit in a program may be compared to including the declaration

```
var
  Input, Output: Text;
```

and executing the following procedure calls at the beginning of the program:

```
Reset(Input, 'Console:');
Rewrite(Output, 'Console:');
```

PasConsole also takes care of initializing all Macintosh ROM Managers, and brings up the Console Window, which emulates an 80-character × 25-line terminal.

The Console device is *line oriented.* When reading from the standard file *Input* or a textfile that is associated with the Console device, lines are read one at a time and stored in a line buffer. The *Read* and *ReadLn* procedures obtain their input from this line buffer. Whenever the line buffer becomes empty, typically due to a *ReadLn*, a new line is input at the next *Read* or *ReadLn*.

The following editing keys are available when inputting lines:

⌫	Backspaces one character
Clear or ⌘X	Clears the entire input line
↵	Terminates the input line
⌘.	Generates end-of-file status

Typing ⌘. has the effect of generating an end-of-file status for the file being read. *Eof(f)* becomes true and stays true until *Reset(f)* is executed.

The Printer Device

The Printer device is used to communicate with the printer. Its device name is 'Printer:'.

The Printer device is implemented by the *PasPrinter* unit. The *PasPrinter* unit furthermore defines a textfile variable called *Printer* and associates it with the Printer device. If you want to use the Printer device, your program must name the *PasPrinter* unit in its uses-clause.

Using the *PasPrinter* unit in a program may be compared to including the declaration:

```
var
  Printer: Text;
```

and executing the following procedure call at the beginning of the program:

```
Rewrite(Printer, 'Printer:');
```

The Printer device uses *text streaming* when sending data to the printer. Basically this means that the characters are sent directly to the printer with no interpretation.

More information on printing can be found in the "Printer Driver" chapter of *Inside Macintosh*.

CHAPTER 25

Standard Procedures and Functions

This chapter describes all the standard (built-in) procedures and functions in Turbo Pascal, except for the I/O procedures and functions discussed in Chapter 23.

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a block surrounding the program, no conflict arises from a declaration that redefines the same identifier within the program.

Exit and Halt Procedures

The Exit Procedure

Syntax: Exit

Exits immediately from the current block. When *Exit* is executed in a subroutine, it causes the subroutine to return. When it is executed in the statement-part of a program, it causes the program to terminate. A call to *Exit* may be compared to a **goto**-statement addressing a label just before the **end** of a block.

The Halt Procedure

Syntax: Halt

Immediately stops execution of the program. A call to *Halt* corresponds to a call to the *ExitToShell* routine in the Macintosh operating system.

Dynamic Allocation Procedures and Functions

These procedures and functions are used to manage the heap, a memory area that is unallocated when a program begins execution. The dynamic allocation procedures and functions operate on the Application Heap, and the routines are implemented using the Macintosh Memory Manager.

The New Procedure

Syntax: New (p)

Creates a new dynamic variable and sets a pointer variable to point to it. *p* is a pointer variable of any pointer-type. The size of the allocated memory block corresponds to the size of the type that *p* points to. The memory block is allocated through a call to the *NewPtr* routine of the Macintosh Memory Manager. The newly created variable can be referenced as *p*^. If there isn't enough free space in the heap to allocate the new variable, *p* is set to **nil**.

The Dispose Procedure

Syntax: Dispose (p)

Disposes a dynamic variable. *p* is a pointer variable of any pointer-type that was previously assigned by the *New* procedure or was assigned a meaningful value by an assignment statement. *Dispose* destroys the variable referenced by *p* and returns its memory region to the heap. It does so by calling the *DisposPtr* routine of the Macintosh Memory Manager. The value of *p* then becomes undefined, and it is an error to subsequently reference *p*^.

The MemAvail Function

Syntax: MemAvail
Result type: *LongInt*

Returns the number of free bytes of heap storage available. This number is calculated from the free size of the heap plus the size by which the heap may grow. Note that a block of storage the size of the returned value is unlikely to be available due to fragmentation of the heap. To find the largest free block, call *MaxAvail*.

The MaxAvail Function

Syntax: MaxAvail
Result type: *LongInt*

Returns the size of the largest contiguous free block in the heap. *MaxAvail* expands the heap to its maximum limit and then compacts the heap. This corresponds to a call to the Memory Manager's *MaxApplZone* routine, followed by a call to the *CompactMem* routine.

Transfer Functions

The procedures *Pack* and *Unpack*, as defined in Standard Pascal, are not implemented by Turbo Pascal.

The Chr Function

Syntax: Chr (x)
Result type: *Char*

Returns a character with a specified ordinal number. *x* is an integer-type expression. The result is the character whose ASCII value is *x*.

The Ord Function

Syntax: Ord (x)
Result type: *Integer* or *LongInt*

Returns the ordinal number of an ordinal-type or pointer-type value. If *x* is an ordinal-type expression, the result is of type *Integer* and the value is the ordinality of *x*. If *x* is a pointer-type expression, the result is of type *LongInt*, and the value is the address of the dynamic variable pointed to by *x*.

The Ord4 Function

Syntax: Ord4 (x)
Result type: *LongInt*

Returns the ordinal number of an ordinal-type or pointer-type value. *Ord4* corresponds to *Ord*, except that the type of the result is always *LongInt*.

The Pointer Function

Syntax: Pointer (x)
Result type: *Pointer*

Converts an integer-type or pointer-type value to a pointer-type value. If *x* is an integer-type expression, *Pointer* returns a pointer value that points to the address given by *x*. If *x* is a pointer-type expression, *Pointer* returns a pointer to the same location. The type of the result is the same **nil**, that is, it is assignment compatible with any pointer-type.

The Trunc Function

Syntax: Trunc (x)
Result type: *LongInt*

Truncates a real-type value to an integer-type value. *x* is a real-type expression. *Trunc* returns a *LongInt* value that is the value of *x* rounded towards zero.

The Round Function

Syntax: Round (x)
Result type: *LongInt*

Rounds a real-type value to an integer-type value. *x* is a real-type expression. *Round* returns a *LongInt* value that is the value of *x* rounded to the nearest whole number. If *x* is exactly halfway between two whole numbers, the result is the number with the greatest absolute magnitude.

The Float Function

Syntax: Float (x)
Result type: *Real*

Converts an integer-type value to a real-type value. *x* is an integer-type expression. Note that the compiler automatically converts integer-type values to real-type values when they appear in real-type expressions. You need not use *Float* unless you want to explicitly force this conversion.

The Abs Function

Syntax: Abs (x)
Result type: Same type as parameter

Returns the absolute value of the argument. *x* is an integer-type or real-type expression. The result, of the same type as *x*, is the absolute value of *x*.

The Sqr Function

Syntax: Sqr (x)
Result type: Same type as parameter

Returns the square of the argument. *x* is an integer-type or real-type expression. The result, of the same type as *x*, is the square of *x*, i.e., *x* * *x*.

The Int Function

Syntax: Int (x)
Result type: *Real*

Returns the integer part of the argument. *x* is a real-type expression. The result is the integer part of *x*, i.e., *x* rounded towards zero.

The Sqrt Function

Syntax: Sqrt (x)
Result type: *Real*

Returns the square root of the argument. *x* is a real-type expression. The result is the square root of *x*.

The Sin Function

Syntax: Sin (x)
Result type: *Real*

Returns the sine of the argument. *x* is a real-type expression. The result is the sine of *x*. *x* is assumed to represent an angle in radians.

The Cos Function

Syntax: Cos (x)
Result type: *Real*

Returns the cosine of the argument. *x* is a real-type expression. The result is the cosine of *x*. *x* is assumed to represent an angle in radians.

The Exp Function

Syntax: Exp (x)
Result type: *Real*

Returns the exponential the argument. *x* is a real-type expression. The result is the exponential of *x*, i.e., the value *e* raised to the power of *x*, where *e* is the base of the natural logarithm.

The Ln Function

Syntax: Ln (x)
Result type: *Real*

Returns the natural logarithm of the argument. *x* is a real-type expression. The result is the natural logarithm of *x*.

The ArcTan Function

Syntax: ArcTan (x)
Result type: *Real*

Returns the arctangent of the argument. *x* is a real-type expression. The result is the principal value, in radians, of the arctangent of *x*.

Ordinal Functions

The Succ Function

Syntax: Succ (x)
Result type: Same type as parameter

Returns the successor of the argument. *x* is an ordinal-type expression. The result, of the same type as *x*, is the successor of *x*.

The Pred Function

Syntax: Pred (x)
Result type: Same type as parameter

Returns the predecessor of the argument. *x* is an ordinal-type expression. The result, of the same type as *x*, is the predecessor of *x*.

The Odd Function

Syntax: Odd (x)
Result type: *Boolean*

Tests if the argument is an odd number. *x* is an integer-type expression. The result is *true* if *x* is an odd number, and *false* if *x* is an even number.

String Procedures and Functions

The Length Function

Syntax: Length (s)
Result type: *Integer*

Returns the dynamic length of a string. *s* is a string-type expression. The result is the length of *s*.

The Pos Function

Syntax: Pos (substr , s)
Result type: *Integer*

Searches for a substring in a string. *substr* and *s* are string-type expressions. *Pos* searches for *substr* within *s*, and returns an *Integer* value that is the index of the first character of *substr* within *s*. If *substr* is not found, *Pos* returns zero.

The Concat Function

Syntax: Concat (s1 [, s2 ,..., sn])
Result type: String

Concatenates a sequence of strings. Each parameter is a string-type expression. The result is the concatenation of all the string parameters. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

The Copy Function

Syntax: Copy (s , index , count)
Result type: String

Returns a substring of a string. *s* is a string-type expression. *index* and *count* are integer-type expressions. *Copy* returns a string containing *count* characters starting with the *index*th character in *s*. If *index* is larger than the length of *s*, an empty string is returned. If *count* specifies more characters than remain starting at the *index*th position, only the remainder of the string is returned.

The Delete Procedure

Syntax: Delete (s , index , count)

Deletes a substring from a string. *s* is a string-type variable. *index* and *count* are integer-type expressions. *Delete* deletes *count* characters from *s* starting at the *index*th position. If *index* is larger than the length of *s*, no characters are deleted. If *count* specifies more characters than remain starting at the *index*th position, the remainder of the string is deleted.

The Insert Procedure

Syntax: Insert (source , s , index)

Inserts a substring into a string. *source* is a string-type expression. *s* is a string-type variable. *index* is an integer-type expression. *Insert* inserts *source* into *s* at the *index*th position. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

Console Handling Procedures and Functions

The routines described in this section reside in the *PasConsole* unit. Thus, if you are compiling in the {$U−} state, your program must name the *PasConsole* unit in its uses-clause.

The ClearScreen Procedure

Syntax: ClearScreen

Clears the screen and places the cursor in the upper left-hand corner.

The ClearEOL Procedure

Syntax: ClearEOL

Clears all characters from the cursor position to the end of the line without moving the cursor.

The DeleteLine Procedure

Syntax: DeleteLine

Deletes the line containing the cursor and moves all lines below it one line up.

The InsertLine Procedure

Syntax: InsertLine

Inserts an empty line at the cursor position. All lines below the are moved one line down, and the bottom line scrolls off the screen.

The GotoXY Procedure

Syntax: GotoXY (x , y)

Moves the cursor to the position on the screen specified by the *Integer* expressions *x* (horizontal position) and *y* (vertical position). The upper left corner is (1,1). If *x* is outside the range 1..80 or if *y* is outside the range 1..25, the cursor does not move.

The KeyPressed Function

Syntax: KeyPressed
Result type: *Boolean*

Returns *True* if a key has been pressed on the keyboard, and *False* if no key has been pressed.

The ReadChar Function

Syntax: ReadChar
Result type: *Char*

Causes the cursor to blink while waiting for a key to be pressed. When that happens, the character is returned. The character is not echoed on the screen. The Command key (⌘) functions as the Control key on a standard keyboard: If

a key is pressed while the Command key is held down, *ReadChar* returns the control-value of the key, that is, Chr(1) for ⌘ A, Chr(2) for ⌘ B, and so on.

Miscellaneous Procedures and Functions

The SizeOf Function

Syntax: SizeOf (x)
Result type: *Integer*

Returns the number of bytes occupied by the argument. *x* is either a variable-identifier or a type-identifier. *SizeOf* returns the number of bytes of memory occupied by *x*.

The MoveLeft Procedure

Syntax: MoveLeft (source , dest , count)

Copies a specified number of contiguous bytes from a source range to a destination range (starting at the lowest address). *source* and *dest* are variable references of any type. *count* is an integer-type expression. *MoveLeft* copies a block of *count* bytes from *source* to *dest*, starting with the first byte occupied by *source*. When *source* and *dest* overlap, you should use this procedure if *source* is at the higher address. No checking whatsoever is performed, so be careful with this procedure.

The MoveRight Procedure

Syntax: MoveRight (source , dest , count)

Copies a specified number of contiguous bytes from a source range to a destination range (starting at the highest address). *source* and *dest* are variable references of any type. *count* is an integer-type expression. *MoveRight* copies a block of *count* bytes from *source* to *dest*, starting with the last byte occupied by *source*. If *source* and *dest* overlap and *dest* is at the higher address, you should use this procedure. No checking whatsoever is performed, so be careful.

The FillChar Procedure

Syntax: FillChar (x , count , ch)

Fills a specified number of contiguous bytes with a specified value. *x* is a variable reference of any type. *count* is an integer-type expression. *ch* is an ordinal-type expression. *FillChar* writes the value of *ch* into *count* contiguous bytes of memory starting at the first byte occupied by *x*. No checking whatsoever is performed, so be careful.

The ScanEQ Function

Syntax: ScanEQ (limit , ch , x)
Result type: *Integer*

Scans a range of bytes for the first occurrence of a given value. *limit* is an integer-type expression. *ch* is an ordinal-type expression. *x* is a variable reference of any type. *ScanEQ* scans *x*, looking for the first occurrence of the byte value given by *ch*. The scan begins with the first byte in *x*. If *ch* is not found within *limit* characters from the beginning of *x*, the value returned is equal to *limit*. Otherwise, the value returned is the number of bytes scanned before *ch* was found.

The ScanNE Function

Syntax: ScanNE (limit , ch , x)
Result type: *Integer*

ScanNE does the same as *ScanEQ*, except that it searches for a byte value that *does not* match the *ch* parameter.

The Hi Function

Syntax: Hi (x)
Result type: −128 .. 127

Returns the high-order byte of the argument. *x* is an expression of type *Integer*. *Hi* returns the high-order byte of *x* as a signed value.

The Lo Function

Syntax: Lo (x)
Result type: −128 .. 127

Returns the low-order byte of the argument. *x* is an expression of type *Integer*. *Lo* returns the low-order byte of *x* as a signed value.

The Swap Function

Syntax: Swap (x)
Result type: *Integer*

Swaps the high- and low-order bytes of the argument. *x* is an expression of type *Integer*.

The HiWord Function

Syntax: HiWord (x)
Result type: *Integer*

Returns the high-order word of the argument. *x* is an expression of type *LongInt*. *HiWord* returns the high-order word of *x* as a signed value.

The LoWord Function

Syntax: LoWord (x)
Result type: *Integer*

Returns the low-order word of the argument. *x* is an expression of type *LongInt*. *HiWord* returns the low-order word of *x* as a signed value.

The SwapWord Function

Syntax: SwapWord (x)
Result type: *LongInt*

Swaps the high- and low-order words of the argument. *x* is an expression of type *LongInt*.

CHAPTER 26

The Standard Apple Numeric Environment (SANE) Library

This chapter discusses the features and uses of the Standard Apple Numeric Environment (SANE) and the routines contained in the SANE library. You'll learn about the data types provided by SANE. You'll also delve into each of the types, procedures, and functions contained in the SANE library.

SANE's main features are based on a standard proposed by the Institute of Electrical and Electronics Engineers (IEEE's Standard 754 for Binary Floating-Point Arithmetic). This standard specifies standardized data types, arithmetic, and conversions.

All floating-point mathematical calculations performed by Turbo Pascal use SANE's standards. This means uniform floating-point operations, which return the most accurate results.

To surpass real-type precision, SANE offers floating-point type extensions. The SANE library provides numerical functions beyond those in Standard Pascal and routines to control the environment for floating-point calculations.

The SANE Data Types

SANE provides three application data types (*Single*, *Double*, and *Comp*) and an arithmetic type (*Extended*). The original specification for Pascal had one data type for use with floating-point numbers, the *Real* type. Turbo Pascal offers—in addition to the *Real* type—the *Single*, *Double*, *Extended*, and *Comp* types. *Extended* alludes to the extended precision with which Turbo Pascal performs all arithmetic operations. Values that can be represented as any of the first three types can be represented also in *Extended*. Turbo Pascal's *Real* type is identical to SANE's *Single* type.

The *Single* type is the smallest format you can use with floating-point numbers. It uses 32 bits of memory.

The *Double* type uses 64 bits for storage.

The *Extended* type uses an 80-bit format. Any arithmetic involving real-type values is performed with the *Extended* type.

The *Comp* type stores integral values in 64 bits. It is considered a real-type because arithmetic done with operands of type *Comp* uses the *Extended* type.

Choosing a Data Type

Four factors to consider when selecting a data type are format (fixed or floating-point), precision, range, and memory usage.

The precision, range, and memory usage for each SANE data type are shown in Table 26-1.

Table 26-1 SANE Data Types

Type Identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	−126	−1022	—	−16383
Maximum	127	1023	—	16384
Significand precision				
Bits	24	53	63	64
Decimal digits	7–8	15–16	18–19	19–20
Decimal range (approximate)				
Min negative	−3.4E+38	−1.7E+308	≈−9.2E18	−1.1E+4932
Max neg norm	−1.2E–38	−2.3E–308		1.7E–4932
Max neg denorm	−1.5E–45	−5.0E–324		−1.9E–4951
Min pos norm	1.2E–38	2.3E–308		1.7E–4932
Min pos denorm	−1.5E–45	−5.0E–324		−1.9E–4951
Max positive	3.4E+38	1.7E+308	≈9.2E18	1.1E+4932
Infinities	Yes	Yes	No	Yes
NaNs	Yes	Yes	Yes	Yes

Many programs require a counting type that counts things exactly. The SANE type *Comp* can be used for this purpose, for instance by representing monetary values as integral numbers of cents or mils (thousands). The sum, difference, or product of any two *Comp* values can be calculated exactly if the magnitude of the result does not exceed 2^{63-1} (9,223,372,036,854,775,807). *Comp* values can be combined with extended values in floating-point computations (such as calculating compound interest).

Arithmetic with *Comp* type variables, like all SANE arithmetic, is performed internally using the *Extended* type. Precision is not affected, as conversion from *Comp* to *Extended* is always exact.

Values Represented

The floating-point types (*Single*, *Double*, and *Extended*) store binary representations of a *sign* (+ or −), an *exponent*, and a *significand*. A represented number has the value

```
± significand * 2^exponent
```

where the significand has a single bit to the left of the binary point (that is, 0 ≤ significand < 2).

Range and Precision

Table 26-1 shows the range and precision of the real-types supported by SANE and Turbo Pascal. Decimal ranges are shown as chopped two-digit decimal representations of the exact binary values.

Using the *Single* type, the largest representable number is:

```
significand = 2 - 2^23
            = 1.11111111111111111111111₂
exponent    = 127
value       = (2 - 2^-23) * 2^127
            ≈ 3.403 * 10^38
```

The smallest representable positive normalized number is:

```
significand = 1
            = 1.00000000000000000000000₂
exponent    = -126
value       = 1 * 2^-126
            ≈ 1.175 * 10^-38
```

The smallest representable positive denormalized number is:

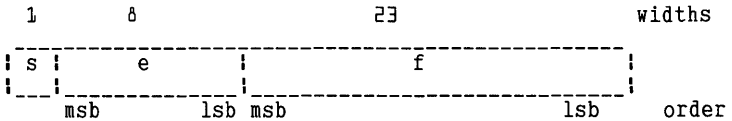
```
significand = 2^-23
            = 0.00000000000000000000001₂
exponent    = -126
value       = 2^-23 * 2^-126
            ≈ 1.401 * 10^-45
```

Formats

Following are the formats of the four SANE real-types, showing the fields that each type divides a number into.

The Single Type

A 32-bit *Single* number is divided into three fields:



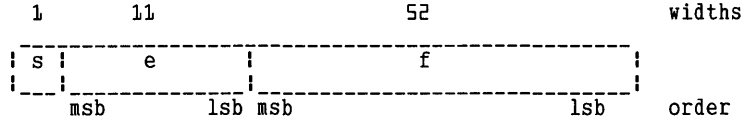
NOTE: *msb* means most significant bit and *lsb* means least significant bit.

The value *v* of the number is determined by these fields:

```
If 0 < e < 255,          then v = (-1)^s * 2^(e-127) * (1.f).
If e = 0 and f ≠ 0,      then v = (-1)^s * 2^(-126) * (0.f).
If e = 0 and f = 0,      then v = (-1)^s * 0.
If e = 255 and f = 0,    then v = (-1)^s * ∞.
If e = 255 and f ≠ 0,    then v is a NaN.
```

The Double Type

A 64-bit *Double* number is divided into three fields:

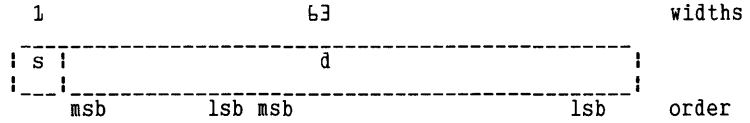


The value *v* of the number is determined by these fields:

```
If 0 < e < 2047,         then v = (-1)^s * 2^(e-1023) * (1.f).
If e = 0 and f ≠ 0,      then v = (-1)^s * 2^(-1022) * (0.f).
If e = 0 and f = 0,      then v = (-1)^s * 0.
If e = 2047 and f = 0,   then v = (-1)^s * ∞.
If e = 2047 and f ≠ 0,   then v is a NaN.
```

The Comp Type

A 64-bit *Comp* number is divided into two fields:

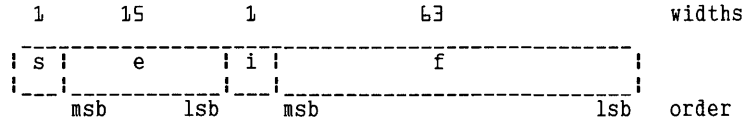


The value *v* of the number is determined by these fields:

```
If s = 1 and d = 0,      then v is the unique Comp NaN.
Otherwise,               v is the two's-complement value
                         of the 64-bit representation.
```

The Extended Type

An 80-bit *Extended* format number is divided into four fields:



The value v of the number is determined by these fields:

```
If 0 <= e < 32767,          then v = (-1)^s * 2^(e-16383) * (i.f).
If e = 32767 and f = 0,     then v = (-1)^s * ∞, regardless of i.
If e = 32767 and f ≠ 0,     then v is a NaN, regardless of i.
```

The SANE Engine

This section describes the features of SANE and the interface between the SANE engine and Turbo Pascal.

Extended Arithmetic

The *Extended* type is the basis of all arithmetic computation. With *Extended* results, computations are accurate to a precision of 19 decimal digits through a range of 10^{-4900} to 10^{+4900}

Turbo Pascal uses the *Extended* format to store all non-integer numeric constants and evaluates all non-integer numeric expressions to *Extended*. The entire right side of the following assignment, for instance, will be computed in *Extended* before being converted to the type of the left side:

```
var
  X,A,B,C: Real;
begin
  X := (B + Sqrt(B * B - A * C)) / A;
end
```

With no special effort by the programmer, Turbo Pascal does computations using the precision and range of the *Extended* type. The added precision means smaller roundoff errors, and the additional range means overflow and underflow are less common, so that programs work more often.

You can go beyond Pascal's automatic *Extended* capabilities. For example, you can declare variables used for intermediate results to be of type *Extended*. The following example computes a sum of products:

```
var
  Sum: Real;
  X,Y: array[1..100] of Real;
  I: Integer;
  T: Extended;                    { for intermediate results }
begin
  T := 0.0;
  for I := 1 to 100 do T := T + X[I] * Y[I];
  Sum := T;
end
```

Had *T* been declared *Real*, the assignment to *T* would have caused a roundoff error at the limit of single precision at each loop entry. But *T* being *Extended*, all roundoff errors are at the limit of *Extended* precision, except for the one resulting from the assignment of *T* to *Sum*. Fewer roundoff errors mean more accurate results.

You can also declare formal value parameters and function results to be of type *Extended* to avoid unnecessary conversions between numeric types, which may result in loss of accuracy. For example:

```
function Area(radius: Extended): Extended;
begin
  Area := pi * radius * radius;
end;
```

Although the *Extended* type makes programs less sensitive to certain errors, exceptional cases do arise. For example, if all variables in the example below are of type *Real*:

```
Average := Sum / Count; Area := Side * Side; +
```

what happens if *Count* is zero or if the product *Side* * *Side* is too large to be represented in the *Real* format? Ordinarily, your program stops, displaying an error message. However, this is not the only way Turbo Pascal can treat such errors. Instead, Turbo Pascal can assign special values to *Average* and *Area*, so your program can continue. In fact, the IEEE standard refers to "exceptions" rather than "errors," and it specifies "no halts" as the default mode of operation for its arithmetic. To install the IEEE defaults, use this statement:

```
SetEnvironment(0);
```

The SANE library also contains functions and procedures for determining when exceptional cases occur.

Number Classes

There are five classes of representations in the SANE data formats.

- **Normalized numbers** Binary floating-point numbers with a leading significand bit of 1.
- **Zero** +0 and −0.
- **Infinities** Special bit patterns resulting when floating-point operations attempt to produce numbers beyond the largest representable number of the intended format.
- **NaNs** A bit pattern resulting when a meaningful result cannot be produced by a floating-point operation.

- **Denormalized numbers** Non-zero binary floating-point numbers whose significands have leading bits of zero and whose exponents are the minimum exponents for the number's storage type.

Infinities

Infinities are special SANE representations that can arise in two ways from operations on finite values:

- When an operation should produce an exact mathematical infinity (such as 1/0), the result is an infinity.
- When an operation produces a number with magnitude too large for the intended floating-point format, the result may (depending on the current rounding direction) be an infinity.

Turbo Pascal predefines a constant, *Inf*, to have the value *positive infinity. Inf* also represents infinity for input and output of floating-point values. Infinities act like mathematical infinities, for example, $1-Inf = -Inf$.

Here's an example of the use of infinity values:

```
program UseInf;
uses SANE;
var
  X: Extended;
begin
  SetEnvironment(0);
  X := 1e4000;
  WriteLn('X*X = ',X*X);
  WriteLn('1/(X*X) = ',1/(X*X));
  WriteLn('1+1/(X*X) = ',1+1/(X*X));
end.
```

NaNs

Another special SANE representation is NaN (Not-a-Number). A NaN is produced whenever an operation cannot return a meaningful result. For example, *0/0* and *Sqrt(−1)* produce NaNs.

Each time a NaN is generated, an associated NaN code is returned as part of the NaN's representation. The code tells you what kind of operation caused the NaN. Table 26-2 shows these NaN codes, which you can use in debugging.

Table 26-2 NaN Codes

Code	Meaning
1	Invalid square root, such as *Sqrt*(−1)
2	Invalid addition, such as (+*Inf*) − (+*Inf*)
4	Invalid division, such as 0/0
8	Invalid multiplication, such as 0 × *Inf*
9	Invalid remainder, such as *Remainder*(*X*,0,*Q*)
17	Attempt to convert invalid ASCII string
20	Result of converting the *Comp* NaN to floating-point format
21	Attempt to create a NaN with a zero code
33	Invalid argument to trig routine
34	Invalid argument to inverse trig routine
36	Invalid argument to log routine
37	Invalid argument to x^i or x^y routine
38	Invalid argument to financial function
255	Uninitialized storage (signaling NaN)

The statement *WriteLn(0/0)* produces the result NAN(004) (assuming the invalid operation halt is off). NAN(004), nan(4), and NaN are all acceptable ways of reading a NaN into a SANE variable.

Denormalized Numbers

When possible, SANE stores values in normalized form; that is, the most significant bit of the significand is a one rather than a zero.

However, when a very small number is being stored, and the exponent is the smallest possible value, you can store still smaller values by storing leading zeroes. For example, $1.0..0_2 \times 2^{-126}$ is the smallest normalized *Real*, and $0.1..0_2 \times 2^{-126}$ is a still smaller denormalized *Real*.

The Environment

The SANE environment consists of various settings that are global to all routines in the SANE library:

- Rounding direction
- Rounding precision
- Exception flags
- Halt settings

The SANE library includes procedures and functions that let you determine the current status of the environment, as well as change any of its settings.

When your program begins, the environment is set to these Turbo Pascal defaults:

- Rounding direction: To nearest
- Rounding precision: Extended
- All exception flags cleared
- Halts on invalid operation, underflow, and divide-by-zero

The entire SANE environment can be encoded in a value of the SANE type *Environment*. The *GetEnvironment*, *SetEnvironment*, *ProcEntry*, and *ProcExit* procedures access the current SANE environment as a whole.

Rounding Direction

The rounding direction can be set in four ways:

- To nearest
- Upward
- Downward
- Toward zero

The default rounding direction is to nearest. You can find the current rounding direction using the *GetRound* function and change it using the *SetRound* function.

The following code shows how to save the current rounding direction, compute a function using toward zero rounding, and then restore the saved rounding direction.

```
var
  R: RoundDir;
  X,Y: Extended;
begin
  R := GetRound;
  SetRound(TowardZero);
  Y := F(X);
  SetRound(R);
end
```

Rounding Precision

The rounding precision can be set in three ways:

- Extended precision
- Double precision
- Real precision

The default rounding precision is extended. You can use SANE to perform calculations and then simulate the results you would get using a system without extended precision arithmetic. The rounding precision is accessed through the *SetPrecision* procedure and the *GetPrecision* function.

Exception Flags

Exceptions can arise from floating-point calculations in a number of ways. For example, multiplying two very large values can result in a value too large to be represented in one of the SANE data types.

SANE lets your program determine when a floating-point calculation has resulted in one of these exceptions. Exceptions fall into five categories:

- Invalid operation
- Underflow
- Overflow
- Divide-by-zero
- Inexact

Whenever one of these exceptions occurs, a corresponding flag is set in the environment. The flag remains set until explicitly cleared. The exception flags are accessed through the *SetException* procedure and the *TestException* function. Below follows a description of each of the individual exceptions.

Invalid Operation: The invalid operation exception occurs when the operands of an operation are invalid, so that a meaningful numeric result is impossible, for example, *0/0* and *Sqrt(−1)*.

Underflow: Underflow occurs when a result is both denormalized and has lost significant digits through rounding. For example, to return the result of

$(1.00000000000000000000001_2 * 2^{-126}) / 2$

to the real format, a leading zero would be introduced and last significant bit would be lost in rounding. The result

$0.100000000000000000000000_2 * 2^{-126}$

would be returned, signaling an underflow.

Overflow: Calculating a value that is too large to fit in the format of its designated type is an overflow. The destination format must be a floating-point type; if the destination format is an integral type, the invalid exception occurs.

Divide-by-zero: This exception occurs when a finite non-zero number is divided by zero. It also occurs when an operation on finite operands produces an exact infinite result. For example, *1/0* (which results in *Inf*) and *Ln(0)* (which results in *-Inf*) both signal divide-by-zero.

Inexact: The inexact exception occurs if the rounded result of an operation is not identical to its mathematical (exact) result. Whenever an overflow or underflow occurs, inexact is also signaled. For instance, 2/3 signals inexact, regardless of the floating-point format used.

Halt Settings

The SANE environment includes a halt setting for each exception that determines whether occurrence of the exception halts the program. By default, Turbo Pascal sets a halt on invalid operation, overflow, and divide-by-zero. The IEEE standard default calls for all halts off.

You can access the halt settings using the *TestHalt* function and the *SetHalt* procedure.

The SANE Library

The SANE library is implemented as a unit in Turbo Pascal. To use the features provided by SANE, you must specify *SANE* in the uses-clause of your program or unit:

```
uses SANE;
```

The rest of this chapter explains each of the constants, types, functions, and procedures contained in the SANE library. Some advanced and rarely used features have been left out; refer to the *Apple Numerics Manual* for a discussion of these.

Constants and Types

Each of the constants and types defined by SANE are briefly discussed in this section. For more information, see the descriptions of the procedures and functions that depend on these constants and types.

The DecStrLen Constant

The *DecStrLen* constant is defined by:

```
DecStrLen = 255;
```

DecStrLen is the maximum length of a decimal numeric string. It is the size attribute of variables of type *DecStr*.

Exception Condition Constants

These declarations specify the exception condition constants:

```
Invalid   = 1;
Underflow = 2;
Overflow  = 4;
DivByZero = 8;
Inexact   = 16;
```

These constants are used to form a value of the *Exception* type. For example, if *E* is a variable of type *Exception*, then:

```
E := Invalid + Overflow + DivByZero;
```

gives *E* a value that represents these three exceptions collectively.

The *SetException* and *SetHalt* procedures take arguments of type *Exception*. The *TestException* and *TestHalt* functions return a value of type *Exception*.

The DecStr Type

This declaration defines the *DecStr* (decimal string) type:

```
DecStr = string[DecStrLen];
```

It is a string with a size attribute of *DecStrLen* (255 characters). It is used to hold the decimal representation, in ASCII characters, of a number.

The DecForm Type

The following declaration defines the decimal format record type:

```
DecForm = record
            Style: (FloatDecimal, FixedDecimal);
            Digits: Integer;
          end;
```

A *DecForm* record holds the specifications for the format of a decimal number. Its *Style* field specifies whether the decimal representation will be floating-point or fixed-point. Its *Digits* field holds the number of significant digits for float style or the number of digits to the right of the decimal point for fixed style.

The *Num2Str* procedure takes a *DecForm* argument.

The RelOp Type

The relational operator type is defined by

```
RelOp = (GreaterThan, LessThan, EqualTo, Unordered);
```

A result of this type is returned by the *Relation* function.

The NumClass Type

The number class type is defined by

```
NumClass = (SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum);
```

whose members are described in Table 26-3.

Table 26-3 *Number Class Descriptions*

Number Class	Meaning
SNaN	Signaling NaN
QNaN	Quiet NaN
Infinite	Infinity or −Infinity
ZeroNum	0 or −0
NormalNum	Normalized number
DenormalNum	Denormalized number

A Quiet NaN moves through floating-point operations without signaling an exception (or halting a program). Signaling NaNs are not used in SANE operations. They signal an Invalid exception when the NaN is an operand of an arithmetic operation.

The inquiry functions return results of type *NumClass*.

The Exception Type

A variable of type *Exception* has an integer value corresponding to the value of an *Exception* constant or to a sum of two or more of the *Exception* constants. The *Exception* type is defined by

```
Exception = Integer;
```

The *SetException*, *TestException*, *SetHalt*, and *TestHalt* routines all take arguments of this type.

The RoundDir Type

The rounding direction type is defined by

```
RoundDir = (ToNearest, Upward, Downward, TowardZero);
```

The *RoundDir* type determines how values are to be rounded when rounding becomes necessary during arithmetic operations or conversions. The *SetRound* procedure takes an argument of type *RoundDir*, while the *GetRound* function returns a value of type *RoundDir*.

The RoundPre Type

The rounding precision type is defined by

```
RoundPre = (ExtPrecision, DblPrecision, RealPrecision);
```

Rounding precision can be used to simulate arithmetic using only single or double precision. The *SetPrecision* procedure takes an argument of type *RoundPre*, while the *GetPrecision* function returns a value of type *RoundPre*.

The Environment Type

A variable of type *Environment* represents the settings of the SANE environment. A value of 0, for example, represents the default IEEE settings. The *Environment* type is defined by

```
Environment = Integer;
```

Use a variable of type *Environment* with the environmental access routines *SetEnvironment*, *GetEnvironment*, *ProcEntry*, and *ProcExit*.

Conversion Procedures and Functions

The SANE library contains procedures and functions that convert numeric values from one binary format to another, from binary to decimal, and from decimal to binary. These conversion procedures and functions are described in the following sections.

The Num2Integer and Num2Longint Functions

```
function Num2Integer(x: Extended): Integer;
function Num2LongInt(x: Extended): LongInt;
```

The *Num2Integer* function takes an *Extended* argument and returns a result of type *Integer*. The *Num2LongInt* function takes an *Extended* argument and returns a result of type *LongInt*.

The value returned by these functions depend on the rounding direction (set with the *RoundDir* procedure). If you were to use the standard rounding direction ToNearest, for example,

```
Num2Integer(99.6);
Num2LongInt(99.6);
```

return the value 100.

Num2Integer and *Num2Longint* are like Turbo Pascal's *Round* and *Trunc* functions. However, *Num2Integer* and *Num2Longint* are affected by the current rounding direction, whereas the *Round* function always returns the nearest *Longint* value, and the *Trunc* function always rounds toward zero.

Using the ToNearest rounding direction, *Num2Integer* and *Num2Longint* round values that are halfway between two integers to the nearest even integer as indicated by the IEEE standard. For example, *Num2Integer(2.5)* returns 2. The *Round* function rounds these halfway values away from zero—*Round(2.5)* returns 3.

The Num2Extended Function

```
function Num2Extended(x: Extended): Extended;
```

Any real-type or integer-type argument can be passed to the *Num2Extended* function. It converts its argument to the *Extended* format, which forces floating-point arithmetic when all variables involved are of integer-types.

The Num2Str Procedure

```
procedure Num2Str(f: DecForm; x: Extended; var s: DecStr);
```

Converts an extended value *x* to a decimal string, returned in *s*, using the specifications in the *DecForm* record *f*. The *Style* field of *f* determines the formatting style. If *f.Style* is *FloatDecimal*, the number is formatted in floating-point style, and *f.Digits* determines the number of significant digits:

```
[ + | - ] <digit> [ . <decimals> ] e [ + | - ] <exponent>
```

These are the components of the output string:

`[ ¦ - ]`	" " or "−" according to the sign of *x*.
`‹digit›`	Single digit, "0" only if *x* is 0.
`[ . ‹decimals› ]`	Digit string, present if *f.Digits* ⟩ 1.
`e`	Lowercase "e" character.
`[ + ¦ - ]`	"+" or "−" according to sign of exponent.
`‹exponent›`	One to four exponent digits.

If *f.Style* is *FixedDecimal*, the number is formatted in fixed-point style, and *f.Digits* determines the number of digits to follow the decimal point:

```
[ - ] ‹digits› [ . ‹decimals› ]
```

These are the components of the output string:

`[ - ]`	"−" if *x* is negative.
`‹digits›`	At least one digit, but no leading zeros.
`[ . ‹decimals› ]`	Decimals if *f.Digits* ⟩ 0.

If *f.Style* is *FixedDecimal* and *x* is greater than or equal to 10 ^ (19 − *f.Digits*), the formatter will select floating-point style with 19 significant digits. In general, if *f.Digits* is outside the range 0..72, it is truncated to be within that range.

NaNs (Not a Number values) are formatted as NAN(ddd) where ddd is a three-decimal-digit code telling the origin of the NaN. Infinities are formatted as INF. A sign or space is prepended according to the selected style.

The Str2Num Function

```
function Str2Num(s: DecStr): Extended;
```

Converts a decimal string argument of type *DecStr* to a value of type *Extended*. The string may contain leading blanks or TABs, but no trailing characters are allowed. Examples of acceptable input are

```
123     123.4E-12     -123.     .456     3e9     -0
-INF     Inf     NAN(12)     -NaN()     nan
```

The accepted syntax is presented below using Backus-Naur form:

`‹decimal number›`	::= [{space ¦ tab}] ⟨left decimal⟩
`‹left decimal›`	::= [+¦−] ⟨unsigned decimal⟩
`‹unsigned decimal›`	::= ⟨finite number⟩ ¦ ⟨infinity⟩ ¦ ⟨NAN⟩
`‹finite number›`	::= ⟨significand⟩ [⟨exponent⟩]
`‹significand›`	::= ⟨integer⟩ ¦ ⟨mixed⟩
`‹integer›`	::= ⟨digits⟩ [.]

‹digits›	::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |8| 9}
‹mixed›	::= [⟨digits⟩] . ⟨digits⟩
‹exponent›	::= E [+|−] ⟨digits⟩
‹infinity›	::= [+|−] INF
‹NAN›	::= NAN [([⟨digits⟩])]

Note: In the table, square brackets enclose optional items, braces (curly brackets) enclose elements to be repeated at least once, and vertical bars separate alternative elements. Letters that appear literally, like the *E* marking the exponent field, may be either uppercase or lowercase.

If the string is syntactically incorrect, *Str2Num* returns NAN(017), which is the code for invalid Decimal to Binary conversion. If the resulting value is outside the floating-point range, *Str2Num* returns INF or −INF according to the sign of the value.

Arithmetic and Auxiliary Functions

The SANE library includes a set of functions that supplement the standard functions described in Chapter 25.

The Remainder Function

```
function Remainder(x,y: Extended; var quo: Integer);
```

Returns an exact remainder of the smallest possible magnitude resulting from the division of its two *Extended* arguments *x* and *y*, as prescribed by the IEEE standard. The result is computed as

```
x-n*y
```

where *n* is the nearest integral approximation to the quotient *x/y*. For example, *Remainder(9,5,q)* returns −1, since −1 = 9−2×5. The integer variable argument *quo* receives the seven low-order bits of *n* as a value between −127 and 127. This is handy when programming a function that requires argument reduction.

The Pascal operator **mod** can be used only with integral values. The *Remainder* function deals with real-type or integer-type values.

The Rint Function

```
function Rint(x: Extended): Extended;
```

Rounds x to an integral value. All sufficiently large floating-point values are integral. Use the *SetRound* procedure to change the rounding direction for the result you want. *Rint* is the same as the standard function *Int*, except that *Int* always rounds toward zero.

The Scalb Function

```
function Scalb(n: Integer; x: Extended): Extended;
```

Scales x by the power to two specified by n. The value $2^n \times x$ is returned in *Extended* format.

The Logb Function

```
function Logb(x: Extended): Extended;
```

Returns the largest power of two that does not exceed the magnitude of x.

The CopySign Function

```
function CopySign(x,y:extended):extended;
```

Returns the value of y with the sign of x.

The NextReal Function

```
function NextReal(x,y: Real): Extended;
```

Returns the next *Real* format value after x in the direction of y.

The NextDouble Function

```
function NextDouble(x,y: Double): Extended;
```

Returns the next *Double* format value after x in the direction of y.

The NextExtended Function

```
function NextExtended(x,y: Extended): Extended;
```

Returns the next *Extended* format value after x in the direction of y.

Turbo Pascal provides the predefined *Ln*, *Exp*, *Sin*, *Cos*, and *ArcTan* functions. The SANE library complements these with the *Log2*, *Ln1*, *Exp2*, *Exp1*, *XpwrI*, *XpwrY*, and *Tan* functions.

The Log2 Function

```
function Log2(x: Extended): Extended;
```

Returns the base-2 logarithm of x.

The Ln1 Function

```
function Ln1(x: Extended): Extended;
```

Returns the base-e logarithm of 1 plus x, that is, $Ln1(x) = Ln(1+x)$. For x near 0, $Ln1(x)$ is more accurate than $Ln(1+x)$.

The Exp2 Function

```
function Exp2(x: Extended): Extended;
```

Returns 2 raised to the power of x, that is, 2^x.

The Exp1 Function

```
function Exp1(x: Extended): Extended;
```

Returns e raised to the power of x, minus one, that is, e^x-1. For x near 0, $Exp1(x)$ is more accurate than $Exp(x)-1$.

The XpwrI Function

```
function XpwrI(x: Extended; i: Integer): Extended;
```

Returns x raised to the power of i, that is, x^i.

The XpwrY Function

```
function XpwrY(x,y: Extended): Extended;
```

Returns x raised to the power of y, that is, x^y.

The Tan Function

```
function Tan(x: Extended): Extended;
```

Returns the tangent of x. In a right triangle, *Tan(x)* is the ratio of the length of the side opposite an angle of x radians to the length of the side adjacent to it. x must be expressed in radians.

Financial Functions

SANE offers two functions for financial applications: *Compound* and *Annuity.*

The Compound Function

```
function Compound(r,n: Extended): Extended;
```

Returns the compound interest. r specifies the interest rate and n specifies the number of periods. The value returned is $(1+r)^n$, which is the principal plus accrued compound interest on an original one-unit investment.

The Annuity Function

```
function Annuity(r,n: Extended): Extended
```

r specifies the interest rate and n specifies the number of periods. *Annuity* returns $(1-(1+r)^{-n})/r$, the present value factor of an ordinary annuity. It returns an *Extended* value. Following is an example of use of the *Annuity* function:

```
program PayBack;
var
  Loan,Payment,Interest,Periods: Extended;
begin
  Write('Loan amount: ');
  ReadLn(Loan);
  Write('Annual interest rate (enter as a decminal): ');
  ReadLn(Interest);
  Write('Number of years: ');
  ReadLn(Periods);
  Payment := Loan / Annuity(Interest / 12, Periods * 12);
  WriteLn('Your payment is: ',Payment:8:2);
end.
```

SANE offers four functions that let you determine the class of a numeric value, and one that returns the sign of a numeric value. The result of the four classification functions is of type *NumClass*.

The ClassReal Function

```
function ClassReal(x: Real): NumClass;
```

Returns the number class of the *Real* type value *x*. For example, *ClassReal(1)* returns *NormalNum*, which is the code for a normalized number. *ClassReal (1e–310)* returns *ZeroNum*, the code for zero, because 1e−310 rounds to +0 in the *Real* format.

The ClassDouble Function

```
function ClassDouble(x: Double): NumClass;
```

Returns the number class of the *Double* type value *x*. For example, *ClassDouble(0.0/0.0)* returns *QNan*, and *ClassDouble(1e–310)* returns *DenormalNum*, because 1e–310 is denormalized in the *Double* format.

The ClassExtended Function

```
function ClassExtended(x: Extended): NumClass;
```

Returns the number class of the *Extended* type value *x*. For example, *ClassExtended(1/0)* returns *Infinite*, and *ClassExtended(1e–310)* returns *NormalNum*.

The ClassComp Function

```
function ClassComp(x: Comp): NumClass;
```

Returns the number class of the *Comp* type value *x*. For example, *ClassComp(1)* returns *NormalNum*, while *ClassComp(0.1)* returns *ZeroNum*, as *Comp* stores only integral values.

The SignNum Function

```
function SignNum(x: Extended): Integer;
```

Returns an integer value that reflects the sign of *x*: 1 is returned if *x* is negative, and 0 is returned if *x* is positive.

Miscellaneous Functions

This section describes the *RandomX*, *NaN*, and *Relation* functions.

The RandomX Function

```
function RandomX(var x: Extended): Extended;
```

RandomX takes a variable argument of type *Extended* that contains an integral value in the range $1 < r < 2^{31}-2$. It returns the next random number (in *Extended* format) in sequence within the same range. The variable argument is updated to the value returned. *RandomX* uses the algorithm:

```
NewX = (7^5 * OldX) mod (2^31-1)
```

The NaN Function

```
function NaN(x: Integer): Extended;
```

Returns a NaN with the code specified by x. See Table 26-2 for a list of defined NaN codes.

The Relation Function

```
function Relation(x,y: Extended): RelOp;
```

Returns a value of type *RelOp* that specifies the relationship between x and y. For example,

```
Relation(0.1,NaN(0));
```

returns *Unordered*, since all comparisons involving NaNs are unordered.

Environmental Access Procedures and Functions

The SANE library provides a number of procedures and functions to access the SANE environment. They are described in this section.

The GetRound Function

```
function GetRound: RoundDir;
```

Returns the current rounding direction.

The SetRound Procedure

```
procedure SetRound(r: RoundDir);
```

Sets the rounding direction to the value specified by *r*.

The GetPrecision Function

```
function GetPrecision: RoundPre;
```

Returns the current rounding precision.

The SetPrecision Procedure

```
procedure SetPrecision(p: RoundPre);
```

Sets the rounding precision to the value specified by *p*.

The TextException Function

```
function TestException(e: Exception): Boolean;
```

Returns *true* if any of the exceptions encoded in *e* are set. For example,

```
Error := TestException(Overflow + Inexact);
```

would set *Error* to *true* if the overflow and/or inexact exception flags were set.

The SetException Procedure

```
procedure SetException(e: Exception; b: Boolean);
```

The exceptions encoded in *e* are set or cleared according to the value of *b*: If *b* is true, the flags are set; if *b* is false, the flags are cleared. For example,

```
SetException(Overflow + Inexact, true);
```

This statement signals the overflow and inexact exceptions. If halt on overflow or inexact were set, this statement would halt the program.

The TestHalt Function

```
function TestHalt(e: Exception): Boolean;
```

Returns true if any of the halt flags encoded in *e* are set, that is, if halts are enabled for any of the specified exceptions.

The SetHalt Procedure

```
procedure SetHalt(e: Exception; b: Boolean);
```

The halt flags encoded in *e* are set or cleared according to the value of *b*: If *b* is true the flags are set, and if *b* is false the flags are cleared. When a specific flag is set and the corresponding exception occurs, the program comes to a halt.

The GetEnvironment Procedure

```
procedure GetEnvironment(var e: Environment);
```

Stores the current settings of the environment in *e*.

The SetEnvironment Procedure

```
procedure SetEnvironment(e: Environment);
```

Sets the environment to the value encoded in *e*. To install the IEEE standard defaults, use the statement

```
SetEnvironment(0);
```

The following procedure runs with the IEEE default environment, while preserving the caller's environment:

```
procedure P;
var
  E: Environment;
begin
  GetEnvironment(E);
  SetEnvironment(0);
  :
  :
  :
  SetEnvironment(E);
end;
```

The ProcEntry Procedure

```
procedure ProcEntry(var e: Environment);
```

Stores the current settings of the environment in *e*, and then sets the environment to the IEEE defaults. The statement

```
ProcEntry(E);
```

is equivalent to

```
GetEnvironment(E);
SetEnvironment(0);
```

The ProcExit Procedure

```
procedure ProcExit(e: Environment);
```

ProcExit temporarily saves the current exception flags, then sets the environment to the value encoded in *e*, and finally signals the temporarily saved exceptions.

ProcEntry and *ProcExit* can be used in routines to hide specific spurious exceptions from the caller, for example,

```
function ArcCos(x: Extended): Extended;
var
  E: Environment;
begin
  ProcEntry(E);
  ArcCos := ArcTan(Sqrt(1.0-x)/(1.0+x)));
  SetException(DivByZero,false);
  ProcExit(E);
end;
```

ProcEntry(E) saves the caller's environment in *E* and sets IEEE defaults, so exceptions cannot halt the routine. If $x = 1$, the computation of *ArcCos* signals divide-by-zero, even though *ArcCos* is assigned the correct value (Pi/2). The call to *SetException* clears the divide-by-zero flag, so the caller never sees it. If $x > 1$ or $x < -1$, the computation of *ArcCos* signals invalid operation. The *ProcExit* procedure will resignal invalid operation after it restores the caller's environment, and if the caller's environment calls for halts on invalid operation, the halt occurs.

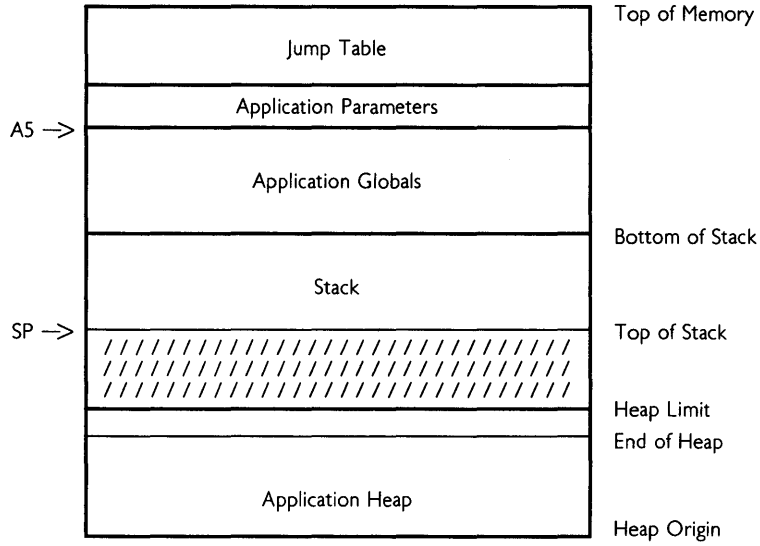
CHAPTER 27

Inside Turbo Pascal

This chapter provides additional information for advanced Pascal programmers. It covers Macintosh architecture, internal data formats, interfacing with assembly language, and defining your own device drivers.

Macintosh Architecture

The environment in which a Macintosh application runs may be divided into five basic components: the *Jump Table*, the *Application Parameters*, the *Application Globals*, the *Stack*, and the *Application Heap*. The organization is shown on the following page.



When an application starts up, the processor's A5 register is set to point to the boundary between the Application Parameters and the Application Globals. The Application Parameters and the Jump Table thus reside at positive offsets from A5 (above A5) and the Application Globals reside at negative offsets from A5 (below A5).

The Application Parameters area occupies the first 32 bytes above A5. It contains the *QuickDraw* Globals Pointer at 0(A5) and the FINDER Startup Handle at 16(A5).

The remainder of the area above A5 is occupied by the Jump Table, which is maintained by the *Segment Loader*. For each segment, it contains one eight-byte entry for every procedure or function that is referenced from another segment. When a segment is in the "unloaded" state, its Jump Table entries contain code that cause the segment to be loaded. When a segment is in the "loaded" state, its Jump Table entries contain instructions that jump to the routines.

The area below A5 contains the application's global variables and the *QuickDraw* global variables.

The Stack follows below the Application Globals. When an application starts up, the processor's *stack pointer* register (referred to as A7 or SP) is set to point just below the Application Globals.

Stack space is always allocated and released in LIFO (last-in/first-out) order: The last item allocated is always the first to be released. The SP register always points to the "top" of the stack — note that the stack grows *downwards*, and that the "top" of the stack is actually the lower end of the stack in memory.

The LIFO nature of the stack makes it convenient for memory allocation connected with the activation and deactivation of procedures and functions. Each time a routine is called, space is allocated for a *stack frame*. The stack frame holds the routine's parameters, local variables, and return address. Upon exiting, the stack frame is released, restoring the stack to the same state it was in when the routine was called. The processor's A6 register functions as a stack frame pointer.

The Application Heap is an area in memory from which storage can be allocated or deallocated in any order. The heap is maintained by the *Memory Manager*. All dynamic storage required by a program is allocated on the heap. This includes dynamic variables (*New* and *Dispose*), code segments, resources, windows, menus, and dialogs.

Two types of blocks can be allocated on the heap: *Non-relocatable* blocks and *relocatable* blocks.

Non-relocatable blocks reside at a fixed locations in the heap, and cannot be moved once they have been allocated. A non-relocatable block is referenced through a *pointer*.

Relocatable blocks may be moved by the Memory Manager to make room for new allocation requests. As a relocatable block may move, it cannot be referenced through a pointer. Instead, the Memory Manager maintains a single non-relocatable *master pointer* to each relocatable block: When the block moves, the master pointer is updated to reflect the new position of the block. You access the block through a *handle*, which is a pointer to the master pointer. To get at a block through a handle, the handle is de-referenced twice, for instance *MYHANDLE*^^.

More information on memory management can be found in the Memory Manager chapter of *Inside Macintosh*.

Internal Data Formats

The compiler always aligns variables to even addresses (word boundaries) unless they occupy a single byte.

Integer-Types

An *Integer* is stored as a 16-bit two's-complement number with a range of −32768 to 32767.

A *LongInt* is stored as a 32-bit two's-complement number with a range of −2147483648 to 2147483647.

A subrange of an integer-type is stored as a signed quantity. If the range is within −128 to 127, a byte is used; if the range is within −32768 to 32767, a word is used; otherwise, a longword is used.

Note: The integer-type subrange 0..255 is stored as an unsigned byte if it is part of a packed structure.

Char-Types

A *Char* is stored as a word with the ASCII code in the low-order byte, and 0 in the high-order byte.

A subrange of a char-type is stored as a byte if the range is within #0 to #127; otherwise, it is stored as a word.

Note: A *Char* is stored as an unsigned byte if it is part of a packed structure.

Boolean-Type

A *Boolean* is stored as a byte that may assume the values 0 (*False*) or 1 (*True*).

Enumerated-Types

An enumerated-type is stored as a byte if the enumeration has 128 or fewer values; otherwise, it is stored as a word.

Real-Types

A *Real* or a *Single* is stored in 4 bytes using the IEEE Single-Precision format. A *Double* is stored in 8 bytes using the IEEE Double-Precision format. An *Extended* is stored 10 bytes using the IEEE Extended-Precision format. A *Comp* is stored as a 64-bit two's-complement number.

For further details on floating-point formats, please refer to Chapter 26.

Pointer-Types

A pointer-type is stored as a 32-bit address value. The pointer value **nil** is stored as zero.

String-Types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string. The length byte and the characters are considered unsigned values.

Set-Types

A set is a bit-array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is calculated as

```
(Max div 8) - (Min div 8) + 1
```

where *Min* and *Max* are the lower and upper bounds of the base-type of that set. The byte number of a specific element *E* is

```
ByteNumber = (E div 8) - (Min div 8)
```

and the bit number within that byte is

```
BitNumber = E mod 8
```

where *E* denotes the ordinal value of the element.

Array-Types

An array is stored as a contiguous sequence of variables of the component-type of the array. If the size of the component-type is greater than one, it is rounded up to an even value so that each component resides on an even boundary. The components with the lowest indices are stored at the lowest memory addresses. A multi-dimensional array is stored with the right-most dimension increasing first.

A packed array is identical to the corresponding unpacked array, unless the component-type is *Char* or the integer subrange 0..255. In that case the components are stored as single-byte unsigned quantities.

Record-Types

The fields of a record are stored as a contiguous sequence of variables. If the size of a specific field is greater than one, it is aligned to an even boundary. The first field is stored at the lowest memory address. If the record contains variant parts, then each variant starts at the same address.

A packed record is identical to the corresponding unpacked record, unless any of the fields are of type *Char* or the integer subrange 0..255. In that case, these fields are stored as single-byte unsigned quantities.

File-Types

File-types (typed-files and textfiles) are represented as records that contain 20 bytes of file status information:

```
type

Buffer    = packed array[0..MaxInt] of Char;
BufferPtr = ^Buffer;
ProcPtr   = ^Integer;

FileRec   = record
              fInpFlag: Boolean;
              fOutFlag: Boolean;
              fRefNum: Integer;
              fVRefNum: Integer;
              fBufSize: Integer;
              fBufPos: Integer;
              fBufEnd: Integer;
              fBuffer: BufferPtr;
              fInOutProc: ProcPtr;
            end;
```

fInpFlag is true if the file was opened with *Reset*. *fOutFlag* is true if the file was opened with *Rewrite*. Both are false if the file is closed. *fRefNum* and *fVRefNum* contain the file reference number and the volume reference number.

For typed files, *fBufSize* contains the record length in bytes, and *fBufPos*, *fBufEnd*, *fBuffer*, and *fInOutProc* are unused.

When a textfile is opened, Turbo Pascal allocates an I/O buffer by calling the Memory Manager's *NewPtr* routine and stores a pointer to it in *fBuffer*. The

buffer is deallocated by a call to the Memory Manager's *DisposPtr* routine when the file is closed. *fBufSize* contains the size of the buffer (default 512), *fBufPos* contains the index of the next character to read or write, and *fBufEnd* contains a count of valid characters in the buffer.

If a textfile is associated with a device, *fRefNum* and *fVRefNum* are zero, and *fInOutProc* contains a pointer to the I/O routine that handles I/O for that device. The section "Defining Your Own Devices" in the following pages provides that information.

Calling Conventions

Turbo Pascal uses the standard stack-based parameter passing conventions as defined in *Inside Macintosh*.

Before calling a procedure or function, the parameters are pushed onto the stack in their order of declaration. If a function is being called, storage for the function result is allocated on the stack before any parameters are pushed. Before returning, the procedure or function removes all parameters from the stack, but leaves the function result (if any) on the stack.

The skeleton code for a procedure call is

```
MOVE    pppp,-(SP)      ;Push first parameter
:
MOVE    pppp,-(SP)      ;Push last parameter
JSR     Procedure       ;Call procedure
```

The skeleton code for a function call is

```
SUBQ.L  #nn,-(SP)       ;Make room for result
MOVE    pppp,-(SP)      ;Push first parameter
:
MOVE    pppp,-(SP)      ;Push last parameter
JSR     Function        ;Call function
MOVE    (SP)+,rrrr      ;Get result
```

Parameters are passed either by *reference* or by *value*. When a parameter is passed by reference, a pointer that points to the actual storage location is pushed onto the stack. When a parameter is passed by value, the actual value is pushed onto the stack.

Variable Parameters

Variable parameters (**var** parameters) are always passed by reference, that is, as a pointer that points to the actual storage location.

Value Parameters

Value parameters are passed by value or by reference depending on the type and size of the parameter. In general, if the value parameter occupies 4 bytes or less, the value is pushed directly onto the stack. Otherwise a pointer to the value is pushed, and the procedure or function then copies the value into a local storage location.

NOTE: To keep the stack properly aligned, the 68000 automatically adjusts the stack pointer by 2 instead of 1 when moving a byte-size value to or from the stack. Thus, a byte-size value occupies a word on the stack, where the high-order byte contains the value and the low-order byte is unused.

An *Integer* is passed as a word. A *LongInt* is passed as a long. An integer-type subrange is passed: as a byte if the range is within −128 to 127; as a word if the range is within −32768 to 32767; otherwise, as a long.

A *Char* is passed as a word with the ASCII code in the low-order byte and 0 in the high-order byte. A char-type subrange is passed as a byte if the range is within #0 to #127, otherwise as a word.

A *Boolean* is passed as a byte with the value 0 or 1.

An enumerated-type is passed as a byte if the enumeration has 128 or fewer values; otherwise, it's passed as a word.

A real-type parameter (*Real*, *Single*, *Double*, *Extended*, and *Comp*) is passed as a pointer to an *Extended* value.

A string-type parameter is passed as a pointer to the value.

A set-type parameter is passed as a pointer to an "unpacked" set, which occupies 32 bytes.

Arrays and records whose sizes are less than or equal to 4 bytes are passed by pushing their value onto the stack. Larger arrays and records are passed as a pointer to the value.

Function Results

For function results of type *Integer*, *LongInt*, *Char*, and *Boolean*, and for function results of any subrange or enumerated type, the caller allocates 2 or 4 bytes on the stack before pushing any parameters. The function places the result in these bytes using the same formats as value parameters; that is, the value is returned as a byte, a word, or a long.

For a real-type function result, the caller pushes a pointer to a temporary storage location before pushing any parameters, and the function returns an *Extended* value in that temporary. The caller removes the pointer from the stack when the function returns.

For a string-type function result, the caller pushes a pointer to a temporary storage location before pushing any parameters, and the function returns a string value in that temporary. The caller removes the pointer from the stack when the function returns.

Entry and Exit Code

Each Pascal procedure and function begins and ends with standard entry and exit code which creates and removes its activation.

The standard entry code is

```
LINK    A6,#-dd             ;Set up stack frame
MOVEM.L D3-D7/A2-A4,-(SP)   ;Save registers
```

where *dd* is the number of bytes of local workspace to allocate. The MOVEM instruction is only present if the procedure or function uses any non-scratch registers, and it only saves the registers that are actually used.

The standard exit code is

```
MOVEM.L (SP)+,D3-D7/A2-A4   ;Restore registers
UNLK    A6                  ;Remove stack frame
MOVE.L  (SP)+,A0            ;Get return address
LEA     pp(SP),SP           ;Remove parameters
JMP     (A0)                ;Return
```

where *pp* is the size of the parameters. (If *pp* is less than or equal to 8, an ADDQ.L instruction is used instead of a LEA instruction.) The MOVEM instruction (if present) lists the same registers as its counterpart in the entry code.

If the procedure or function has no parameters the exit code is

```
MOVEM.L (SP)+,D3-D7/A2-A4   ;Restore registers
UNLK    A6                  ;Remove stack frame
RTS                         ;Return
```

Linking with Assembly Language

Procedures and functions written in assembly language may be linked with Turbo Pascal programs or units through {$L FileName} compiler directives. The assembly-language source file must be assembled into an object file using Apple's MDS assembler or an equivalent.

Multiple object files may be linked with a program or unit through multiple directives. Note that all $L directives must appear before the **begin** that heads the main statement-part of the program or unit. If a $L directive appears in a program, the object file is linked into the *blank segment*, that is, the segment in which the main program resides.

Procedures and Functions

Procedures and functions written in assembly language must be declared as **external** in the Pascal program or unit, for instance:

```
function UpCase(Ch: Char): Char; external;
```

In the corresponding assembly-language source file, externally defined procedures and functions must appear in XDEF directives, for instance:

```
XDEF    UpCase          ;Define UpCase

UpCase  MOVE.L  (SP)+,A0        ;Get return address
        MOVE.W  (SP)+,D0        ;Get Ch
        CMP.W   #'a',D0         ;Skip if not lower case
        BLT.S   @1
        CMP.W   #'z',D0
        BGT.S   @1
        SUB.W   #$20,D0         ;Convert to upper case
@1      MOVE.W  D0,(SP)         ;Store return value
        JMP     (A0)            ;Return
```

It is up to you to ensure that an assembly-language procedure or function matches its Pascal definition with respect to the number of parameters, the types of the parameters, and the result type.

An assembly-language source file may reference Pascal procedures and functions via XREF directives, for instance:

```
XREF    ReadFile        ;ReadFile is a Pascal routine
:
JSR     ReadFile        ;Call ReadFile
```

You *must not* specify the addressing base, such as (A5) or (PC), when calling an XDEFed or XREFed procedure or function. Turbo Pascal itself figures out which calling method to use when it links the object file with the program.

Variables

An assembly-language source file may declare variables via DS directives. Such variables are private to the assembly-language source file and cannot be referenced from the Pascal program or unit.

Global variables defined in the Pascal program or unit are referenced in assembly language via XREF directives, for instance:

```
XREF    ErrCnt          ;ErrCnt is a Pascal variable
:
ADDQ.W  #1,ErrCnt(A5)   ;Increment ErrCnt
```

The (A5) addressing base *must* be specified, just as it is for private variables defined via DS directives.

Operations on Relocatable Symbols

An assembly-language symbol is *relocatable* if it is created as a label or through an XREF directive. When a relocatable symbol appears in an assembly-language expression, the assembler cannot calculate the true value of the expression, since the value of the symbol is not determined until the object file is linked. For instance, if *Symbol* is defined via a DS directive, the address expression `Symbol(A5)` is not resolved until the object file is linked.

When an object file appears in a $L compiler directive, Turbo Pascal converts the object file from MDS Link format to its own internal link format. This conversion is possible only if the following rules are observed in the Assembly Language source file:

- XDEFed and XREFed procedure and function symbols may only be used as operands in JSR, JMP, PEA, and LEA instructions. For instance, if *Proc* is a procedure or function symbol, the instruction `BRA Proc` is not allowed. Furthermore, procedure and function symbols may not participate in assembly-language expressions. For instance, the instruction `JSR Proc+4` is not allowed.
- When a relocatable symbol is part of an assembly-language expression, the expression may only add an absolute value to the symbol or subtract an absolute value from the symbol. For instance, if *Sym1* and *Sym2* are relocatable symbols, the expressions `Sym1+4` and `Sym2-8` are allowed, whereas `Sym1*8`, `8-Sym2`, and `Sym2-Sym1` are not.
- Assembly-language expressions that involve relocatable symbols are evaluated as 16-bit integers and can only be coded as such. For instance, if *Symbol* is a relocatable symbol, the address expression `Symbol(A5,D0.W)` is not allowed, since it would produce an 8-bit displacement. Likewise, the directive `DC.L Symbol-*` is not allowed, since it would produce a 32-bit value.

Register Saving Conventions

An assembly-language procedure or function may modify registers D0–D2, A0, and A1 (the scratch registers). All other registers must be saved and restored. In particular, you should avoid modifying A5 and A7, and only modify A6 in connection with LINK and UNLK instructions.

Defining Your Own Devices

In addition to the standard Console and Printer devices, Turbo Pascal allows you to define your own devices.

The Device Procedure

To define a device, you call the *Device* standard procedure, which supplies a *device name* and a pointer to a *device I/O function*. The device name identifies the device, and the device I/O function handles all input and output requests for the device. The *Device* procedure resides in the *PasInOut* unit, so if you are compiling in the {$U-} state, your program must name the *PasInOut* in its uses-clause.

The syntax of a call to *Device* is

```
Device ( name , inoutptr )
```

where *name* is a string-type expression that names the device, and *inoutptr* is an expression of any pointer-type that points to the device I/O function. An example is

```
Device('MyDevice:', @MyInOut);
```

When a file is opened with *Reset* or *Rewrite*, Turbo Pascal scans the device name list, and associates the file with a device if a matching device name is found. Otherwise the file variable is associated with a disk file.

The function header of a device I/O function is

```
function DevInOut(var F: FileRec): Integer;
```

where *FileRec* is the file record type defined in the previous section, "File-Types."

NOTE: *FileRec* is not a predefined type; you must define it yourself. The names of the device I/O function and the parameter are unimportant, but the parameter must be a **var** parameter of a *FileRec* like type, and the function result must be *Integer*.

The device I/O function is called by the *Read*, *ReadLn*, *Write*, *WriteLn*, *Close*, *Eof*, *Eoln*, *SeekEof*, and *SeekEoln* standard procedures and functions whenever input from the device or output to the device is required.

The device I/O function determines whether it should read or write by looking at the *fInpFlag* and *fOutFlag* fields of the file record (both flags are never set at the same time).

If *fInpFlag* is set, the device I/O function should read *fBufSize* or less characters into the buffer pointed to by *fBuffer*, and it should return the number of characters actually read in *fBufEnd* and zero if *fBufPos*. If the device I/O function returns zero in *fBufEnd* as a result of an input request, *Eof(f)* becomes *True* for the file.

If *fOutFlag* is set, the device I/O function should write *fBufPos* characters from the buffer pointed to by *fBuffer*, and return zero in *fBufPos*. The device I/O function is called after each write-parameter in a *Write* or *WriteLn* statement. This ensures that text written to the device appears on the device immediately. If this is not required, the device I/O function may choose to ignore write requests if *fBufPos* does not equal *fBufEnd*. In that case, the buffer is not emptied until it is completely full.

The return value of the device I/O function becomes the value returned by *IOResult*. Zero indicates a successful operation.

This section presents two simple device I/O functions that illustrate different ways of implementing device I/O in Turbo Pascal.

The examples assume that three low-level procedures and functions exist: An *InputChar(Ch)* procedure that inputs a character from the device and stores it in *Ch*, an *OutputChar(Ch)* procedure that outputs the character in *Ch* to the device, and a *CharReady* function that returns true as long as there are still characters to be read from the device.

The device I/O functions implement two devices called 'BlockDev:' and 'LineDev:'. They are defined by executing

```
Device('BlockDev:', @BlockInOut);
Device('LineDev:', @LineInOut);
```

The *BlockDev* function shown below uses the *block-oriented* method for doing I/O. When requested for input, it fills the entire buffer; when requested for output, it only starts sending data out when the buffer is completely full. This method resembles the one used to input from and output to disk files.

```
function BlockInOut(var F: FileRec): Integer;
var
  Ch: Char;
  P: Integer;
begin
  MyInOut := 0;
  with F do
  if fInpFlag then
  begin
    fBufEnd := 0;
    while CharReady and (fBufEnd < fBufSize) do
    begin
      InputChar(Ch);
      fBuffer^[fBufEnd] := Ch;
      fBufEnd := fBufEnd + 1;
    end;
    fBufPos := 0;
  end else
  if (fBufPos = fBufEnd) then
  begin
    for P := 0 to fBufPos - 1 do
    OutputChar(fBuffer^[P]);
    fBufPos := 0;
  end;
end;
```

The *LineInOut* function shown below uses the *line-oriented* method for doing I/O. When requested for input, it inputs one line (which is ended by a CR character), and when requested for output, it outputs the contents of the buffer immediately. This method resembles the one used by the Console device.

```
function TextInOut(var F: FileRec): Integer;
var
  Ch: Char;
  P: Integer;
begin
  MyInOut := 0;
  with F do
  if fInpFlag then
  begin
    fBufEnd := 0;
    if CharReady then
    repeat
      InputChar(Ch);
      fBuffer^[fBufEnd] := Ch;
      fBufEnd := fBufEnd + 1;
    until not CharReady or (fBufEnd = fBufSize) or (Ch = #13);
    fBufPos := 0;
  end else
  begin
    for P := 0 to fBufPos - 1 do
    OutputChar(fBuffer^[P]);
    fBufPos := 0;
  end;
end;
```

PART III

Appendices

APPENDIX A

Comparing Turbo Pascal with Other Pascals

This appendix compares Turbo Pascal with the American National Standard (ANS) Pascal and Lisa Pascal (Apple Computer's Pascal compiler for the Lisa computer).

Turbo Pascal Compared to ANS Pascal

This section compares Turbo Pascal to ANS Pascal as defined by ANSI/IEEE770X3.97-1983 in the book *American National Standard Pascal Computer Programming Language* (ISBN 0-471-88944-X, published by The Institute of Electrical and Electronics Engineers in New York).

Exceptions to ANS Pascal Requirements

Turbo Pascal complies with the requirements of ANSI/IEEE770X3.97-1983 with the following exceptions:

- In ANS Pascal, an identifier may be of any length and all characters are significant. In Turbo Pascal, an identifier may be of any length, but only the first 63 characters are significant.

- In ANS Pascal, the @ symbol is an alternative for the ^ symbol. In Turbo Pascal, the @ symbol is an operator, which is never treated identically with the ^ symbol.
- In ANS Pascal, a comment may begin with { and end with *), or begin with (* and end with }. In Turbo Pascal, comments must begin and end with the same set of symbols.
- In ANS Pascal, each possible value of the tag-type in a variant-part must appear once. In Turbo Pascal, this requirement is not enforced.
- In ANS Pascal, the component-type of a file-type may not be a structured-type having a component of a file-type. In Turbo Pascal, this requirement is not enforced.
- In ANS Pascal, a file-variable has an associated buffer-variable, which is referenced by writing the ^ symbol after the file-variable. In Turbo Pascal, a file-variable does not have an associated buffer-variable, and writing the ^ symbol after a file-variable is an error.
- In ANS Pascal, the statement-part of a function must contain at least one assignment to the function identifier. In Turbo Pascal, this requirement is not enforced.
- In ANS Pascal, a field that is the selector of a variant-part may not be an actual variable parameter. In Turbo Pascal, this requirement is not enforced.
- In ANS Pascal, procedures and functions allow procedural and functional parameters; these parameters are not allowed in Turbo Pascal.
- In ANS Pascal, the standard procedures *Reset* and *Rewrite* take only one parameter, a file variable. In Turbo Pascal, *Reset* and *Rewrite* require a second parameter, a string-type expression, which names an external file.
- ANS Pascal defines the standard procedures *Get* and *Put*, which are used to read from and write to files. These procedures are not defined in Turbo Pascal.
- In ANS Pascal, the standard procedures *Read* and *Write* are defined in terms of *Get* and *Put* and references to buffer-variables. In Turbo Pascal, *Read* and *Write* function as in ANS Pascal, but they are automatic operations.
- In ANS Pascal, the syntax *New(p,c1,...,cn)* creates a dynamic variable with a specific active variant. In Turbo Pascal, this syntax is not allowed.
- In ANS Pascal, the syntax *Dispose(q,k1,...,km)* removes a dynamic variable with a specific active variant. In Turbo Pascal, this syntax is not allowed.
- ANS Pascal defines the standard procedures *Pack* and *Unpack*, which are used to "pack" and "unpack" packed variables. These procedures are not defined in Turbo Pascal.
- In ANS Pascal, the term *i* **mod** *j* always computes a positive value, and it is an error if *j* is zero or negative. In Turbo Pascal, *i* **mod** *j* is computed as *i* - (*i* **div** *j*) * *j*, and it is not an error if *j* is negative.

- In ANS Pascal, a **goto** statement within a block may refer to a label in an enclosing block. In Turbo Pascal, this is an error.
- In ANS Pascal, it is an error if the value of the selector in a **case** statement is not equal to any of the case-constants. In Turbo Pascal, this is not an error; instead, the **case** statement is ignored unless it contains an **otherwise** clause.
- In ANS Pascal, statements that *threaten* the control-variable of a **for** statement are not allowed. In Turbo Pascal, this requirement is not enforced.
- In ANS Pascal, a *Read* from a text file with a char-type variable assigns a blank to the variable if *Eoln* was *True* before the *Read*. In Turbo Pascal, a carriage-return character (ASCII 13) is assigned to the variable in this situation.
- In ANS Pascal, a *Read* from a text file with an integer-type or real-type variable ceases as soon as the next character in the file is not part of a signed-integer or a signed-number. In Turbo Pascal, reading ceases when the next character in the file is a blank or a control character (including the end-of-line character).
- In ANS Pascal, a *Write* to a text file with a packed-string-type value causes the string to be truncated if the specified field width is less than the length of the string. In Turbo Pascal, the string is always written in full, even if it is longer than the specified field width.

Note: Turbo Pascal is unable to detect whether or not a program violates any of the exceptions listed here.

Extensions to ANS Pascal

The following Turbo Pascal features are extensions to Pascal as specified by ANSI/IEEE770X3.97-1983.

- The following are reserved words in Turbo Pascal:

```
implementation    otherwise    shr       unit     xor
interface         shl          string    uses
```

- An identifier may contain underscore characters after the first character.
- Integer constants may be written in hexadecimal notation. Such constants are prefixed by a $.
- Identifiers may serve as labels.
- String constants are compatible with the Turbo Pascal string-types, and may contain control characters and other non-printable characters.
- Label, constant, type, variable, procedure, and function declarations may occur any number of times in any order in a block.

- A signed constant identifier may denote a value of type *Integer*, *LongInt*, or *Extended*.
- Turbo Pascal implements the additional integer-type *LongInt*, and the additional real-types *Double*, *Comp*, and *Extended*.
- Arithmetic operations on *Integer* operands produce *Integer* results. Arithmetic on *LongInt* operands or mixed *Integer* and *LongInt* operands produce *LongInt* results. *LongInt* values are compatible with the *Integer* type provided they are in the *Integer* range.
- Arithmetic operations on real-type operands or mixed integer-type and real-type operands produce *Extended* results. *Extended* values are compatible with the *Real*, *Double*, and *Comp* types, provided they are in the range of those types.
- Turbo Pascal implements string-types, which differ from the packed-string-types defined by ANS Pascal in that they include a dynamic-length attribute that may vary during execution.
- The type compatibility rules are extended to make char-types and packed-string-types compatible with string-types.
- String-type variables can be indexed as arrays to access individual characters in a string.
- The type of a variable-reference can be changed to another type through a variable-type-cast.
- Turbo Pascal implements three new logical operators: **xor**, **shl**, and **shr**.
- The **not**, **and**, **or**, and **xor** operators may be used with integer-type operands to perform bitwise logical operations.
- The + operator can be used to concatenate strings.
- The relational operators can be used to compare strings.
- Turbo Pascal implements the @ operator, which is used for obtaining the address of a variable or a procedure or function.
- The type of an expression can be changed to another type through a value-type-cast.
- The **case** statement allows constant ranges in case label lists, and provides an optional **otherwise** part.
- Procedures and functions can be declared as **external** (assembly-language subroutines) and **inline** (inline machine code).
- A variable parameter can be untyped (typeless), in which case any variable-reference may serve as the actual parameter.
- Turbo Pascal implements *units* to facilitate modular programming and separate compilation.

- Turbo Pascal implements the following file-handling procedures and functions, which are not available in ANS Pascal:

```
Close     Rename     Seek       FileSize     SeekEoln
Erase     IOResult   FilePos    SeekEof
```

- String-type values may be input and output with the *Read*, *ReadLn*, *Write*, and *WriteLn* standard procedures.
- Turbo Pascal implements two standard devices, *Console:* and *Printer:*, and furthermore supports user-defined devices.
- Turbo Pascal implements the following standard procedures and functions, which are not found in ANS Pascal:

```
Exit        Int        ClearScreen    SizeOf       Lo
Halt        Length     ClearEOL       MoveLeft     Swap
MemAvail    Pos        DeleteLine     MoveRight    HiWord
MaxAvail    Concat     InsertLine     FillChar     LoWord
Ord4        Copy       GotoXY         ScanEQ       SwapWord
Pointer     Delete     KeyPressed     ScanNE
Float       Insert     ReadChar       Hi
```

Note: Turbo Pascal is unable to detect whether or not a program uses any of the extensions listed here.

Implementation-Dependent Features

The effect of using an implementation-dependent feature of Pascal, as defined by ANSI/IEEE770X3.97-1983, is unspecified. Programs should not depend on any specific path being taken in cases where an implementation-dependent feature is being used. Implementation-dependent features include:

- The order of evaluation of index-expressions in a variable-reference.
- The order of evaluation of expressions in a set-constructor.
- The order of evaluation of operands of a binary operator.
- The order of evaluation of actual parameters in a function call.
- The order of evaluation of the left and right sides of an assignment.
- The order of evaluation of actual parameters in a procedure statement.
- The effect of reading a text file to which the procedure *Page* was applied during its creation.
- The binding of variables denoted by the program parameters to entities external to the program.

This section lists those errors from Appendix D of the ANS Pascal Standard that are not automatically detected by Turbo Pascal. The numbers referred to here are the numbers used in the ANS Pascal Standard. Errors 6, 19-22, and 25–31 are not detected, because they are not applicable to Turbo Pascal.

2. If *t* is a tag-field in a variant-part and *f* is a field within the active variant of that variant-part, it is an error to alter the value of *t* while a reference to *f* exists. This error is not detected.

3. If *p* is a pointer variable, it is an error to reference *p*^ if *p* is **nil**. This error is not detected.

4. If *p* is a pointer variable, it is an error to reference *p*^ if *p* is undefined. This error is not detected.

5. If *p* is a pointer variable, it is an error to alter the value of *p* while a reference to *p*^ exists. This error is not detected.

24. If *p* is a pointer variable, the procedure call *Dispose(p)* is an error if *p* is undefined. This error is not detected.

42. The function call *Eoln(f)* is an error if *Eof(f)* is *True*. In Turbo Pascal this is not an error, and *Eoln(f)* is *True* when *Eof(f)* is *True*.

43. It is an error to reference a variable in an expression if the value of that variable is undefined. This error is not detected.

46. A term of the form *i* **mod** *j* is an error if *j* is zero or negative. In Turbo Pascal, it is not an error if *j* is negative.

48. It is an error if a function does not assign a result value to the function identifier. This error is not detected.

51. It is an error if the value of the selector in a **case** statement is not equal to any of the case-constants. In Turbo Pascal, this is not an error; instead, the **case** statement is ignored unless it contains an **otherwise** clause.

Turbo Pascal Compared to Lisa Pascal

This section compares Turbo Pascal to Lisa Pascal. Lisa Pascal was the first Pascal compiler made available by Apple Computer for its line of 68000-based computers.

- **Identifier length.** In Lisa Pascal, only the first 8 characters of an identifier are significant. In Turbo Pascal, the first 63 characters are significant. Lisa Pascal does not detect abbreviations and spelling errors after the 8th character; Turbo Pascal does.
- **Constant expressions.** Lisa Pascal supports constant expressions; that is, expressions are allowed where constants are expected, as long as they evaluate to a constant value. Turbo Pascal does not support this.
- **Bit packing.** Lisa Pascal performs data packing to the bit level. For example, in Lisa Pascal, an array variable of the type

```
packed array[0..127] of Boolean
```

  occupies only 8 bytes, and each of the 8 bits in a byte represent a single *Boolean* component. In Turbo Pascal, data packing is performed only to the byte level, so an array variable of the preceding type would occupy 128 bytes.
- **Set limits.** In Lisa Pascal, the base-type of a set must not have more than 4088 possible values, and the ordinal values of the lower and upper bounds must be within the range 0..4087. In Turbo Pascal, the base-type of a set must not have more than 256 possible values, and the ordinal values of the lower and upper bounds must be within the range 0..255.
- **Type casting.** Type casting in Lisa Pascal is more permissive than in Turbo Pascal. Specifically, Lisa Pascal allows types other than ordinal-types and pointer-types in value-type-casts. For example, assuming the declarations

```
type
  Point = record
            x,y: Integer;
          end;
var
  P: Point;
```

  then the following statement is allowed in Lisa Pascal, but not in Turbo Pascal:

```
P := Point(180 * 65536 + 32)
```

  However, the following statement is allowed in both Turbo Pascal and Lisa Pascal, since a variable-type-cast may involve any two types as long as they are of the same size:

```
Longint(P) := 180 * 65536 + 32;
```

- **Exponentation operator.** Lisa Pascal implements an exponentation operator. This operator is not supported by Turbo Pascal. The construct $x^{**}y$ in Lisa Pascal is equivalent to $Exp(Ln(x) * y)$ in Turbo Pascal.
- **Short circuit *Boolean* expressions.** Lisa Pascal implements "short circuit" *Boolean* expression evaluation through the operators & (ampersand) and | (vertical bar). In Lisa Pascal, the following construct is allowed:

```
if (p <> nil) & (p^.count = 0) then ...
```

In Turbo Pascal, this must be written as

```
if p <> nil then if p^.count = 0 then ...
```

- **goto statements.** In Lisa Pascal, a **goto** statement may leave the current block, that is, a **goto** statement may jump out of a procedure or a function. This is not allowed in Turbo Pascal.
- **cycle and leave statements.** These Lisa Pascal statements are not found in Turbo Pascal, but they are easily programmed with **goto** statements.
- **Arbitrary typed functions.** Lisa Pascal allows arbitrary typed functions, that is, functions that return values of types other than simple-types, string-types, and pointer-types. Such functions must be changed to procedures in Turbo Pascal. Furthermore, Lisa Pascal allows function results to be treated as variables, such as

```
FileFlags := GetMenu(fileID)^^.enableFlags;
```

In Turbo Pascal, this would require a temporary variable:

```
FileMenu := GetMenu(fileID);
FileFlags := FileMenu^^.enableFlags;
```

- **Procedural and functional parameters.** In Lisa Pascal, procedures and functions allow procedural and functional parameters; these parameters are not allowed in Turbo Pascal.
- **univ parameters.** In Lisa Pascal, when a formal parameter is declared with the word **univ**, any actual parameter type is acceptable as long as it has the same size as the formal parameter type. Turbo Pascal does not support **univ** parameters, but they are easily circumvented through type casting. Alternatively, routines that use **univ** parameters can be reprogrammed to use Turbo Pascal's untyped variable parameters, which offer greater flexibility.
- **Unit numbers.** A Turbo Pascal unit must be given a unit number, which is not required in Lisa Pascal. Turbo Pascal stores the interface part of a compiled unit using the compiler's internal binary format. This provides for very fast processing when the unit is used, but requires a unique unit number for identification purposes. Lisa Pascal does not require unit numbers, since the interface part of a unit is recompiled every time the unit is used (like an include file).
- **external routines in units.** In Lisa Pascal, when external procedures and functions are declared in a unit, the procedure and function headers must appear in the **interface** part, and be repeated in the **implementation** part, followed by **external** directives. In Turbo Pascal, **external** and **inline** directives are specified along with the procedure and function headers in the **interface** part, and the headers must not be repeated in the **implementation** part.
- **Segmentation.** In Lisa Pascal, segmentation is always enabled. In Turbo Pascal, a {$S+} directive must be placed in the beginning of a program to enable segmentation.

- **Units and segmentation.** In Lisa Pascal, the segment in which a unit is to reside is determined by the unit itself, that is, {$S segname} directives in the unit determine its segment(s) in the final program. In Turbo Pascal, segment directives are ignored in units, and the segment in which a unit ultimately resides is determined by {$S segname} directives in the uses-clause of the final program.
- **File I/O.** Turbo Pascal does not implement the standard procedures *Get* and *Put*, nor does Turbo Pascal implement file buffer variables; that is, the syntax *f*^, where *f* is a file-type variable, is not allowed. Lisa Pascal programs that use these features must be changed to use the standard procedures *Read* and *Write*.
- **The Close procedure.** In Turbo Pascal, the *Close* procedure does not allow an *option* parameter (such as *lock*, *purge*, or *crunch*).
- **The *Exit* procedure.** In Turbo Pascal, the *Exit* procedure takes no parameter, and can be used only to exit the current block.
- **Standard procedures and functions.** The following Lisa Pascal standard procedures and functions are not implemented in Turbo Pascal:

```
Get Release     BitNOT       ClearBit
Put             HeapResult    BitSL
SetBit          BlockRead     BitAND
BitSR           BlockWrite    BitOR
BitRotL         Mark          BitXOR
BitTest
```

  The bit-manipulation routines are easily coded with Turbo Pascal's **not**, **and**, **or**, **xor**, **shl**, and **shr** operators.
- **Compiler directives.** The following Lisa Pascal compiler directives are directly supported by Turbo Pascal: $D, $I, $R, and $S. The $U directive is also supported in Turbo Pascal, but it works differently. The remaining Lisa Pascal compiler directives are not supported by Turbo Pascal. In general, it is recommended that you carefully examine all compiler directives when porting a Lisa Pascal program to Turbo Pascal.

APPENDIX **B**

Error Messages and Codes

This appendix lists all the compiler and system error messages, and the *IOResult* and NaN codes. Explanatory notes follow some messages and codes. In some cases, solutions are suggested.

Compiler Error Messages

```
01  ';' expected.
02  ':' expected.
03  ',' expected.
04  '(' expected.
05  ')' expected.
06  '=' expected.
07  ':=' expected.
08  '[' expected.
09  ']' expected.
10  '.' expected.
11  '..' expected.
12  begin expected.
13  do expected.
14  end expected.
15  of expected.
16  interface expected.
17  then expected.
18  to or downto expected.
19  implementation expected.
20  Boolean expression expected.
21  File variable expected.
22  Integer constant expected.
23  Integer expression expected.
```

```
24  Integer variable expected.
25  Integer or real constant expected.
26  Integer or real expression expected.
27  Integer or real variable expected.
28  Pointer variable expected.
29  Record variable expected.
30  Ordinal type expected.
```

All simple types except real types are ordinal types.

```
31  Ordinal expression expected.
32  String constant expected.
33  String expression expected.
34  String variable expected.
35  Identifier expected.
36  Type identifier expected.
37  Field identifier expected.
```

The identifier does not denote a field in a record structure.

```
38  Constant expected.
39  Variable expected.
40  Undefined label.
41  Unknown identifier.
42  Undefined type in pointer definition.
```

A preceding pointer type definition refers to an unknown type identifier.

```
43  Duplicate identifier.
```

The identifier has already been used within the current block.

```
44  Type mismatch.
```

This message means one of the following conditions exists:

1. incompatible types of the variable and the expression in an assignment statement;
2. incompatible types of the actual and formal parameter in a call to a subprogram;
3. expression type incompatible with index type in array indexing;
4. incompatible types of operands in an expression.

```
45  Constant out of range.
46  Constant and case types do not match.
```

The type of the case constant is incompatible with the **case** statement's selector expression.

```
47  Operand types do not match operator.
```

The operator cannot be applied to operands of this type; for example, 'A' div '2'.

```
48  Invalid result type.
```

Valid types are all simple types, string types, and pointer types.

```
49  Invalid string length.
```

The length of a string must be in the range 1..255.

```
51  Invalid subrange base type.
```

All ordinal types are valid base types.

```
52  Lower bound greater than upper bound.
53  Invalid for control variable.
```

A **for** statement control variable must be a single variable defined in the declaration part of the program or in the declaration part of the current subprogram.

```
54  Illegal assignment.
```

1. Files may not be assigned values.
2. A function identifier can only be assigned values within the statement part of the function.

```
55  String constant exceeds line.
56  Error in integer constant.
```

The syntax of *Integer* constants is defined in Chapter 16. Note that whole real numbers should be followed by a decimal point and a zero; for example, 123456789.0.

```
57  Error in real constant.
```

The syntax of *Real* constants is defined in Chapter 16.

```
58  Division by zero.
59  Structure too large.
```

The size of a structure may not exceed 32K bytes.

```
60  Constants are not allowed here.
62  Invalid type cast argument.
```

1. If a type cast is used in a position where a variable is expected, the argument can't be an expression, and the sizes of the argument and the result must be identical.
2. If the type cast argument is an expression, the argument and result types must be ordinal types or pointer types.
3. If the sizes of the argument and the result are not identical, the argument and result types must be scalar or pointer types.

```
63  Invalid '@' argument.
```

Valid arguments are variables and procedure or function identifiers.

```
64  Label already defined.
65  Invalid file type.
```

The file type is not supported by the file-handling procedure; for example, *ReadLn* with a typed file or *Seek* with a text file.

```
66  Cannot read or write variables of this type.
67  Files must be var parameters.
68  File components may not be files.
```

file of file constructs are not allowed.

```
70  Set base type out of range.
```

The base type of a set must be an enumerated type with no more than 256 possible values or a subrange with bounds in the range 0..255.

```
71  Invalid goto.
```

A **goto** statement in the preceding statement part references a label within a **for** statement from outside that **for** statement.

```
72  Label not within current block.
```

A **goto** statement cannot reference a label outside the current block.

```
73  Undefined forward procedure(s).
```

A subprogram has been **forward** declared in the preceding declaration part, but the body never occurred.

```
74  program or unit expected.
75  Error in type.
```

This symbol is not a type identifier, nor is it one of the reserved words that start a type definition.

```
76  Error in statement.
```

This symbol is not a variable, procedure, or label identifier, nor is it one of the reserved words that start a statement.

```
77  Error in expression.
```

This symbol cannot be used in an expression in the way specified.

```
78  Invalid external definition.
```

An object file defines a symbol that is not the identifier of an **external** declared procedure or function.

```
79  Invalid external reference.
```

An object file references a symbol that is not the identifier of a variable, procedure, or function.

```
80  Too many symbols.
```

Try increasing the symbol table size in the **Compile Options Dialog**. If the symbol table size is already set at the 32K byte maximum, divide your program or unit into two or more units.

```
81  Too many nested scopes.
```

The total sum of "used" units, nested subprograms, and active **with** statements can't exceed 64 at any time.

```
82 Driver header not found.
```

The driver header resource named in the $D compiler directive does not exist.

```
83 Too many variables.
```

The total size of variables declared within the program or within a subprogram may not exceed 32K bytes.

```
84 Expression too complicated.
```

The code generator ran out of registers. Simplify the expression by breaking it into two or more expressions.

```
85 Segment too large.
```

The size of a segment may not exceed 32K bytes. Introduce a new segment or move some units or subprograms to another segment. Also, make sure that segmentation is enabled with a {$S+} directive in the beginning of the program.

```
86 Unit not found.
```

This unit is not a resident unit, it is not contained in any of the unit library files specified by $U compiler directives, and it has not been compiled to memory in a window.

```
87 Duplicate or invalid unit number.
```

1. This unit has the same reference number as one of the units named before it in the uses-clause.
2. The unit number is not an integer within the range 0 to 32767.

```
88 Unit missing.
```

One or more of the units used by this unit have not been named in the **uses** clause.

```
89 Incompatible unit versions.
```

One or more of the units used by this unit have been changed since the unit was compiled.

```
90 Syntax error.
91 Unexpected end of text.
```

Your program cannot end in its current set-up. It probably has more **begin**s than **end**s.

```
92 Line too long.
```

The maximum line length is 128 characters.

```
93 Invalid compiler directive.
```

1. The compiler directive letter is unknown.
2. The compiler directive parameter is invalid.
3. You are using a global compiler directive when compilation of the body of the program has begun.

```
94  Target address found in unit.
```

This error is reported by the Compile menu's Find Error command when the address at which the execution error occurred is within one of the units used by the program. When reporting the error, the compiler also adjusts the target address so that the correct statement will be located if you load the source text of the unit and issue a new Find Error command.

```
95  Undefined external procedure(s).
```

A subprogram has been **external** declared, but it was not defined by any of the object file(s) linked with $L compiler directive(s).

```
96  Object file format error.
```

An object file uses an MDS Linker feature which is not supported by Turbo Pascal. For further details, please refer to Section 26.4.3.

```
97  Runtime support unit missing.
```

The unit that defines your predefined procedure or function was not named in your **uses** clause; for example, *WriteLn* without the *PasInOut* unit.

```
98  Target address not found.
```

This error is reported by the Compile menu's Find Error command when the address at which the execution error occurred is not within the program itself. A probable cause is passing invalid parameters to ROM-based system routines.

```
99  Not enough memory.
```

This error occurs when the program being compiled is too large to fit in memory. Close some windows to free some space, or compile the program to disk.

System Error Messages

This section lists all system errors that may be reported by the Macintosh System Error handler. The Compile menu's Find Error command only shows messages for system errors 02, 04, 05, 16, 25, 28, and 99. Other errors are not likely to occur in a Turbo Pascal program, but they are included here for reference.

```
01  Bus error.
02  Address error.
```

Word or long-word reference was made to an odd address. This typically indicates use of an uninitialized pointer variable.

```
03  Illegal instruction.
04  Division by zero.
```

A division (**div** operator) or modulo (**mod** operator) with a divisor of 0 was attempted.

```
05  Range check failed.
```

An assignment or array indexing operation compiled in the {*$R+*} state involved a value that was not within the allowed range.

```
06  TrapV exception.
07  Privilege violation.
08  Trace exception.
09  Line 1010 exception.
10  Line 1111 exception.
11  Miscellaneous exception.
12  Unimplemented core routine.
13  Spurious interrupt.
14  I/O system error.
15  Segment loader error.
16  Floating point error.
```

The halt bit in the floating-point environment word was set. By default, Turbo Pascal enables halts for invalid operation, division by zero, and overflow.

```
17  Can't load package 0.
24  Can't load package 7.
25  Memory allocation error.
26  Segment loader error.
27  File map trashed.
28  Stack overflow error.
```

The stack has expanded into the heap.

```
32  Memory manager error.
53  Memory manager error.
99  Input/Output check failed.
```

A standard I/O procedure compiled in the {$I+} state returned a nonzero *IOResult*.

IOResult codes

This section lists all result codes that may be returned by the *IOResult* function. The codes correspond to those returned by the routines in the Macintosh Operating System, except for the codes −128, −129, and −130, which are generated by Turbo Pascal itself. Although many of the codes are not likely to be returned by the *IOResult* function, they are included here for reference.

```
-33  File directory full.
-34  All allocation blocks on the volume are full.
-35  Specified volume doesn't exist.
-36  Disk I/O error.
-37  Bad file name or volume name (perhaps zero-length).
-38  File not open.
-39  Logical end-of-file reached during read operation.
-40  Attempt to position before start of file.
-41  System heap is full.
-42  Too many files open.
-43  File not found.
-44  Volume is locked by a hardware setting.
-45  File is locked.
-46  Volume is locked by a software flag.
-47  One or more files are open.
-48  A file with the specified name already exists.
-49  Only one access path to a file can allow writing.
-50  No default volume.
-51  Bad file reference number.
-53  Volume not on-line.
-54  Read/write permission doesn't allow writing.
-55  Specified volume is already mounted and on-line.
-56  No such drive number.
-57  Volume lacks Macintosh-format directory.
-58  External file system error.
-59  Problem during Rename.
-60  Master directory block is bad; must re-initialize volume.
-61  Read/write permission doesn't allow writing.
-108  Not enough room in heap zone.
-120  Directory not found.
-121  Too many working directories open.
-122  Attempted to move into offspring.
-123  Attempt to do HFS operation on non-HFS volume.
-127  Internal file system error.
-128  Textfile not open for input.
-129  Textfile not open for output.
-130  Error in numeric value during read from textfile.
```

NaN codes

When a floating-point operation cannot produce a meaningful result, the operation delivers a special bit pattern called a *NaN* (Not a Number). For example, 0 divided by 0 yields *NaN(004)*. The following *NaN* codes may be reported.

```
001  Invalid square root, such as Sqrt(-1).
002  Invalid addition or subtraction, such as INF - INF.
004  Invalid division, such as 0 / 0.
008  Invalid multiplication, such as 0 * INF.
009  Invalid remainder or mod, such as x rem 0.
017  Attempt to convert invalid ASCII string to binary.
020  Result of converting comp NaN to floating.
021  Attempt to create NaN with a zero code.
033  Invalid argument to trig routine.
034  Invalid argument to inverse trig routine.
036  Invalid argument to log routine.
037  Invalid argument to x^i or x^y routine.
038  Invalid argument to financial function.
255  Uninitialized storage.
```

APPENDIX C

Compiler Directives

Some of the Turbo Pascal compiler's features are controlled through *compiler directives*. A compiler directive is introduced as a comment with a special syntax; Turbo Pascal allows compiler directives wherever comments are allowed.

A compiler directive starts with a $ character as the first character after the opening comment delimiter. The $ is immediately followed by a letter that designates the particular directive.

There are two types of directives: *switch* directives and *parameter* directives. A switch directive turns a particular compiler feature on or off by specifying + or − immediately after the directive letter. A parameter directive is followed by a string, such as a file name or a segment name. The string argument is terminated by the closing comment delimiter.

Compiler directives are either *global* or *local*. Global directives affect the entire compilation, whereas local directives affect only the part of the compilation that extends from the directive until the next occurrence of the same directive. Global directives must appear before the declaration part of the program or the unit being compiled; that is, before the first **uses**, **label**, **const**, **type**, **procedure**, **function**, or **begin** keyword of a program or before the **interface** keyword of a unit. Local directives, on the other hand, may appear anywhere in the program or unit.

Examples of compiler directives follow:

```
{$B+}
{$R- Turn off range checking}
{$I TypeDefs.Pas}
{$U Turbo:Units:MacLibrary}
{$T APPLMYED}
```

Set Bundle Bit

Syntax: {$B+} or {$B−}
Default: {$B−}
Type: Global

This switch controls the bundle-bit setting in the compiled application file. You should only set the bundle bit if your application's resource file contains a BNDL resource (see the *R* parameter directive). The *B* switch only takes effect if the compiler was invoked with the Compile To Disk command.

Generate Debug Symbols

Syntax: {$D+} or {$D−}
Default: {$D−}
Type: Local

This switch turns the generation of procedure names in the object code on or off. Debug symbols allow you to see the names of your procedures when you debug your program, using MACSBUG, for instance.

Compile Desk Accessory

Syntax: {$D PasDeskAcc}
Type: Global

The appearance of this parameter directive tells the compiler that you are compiling a desk-accessory program. Instead of generating CODE resources in the output file, the compiler generates a DRVR resource with a resource ID of 12. The *D* directive also has the effect of turning off segmentation (corresponding to a {$S-} directive) and changing the output file type to DFIL and the output file creator to DMOV (corresponding to a {$T DFILDMOV} directive). The latter causes the FINDER to display the desk-accessory icon for the generated file and to launch the FONT/DA MOVER when the file is double-clicked. For further details on compiling desk accessories, refer to Chapter 10.

Check I/O Results

Syntax: {$I+} or {$I−}
Default: {$I+}
Type: Local

This switch turns on or off the automatic generation of code that checks the I/O result of a call to an I/O procedure from the *PasInOut* unit. If an I/O procedure returns a non-zero I/O result when this switch is on, the program terminates by displaying the system error bomb box with an ID of 99. When this switch is off, it is up to you to check for I/O errors through the *IOResult* function.

Include File

Syntax: {$I FileName}
Type: Local

This parameter directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text after the line containing the Include directive. If *FileName* does not specify a directory, the program searches for the file in the directory specified by the $I entry of the Compile Options Dialog, or in the current directory if the $I entry is empty.

Link Object File

Syntax: {$L FileName}
Type: Local

This parameter directive instructs the compiler to link the named file with the program or unit being compiled. The *L* directive is typically used to link code written in another language (for example, the MDS assembler) for subprograms declared to be **external**. The named file must be an MDS object-format .REL file. Files named in *L* directives are always linked into the *blank segment*, that is, the segment that also contains the main statement part. A program or unit may contain multiple *L* directives, but all of them must appear before the **begin** keyword that heads the main statement part. If *FileName* does not specify a directory, the program searches for a file in the directory specified by the $L entry of the Compile Options Dialog, or in the current directory if the $L entry is empty.

Define Output File

Syntax: {$O FileName}
Default: {$O ProgName}
Type: Global

This parameter directive defines the name of the output file generated by the Compile to Disk command. The default output file name PROGNAME is the name of the program or unit; that is, the name specified after the **program** or **unit** keyword. If the output file of a unit you are compiling already exists, and if it contains a unit of the same name, the new unit replaces the older version stored in the file. If *FileName* specifies a directory, the file is created in that directory. Otherwise, the output file is created in the directory specified by the $O entry in the Compile Options Dialog, or in the current directory if the $O entry is empty.

Generate Range Checks

Syntax: {$R+} or {$R−}
Default: {$R−}
Type: Local

This switch instructs the compiler to turn the generation of range checking code on or off. When range checking is on, all array and string indexing expressions are checked to be within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, the program terminates by displaying the system error bomb box with an ID of 5.

Define Resource File

Syntax: {$R FileName}
Type: Global

The appearance of this parameter directive indicates to the compiler that your program uses the resources stored in the named file. The file is typically a .RSRC file generated from a .R file by the resource compiler (RMAKER). If the compiler was invoked using the Compile to Disk command, all resources stored in the named file are copied to the output file. If the compiler was invoked using the Compile to Memory command, the named file will automatically be opened when the program is executed by a Compile Run command. If *FileName* does not specify a directory, the file is searched for in the directory specified by the $R entry of the Compile Options Dialog, or in the current directory if the $R entry is empty.

Generate Segmented Code

Syntax: {$S+} or {$S−}
Default: {$S−}
Type: Global

This switch instructs the compiler to turn the generation of segmented code on or off. When segmentation is off (the default state), the size of the generated code cannot exceed 32K, which is the maximum size of a single segment. Segmentation is always off when compiling a unit or a desk accessory, so the compiled code of a unit or a desk accessory cannot be larger than 32K. When segmentation is turned on with a {$S+} directive, there is no limit to the size of a program, as long as its code is divided into segments of less than 32K using the $S parameter directive. When segmentation is off, all subprogram calls and subprogram address references are coded using PC-relative instructions. When segmentation is on, all calls and address references are routed through the segment loader jump table.

Define Segment Name

Syntax: {$S SegName}
Default: {$S }
Type: Local

This parameter directive defines the name of the segment in which the code of the following units or subprograms is to be stored. The $S parameter directive is ignored when compiling units and desk accessories; for programs, it only takes effect when segmentation has been enabled with a {$S+} directive. *SegName* is a string of up to eight case-sensitive characters. If less than eight characters are specified, the remaining characters are assumed to be blanks. If the specified segment does not exist, a new segment is created. The default segment, also called the *blank segment*, is a segment whose name consists of eight blanks.

Note: Segment names only exist within the compiler; in the finished code they are turned into code resource numbers starting from 1.

Define Type and Creator

Syntax: {$T ttttcccc}
Default: {$T APPL????}
Type: Global

This parameter directive is used to define the type and creator of an application file generated by the Compile to Disk command. The first four characters of the string argument define the type; the next four characters define the creator (note that no separators are placed between the two definitions). The *T* directive is typically used in connection with the *B* switch and a resource file that contains a bundle (BNDL) resource.

Use Standard Units

Syntax: {$U+} or {$U−}
Default: {$U+}
Type: Global

This switch determines whether or not the standard units *PasInOut* and *PasConsole* should be included in the compilation. In effect, the default {$U+} directive corresponds to inserting `uses PasInOut,PasConsole` before the declaration part of the program or unit being compiled.

Note: The *PasSystem* unit is always included in a compilation regardless of the *U* switch. The *PasPrinter* unit is never included automatically.

Search Unit Library

Syntax: {$U FileName}
Type: Global

This parameter directive is used to specify the name of a unit library file to search for, in addition to the resident units installed in Turbo itself. If *FileName* specifies a directory, the unit library file is opened in that directory. Otherwise, the unit library file is opened in the directory specified by the $U entry in the Compile Options Dialog, or in the current directory if the $U entry is empty.

When searching for a unit named in a uses-clause, Turbo Pascal first inspects all open windows that contain a unit that has been compiled to memory. Next, it searches all unit library files named in *U* directives in order of appearance. Finally, it looks to itself to see if the unit is installed as a resident unit.

If a unit library file is named in both the *O* and *U* directives (which may be the case when compiling a unit), the *U* directive naming the file must be the first *U* directive.

APPENDIX D

Macintosh Interface Units

The following pages contain the 15 Macintosh interface units used by Turbo Pascal. Below each head is a brief explanation of what the unit is and what it can do, followed by the interface listing. The units are discussed in this order:

- *PasInOut*
- *PasConsole*
- *PasPrinter*
- *SANE*
- *MemTypes*
- *QuickDraw*
- *OSIntf*
- *ToolIntf*
- *PackIntf*
- *MacPrint*
- *FixMath*
- *Graf3D*
- *AppleTalk*
- *SpeechIntf*
- *SCSIIntf*

PasInOut

PasInOut implements the standard Pascal input/output (I/O) routines (*Read*, *ReadLn*, *Write*, *WriteLn*, *Reset*, *Rewrite*, and so on), as well as the Turbo Pascal-specific ones (*Close*, *Seek*, *Rename*, *Erase*, and so forth). It also does all I/O and range-error checking. If you look at the interface listing below, you'll find that there is very little you can use directly; instead, the compiler makes calls to specific hidden routines in the implementation.

```
unit PasInOut(-2);   { PasInOut - standard I/O unit }

interface

var
  TextType,TextCreator,
  FileType,FileCreator: packed array[1..4] of Char;
```

PasConsole

PasConsole is the unit that makes it easy to write textbook Pascal programs. It creates a window that emulates a terminal screen 80 characters wide by 25 lines deep. When this unit is used by a program or unit, any calls to *Read* or *ReadLn* are made from the keyboard and automatically echoed to this window; likewise, any calls to *Write* or *WriteLn* write to this window. A number of cursor- and screen-control routines are available: *ClearScreen*, *ClearEOL*, *InsertLine*, *DeleteLine*, and *GoToXY*. The functions *KeyPressed* and *ReadChar* are included, as are the file variables *Input* and *Output*. This unit also creates a new device (Console:) that can be assigned to any file of type text. The user can then send output to the screen (instead of to a disk file).

```
unit PasConsole(-3);      { PasConsole - console unit }

interface

uses PasInOut;

var
  Input,Output: Text;

function KeyPressed: Boolean; external;
function ReadChar: Char; external;

procedure ClearScreen; external;
procedure ClearEOL; external;
procedure InsertLine; external;
procedure DeleteLine; external;
procedure GotoXY(X,Y: Integer); external;
```

PasPrinter

PasPrinter declares the text-file variable *Printer* and connects it to a device driver that allows you to send standard Pascal output to the printer using *Write* and *WriteLn*.

Like *PasConsole*, this unit creates a new device (Printer:) that can be assigned to any file of type text; it will then send output to the printer (instead of to a disk file).

```
unit PasPrinter(-4);      { PasPrinter - printer unit }

interface

uses PasInOut;

var
  Printer: Text;
```

SANE

The *SANE* unit implements the Standard Apple Numeric Environment. *SANE* is the basis for all floating-point mathematical calculations performed by Turbo Pascal. Programmers who are interested in using *SANE* features not directly supported by Turbo Pascal can access these features through the *SANE* unit. For detailed instructions about SANE, see Chapter 26 and the *Apple Numerics Manual.*

```
unit SANE(-5);

{ SANE - Standard Apple Numeric Environment interface unit }

{ Implements floating-point operations not directly supported }
{ by Turbo Pascal }

interface

const

  DecStrLen = 255;
  SigDigLen =  20;

  Invalid     =  1;
  Underflow   =  2;
  Overflow    =  4;
  DivByZero   =  8;
  Inexact     = 16;

type

  DecStr        = string[DecStrLen];

  CStrPtr       = ^Char;

  Decimal       = record
                    sgn : 0..1;
                    exp : Integer;
                    sig : string[SigDigLen]
                  end;

  DecForm       = record
                    Style        : ( FloatDecimal, FixedDecimal );
                    Digits : Integer;
                  end;

  RelOp         = ( GreaterThan, LessThan, EqualTo, Unordered );

  NumClass      = ( SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum );

  Exception     = Integer;

  RoundDir      = ( ToNearest, Upward, Downward, TowardZero );

  RoundPre      = ( ExtPrecision, DblPrecision, RealPrecision );

  Environment   = Integer;

function  Num2Integer    ( x : Extended ) : Integer; external;
function  Num2Longint    ( x : Extended ) : LongInt; external;
function  Num2Real       ( x : Extended ) : Real; external;
```

```
function  Num2Double     ( x : Extended ) : Double; external;
function  Num2Extended   ( x : Extended ) : Extended; external;
function  Num2Comp       ( x : Extended ) : Comp; external;

procedure Num2Dec ( var f : DecForm; x : Extended; var d : Decimal ); external;
function  Dec2Num ( d : Decimal ) : Extended; external;
procedure Num2Str ( f : DecForm; x : Extended; var s : DecStr ); external;
function  Str2Num ( s : DecStr ) : Extended; external;

function  Remainder ( x, y : Extended; var quo : Integer) : Extended; external;

function  Rint ( x : Extended ) : Extended; external;

function  Scalb ( n : Integer; x : Extended ) : Extended; external;

function  Logb ( x : Extended ) : Extended; external;

function  CopySign ( x, y : Extended ) : Extended; external;

function  NextReal     ( x, y : Real )     : Real; external;
function  NextDouble   ( x, y : Double )   : Double; external;
function  NextExtended ( x, y : Extended ) : Extended; external;

function  Log2 ( x : Extended )              : Extended; external;
function  Ln1 ( x : Extended )               : Extended; external;
function  Exp2 ( x : Extended )              : Extended; external;
function  Exp1 ( x : Extended )              : Extended; external;
function  XpwrI ( x : Extended; i : Integer ) : Extended; external;
function  XpwrY ( x, y : Extended )          : Extended; external;
function  Compound ( r, n : Extended )       : Extended; external;
function  Annuity ( r, n : Extended )        : Extended; external;
function  Tan ( x : Extended )               : Extended; external;
function  RandomX ( var x : Extended )      : Extended; external;

function  ClassReal      ( x : Real )     : NumClass; external;
function  ClassDouble    ( x : Double )   : NumClass; external;
function  ClassComp      ( x : Comp )     : NumClass; external;
function  ClassExtended ( x : Extended ) : NumClass; external;

function  SignNum ( x : Extended ) : Integer; external;
function  NAN ( i : Integer )      : Extended; external;

procedure SetException ( e : Exception; b : Boolean ); external;
function  TestException ( e : Exception)  : Boolean; external;

procedure SetHalt ( e : Exception; b : Boolean ); external;
function  TestHalt ( e : Exception)  : Boolean; external;

procedure SetRound ( r : RoundDir ); external;
function  GetRound : RoundDir; external;

procedure SetPrecision ( p : RoundPre ); external;
function  GetPrecision : RoundPre; external;

procedure SetEnvironment ( e : Environment ); external;
procedure GetEnvironment ( var e : Environment ); external;

procedure ProcEntry ( var e : Environment ); external;
procedure ProcExit ( e : Environment ); external;

function  GetHaltVector : LongInt ; external;
procedure SetHaltVector ( v : LongInt ) ; external;

function  Relation ( x, y : Extended) : Relop; external;
```

MemTypes

MemTypes defines special Mac data types, such as *SignedByte*, *Ptr*, *Handle*, and *Str255*. That's all it does; it doesn't define any constants, variables, or routines. It is used by every unit in this list and must be in any Mac-style application.

```
unit MemTypes(-6);

interface

type

   SignedByte   = -128..127;     { any byte in memory }
   Byte         = 0..255;        { unsigned byte for fontmgr }
   Ptr          = ^SignedByte;   { blind pointer }
   Handle       = ^Ptr;          { pointer to a master pointer }
   ProcPtr      = Ptr;           { pointer to a procedure }
   Fixed        = LongInt;       { fixed point arithmetic type }

   Str255       =  String[255];  { maximum string size }
   StringPtr    = ^Str255;       { pointer to maximum string }
   StringHandle = ^StringPtr;    { handle to maximum string }
```

QuickDraw

QuickDraw is a Macintosh graphics package that lets you perform complex graphic operations quickly and easily. This unit defines all the constants, types, variables, procedures, and functions needed to use *QuickDraw*.

```
unit QuickDraw(-7);

interface

uses MemTypes;

const srcCopy    =  0;  { the 16 transfer modes }
      srcOr      =  1;
      srcXor     =  2;
      srcBic     =  3;
      notSrcCopy =  4;
      notSrcOr   =  5;
      notSrcXor  =  6;
      notSrcBic  =  7;
      patCopy    =  8;
      patOr      =  9;
      patXor     = 10;
      patBic     = 11;
      notPatCopy = 12;
      notPatOr   = 13;
      notPatXor  = 14;
      notPatBic  = 15;

{ QuickDraw color separation constants }

      normalBit    =  0;       { normal screen mapping }
      inverseBit   =  1;       { inverse screen mapping }
      redBit       =  4;       { RGB additive mapping }
      greenBit     =  3;
      blueBit      =  2;
      cyanBit      =  8;       { CMYBk subtractive mapping }
      magentaBit   =  7;
      yellowBit    =  6;
      blackBit     =  5;

      blackColor   =  33;      { colors expressed in these mappings }
      whiteColor   =  30;
      redColor     = 205;
      greenColor   = 341;
      blueColor    = 409;
      cyanColor    = 273;
      magentaColor = 137;
      yellowColor  =  69;

      picLParen    = 0;       { standard picture comments }
      picRParen    = 1;


type QDByte    =  SignedByte;
     QDPtr     =  Ptr;              { blind pointer }
     QDHandle  =  Handle;           { blind handle  }
     Pattern   =  packed array[0..7] of 0..255;
     Bits16    =  array[0..15] of integer;
     VHSelect  =  (v,h);
     GrafVerb  =  (frame,paint,erase,invert,fill);
     StyleItem =  (bold,italic,underline,outline,shadow,condense,extend);
```

```
Style     =  set of StyleItem;

FontInfo  =  record
               ascent:  Integer;
               descent: Integer;
               widMax:  Integer;
               leading: Integer;
             end;

Point = record case Integer of
          0: (v: Integer;
              h: Integer);
          1: (vh: array[VHSelect] of Integer);
        end;

Rect = record case Integer of
         0: (top:      Integer;
             left:     Integer;
             bottom:   Integer;
             right:    Integer);
         1: (topLeft:  Point;
             botRight: Point);
       end;

BitMap = record
           baseAddr: Ptr;
           rowBytes: Integer;
           bounds:   Rect;
         end;

Cursor = record
           data:    Bits16;
           mask:    Bits16;
           hotSpot: Point;
         end;

PenState = record
             pnLoc:   Point;
             pnSize:  Point;
             pnMode:  Integer;
             pnPat:   Pattern;
           end;

PolyHandle = ^PolyPtr;
PolyPtr    = ^Polygon;
Polygon    = record
               polySize:   Integer;
               polyBBox:   Rect;
               polyPoints: array[0..0] of Point;
             end;


RgnHandle = ^RgnPtr;
RgnPtr    = ^Region;
Region    =  record
               rgnSize:  Integer;  { rgnSize = 10 for rectangular }
               rgnBBox:  Rect;
               { plus more data if not rectangular }
             end;


PicHandle = ^PicPtr;
PicPtr    = ^Picture;
Picture   =  record
               picSize:    Integer;
               picFrame:   Rect;
```

```
                { plus byte codes for picture content }
              end;

    QDProcsPtr = ^QDProcs;
    QDProcs = record
                textProc:    Ptr;
                lineProc:    Ptr;
                rectProc:    Ptr;
                rRectProc:   Ptr;
                ovalProc:    Ptr;
                arcProc:     Ptr;
                polyProc:    Ptr;
                rgnProc:     Ptr;
                bitsProc:    Ptr;
                commentProc: Ptr;
                txMeasProc:  Ptr;
                getPicProc:  Ptr;
                putPicProc:  Ptr;
              end;

    GrafPtr  = ^GrafPort;
    GrafPort = record
                 device:     Integer;
                 portBits:   BitMap;
                 portRect:   Rect;
                 visRgn:     RgnHandle;
                 clipRgn:    RgnHandle;
                 bkPat:      Pattern;
                 fillPat:    Pattern;
                 pnLoc:      Point;
                 pnSize:     Point;
                 pnMode:     Integer;
                 pnPat:      Pattern;
                 pnVis:      Integer;
                 txFont:     Integer;
                 txFace:     Style;
                 txMode:     Integer;
                 txSize:     Integer;
                 spExtra:    Fixed;
                 fgColor:    LongInt;
                 bkColor:    LongInt;
                 colrBit:    Integer;
                 patStretch: Integer;
                 picSave:    Handle;
                 rgnSave:    Handle;
                 polySave:   Handle;
                 grafProcs:  QDProcsPtr;
               end;

var thePort:    GrafPtr;
    white:      Pattern;
    black:      Pattern;
    gray:       Pattern;
    ltGray:     Pattern;
    dkGray:     Pattern;
    arrow:      Cursor;
    screenBits: BitMap;
    randSeed:   LongInt;

{ GrafPort routines }

procedure InitGraf   (globalPtr: Ptr);          inline $A86E;
procedure OpenPort   (port: GrafPtr);           inline $A86F;
procedure InitPort   (port: GrafPtr);           inline $A86D;
procedure ClosePort  (port: GrafPtr);           inline $A87D;
procedure SetPort    (port: GrafPtr);           inline $A873;
```

```
procedure ClosePort  (port: GrafPtr);                          inline $A87D;
procedure SetPort    (port: GrafPtr);                          inline $A873;
procedure GetPort    (var port: GrafPtr);                      inline $A874;
procedure GrafDevice (device: Integer);                        inline $A872;
procedure SetPortBits(bm: BitMap);                             inline $A875;
procedure PortSize   (width,height: Integer);                  inline $A876;
procedure MovePortTo (leftGlobal,topGlobal: Integer);          inline $A877;
procedure SetOrigin  (h,v: Integer);                           inline $A878;
procedure SetClip    (rgn: RgnHandle);                         inline $A879;
procedure GetClip    (rgn: RgnHandle);                         inline $A87A;
procedure ClipRect   (r: Rect);                                inline $A87B;
procedure BackPat    (pat: Pattern);                           inline $A87C;

{ Cursor routines }

procedure InitCursor;                                          inline $A850;
procedure SetCursor(crsr: Cursor);                             inline $A851;
procedure HideCursor;                                          inline $A852;
procedure ShowCursor;                                          inline $A853;
procedure ObscureCursor;                                       inline $A856;

{ Line routines }

procedure HidePen;                                             inline $A896;
procedure ShowPen;                                             inline $A897;
procedure GetPen     (var pt: Point);                          inline $A89A;
procedure GetPenState(var pnState: PenState);                   inline $A898;
procedure SetPenState(pnState: PenState);                       inline $A899;
procedure PenSize    (width,height: Integer);                  inline $A89B;
procedure PenMode    (mode: Integer);                          inline $A89C;
procedure PenPat     (pat: Pattern);                           inline $A89D;
procedure PenNormal;                                           inline $A89E;
procedure MoveTo     (h,v: Integer);                           inline $A893;
procedure Move       (dh,dv: Integer);                         inline $A894;
procedure LineTo     (h,v: Integer);                           inline $A891;
procedure Line       (dh,dv: Integer);                         inline $A892;

{ Text routines }

procedure TextFont     (font: Integer);                          inline $A887;
procedure TextFace     (face: Style);                          inline $205F,$1010,$3F00,$A888;
procedure TextMode     (mode: Integer);                          inline $A889;
procedure TextSize     (size: Integer);                          inline $A88A;
procedure SpaceExtra   (extra: Fixed);                           inline $A88E;
procedure DrawChar     (ch: Char);                               inline $A883;
procedure DrawString   (s: Str255);                              inline $A884;
procedure DrawText     (textBuf: Ptr;  firstByte,byteCount: Integer);  inline $A885;
function  CharWidth    (ch: Char): Integer;                      inline $A88D;
function  StringWidth  (s: Str255): Integer;                     inline $A88C;
function TextWidth (textBuf: Ptr; firstByte,byteCount: Integer): Integer; inline $A886;

procedure GetFontInfo  (var info: FontInfo);                     inline $A88B;

procedure MeasureText  (count: Integer; textAddr, charLocs: Ptr); inline $A837;

{ point calculations }

procedure AddPt        (src: Point; var dst: Point);   inline $A87E;
procedure SubPt        (src: Point; var dst: Point);   inline $A87F;
procedure SetPt        (var pt: Point; h,v: Integer);  inline $A880;
function  EqualPt      (pt1,pt2: Point): Boolean;      inline $A881;
procedure ScalePt (var pt: Point; fromRect,toRect: Rect); inline $A8F8;
procedure MapPt        (var pt: Point;fromRect,toRect: Rect); inline $A8F9;
```

```
procedure LocalToGlobal (var pt: Point);                    inline $A870;
procedure GlobalToLocal (var pt: Point);                    inline $A871;

{ rectangle calculations }

procedure SetRect    (var r: Rect;left,top,right,bottom: Integer);  inline $A8A7;
function  EqualRect  (rect1,rect2: rect): Boolean;          inline $A8A6;
function  EmptyRect  (r: rect): Boolean;                    inline $A8AE;
procedure OffsetRect (var r: Rect; dh,dv: Integer);         inline $A8A8;
procedure MapRect    (var r:  Rect; fromRect,toRect: Rect);       inline $A8FA;
procedure InsetRect  (var r: Rect; dh,dv: Integer);         inline $A8A9;
function  SectRect   (src1,src2: Rect;  var dstRect: Rect): Boolean;   inline $A8AA;
procedure UnionRect  (src1,src2: Rect; var dstRect: Rect);       inline $A8AB;
function  PtInRect   (pt: Point; r: Rect): Boolean;         inline $A8AD;
procedure Pt2Rect    (pt1,pt2: Point; var dstRect: Rect);        inline $A8AC;

{ graphical operations on rectangles }

procedure FrameRect  (r: Rect);                             inline $A8A1;
procedure PaintRect  (r: Rect);                             inline $A8A2;
procedure EraseRect  (r: Rect);                             inline $A8A3;
procedure InvertRect (r: Rect);                             inline $A8A4;
procedure FillRect   (r: Rect; pat: Pattern);               inline $A8A5;

{ RoundRect routines }

procedure FrameRoundRect  (r: Rect; ovWd,ovHt: Integer); inline $A8B0;
procedure PaintRoundRect  (r: Rect; ovWd,ovHt: Integer); inline $A8B1;
procedure EraseRoundRect  (r: Rect; ovWd,ovHt: Integer); inline $A8B2;
procedure InvertRoundRect (r: Rect; ovWd,ovHt: Integer); inline $A8B3;
procedure FillRoundRect (r: Rect; ovWd,ovHt: Integer; pat: Pattern); inline $A8B4;

{ oval routines }

procedure FrameOval  (r: Rect);               inline $A8B7;
procedure PaintOval  (r: Rect);               inline $A8B8;
procedure EraseOval  (r: Rect);               inline $A8B9;
procedure InvertOval (r: Rect);               inline $A8BA;
procedure FillOval   (r: Rect; pat: Pattern); inline $A8BB;

{ arc routines }

procedure FrameArc  (r: Rect; startAngle,arcAngle: Integer);    inline $A8BE;
procedure PaintArc  (r: Rect; startAngle,arcAngle: Integer);    inline $A8BF;
procedure EraseArc  (r: Rect; startAngle,arcAngle: Integer);    inline $A8C0;
procedure InvertArc (r: Rect; startAngle,arcAngle: Integer);    inline $A8C1;
procedure FillArc (r: Rect; startAngle,arcAngle: Integer; pat: Pattern); inline $A8C2;

procedure PtToAngle (r: Rect; pt: Point; var angle: Integer);   inline $A8C3;

{ polygon routines }

function  OpenPoly:   PolyHandle;                               inline $A8CB;
procedure ClosePoly;                                            inline $A8CC;
procedure KillPoly   (poly: PolyHandle);                        inline $A8CD;
procedure OffsetPoly (poly: PolyHandle; dh,dv: Integer);        inline $A8CE;
procedure MapPoly    (poly: PolyHandle; fromRect,toRect: Rect); inline $A8FC;
procedure FramePoly  (poly: PolyHandle);                        inline $A8C6;
procedure PaintPoly  (poly: PolyHandle);                        inline $A8C7;
```

```
procedure ErasePoly  (poly: PolyHandle);                          inline $A8C8;
procedure InvertPoly (poly: PolyHandle);                          inline $A8C9;
procedure FillPoly   (poly: PolyHandle; pat: Pattern);            inline $A8CA;

{ region calculations }

function  NewRgn:   RgnHandle;                                    inline $A8D8;
procedure DisposeRgn(rgn: RgnHandle);                             inline $A8D9;
procedure CopyRgn   (srcRgn,dstRgn: RgnHandle);                   inline $A8DC;
procedure SetEmptyRgn(rgn: RgnHandle);                            inline $A8DD;
procedure SetRectRgn(rgn: RgnHandle; left,top,right,bottom: Integer); inline $A8DE;
procedure RectRgn   (rgn: RgnHandle; r: Rect);                    inline $A8DF;
procedure OpenRgn;                                                inline $A8DA;
procedure CloseRgn  (dstRgn: RgnHandle);                          inline $A8DB;
procedure OffsetRgn (rgn: RgnHandle; dh,dv: Integer);             inline $A8E0;
procedure MapRgn    (rgn: RgnHandle;   fromRect,toRect: Rect);    inline $A8FB;
procedure InsetRgn  (rgn: RgnHandle; dh,dv: Integer);             inline $A8E1;
procedure SectRgn   (srcRgnA,srcRgnB,dstRgn: RgnHandle);          inline $A8E4;
procedure UnionRgn  (srcRgnA,srcRgnB,dstRgn: RgnHandle);          inline $A8E5;
procedure DiffRgn   (srcRgnA,srcRgnB,dstRgn: RgnHandle);          inline $A8E6;
procedure XorRgn    (srcRgnA,srcRgnB,dstRgn: RgnHandle);          inline $A8E7;
function  EqualRgn  (rgnA,rgnB: RgnHandle): Boolean;              inline $A8E3;
function  EmptyRgn  (rgn: RgnHandle): Boolean;                    inline $A8E2;
function  PtInRgn   (pt: Point; rgn: RgnHandle): Boolean;         inline $A8E8;
function  RectInRgn (r: Rect; rgn: RgnHandle): Boolean;           inline $A8E9;

{ graphical operations on regions }

procedure FrameRgn  (rgn: RgnHandle);               inline $A8D2;
procedure PaintRgn  (rgn: RgnHandle);               inline $A8D3;
procedure EraseRgn  (rgn: RgnHandle);               inline $A8D4;
procedure InvertRgn (rgn: RgnHandle);               inline $A8D5;
procedure FillRgn   (rgn: RgnHandle; pat: Pattern); inline $A8D6;

{ graphical operations on BitMaps }

procedure ScrollRect(dstRect: Rect; dh,dv: Integer; updateRgn: rgnHandle); inline
                                                                    $A8EF;
procedure CopyBits  (srcBits,dstBits: BitMap;
                     srcRect,dstRect: Rect;
                     mode:           Integer;
                     maskRgn:        RgnHandle);        inline $A8EC;
procedure SeedFill(srcPtr,dstPtr:Ptr;
                   srcRow,dstRow,height,words: Integer;
                   seedH,seedV: Integer);               inline $A839;
procedure CalcMask(srcPtr,dstPtr:Ptr;
                   srcRow,dstRow,height,words: Integer); inline $A838;
procedure CopyMask(srcBits,maskBits, dstBits: BitMap; srcRect,maskRect,dstRect: rect);
                                                        inline $A817;
function  GetMaskTable: Ptr;                            inline $A836,$2E88;

{ picture routines }

function  OpenPicture(picFrame: Rect): PicHandle;          inline $A8F3;
procedure ClosePicture;                                    inline $A8F4;
procedure DrawPicture(myPicture: PicHandle; dstRect: Rect); inline $A8F6;
procedure PicComment(kind,dataSize: Integer; dataHandle: Handle); inline $A8F2;
procedure KillPicture(myPicture: PicHandle);               inline $A8F5;

{ the bottleneck interface }

procedure SetStdProcs(var procs: QDProcs);              inline $A8EA;
procedure StdText (count: Integer; textAddr: Ptr; numer,denom: Point); inline $A882;
```

```
procedure StdLine    (newPt: Point);                        inline $A89D;
procedure StdRect    (verb: GrafVerb; r: Rect);             inline $A8A0;
procedure StdRRect   (verb: GrafVerb; r: Rect; ovWd,ovHt: Integer); inline $A8AF;
procedure StdOval    (verb: GrafVerb; r: Rect);             inline $A8B6;
procedure StdArc (verb: GrafVerb; r: Rect; startAngle,arcAngle: Integer); inline $A8BD;

procedure StdPoly    (verb: GrafVerb; poly: PolyHandle);    inline $A8C5;
procedure StdRgn     (verb: GrafVerb; rgn: RgnHandle);      inline $A8D1;
procedure StdBits (var srcBits: BitMap; var srcRect,dstRect: Rect; mode:
                                        Integer; maskRgn: RgnHandle); inline $A8EB;
procedure StdComment (kind,dataSize: Integer;
                      dataHandle: Handle);                    inline $A8F1;
function  StdTxMeas  (count: Integer; textAddr: Ptr;
                      var numer,denom: Point;
                      var info: FontInfo): Integer;           inline $A8ED;
procedure StdGetPic  (dataPtr: Ptr; byteCount: Integer);     inline $A8EE;
procedure StdPutPic  (dataPtr: Ptr; byteCount: Integer);     inline $A8F0;

{ miscellaneous utility routines }

function GetPixel  (h,v: Integer): Boolean;      inline $A865;
function  Random:   Integer;                     inline $A861;
procedure StuffHex  (thingptr: Ptr; s:Str255);   inline $A866;
procedure ForeColor (color: LongInt);            inline $A862;
procedure BackColor (color: LongInt);            inline $A863;
procedure ColorBit  (whichBit: Integer);         inline $A864;
```

OSIntf

The Macintosh operating system (Mac OS) is at the lowest level of Macintosh operations. It performs basic tasks such as input/output memory management and interrupt handling. Many of the Toolbox procedures and functions call Mac OS routines to support their operations. The *OSIntf* unit declares the Pascal interface to the Mac OS, naming the many constants, data types, variables, and routines.

```
unit OSIntf(-8);

interface

uses MemTypes,QuickDraw;

const       { for EventManager }

  everyEvent  = -1;
  NullEvent   =  0;
  mouseDown   =  1;
  mouseUp     =  2;
  keyDown     =  3;
  keyUp       =  4;
  autoKey     =  5;
  updateEvt   =  6;
  diskEvt     =  7;
  activateEvt =  8;
  networkEvt  = 10;
  driverEvt   = 11;
  app1Evt     = 12;
  app2Evt     = 13;
  app3Evt     = 14;
  app4Evt     = 15;

  { event mask equates }
  mDownMask     =      2;
  mUpMask       =      4;
  keyDownMask   =      8;
  keyUpMask     =     16;
  autoKeyMask   =     32;
  updateMask    =     64;
  diskMask      =    128;
  activMask     =    256;
  networkMask   =   1024;
  driverMask    =   2048;
  app1Mask      =   4096;
  app2Mask      =   8192;
  app3Mask      =  16384;
  app4Mask      = -32768;

  { to decipher event message for keyDown events }
  charCodeMask = $000000FF;
  keyCodeMask  = $0000FF00;

  { modifiers }
  optionKey= 2048;   { Bit 3 of high byte }
  alphaLock= 1024;   { Bit 2 }
  ShiftKey=   512;   { Bit 1 }
  CmdKey=     256;   { Bit 0 }
  BtnState=   128;   { Bit 7 of low byte is mouse button state }
```

```
activeFlag  = 1;   { bit 0 of modifiers for activate event }

EvtNotEnb = 1;            { error for PostEvent }

{ for Memory Manager }
MemFullErr  =   -108;     { not enough room in heap zone }
NilHandleErr =  -109;     { master pointer was NIL in HandleZone or other }
MemWZErr    =   -111;     { WhichZone failed (applied to free block) }
MemPurErr   =   -112;     { trying to purge a locked or non-purgeable block }
MemLockedErr =  -117;     { block is locked }
NoErr       =     0;      { all is well }

{ file system error codes }
DirFulErr   =    -33;     { directory full }
DskFulErr   =    -34;     { disk full }
NSVErr      =    -35;     { no such volume }
IOErr       =    -36;     { I/O error }
BdNamErr    =    -37;     { bad name }
FNOpnErr    =    -38;     { file not open }
EOFErr      =    -39;     { end of file }
PosErr      =    -40;     { tried to position to before start of file (r/w) }
MFulErr     =    -41;     { memory full(open) or file won't fit (load) }
TMFOErr     =    -42;     { too many files open }
FNFErr      =    -43;     { file not found }

WPrErr      =    -44;     { diskette is write protected }
FLckdErr    =    -45;     { file is locked }
VLckdErr    =    -46;     { volume is locked }
FBsyErr     =    -47;     { File is busy (delete) }
DupFNErr    =    -48;     { duplicate filename (rename) }
OpWrErr     =    -49;     { file already open with with write permission }
ParamErr    =    -50;     { error in user parameter list }
RFNumErr    =    -51;     { refnum error }
GFPErr      =    -52;     { get file position error }
VolOffLinErr =   -53;     { volume not on line error (was Ejected) }
PermErr     =    -54;     { permissions error (on file open) }
VolOnLinErr =    -55;     { drive volume already on-line at MountVol }
NSDrvErr    =    -56;     { no such drive (tried to mount a bad drive num) }
NoMacDskErr =    -57;     { not a Mac diskette (sig bytes are wrong) }
ExtFSErr    =    -58;     { volume in question belongs to an external fs }
FSRnErr     =    -59;     { file system rename error }
BadMDBErr   =    -60;     { bad master directory block }
WrPermErr   =    -61;     { write permissions error }

lastDskErr  =    -64;     { last in a range of disk errors }
noDriveErr  =    -64;     { drive not installed }
offLinErr   =    -65;     { r/w requested for an off-line drive }
noNybErr    =    -66;     { couldn't find 5 nybbles in 200 tries }
noAdrMkErr  =    -67;     { couldn't find valid addr mark }
dataVerErr  =    -68;     { read verify compare failed }
badCkSmErr  =    -69;     { addr mark checksum didn't check }
badBtSlpErr =    -70;     { bad addr mark bit slip nybbles }
noDtaMkErr  =    -71;     { couldn't find a data mark header }
badDCkSum   =    -72;     { bad data mark checksum }
badDBtSlp   =    -73;     { bad data mark bit slip nybbles }
wrUnderRun  =    -74;     { write underrun occurred }
cantStepErr =    -75;     { step handshake failed }
tk0BadErr   =    -76;     { track 0 detect doesn't change }
initIWMErr  =    -77;     { unable to initialize IWM }
twoSideErr  =    -78;     { tried to read 2nd side on a 1-sided drive }
spdAdjErr   =    -79;     { unable to correctly adjust disk speed }
seekErr     =    -80;     { track number wrong on address mark }
sectNFErr   =    -81;     { sector number never found on a track }
firstDskErr =    -84;     { first in a range of disk errors }

DirNFErr    =   -120;     { directory not found }
```

```
TMWDOErr     =   -121;    { no free WDCB available }
BadMovErr    =   -122;    { move into offspring error }
WrgVolTypErr =   -123;    { wrong volume type error - operation not supported for MFS}
FSDSIntErr   =   -127;    { internal file system error }


MaxSize    = $800000;     { max data block size is 8 megabytes }

fHasBundle =  8192;       { finder constants }
fInvisible = 16384;
fTrash     =    -3;
fDesktop   =    -2;
fDisk      =     0;

{ I/O constants }
fsAtMark    =  0;         { ioPosMode values }
fsFromStart =  1;
fsFromLEOF  =  2;
fsFromMark  =  3;
rdVerify    = 64;

fsCurPerm    = 0;         { ioPermission values }
fsRdPerm     = 1;
fsWrPerm     = 2;
fsRdWrPerm   = 3;
fsRdWrShPerm = 4;

{ refNums from serial ports }
AinRefNum  = -6;          { serial port A input }
AoutRefNum = -7;          { serial port A output }
BinRefNum  = -8;          { serial port B input }
BoutRefNum = -9;          { serial port B output }

{ baud rate constants }
baud300   = 380;
baud600   = 189;
baud1200  =  94;
baud1800  =  62;
baud2400  =  46;
baud3600  =  30;
baud4800  =  22;
baud7200  =  14;
baud9600  =  10;
baud19200 =   4;
baud57600 =   0;

{ SCC channel configuration word }
{ driver reset information masks }
stop10    =  16384;
stop15    = -32768;
stop20    = -16384;

noParity   =     0;
oddParity  =  4096;
evenParity = 12288;

data5 =    0;
data6 = 2048;
data7 = 1024;
data8 = 3072;

{ serial driver error masks }
swOverrunErr =  1;
parityErr    = 16;
hwOverrunErr = 32;
framingErr   = 64;
```

```
{ serial port configuration usage constants for Config field of SysParmType }
 useFree  = 0;
 useATalk = 1;
 useAsync = 2;

 xOffWasSent = $80;        { serial driver message constant }

 { for application parameter }
 { constants for message returned by the finder on launch }
 appOpen  = 0;
 appPrint = 1;

 SWmode = -1;              { for sound driver }
 FTmode =  1;
 FFmode =  0;

 { Desk Accessories - message definitions (in CSCode of Control Call) }
 accEvent        =    64;  { event message from SystemEvent }
 accRun          =    65;  { run message from SystemTask }
 accCursor       =    66;  { cursor message from SystemTask }
 accMenu         =    67;  { menu message from SystemMenu }
 accUndo         =    68;  { undo message from SystemEdit }
 accCut          =    70;  { cut message from SystemEdit }
 accCopy         =    71;  { copy message from SystemEdit }
 accPaste        =    72;  { paste message from SystemEdit }
 accClear        =    73;  { clear message from SystemEdit }

 goodbye         =    -1;  { goodbye message }

 macXLMachine = 0;         { for "machine" parameter of Environs }
 macMachine   = 1;
 { if BitTst(ioDirFlg, myParamBlk^.ioFlAttrib) then }
 ioDirFlg  =  3;
 { if BitAnd(ioDirMask, myParamBlk^.ioFlAttrib) = ioDirMask then }
 ioDirMask = $10;
 FSRtParID =  1;           { DirID of parent's root }
 FSRtDirID =  2;           { Root DirID }

 { result codes for RelString }
 sortsBefore     =      -1;                  { str1 < str2 }
 sortsEqual      =       0;                  { str1 = str2 }
 sortsAfter      =       1;                  { str1 > str2 }

type
 EventRecord = record      { for Event Manager }
               what     : Integer;
               message  : LongInt;
               when     : LongInt;
               where    : Point;
               modifiers: Integer;
               end;

 Zone       = record
               BkLim     : Ptr;
               PurgePtr  : Ptr;
               HFstFree  : Ptr;
               ZCBFree   : LongInt;
               GZProc    : ProcPtr;
               MoreMast  : Integer;
               Flags     : Integer;
               CntRel    : Integer;
               MaxRel    : Integer;
               CntNRel   : Integer;
               MaxNRel   : Integer;
               CntEmpty  : Integer;
```

```
                CntHandles: Integer;
                MinCBFree : LongInt;
                PurgeProc : ProcPtr;
                SparePtr  : Ptr;          { reserved for future }
                AllocPtr  : Ptr;
                HeapData  : Integer;
              end;
THz         = ^Zone;        { pointer to the start of a heap zone }
Size        = LongInt;      { size of a block in bytes }
OSErr       = Integer;      { error code }

QElemPtr = ^QElem;          { ptr to generic queue element }

{ vertical blanking control block queue element }
VBLTask = record
    qLink: QElemPtr;        { link to next element }
    qType: Integer;         { unique ID for validity check }
    vblAddr: ProcPtr;       { address of service routine }
    vblCount: Integer;      { count field for timeout }
    vblPhase: Integer;      { phase to allow synchronization }
  end;                    { VBLCntrlBlk }
                          { VBLQElPtr = ^VBLTask; }

evQEl = record
    qLink:      QElemPtr;
    qType:      Integer;
    evtQwhat:   Integer;  { this part is identical to the EventRecord as... }
    evtQmessage:LongInt;  { defined in ToolIntf }
    evtQwhen:   LongInt;
    evtQwhere:  Point;
    evtQmodifiers: Integer;
  end;

DrvQEl = record             { drive queue elements }
    qLink:      QElemPtr;
    qType:      Integer;
    dQDrive:    Integer;
    dQRefNum:   Integer;    { ref num of the driver that handles this drive }
    dQFSID:     Integer;    { id of file system that handles this drive }
    dQDrvSize:  Integer;    { size of drive (512-byte blocks); }
                            { not for drvs 1&2 }
  end;
DrvQElPtr = ^DrvQEl;

TrapType = (OSTrap, ToolTrap);      { for NGet and NSet TrapAdress }

{ file system }
ParamBlkType = (IOParam,FileParam,VolumeParam,CntrlParam);

OSType = packed array[1..4] of Char; { same as rsrc mgr's Restype }

FInfo = record              { record of finder info }
          fdType: OSType;     { the type of the file }
          fdCreator: OSType;  { file's creator }
          fdFlags: Integer;   { flags, e.g., hasbundle,invisible,locked, etc. }
          fdLocation: Point;  { file's location in folder }
          fdFldr: Integer;    { folder containing file }
        end;                { FInfo }

{ new HFS }

FXInfo = record
           fdIconID: Integer;                   { Icon ID }
           fdUnused: array[1..4] of Integer; { unused but reserved }
           fdComment: Integer;                  { comment ID }
           fdPutAway: LongInt;                  { home dir ID }
```

```
        end;

DInfo = record
          frRect: Rect;          { folder rect }
          frFlags: Integer;      { flags }
          frLocation: Point;     { folder location }
          frView: Integer;       { folder view }
        end;

DXInfo = record
           frScroll: Point;      { scroll position }
           frOpenChain: LongInt; { dir ID chain of open folders }
           frUnused: Integer;    { unused but reserved }
           frComment: Integer;   { comment }
           frPutAway: LongInt;   { dir ID }
         end;

ParamBlockRec = record
      { 12 byte header used by file and IO system }
      qLink: QElemPtr;           { queue link in header }
      qType: Integer;            { type byte for safety check }
      ioTrap: Integer;           { FS: the Trap }
      ioCmdAddr: Ptr;            { FS: address to dispatch to }

      { common header to all variants }
      ioCompletion: ProcPtr;     { completion routine addr (0 for synch calls) }
      ioResult: OSErr;           { result code }
      ioNamePtr: StringPtr;      { ptr to Vol:FileName string }
      ioVRefNum: Integer;        { volume refnum (DrvNum for Eject and MountVol) }

      { different components for the different type of parameter blocks }
      case ParamBlkType of
      ioParam:
        (ioRefNum: Integer;      { refNum for I/O operation }
         ioVersNum: SignedByte;  { version number }
         ioPermssn: SignedByte;  { Open: permissions (byte) }

         ioMisc: Ptr;            { Rename: new name }
                                 { GetEOF,SetEOF: logical end of file }
                                 { Open: optional ptr to buffer }
                                 { SetFileType: new type }
         ioBuffer: Ptr;          { data buffer Ptr }
         ioReqCount: LongInt;    { requested byte count; also = ioNewDirID }
         ioActCount: LongInt;    { actual byte count completed }
         ioPosMode: Integer;     { initial file positioning }
         ioPosOffset: LongInt);  { file position offset }

      FileParam:
        (ioFRefNum: Integer;     { reference number }
         ioFVersNum: SignedByte; { version number }
         filler1: SignedByte;
         ioFDirIndex: Integer;   { GetFInfo directory index }
         ioFlAttrib: SignedByte; { GetFInfo: in-use bit=7, lock bit=0 }
         ioFlVersNum: SignedByte; { file version number }
         ioFlFndrInfo: FInfo;    { user info }
         ioFlNum: LongInt;       { GetFInfo: file number; TF- ioDirID }
         ioFlStBlk: Integer;     { start file block (0 if none) }
         ioFlLgLen: LongInt;     { logical length (EOF) }
         ioFlPyLen: LongInt;     { physical length }
         ioFlRStBlk: Integer;    { start block rsrc fork }
         ioFlRLgLen: LongInt;    { file logical length rsrc fork }
         ioFlRPyLen: LongInt;    { file physical length rsrc fork }
         ioFlCrDat: LongInt;     { file creation date & time (32 bits in secs) }
         ioFlMdDat: LongInt);    { last modified date and time }
```

```
        VolumeParam:
          (filler2: LongInt;
           ioVolIndex: Integer;     { volume index number }
           ioVCrDate: LongInt;      { creation date and time }
           ioVLsBkUp: LongInt;      { last backup date and time }
           ioVAtrb: Integer;        { volume attrib }
           ioVNmFls: Integer;       { number of files in directory }
           ioVDirSt: Integer;       { start block of file directory }
           ioVBlLn: Integer;        { GetVolInfo: length of dir in blocks }
           ioVNmAlBlks: Integer;    { GetVolInfo: num blks (of alloc size) }
           ioVAlBlkSiz: LongInt;    { GetVolInfo: alloc blk byte size }
           ioVClpSiz: LongInt;      { GetVolInfo: bytes to allocate at a time }
           ioAlBlSt: Integer;       { starting disk(512-byte) block in block map }
           ioVNxtFNum: LongInt;     { GetVolInfo: next free file number }
           ioVFrBlk: Integer);      { GetVolInfo: # free alloc blks for this vol }

         CntrlParam:
           (ioCRefNum: Integer;                    { refNum for I/O operation }
            CSCode: Integer;                       { word for control status code }
            CSParam: array[0..10] of Integer);  { operation-defined parameters }
         end;                                      { ParamBlockRec }

ParmBlkPtr = ^ParamBlockRec;

HParamBlockRec = record
      { 12 byte header used by the file system }
      qLink: QElemPtr;
      qType: Integer;
      ioTrap: Integer;
      ioCmdAddr: Ptr;

      { common header to all variants }
      ioCompletion: ProcPtr;      { completion routine, or NIL if none }
      ioResult: OSErr;            { result code }
      ioNamePtr: StringPtr;       { ptr to pathname }
      ioVRefNum: Integer;         { volume refnum }

      { different components for the different type of parameter blocks }
      case ParamBlkType of
      ioParam:
        (ioRefNum: Integer;       { refNum for I/O operation }
         ioVersNum: SignedByte;   { version number }
         ioPermssn: SignedByte;   { Open: permissions (byte) }

         ioMisc: Ptr;             { HRename: new name }
                                  { HOpen: optional ptr to buffer }
         ioBuffer: Ptr;           { data buffer Ptr }
         ioReqCount: LongInt;     { requested byte count }
         ioActCount: LongInt;     { actual byte count completed }
         ioPosMode: Integer;      { initial file positioning }
         ioPosOffset: LongInt);   { file position offset }


      fileParam:
        (ioFRefNum: Integer;      { reference number } (*choose either this or ioRefNum *)
         ioFVersNum: SignedByte;  { version number, normally 0 }
         filler1: SignedByte;
         ioFDirIndex: Integer;    { HGetFInfo directory index }
         ioFlAttrib: SignedByte;  { HGetFInfo: in-use bit=7, lock bit=0 }
         ioFlVersNum: SignedByte; { file version number returned by GetFInfoz }
         ioFlFndrInfo: FInfo;     { user info }
         ioDirID: LongInt;        { directory ID }
         ioFlStBlk: Integer;      { start file block (0 if none) }
         ioFlLgLen: LongInt;      { logical length (EOF) }
         ioFlPyLen: LongInt;      { physical length }
         ioFlRStBlk: Integer;     { start block rsrc fork }
```

```
        ioFlRLgLen: LongInt;     { file logical length rsrc fork }
        ioFlRPyLen: LongInt;     { file physical length rsrc fork }
        ioFlCrDat: LongInt;      { file creation date & time (32 bits in secs) }
        ioFlMdDat: LongInt);     { last modified date and time }

      volumeParam:
        (filler2: LongInt;
        ioVolIndex: Integer;     { volume index number }
        ioVCrDate: LongInt;      { creation date and time }
        ioVLsMod: LongInt;       { last date and time volume was flushed }
        ioVAtrb: Integer;        { volume attrib }
        ioVNmFls: Integer;       { number of files in directory }
        ioVBitMap: Integer;      { start block of volume bitmap }
        ioAllocPtr: Integer;     { HGetVInfo: length of dir in blocks }
        ioVNmAlBlks: Integer;    { HGetVInfo: num blks (of alloc size) }
        ioVAlBlkSiz: LongInt;    { HGetVInfo: alloc blk byte size }
        ioVClpSiz: LongInt;      { HGetVInfo: bytes to allocate at a time }
        ioAlBlSt: Integer;       { starting disk(512-byte) block in block map }
        ioVNxtCNID: LongInt;     { HGetVInfo: next free file number }
        ioVFrBlk: Integer;       { HGetVInfo: # free alloc blks for this vol }
        ioVSigWord: Integer;     { volume signature }
        ioVDrvInfo: Integer;     { drive number }
        ioVDRefNum: Integer;     { driver refNum }
        ioVFSID: Integer;        { ID of file system handling this volume }
        ioVBkUp: LongInt;        { last backup date (0 if never backed up) }
        ioVSeqNum: Integer;      { sequence number of this volume in volume set }
        ioVWrCnt: LongInt;       { volume write count }
        ioVFilCnt: LongInt;      { volume file count }
        ioVDirCnt: LongInt;      { volume directory count }
        ioVFndrInfo: array [1..8] of LongInt);  { finder info. for volume }
      end;                                      { HParamBlockRec }

HParmBlkPtr = ^HParamBlockRec;

FCBPBRec = record              { for PBGetFCBInfo }
   { 12 byte header used by the file and IO system }
   qLink: QElemPtr;            { queue link in header }
   qType: Integer;             { type byte for safety check }
   ioTrap: Integer;            { FS: the Trap }
   ioCmdAddr: Ptr;             { FS: address to dispatch to }
   ioCompletion: ProcPtr;      { completion routine addr (0 for synch calls) }
   ioResult: OSErr;            { result code }
   ioNamePtr: StringPtr;       { ptr to Vol:FileName string }
   ioVRefNum: Integer;         { volume refnum (DrvNum for Eject and MountVol) }
   ioRefNum: Integer;          { file to get the FCB about }
   filler: Integer;
   ioFCBIndx: LongInt;         { FCB index for _GetFCBInfo }
   ioFCBFlNm: LongInt;         { file number }
   ioFCBFlags: Integer;        { FCB flags }
   ioFCBStBlk: Integer;        { file start block }
   ioFCBEOF: LongInt;          { logical end-of-file }
   ioFCBPLen: LongInt;         { physical end-of-file }
   ioFCBCrPs: LongInt;         { current file position }
   ioFCBVRefNum: Integer;      { volume refNum }
   ioFCBClpSiz: LongInt;       { file clump size }
   ioFCBParID: LongInt;        { parent directory ID }
end;

FCBPBPtr = ^FCBPBRec;

CMovePBRec = record
   qLink: QElemPtr;            { queue link in header }
   qType: Integer;             { type byte for safety check }
   ioTrap: Integer;            { FS: the Trap }
   ioCmdAddr: Ptr;             { FS: address to dispatch to }
   ioCompletion: ProcPtr;      { completion routine addr (0 for synch calls) }
```

```
    ioResult: OSErr;            { result code }
    ioNamePtr: StringPtr;       { ptr to Vol:FileName string }
    ioVRefNum: Integer;         { volume refnum (DrvNum for Eject and MountVol) }
    filler1: LongInt;
    ioNewName: StringPtr;       { name of new directory }
    filler2: LongInt;
    ioNewDirID: LongInt;        { directory ID of new directory }
    filler3: array [1..2] of LongInt;
    ioDirID: LongInt;           { directory ID of current directory }
end;
CMovePBPtr = ^CMovePBRec;

WDPBRec = record              { for PBGetWDInfo }
    qLink: QElemPtr;            { queue link in header }
    qType: Integer;             { type byte for safety check }
    ioTrap: Integer;            { FS: the Trap }
    ioCmdAddr: Ptr;             { FS: address to dispatch to }
    ioCompletion: ProcPtr;      { completion routine addr (0 for synch calls) }
    ioResult: OSErr;            { result code }
    ioNamePtr: StringPtr;       { ptr to Vol:FileName string }
    ioVRefNum: Integer;         { volume refnum }
    filler1: Integer;           { not used }
    ioWDIndex: Integer;         { working directory index for _GetWDInfo }
    ioWDProcID: LongInt;        { WD's ProcID }
    ioWDVRefNum: Integer;       { WD's Volume RefNum }
    filler2: array[1..7] of Integer;
    ioWDDirID: LongInt;         { WD's DirID }
end;

WDPBPtr = ^WDPBRec;

CInfoType = (inputParam, hFileInfo, dirInfo);

CInfoPBRec = record { ioDirFlg clear; equates for catalog information return }
    qLink: QElemPtr;            { queue link in header }
    qType: Integer;             { type byte for safety check }
    ioTrap: Integer;            { FS: the Trap }
    ioCmdAddr: Ptr;             { FS: address to dispatch to }
    ioCompletion: ProcPtr;      { completion routine addr (0 for synch calls) }
    ioResult: OSErr;            { result code }
    ioNamePtr: StringPtr;       { ptr to Vol:FileName string }
    ioVRefNum: Integer;         { volume refnum (DrvNum for Eject and MountVol) }
    ioFRefNum: Integer;         { file reference number }
    filler1: Integer;
    ioFDirIndex: Integer;       { GetFInfo directory index }
    ioFlAttrib: SignedByte;     { GetFInfo: in-use bit=7, lock bit=0 }
    filler2: SignedByte;
    case CInfoType of
    inputParam:
      (filler3: array[1..8] of Integer;
       ioDirID: LongInt);
    hFileInfo:
      (ioFlFndrInfo: FInfo;     { user info }
       ioFlNum: LongInt;        { GetFInfo: file number }
       ioFlStBlk: Integer;      { start file block (0 if none) }
       ioFlLgLen: LongInt;      { logical length (EOF) }
       ioFlPyLen: LongInt;      { physical length }
       ioFlRStBlk: Integer;     { start block rsrc fork }
       ioFlRLgLen: LongInt;     { file logical length rsrc fork }
       ioFlRPyLen: LongInt;     { file physical length rsrc fork }
       ioFlCrDat: LongInt;      { file creation date & time (32 bits in secs) }
       ioFlMdDat: LongInt;      { last modified date and time }
       ioFlBkDat: LongInt;      { file last backup date }
       ioFlXFndrInfo: FXInfo;   { file additional finder info bytes }
       ioFlParID: LongInt;      { file parent directory ID (integer?) }
       ioFlClpSiz: LongInt);    { file clump size }
```

```
    dirInfo: { equates for directory information return }
      (ioDrUsrWds: DInfo;       { directory's user info bytes }
       ioDrDirID: LongInt;      { directory ID }
       ioDrNmFls: Integer;      { number of files in a directory }
       filler4: array [1..9] of Integer;
       ioDrCrDat: LongInt;      { directory creation date }
       ioDrMdDat: LongInt;      { directory modification date }
       ioDrBkDat: LongInt;      { directory backup date }
       ioDrFndrInfo: DXInfo;    { directory finder info bytes }
       ioDrParID: LongInt);     { directory's parent directory ID }
    end;

  CInfoPBPtr = ^CInfoPBRec;

  { 20 bytes of system parameter area }
  SysParmType = packed record
          Valid: Byte;          { validation field ($A7) }
          ATalkA: Byte;         { AppleTalk node number hint for port A }
          ATalkB: Byte;         { AppleTalk node number hint for port B }
          Config: Byte;         { ATalk port configuration A = bits 4-7, B = 0-3 }
          PortA: Integer;       { SCC port A configuration }
          PortB: Integer;       { SCC port B configuration }
          Alarm: LongInt;       { alarm time }
          Font: Integer;        { default font ID }
          KbdPrint: Integer;    { high byte = kbd repeat }
                                { high nibble = thresh in 4/60ths }
                                { low nibble = rates in 2/60ths }
                                { low byte = print stuff }
          VolClik: Integer;     { low 3 bits of high byte = volume control }
                              { high nibble of low byte = double time in 4/60ths }
                              { low nibble of low byte = caret blink time in 4/60ths }
          Misc: Integer;        { EEEC EEEE PSKB FFHH }
                                { E = extra }
                                { P = paranoia level }
                                { S = mouse scaling }
                                { K = key click }
                                { B = boot disk }
                                { F = menu flash }
                                { H = help level }
    end;                      { SysParmType }
  SysPPtr = ^SysParmType;

  { volume control block data structure }
  VCB = record
      qLink:        QElemPtr;   { link to next element }
      qType:        Integer;    { not used }
      vcbFlags:     Integer;
      vcbSigWord:   Integer;
      vcbCrDate:    LongInt;
      vcbLsMod:     LongInt;
      vcbAtrb:      Integer;
      vcbNmFls:     Integer;
      vcbVBMSt:     Integer;
      vcbAllocPtr:  Integer;
      vcbNmAlBlks:  Integer;
      vcbAlBlkSiz:  LongInt;
      vcbClpSIz:    LongInt;
      vcbAlBlSt:    Integer;
      vcbNxtCNID:   LongInt;
      vcbFreeBks:   Integer;
      vcbVN:        STRING[27];
      vcbDrvNum:    Integer;
      vcbDRefNum:   Integer;
      vcbFSID:      Integer;
      vcbVRefNum:   Integer;
```

```
    vcbMAdr:      Ptr;
    vcbBufAdr:    Ptr;
    vcbMLen:      Integer;
    vcbDirIndex:  Integer;
    vcbDirBlk:    Integer;

    vcbVolBkup:   LongInt;          { new HFS extensions }
    vcbVSegNum:   Integer;
    vcbWrCnt:     LongInt;
    vcbXTClpSiz:  LongInt;
    vcbCTClpSiz:  LongInt;
    vcbNmRtDirs:  Integer;
    vcbFilCnt:    LongInt;
    vcbDirCnt:    LongInt;
    vcbFndrInfo:  FInfo;
    vcbVCSize:    Integer;
    vcbVBMCSiz:   Integer;
    vcbCtlCSiz:   Integer;

    vcbXTAlBlks:  Integer;          { additional VCB info }
    vcbCTAlBlks:  Integer;
    vcbXTRef:     Integer;
    vcbCTRef:     Integer;
    vcbCtlBuf:    LongInt;
    vcbDirIDM:    LongInt;
    vcbOffsM:     Integer;
end;

{ general queue data structure }
QHdr = record
    QFlags: Integer;      { misc flags }
    QHead: QElemPtr;      { first elem }
    QTail: QElemPtr;      { last elem }
  end;                  { QHdr }
QHdrPtr = ^QHdr;
{ there are currently four types of queues: }
{ VType, queue of Vertical Blanking Control Blocks }
{ IOQType, queue of I/O queue elements }
{ DrvType, queue of drivers }
{ EvType, queue of Event Records }
{ FSQType, queue of VCB elements }
{ TimerType no longer is used. }
{ DrvType replaces it here in enum type }
QTypes = (dummyType,vType,ioQType,drvQType,evType,fsQType);

QElem = record
        case QTypes of
        vType:
          (vblQelem: VBLTask);      { vertical blanking }

        ioQType:
          (ioQElem: ParamBlockRec); { I/O parameter block }

        drvQType:
          (drvQElem: DrvQEl);       { drive }

        evType:
          (evQElem: EvQEl);         { event }

        fsQType:
          (vcbQElem: VCB);          { volume control block }

  end;                            { QElem }

 DCtlEntry = record                 { device control entry }
              DCtlDriver: Ptr;       { ptr to ROM or handle to RAM driver }
```

```
                DCtlFlags: Integer;     { flags }
                DCtlQHdr: QHdr;         { driver's I/O queue }
                DCtlPosition: LongInt; { byte pos used by read and write calls }
                DCtlStorage: Handle;    { hndl to RAM drivers private storage }
                DCtlRefNum: Integer;    { driver's reference number }
                DCtlCurTicks: LongInt;{ long counter for timing system task calls }
                DCtlWindow: Ptr;        { ptr to driver's window if any }
                DCtlDelay: Integer;     { number of ticks btwn sysTask calls }
                DCtlEMask: Integer;     { desk acessory event mask }
                DCtlMenu: Integer;      { menu ID of menu associated with driver }
              end;                    { DCtlEntry }
  DCtlPtr = ^DCtlEntry;
  DCtlHandle = ^DCtlPtr;

   { for Serial Driver }
 SerShk = packed record          { handshake control fields }
                  fXOn: Byte;     { XON flow control enabled flag }
                  fCTS: Byte;     { CTS flow control enabled flag }
                  xon:  Char;     { XOn character }
                  xoff: Char;     { XOff character }
                  errs: Byte;     { errors mask bits }
                  evts: Byte;     { event enable mask bits }
                  fInX: Byte;     { input flow control enabled flag }
                  null: Byte;     { unused }
                end;

   { parameter block structure for file and IO routines }
 SerStaRec = packed record
                    cumErrs:  Byte;   { cumulative errors report }
                    XOFFSent: Byte;   { XOff Sent flag }
                    rdPend:   Byte;   { read pending flag }
                    wrPend:   Byte;   { write pending flag }
                    ctsHold:  Byte;   { CTS flow control hold flag }
                    XOFFHold: Byte;   { XOff flow control hold flag }
                  end;

 { for sound driver }
 { for 4-tone sound generation }
 Wave = packed array[0..255] of Byte;
 WavePtr = ^Wave;
 FTSoundRec = record
                duration: Integer;
                sound1Rate: LongInt;
                sound1Phase: LongInt;
                sound2Rate: LongInt;
                sound2Phase: LongInt;
                sound3Rate: LongInt;
                sound3Phase: LongInt;
                sound4Rate: LongInt;
                sound4Phase: LongInt;
                sound1Wave: WavePtr;
                sound2Wave: WavePtr;
                sound3Wave: WavePtr;
                sound4Wave: WavePtr;
              end;
 FTSndRecPtr = ^FTSoundRec;

 FTSynthRec = record
                mode: Integer;
                sndRec: FTSndRecPtr;
              end;
 FTSynthPtr = ^FTSynthRec;

 Tone = record
          count: Integer;
          amplitude: Integer;
```

```
        duration: Integer;
      end;

Tones = array[0..5000] of Tone;

SWSynthRec = record
               mode: Integer;
               triplets: Tones;
             end;

SWSynthPtr = ^SWSynthRec;

freeWave = packed array[0..30000] of Byte;

FFSynthRec = record
               mode: Integer;
               count: Fixed;
               waveBytes: freeWave;
             end

FFSynthPtr = ^FFSynthRec;

DateTimeRec = record                { for date and time }
                Year,                 { 1904,1905,... }
                Month,                { 1,...,12 corresponding to Jan,...,Dec }
                Day,                  { 1,...31 }
                Hour,                 { 0,...,23 }
                Minute,               { 0,...,59 }
                Second,               { 0,...,59 }
                DayOfWeek: Integer; { 1,...,7 corresponding to Sun,...,Sat }
              end;                  { DateTimeRec }


appFile = record                     { for application parameter }
            vRefNum: Integer;
            ftype: OSType;
            versNum: Integer;    { versNum in high byte }
            fName: str255;
          end; {appFile}

SPortSel = (SPortA,SPortB);          { for RAM serial driver }

DriveKind = (sony, hard20);          { for disk driver }

DrvSts = record
         track:              Integer;     { current track }
         writeProt:          SignedByte;  { bit 7 = 1 if volume is locked }
         diskInPlace:        SignedByte;  { disk in drive }
         installed:          SignedByte;  { drive installed }
         sides:              SignedByte;  { -1 for 2-sided, 0 for 1-sided }
         DriveQLink:         QElemPtr;    { next queue entry }
         DriveQVers:         Integer;     { 1 for HD20 }
         dqDrive:            Integer;     { drive number }
         dqRefNum:           Integer;     { driver reference number }
         dqFSID:             Integer;     { file system ID }
         case DriveKind of
          sony:
          (twoSideFmt: SignedByte;        { -1 for 2-sided, 0 for 1-sided; }
                                          { valid after first read or write }
           needsFlush:       SignedByte;  { -1 for Mac Plus drive }
           diskErrs:         Integer);    { soft error count }
          hard20:
           (DriveSize:       Integer;     { drive block size low word }
           DriveS1:          Integer;     { drive block size high word }
           DriveType:        Integer;     { 1 for HD20 }
           DriveManf:        Integer;     { 1 for Apple Computer, Inc. }
```

```
            DriveChar:          Byte;       { 230 ($E6) for HD20 }
            DriveMisc:          SignedByte) { 0 - reserved }
    end;

{ for Event Manager }
function PostEvent(eventNum: Integer; eventMsg: LongInt): OSErr; external;
function  PPostEvent(eventCode: Integer; eventMsg: LongInt;
                                        var qEl: EvQEl): OSErr; external;
procedure FlushEvents(whichMask,stopMask: Integer); inline $201F, $A032;
procedure SetEventMask(theMask: Integer); inline $31DF, $0144;
function  OSEventAvail(mask: Integer; var theEvent: EventRecord): Boolean; external;
function  GetOSEvent(mask: Integer; var theEvent: EventRecord): Boolean; extended;

{ OS utilities }
function  HandToHand(var theHndl: Handle): OSErr; external;
function  PtrToXHand(srcPtr: Ptr; dstHndl: Handle; size: LongInt): OSErr; external;
function  PtrToHand(srcPtr: Ptr; var dstHndl: Handle; size: LongInt): OSErr; external;
function  HandAndHand(hand1,hand2: Handle): OSErr; external;
function  PtrAndHand(ptr1: Ptr; hand2: Handle; size: LongInt): OSErr; external;
procedure SysBeep(duration: Integer); inline $A9C8;
procedure Environs(var rom,machine: Integer); external;
procedure Restart; external;


{ routines to set A5 to CurrentA5 and then restore A5 to previous value }
{ useful for ensuring good world for IOCompletion routines }
procedure SetUpA5; inline $2F0D, $2A78, $0904;
    { MOVE.L A5,-(SP)       ;save old A5 on stack
      MOVE.L CurrentA5,A5   ;get the real A5 }
procedure RestoreA5; inline $2A5F;
    { MOVE.L (A7)+,A5       ;restore A5 }

{ from HEAPZONE.TEXT }
procedure SetApplBase(startPtr: Ptr); external;
procedure InitApplZone; external;
procedure InitZone(pgrowZone:          ProcPtr;
                   cmoreMasters:         Integer;
                   limitPtr,startPtr : Ptr); external;
function GetZone: THz; external;
procedure SetZone(hz: THz); external;

function  ApplicZone: THz; inline $2EB8, $02AA;
function  SystemZone: THz; inline $2EB8, $02A6;

function  CompactMem(cbNeeded: size): size; external;
procedure PurgeMem(cbNeeded: size); external;
function  FreeMem: LongInt; external;
procedure ResrvMem(cbNeeded: size); external;
function  MaxMem(Var grow: size): size; external;
function  TopMem: Ptr; inline $2EB8, $0108;

procedure SetGrowZone(growZone: ProcPtr); external;
procedure SetApplLimit(zoneLimit: Ptr); external;
function  GetApplLimit: Ptr; inline $2EB8, $0130;
function  StackSpace: LongInt; external;

procedure PurgeSpace (var total, contig: LongInt); external;
function  MaxBlock: LongInt; external;

procedure MaxApplZone; external;
procedure MoveHHi (h: handle); external;

function  NewPtr(byteCount: size): Ptr; external;
procedure DisposPtr(p: Ptr); external;
function  GetPtrSize(p: Ptr): size; external;
procedure SetPtrSize(p: Ptr; newSize: size); external;
```

```
function  PtrZone(p: Ptr): THz; external;

function  NewHandle(byteCount: size): handle; external;

function  NewEmptyHandle: handle; external;

procedure DisposHandle(h: handle); external;
function  GetHandleSize(h: handle): size; external;
procedure SetHandleSize(h: handle; newSize: size); external;
function  HandleZone(h: handle): THz; external;
function  RecoverHandle(p: Ptr): handle; external;
procedure EmptyHandle(h: handle); external;
procedure ReAllocHandle(h: handle; byteCount: size); external;

procedure HLock(h: handle); external;
procedure HUnLock(h: handle); external;
procedure HPurge(h: handle); external;
procedure HNoPurge(h: handle); external;

procedure HSetRBit(h: handle); external;
procedure HClrRBit(h: handle); external;
procedure HSetState(h: handle; flags: SignedByte); external;
function  HGetState(h: handle): SignedByte; external;

procedure MoreMasters; external;

procedure BlockMove(srcPtr, destPtr: Ptr; byteCount: size); external;
function  MemError: OSErr; inline $3EB8, $0220;

function  GZSaveHnd: handle; inline $2EB8, $0328;


{ interface for core routines pertaining to the vertical retrace mgr }
{ routines defined in VBLCORE.TEXT }
function  VInstall(VBLTaskPtr: QElemPtr): OSErr; external;
function  VRemove(VBLTaskPtr: QElemPtr): OSErr; external;

{ interface for operating system dispatcher }
{ routines defined in DISPATCH.TEXT }
function  GetTrapAddress(trapNum: Integer): LongInt; external;
procedure SetTrapAddress(trapAddr: LongInt; trapNum: Integer); external;
function  NGetTrapAddress(trapNum: Integer; tTyp: TrapType): LongInt; external;
procedure NSetTrapAddress(trapAddr: LongInt; trapNum: Integer; tTyp: TrapType);  external;


{ interface for utility core routines (defined in sysutil) }
function  GetSysPPtr: SysPPtr; external;
function  WriteParam: OSErr; external;
function  SetDateTime(time:  LongInt):OSErr; external;
function  ReadDateTime(var time: LongInt):OSErr; external;
procedure GetDateTime(var secs: LongInt); external;
procedure SetTime(d: DateTimeRec); external;
procedure GetTime(var d: DateTimeRec); external;
procedure Date2Secs(d: DateTimeRec; var s: LongInt); external;
procedure Secs2Date(s: LongInt; var d: DateTimeRec); external;
procedure Delay(numTicks: LongInt; var finalTicks: LongInt); external;
function  EqualString(str1,str2: Str255; caseSens,diacSens: Boolean):Boolean;  external;


function  RelString(aStr,bStr: Str255; caseSens, diacSens: Boolean): Integer;  external;


procedure UprString(var theString: Str255; diacSens: Boolean); external;
function  InitUtil: OSErr; inline $A03F, $3E80;

procedure UnLoadSeg(routineAddr: Ptr); inline $A9F1;
```

```
procedure ExitToShell; inline $A9F4;
procedure GetAppParms(var apName: str255; var apRefNum: Integer;
                      var apParam: Handle); inline $A9F5;
procedure CountAppFiles(var message: Integer; var count: Integer); external;
procedure GetAppFiles(index: Integer; var theFile: AppFile); external;
procedure ClrAppFiles(index: Integer); external;

{ queue routines - part of Macintosh core utility routines }
procedure FInitQueue; inline $A016;
procedure Enqueue(qElement: QElemPtr; qHeader: QHdrPtr); external;
function  Dequeue(qElement: QElemPtr; qHeader: QHdrPtr): OSErr; external;
function  GetFSQHdr: QHdrPtr;  external;
function  GetDrvQHdr: QHdrPtr; external;
function  GetVCBQHdr: QHdrPtr; external;
function  GetVBLQHdr: QHdrPtr; external;
function  GetEvQHdr: QHdrPtr;  external;
function  GetDCtlEntry(refNum: Integer): DCtlHandle; external;

function  PBOpen(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;    external;
function  PBClose(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;   external;
function  PBRead(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;    external;
function  PBWrite(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;   external;
function  PBControl(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBStatus(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBKillIO(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;

function  PBGetVInfo(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBGetVol(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBSetVol(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBFlushVol(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBCreate(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBDelete(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBOpenRF(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBRename(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBGetFInfo(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBSetFInfo(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBSetFLock(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBRstFLock(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBSetFVers(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBAllocate(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBGetEOF(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBSetEOF(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;  external;
function  PBGetFPos(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBSetFPos(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBFlushFile(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBMountVol(paramBlock: ParmBlkPtr): OSErr; external;
function  PBUnMountVol(paramBlock: ParmBlkPtr): OSErr; external;
function  PBEject(paramBlock: ParmBlkPtr): OSErr; external;
function  PBOffLine(paramBlock: ParmBlkPtr): OSErr; external;
procedure AddDrive(drvrRefNum: Integer; drvNum: Integer; QEl: drvQElPtr); external;

function  PBOpenWD(paramBlock: WDPBPtr; aSync: Boolean): OSErr;  external;
function  PBCloseWD(paramBlock: WDPBPtr; aSync: Boolean): OSErr; external;
function  PBHSetVol(paramBlock: WDPBPtr; aSync: Boolean): OSErr; external;
function  PBHGetVol(paramBlock: WDPBPtr; aSync: Boolean): OSErr; external;
function  PBCatMove(paramBlock: CMovePBPtr; aSync: Boolean): OSErr; external;
function  PBDirCreate(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBGetWDInfo(paramBlock: WDPBPtr; aSync: Boolean): OSErr; external;
function  PBGetFCBInfo(paramBlock: FCBPBPtr; aSync: Boolean): OSErr; external;
function  PBGetCatInfo(paramBlock: CInfoPBPtr; aSync: Boolean): OSErr; external;
function  PBSetCatInfo(paramBlock: CInfoPBPtr; aSync: Boolean): OSErr; external;
function  PBAllocContig(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBLockRange(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBUnLockRange(paramBlock: ParmBlkPtr; aSync: Boolean): OSErr; external;

function  PBSetVInfo(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
```

```
function  PBHGetVInfo(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHOpen(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHOpenRF(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHCreate(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHDelete(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHRename(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHRstFLock(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHSetFLock(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHGetFInfo(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBHSetFInfo(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;
function  PBSetPEOF(paramBlock: hParmBlkPtr; aSync: Boolean): OSErr; external;

function  FSOpen(fileName: Str255; vRefNum: Integer;
              var refNum: Integer): OSErr; external;
function  FSClose(refNum: Integer): OSErr; external;
function  FSRead(refNum: Integer;
              var count: LongInt; buffPtr: Ptr): OSErr; external;
function  FSWrite(refNum: Integer;
              var count: LongInt; buffPtr: Ptr): OSErr; external;
function  Control(refNum: Integer; csCode: Integer;
              csParamPtr: Ptr): OSErr; external;
function  Status(refNum: Integer; csCode: Integer;
              csParamPtr: Ptr): OSErr; external;
function  KillIO(refNum: Integer): OSErr; external;

{ volume level calls }
function  GetVInfo(drvNum: Integer; volName: StringPtr; var vRefNum: Integer;
                   var FreeBytes: LongInt): OSErr; external;
function  GetFInfo(fileName: Str255; vRefNum: Integer;
                   var FndrInfo: FInfo):OSErr; external;
function  GetVol(volName: StringPtr; var vRefNum: Integer):OSErr; external;
function  SetVol(volName: StringPtr; vRefNum: Integer): OSErr; external;
function  UnMountVol(volName: StringPtr; vRefNum: Integer):OSErr; external;
function  Eject(volName: StringPtr; vRefNum: Integer): OSErr; external;
function  FlushVol(volName: StringPtr; vRefNum: Integer):OSErr; external;

{ file level calls for unopened files }
function  Create(fileName: Str255; vRefNum: Integer; creator: OSType;
                 fileType: OSType):OSErr; external;
function  FSDelete(fileName: Str255; vRefNum: Integer):OSErr; external;
function  OpenRF(fileName: Str255; vRefNum: Integer;
                 var refNum: Integer): OSErr; external;
function  Rename(oldName: Str255; vRefNum: Integer;
                 newName: Str255):OSErr; external;
function  SetFInfo(fileName: Str255; vRefNum: Integer;
                 FndrInfo: FInfo):OSErr; external;
function  SetFLock(fileName: Str255; vRefNum: Integer):OSErr; external;
function RstFLock(fileName: Str255; vRefNum: Integer):OSErr; external;

{ file level calls for opened files }
function  Allocate(refNum: Integer; var count: LongInt):OSErr; external;
function  GetEOF(refNum: Integer; var LogEOF: LongInt):OSErr; external;
function  SetEOF(refNum: Integer; LogEOF: LongInt):OSErr; external;
function  GetFPos(refNum: Integer; var filePos: LongInt):OSErr; external;
function  SetFPos(refNum: Integer; posMode: Integer; posOff: LongInt):OSErr;
external;
function  GetVRefNum(fileRefNum: Integer; var vRefNum: Integer):OSErr; external;

{ serial driver interface }
function  OpenDriver(name: Str255; var drvrRefNum: Integer): OSErr; external
function  CloseDriver(refNum:  Integer):OSErr; external

function  SerReset(refNum: Integer; serConfig: Integer): OSErr; external;
function  SerSetBuf(refNum: Integer; serBPtr: Ptr; serBLen: Integer):
                    OSErr; external;
function  SerHShake(refNum: Integer; flags: SerShk): OSErr; external;
```

```
function  SerSetBrk(refNum: Integer): OSErr; external;
function  SerClrBrk(refNum: Integer): OSErr; external;
function  SerGetBuf(refNum: Integer;
                    var count: LongInt): OSErr; external;
function  SerStatus(refNum: Integer;
                    var serSta: SerStaRec): OSErr; external;
function  DiskEject(drvnum: Integer): OSErr; external;
function  SetTagBuffer(buffPtr: Ptr): OSErr; external;
function  DriveStatus(drvNum: Integer; var status: DrvSts): OSErr; external;
function  RamSDOpen(whichPort: SPortSel): OSErr; external;
procedure RamSDClose(whichPort: SPortSel); external;

{ for sound driver }
procedure SetSoundVol(level: Integer); external;
procedure GetSoundVol(var level: Integer); external;
procedure StartSound(synthRec: Ptr; numBytes: LongInt; CompletionRtn: ProcPtr); external;

procedure StopSound; external;
function  SoundDone: Boolean; external;

{ for the system error handler }
procedure SysError(errorCode: Integer); inline $3D1F, $A9C9;
```

ToolIntf

ToolIntf implements the Macintosh's user interface features: windows, menus, controls, dialog boxes, text editing commands, and so on. This unit is needed in any Mac-style program.

```
unit ToolIntf(-9);

interface

uses MemTypes,QuickDraw,OsIntf;

const       { for FontManager }

  commandMark = $11;
  checkMark   = $12;
  diamondMark = $13;
  appleMark   = $14;

  systemFont =  0;
  applFont   =  1;
  newYork    =  2;
  geneva     =  3;
  monaco     =  4;
  venice     =  5;
  london     =  6;
  athens     =  7;
  sanFran    =  8;
  toronto    =  9;
  cairo      = 11;
  losAngeles = 12;
  times      = 20;
  helvetica  = 21;
  courier    = 22;
  symbol     = 23;
  mobile     = 24;

  propFont   = $9000;
  prpFntH    = $9001;
  prpFntW    = $9002;
  prpFntHW   = $9003;

  fixedFont  = $B000;
  fxdFntH    = $B001;
  fxdFntW    = $B002;
  fxdFntHW   = $B003;

  fontWid    = $ACB0;

  { for Window Manager }
  wDraw         = 0;        { window messages }
  wHit          = 1;
  wCalcRgns     = 2;
  wNew          = 3;
  wDispose      = 4;
  wGrow         = 5;
  wDrawGIcon    = 6;

  dialogKind  = 2;          { types of windows }
  userKind    = 8;

  deskPatID = 16;           { desk pattern resource ID }
```

```
documentProc  =  0;       { window definition procedure IDs }
dBoxProc      =  1;
plainDBox     =  2;
altDBoxProc   =  3;
noGrowDocProc =  4;
zoomDocProc   =  8;
zoomNoGrow    = 12;
rDocProc      = 16;

inDesk        = 0;        { FindWindow result codes }
inMenuBar     = 1;
inSysWindow   = 2;
inContent     = 3;
inDrag        = 4;
inGrow        = 5;
inGoAway      = 6;

inZoomIn      = 7;        { new 128K ROM }
inZoomOut     = 8;

wNoHit        = 0;        { defProc hit test codes }
wInContent    = 1;
wInDrag       = 2;
wInGrow       = 3;
wInGoAway     = 4;

wInZoomIn     = 5;        { new 128K ROM }
wInZoomOut    = 6;

noConstraint = 0;         { axis constraints for DragGrayRgn call }
hAxisOnly     = 1;
vAxisOnly     = 2;

teJustLeft    =  0;       { for TextEdit }
teJustRight   = -1;
teJustCenter  =  1;

{ for Resource Manager }
{ resource attribute byte}
resSysHeap    =  64;      { system or application heap? }
resPurgeable  =  32;      { purgeable resource? }
resLocked     =  16;      { load it in locked? }
resProtected  =   8;      { protected? }
resPreload    =   4;      { load in on OpenResFile? }
resChanged    =   2;      { resource changed? }

mapReadOnly   =   128;    { resource file read-only }
mapCompact    =    64;    { compact resource file }
mapChanged    =    32;    { write map out at update }

resNotFound   =  -192;    { resource not found }
resFNotFound  =  -193;    { resource file not found }
addResFailed  =  -194;    { AddResource failed }
rmvResFailed  =  -196;    { RmveResource failed }

{ ID's for resources provided in sysResDef }

{ standard cursor definitions }
iBeamCursor = 1;          { text selection cursor }
crossCursor = 2;          { for drawing graphics }
plusCursor  = 3;          { for structured selection }
watchCursor = 4;          { for indicating a long delay }

stopIcon    = 0;          { icons }
noteIcon    = 1;
```

```
cautionIcon = 2;

{ patterns }
sysPatListID = 0;       { ID of PAT# which contains 38 patterns }

{ for Control Manager }
drawCntl  = 0;          { control messages }
testCntl  = 1;
calcCRgns = 2;
initCntl  = 3;
dispCntl  = 4;
posCntl   = 5;
thumbCntl = 6;
dragCntl  = 7;
autoTrack = 8;

inButton     =  10;     { FindControl result codes }
inCheckbox   =  11;
inUpButton   =  20;
inDownButton =  21;
inPageUp     =  22;
inPageDown   =  23;
inThumb      = 129;

pushButProc   =  0;     { control definition proc ID's }
checkBoxProc  =  1;
radioButProc  =  2;
scrollBarProc = 16;

useWFont      =  8;

userItem =  0;          { for Dialog Manager }
ctrlItem =  4;

btnCtrl =   0;          { low 2 bits specify what kind of control }
chkCtrl =   1;
radCtrl =   2;
resCtrl =   3;

statText    =    8;     { static text }
editText    =   16;     { editable text }
iconItem    =   32;     { icon item }
picItem     =   64;     { picture item }
itemDisable =  128;     { disable item if set }

ok     = 1;             { OK button is first by convention }
cancel = 2;             { cancel button is second by convention }

{ for Menu Manager }
noMark       =  0;      { mark symbol for MarkItem }
TextMenuProc =  0;

{ menu defProc messages }
mDrawMsg    = 0;
mChooseMsg  = 1;
mSizeMsg    = 2;

{ for Scrap Manager }
noScrapErr = -100;      { desk scrap isn't initialized }
noTypeErr  = -102;

{ package manager }
dskInit  = 2;           { disk initializaton }
stdFile  = 3;           { standard file }
flPoint  = 4;           { floating-point arithmetic }
trFunc   = 5;           { transcendental functions }
```

```
  intUtil  = 6;          { international utilities }
  bdConv   = 7;          { binary/decimal conversion }

type

  Int64Bit =             { general utilities }
      record
        hiLong: LongInt;
        loLong: LongInt;
      end;

  CursPtr = ^Cursor;
  CursHandle = ^CursPtr;

  PatPtr = ^Pattern;
  PatHandle = ^PatPtr;

  FMInput = packed record           { for Font Manager }
              family:   Integer;
              size:     Integer;
              face:     Style;
              needBits: Boolean;
              device:   Integer;
              numer:    Point;
              denom:    Point;
            end;

  FMOutPtr = ^FMOutPut;

  FMOutPut = packed record
               errNum:     Integer;
               fontHandle: Handle;
               bold:       Byte;
               italic:     Byte;
               ulOffset:   Byte;
               ulShadow:   Byte;
               ulThick:    Byte;
               shadow:     Byte;
               extra:      SignedByte;
               ascent:     Byte;
               descent:    Byte;
               widMax:     Byte;
               leading:    SignedByte;
               unused:     Byte;
               numer:      Point;
               denom:      Point;
             end;

  FontRec = record
              fontType:    Integer;  { font type }
              firstChar:   Integer;  { ASCII code of first character }
              lastChar:    Integer;  { ASCII code of last character }
              widMax:      Integer;  { maximum character width }
              kernMax:     Integer;  { negative of maximum character kern }
              nDescent:    Integer;  { negative of descent }
              fRectWidth:  Integer;  { width of font rectangle }
              fRectHeight: Integer;  { height of font rectangle }
              oWTLoc:      Integer;  { offset to offset/width table }
              ascent:      Integer;  { ascent }
              descent:     Integer;  { descent }
              leading:     Integer;  { leading }
              rowWords:    Integer;  { row width of bit image / 2 }
              bitImage:    array[1..rowWords,1..fRectHeight] of Integer;
              locTable:    array[firstChar..lastChar+2] of Integer;
              owTable:     array[firstChar..lastChar+2] of Integer;
              widthTable:  array[firstChar..lastChar+2] of Integer;
```

```
          heightTable: array[firstChar..lastChar+2] of Integer; }
        end;

WidthTable = packed record              { new 128K ROM }
               tabData:        array[1..256] of fixed; { character widths }
               tabFont:        Handle;  { font record used to build table }
               sExtra:         LongInt; { extra space used for table }
               style:          LongInt; { extra due to style }
               fID:            Integer; { font family ID }
               fSize:          Integer; { font size request }
               face:           Integer; { style (face) request }
               device:         Integer; { device requested }
               vInScale:       Fixed;
               hInScale:       Fixed;
               aFID:           Integer; { actual font family ID for table }
               fHand:          Handle;  { family record used to build up table }
               usedFam:        Boolean; { used fixed point family widths }
               aFace:          Byte;    { actual face produced }
               vOutput:        Integer; { vertical scale output value }
               hOutput:        Integer; { horizontal scale output value }
               vFactor:        Integer; { vertical scale output value }
               hFactor:        Integer; { horizontal scale output value }
               aSize:          Integer; { actual size of actual font used }
               tabSize:        Integer; { total size of table }
             end;

FMetricRec = record
               ascent:     Fixed;       { baseline to top }
               descent:    Fixed;       { baseline to bottom }
               leading:    Fixed;       { leading between lines }
               widMax:     Fixed;       { maximum character width }
               wTabHandle: Fixed;       { handle to font width table }
             end;

WidTable = record
             numWidths: Integer;    { number of entries - 1 }
             { widList: array[1..numWidths] of WidEntry }
           end;

WidEntry = record
             widStyle: Integer;         { style entry applies to }
             { widRec: array[firstChar..lastChar] of Integer }
           end;

AsscEntry = record
              fontSize:   Integer;
              fontStyle:  Integer;
              fontID:     Integer;      { font resource ID }
            end;

FontAssoc = record
              numAssoc: Integer;       { number of entries - 1 }
              { asscTable: array[1..numAssoc] of AsscEntry }
            end;

StyleTable = record
               fontClass:     Integer;
               offset:        LongInt;
               reserved:      LongInt;
               indexes:       array[0..47] of Byte;
             end;

NameTable = record
              stringCount:   Integer;
              baseFontName:  STR255;
              { strings: array[2..stringCount]  of string }
```

```
            { the lengths of the strings are arbitrary }
          end;

  KernPair = record
              kernFirst:   Char;    { 1st character of kerned pair }
              kernSecond:  Char;    { 2nd character of kerned pair }
              kernWidth:   Integer; { kerning in 1 pt fixed format }
            end;

  KernEntry = record
               kernLength:  Integer;    { length of this entry }
               kernStyle:   Integer;    { style the entry applies to }
               { kernRec: array[1..(kernLength/4)-1] of KernPair }
             end;

  KernTable = record
               numKerns: Integer;  { number of kerning entries }
               { kernList: array[1..numKerns] of KernEntry }
             end;

  FamRec = record
             ffFlags:     Integer; { flags for family }
             ffFamID:     Integer; { family ID number }
             ffFirstChar: Integer; { ASCII code of 1st character }
             ffLastChar:  Integer; { ASCII code of last character }
             ffAscent:    Integer; { maximum ascent for 1 pt font }
             ffDescent:   Integer; { maximum descent for 1 pt font }
             ffLeading:   Integer; { maximum leading for 1 pt font }
             ffWidMax:    Integer; { maximum widMax for 1 pt font }
             ffWTabOff:   LongInt; { offset to width table }
             ffKernOff:   LongInt; { offset to kerning table }
             ffStylOff:   LongInt; { offset to style mapping table }
             ffProperty:  array[1..9] of Integer; { style property info }
             ffIntl:      array[1..2] of Integer; { for international use }
             ffVersion:   Integer;      { version number }
             {ffAssoc:    FontAssoc;}  { font association table }
             {ffWidthTab: WidTable;}   { width table }
             {ffStyTab:   StyleTable;} {style mapping table }
             {ffKernTab:  KernTable;}  { kerning table }
           end;

{ for Event Manager }
{the Event Record is defined in OsIntf}

  KeyMap = packed array[0..127] of Boolean;

  WindowPtr     = GrafPtr;          { for Window Manager }
  WindowPeek    = ^WindowRecord;
  ControlHandle = ^ControlPtr;      { for Control Manager }

  WindowRecord = record
                   port:          GrafPort;
                   windowKind:    Integer;
                   visible:       Boolean;
                   hilited:       Boolean;
                   goAwayFlag:    Boolean;
                   spareFlag:     Boolean;
                   strucRgn:      RgnHandle;
                   contRgn:       RgnHandle;
                   updateRgn:     RgnHandle;
                   windowDefProc: Handle;
                   dataHandle:    Handle;
                   titleHandle:   StringHandle;
                   titleWidth:    Integer;
                   ControlList:   ControlHandle;
                   nextWindow:    WindowPeek;
```

```
                windowPic:      PicHandle;
                refCon:         LongInt;
              end;

TERec = record                          { for TextEdit }
          destRect: Rect;                 { destination rectangle }
          viewRect: Rect;                 { view rectangle }
          selRect: Rect;                  { select rectangle }
          lineHeight: Integer;            { current font line height }
          fontAscent: Integer;            { current font ascent }
          selPoint: Point;                { selection point(mouseLoc) }
          selStart: Integer;              { selection start }
          selEnd: Integer;                { selection end }
          active: Integer;                { <>0 if active }
          wordBreak:  ProcPtr;            { word break routine }
          clikLoop:  ProcPtr;             { click loop routine }
          clickTime: LongInt;             { time of first click }
          clickLoc: Integer;              { char. location of click }
          caretTime: LongInt;             { time for next caret blink }
          caretState: Integer;            { on/active Booleans }
          just: Integer;                  { fill style }
          teLength: Integer;              { length of text below }
          hText: Handle;                  { handle to actual text }
          recalBack: Integer;             { <>0 if recal in background }
          recalLines: Integer;            { line being recal'ed }
          clikStuff: Integer;             { click stuff (internal) }
          crOnly: Integer;                { set to -1 if CR line breaks only }
          txFont: Integer;                { text Font }
          txFace: Style;                  { text Face }
          txMode: Integer;                { text Mode }
          txSize: Integer;                { text Size }
          inPort: GrafPtr;                { Grafport }
          highHook: ProcPtr;              { highlighting hook }
          caretHook: ProcPtr;             { highlighting hook }
          nLines: Integer;                { number of lines }
          lineStarts: array[0..16000] of Integer;
                                          { actual line starts itself }
        end;                            { record }
TEPtr = ^TERec;
TEHandle = ^TEPtr;

CharsHandle = ^CharsPtr;
CharsPtr = ^Chars;
Chars = packed array[0..32000] of Char;

{ for Resource Manager }
ResType = packed array[1..4] of Char;

{ for Control Manager }
ControlPtr    = ^ControlRecord;

ControlRecord = packed record
                  nextControl:      ControlHandle;
                  contrlOwner:      WindowPtr;
                  contrlRect:       Rect;
                  contrlVis:        Byte;
                  contrlHilite:     Byte;
                  contrlValue:      Integer;
                  contrlMin:        Integer;
                  contrlMax:        Integer;
                  contrlDefProc:    Handle;
                  contrlData:       Handle;
                  contrlAction:     ProcPtr;
                  contrlrfCon:      LongInt;
                  contrlTitle:      Str255;
                end;                      { ControlRecord }
```

```
{ for Dialog Manager }
DialogPtr=    WindowPtr;
DialogPeek=   ^DialogRecord;
DialogRecord= record
                window:      WindowRecord;
                Items:       Handle;
                textH:       TEHandle;
                editField:   Integer;
                editOpen:    Integer;
                aDefItem:    Integer;
              end;

DialogTHndl=   ^DialogTPtr;
DialogTPtr=    ^DialogTemplate;
DialogTemplate= record
                  boundsRect:  Rect;
                  procID:      Integer;
                  visible:     Boolean;
                  filler1:     Boolean;
                  goAwayFlag:  Boolean;
                  refCon:      LongInt;
                  ItemsID:     Integer;
                  title:       Str255;
                end;

StageList=    packed record
                boldItm4:  0..1;
                boxDrwn4:  Boolean;
                sound4:    0..3;
                boldItm3:  0..1;
                boxDrwn3:  Boolean;
                sound3:    0..3;
                boldItm2:  0..1;
                boxDrwn2:  Boolean;
                sound2:    0..3;
                boldItm1:  0..1;
                boxDrwn1:  Boolean;
                sound1:    0..3;
              end;

AlertTHndl=   ^AlertTPtr;
AlertTPtr=    ^AlertTemplate;
AlertTemplate= record
                boundsRect:  Rect;
                itemsID:     Integer;
                stages:      StageList;
               end;

{ for Menu Manager }
MenuPtr =    ^MenuInfo;
MenuHandle = ^MenuPtr;
MenuInfo =    record
                menuId:       Integer;
                menuWidth:    Integer;
                menuHeight:   Integer;
                menuProc:     Handle;
                enableFlags:  LongInt;
                menuData:     Str255;
              end;

{ for Scrap Manager }
ScrapStuff = record
               scrapSize:   LongInt;
               scrapHandle: Handle;
               scrapCount:  Integer;
```

```
                scrapState:  Integer;
                scrapName:   StringPtr;
              end;
  pScrapStuff = ^ScrapStuff;

{ general utilities }
function  BitAnd(long1,long2: LongInt): LongInt; inline $201F,$C09F,$2E80;
function  BitOr(long1,long2: LongInt): LongInt; inline $201F,$809F,$2E80;
function  BitXor(long1,long2: LongInt): LongInt; inline $201F,$2E9F,$B197;
function  BitNot(long: LongInt): LongInt; inline $2E9F,$4697;
function  BitShift(long: LongInt; count: Integer): LongInt; inline $A85C;
function  BitTst(bytePtr: Ptr; bitNum: LongInt): Boolean; inline $A85D;
procedure BitSet(bytePtr: Ptr; bitNum: LongInt); inline $A85E;
procedure BitClr(bytePtr: Ptr; bitNum: LongInt); inline $A85F;
procedure LongMul(a,b: LongInt; var dst: Int64Bit); inline $A867;
function  FixMul(a,b: Fixed): Fixed; inline $A868;
function  FixRatio(numer,denom: Integer): Fixed; inline $A869;
function  FixRound(x: Fixed): Integer; inline $A86C;
procedure PackBits(var srcPtr,dstPtr: Ptr; srcBytes: Integer); inline $A8CF;
procedure UnPackBits(var srcPtr,dstPtr: Ptr; dstBytes: Integer); inline $A8D0;
function  SlopeFromAngle(angle: Integer): Fixed; inline $A8BC;
function  AngleFromSlope(slope: Fixed): Integer; inline $A8C4;
function  DeltaPoint(ptA,ptB: Point): LongInt; inline $A94F;

function  NewString(theString:Str255): StringHandle; inline $A906;
procedure SetString(theString:StringHandle; strNew: Str255); inline $A907;
function  GetString(stringID:  Integer): StringHandle; inline $A9BA;
procedure GetIndString(var theString: str255; strListID: Integer;
                       index: Integer); external;

function Munger(h: Handle; offset: LongInt; ptr1: Ptr; len1: LongInt;
                ptr2: Ptr; len2: LongInt): LongInt; inline $A9E0;

function  GetIcon(iconID: Integer): Handle; inline $A9BB;
procedure PlotIcon(theRect: Rect; theIcon: Handle); inline $A94B;
function  GetCursor(cursorID: Integer): CursHandle; inline $A9B9;
function  GetPattern(patID: Integer): PatHandle; inline $A9B8;
function  GetPicture(picID: Integer): PicHandle; inline $A9BC;
procedure GetIndPattern(var thePat: Pattern; patListID: Integer;
                       index: Integer); external;

procedure ShieldCursor(shieldRect: Rect; offsetPt: Point); inline $A855;
procedure ScreenRes(var scrnHRes, scrnVRes: Integer); external;

{ for Font Manager }
procedure InitFonts; inline $A8FE;
procedure GetFontName(familyID: Integer; var theName: Str255); inline $A8FF;
procedure GetFNum(theName: Str255; var familyID: Integer); inline $A900;
procedure SetFontLock(lockFlag: Boolean); inline $A903;
function  FMSwapFont(inRec: FMInput): FMOutPtr; inline $A901;
function  RealFont(famID: Integer; size: Integer): Boolean; inline $A902;

{ new 128K ROM }
procedure SetFScaleDisable(scaleDisable:Boolean); inline $A834;
procedure SetFractEnable(fractEnable:Boolean); external;
procedure FontMetrics(var theMetrics:FMetricRec); inline $A835;

{ for Event Manager }
function  EventAvail(mask:Integer; var theEvent: EventRecord):
                       Boolean; inline $A971;
function  GetNextEvent(mask:Integer; var theEvent: EventRecord):
                       Boolean; inline $A970;
function  StillDown:Boolean; inline $A973;
function  WaitMouseUp:Boolean; inline $A977;
procedure GetMouse(var pt: Point); inline $A972;
function  TickCount:LongInt; inline $A975;
```

```
function  Button: Boolean; inline $A974;
procedure GetKeys(var k: keyMap); inline $A976;

function GetDblTime: LongInt; inline $2EB8, $02F0;
function GetCaretTime: LongInt; inline $2EB8, $02F4;

{ for Window Manager }
procedure ClipAbove(window: WindowPeek); inline $A90B;
procedure PaintOne(window: WindowPeek; clobbered: RgnHandle); inline $A90C;
procedure PaintBehind(startWindow: WindowPeek; clobbered: RgnHandle); inline $A90D;
procedure SaveOld(window: WindowPeek); inline $A90E;
procedure DrawNew(window: WindowPeek; fUpdate: Boolean); inline $A90F;
procedure CalcVis(window: WindowPeek); inline $A909;
procedure CalcVisBehind(startWindow: WindowPeek; clobbered: RgnHandle); inline $A90A;

procedure ShowHide(window: WindowPtr; showFlag: Boolean); inline $A908;

function  CheckUpdate(var theEvent: EventRecord): Boolean; inline $A911;
procedure GetWMgrPort(var wPort: GrafPtr); inline $A910;

procedure InitWindows; inline $A912;
function  NewWindow(wStorage: Ptr; boundsRect: Rect; title: Str255;
                    visible: Boolean; theProc: Integer; behind: WindowPtr;
                    goAwayFlag: Boolean; refCon: LongInt): WindowPtr; inline $A913;

procedure DisposeWindow(theWindow: WindowPtr); inline $A914;
procedure CloseWindow(theWindow: WindowPtr); inline $A92D;
procedure MoveWindow(theWindow: WindowPtr; h,v: Integer; BringToFront:
                       Boolean); inline $A91B;
procedure SizeWindow(theWindow: WindowPtr; width,height: Integer;
                       fUpdate: Boolean); inline $A91D;
function  GrowWindow(theWindow: windowPtr; startPt: Point; bBox: Rect):
                       LongInt; inline $A92B;
procedure DragWindow(theWindow: WindowPtr; startPt: Point;
                       boundsRect: Rect); inline $A925;
procedure ShowWindow(theWindow: WindowPtr); inline $A915;
procedure HideWindow(theWindow: WindowPtr); inline $A916;
procedure SetWTitle(theWindow: WindowPtr; title: Str255); inline $A91A;
procedure GetWTitle(theWindow: WindowPtr; var title: Str255); inline $A919;
procedure HiliteWindow(theWindow: WindowPtr; fHiLite: Boolean); inline $A91C;
procedure BeginUpdate(theWindow: WindowPtr); inline $A922;
procedure EndUpdate(theWindow: WindowPtr); inline $A923;
procedure SetWRefCon(theWindow: WindowPtr; data: LongInt); inline $A918;
function  GetWRefCon(theWindow: WindowPtr): LongInt; inline $A917;
procedure SetWindowPic(theWindow: WindowPtr; thePic: PicHandle); inline $A92E;
function  GetWindowPic(theWindow: WindowPtr): PicHandle; inline $A92F;
procedure BringToFront(theWindow: WindowPtr); inline $A920;
procedure SendBehind(theWindow,behindWindow: WindowPtr); inline $A921;
function  FrontWindow: WindowPtr; inline $A924;
procedure SelectWindow(theWindow: WindowPtr); inline $A91F;
function  TrackGoAway(theWindow: WindowPtr; thePt: Point): Boolean; inline $A91E;
procedure DrawGrowIcon(theWindow: WindowPtr); inline $A904;

procedure ValidRect(goodRect: Rect); inline $A92A;
procedure ValidRgn(goodRgn: RgnHandle); inline $A929;
procedure InvalRect(badRect: Rect); inline $A928;
procedure InvalRgn(badRgn:  RgnHandle); inline $A927;
function  FindWindow(thePoint: Point;
                  var theWindow: WindowPtr): Integer; inline $A92C;
function  GetNewWindow(windowID: Integer; wStorage: Ptr;
                  behind: WindowPtr): WindowPtr; inline $A9BD;
function  PinRect(theRect: Rect; thePt: Point): LongInt; inline $A94E;
function  DragGrayRgn(theRgn: RgnHandle; startPt: Point; boundsRect,
                      slopRect: Rect; axis: Integer; actionProc: ProcPtr):
                      LongInt; inline $A905;
```

```
{ new 128K ROM }
function TrackBox(theWindow:WindowPtr; thePt:Point;
                  partCode:Integer): Boolean; inline $A83B;

procedure ZoomWindow(theWindow:WindowPtr; partCode: Integer;
                     front: Boolean); inline $A83A;

{ for TextEdit }
procedure TEActivate( h: TEHandle ); inline $A9D8;
procedure TECalText( h: TEHandle );  inline $A9DD;
procedure TEClick( pt: Point; extend: Boolean; h: TEHandle ); inline $A9D4;
procedure TECopy( h: TEHandle ); inline $A9D5;
procedure TECut( h: TEHandle );  inline $A9D6;
procedure TEDeActivate( h: TEHandle ); inline $A9D9;
procedure TEDelete( h: TEHandle ); inline $A9D7;
procedure TEDispose( h: TEHandle ); inline $A9CD;
procedure TEIdle( h: TEHandle ); inline $A9DA;
procedure TEInit; inline $A9CC;
procedure TEKey( key: Char; h: TEHandle ); inline $A9DC;
function  TENew( dest, view: Rect ): TEHandle; inline $A9D2;
procedure TEPaste( h: TEHandle ); inline $A9DB;
procedure TEScroll( dh, dv: Integer; h: TEHandle ); inline $A9DD;
procedure TESetSelect( selStart, selEnd: LongInt; h: TEHandle  ); inline $A9D1;
procedure TESetText( inText: Ptr; textLength: LongInt; h: TEHandle ); inline $A9CF;
procedure TEInsert( inText: Ptr; textLength: LongInt; h: TEHandle ); inline $A9DE;
procedure TEUpdate( rUpdate: Rect; h: TEHandle ); inline $A9D3;
procedure TESetJust( just: Integer; h: TEHandle ); inline $A9DF;
function  TEGetText( h: TEHandle ): CharsHandle; inline $A9CB;

function TEScrapHandle: Handle; inline $2EB8, $0AB4;
function TEGetScrapLen: LongInt; external;
procedure TESetScrapLen(length: LongInt); external;
function TEFromScrap: OsErr; external;
function TEToScrap: OsErr; external;

procedure SetWordBreak(wBrkProc: ProcPtr; hTE: TEHandle); external;
procedure SetClikLoop(clikProc: ProcPtr; hTE: TEHandle); external;

{ new 128K ROM }
procedure TESelView(hTE: TEHandle); inline $A811;
procedure TEPinScroll(dh,dv:Integer; hTE:TEHandle); inline $A812;
procedure TEAutoView(auto:Boolean; hTE:TEHandle); inline $A813;

{ box drawing utility }
procedure TextBox( inText: Ptr; textLength: LongInt;
                        r: Rect; style: Integer ); inline $A9CE;

{ for Resource Manager }
function  InitResources: Integer; inline $A995;
procedure RsrcZoneInit; inline $A996;
procedure CreateResFile(fileName:  Str255); inline $A9B1;
function  OpenResFile(fileName: Str255): Integer; inline $A997;
procedure UseResFile(refNum:  Integer); inline $A998;
function  GetResFileAttrs(refNum: Integer): Integer; inline $A9F6;
procedure SetResFileAttrs(refNum: Integer; attrs: Integer); inline $A9F7;
procedure UpdateResFile(refNum: Integer); inline $A999;
procedure CloseResFile(refNum:  Integer); inline $A99A;
procedure SetResPurge(install:  Boolean); inline $A993;
procedure SetResLoad(AutoLoad:  Boolean); inline $A99B;
function  CountResources(theType: ResType): Integer; inline $A99C;
function  GetIndResource(theType: ResType; index:
                        Integer): Handle; inline $A99D;
function  CountTypes:  Integer; inline $A99E;
procedure GetIndType(var theType: ResType; index: Integer); inline $A99F;
function  UniqueID(theType: ResType): Integer; inline $A9C1;
```

```
function  GetResource(theType: ResType; ID: Integer):
                                  Handle; inline $A9A0;
function  GetNamedResource(theType: ResType; name: Str255):
                                  Handle; inline $A9A1;
procedure LoadResource(theResource: Handle); inline $A9A2;
procedure ReleaseResource(theResource: Handle); inline $A9A3;
procedure DetachResource(theResource: Handle); inline $A992;
procedure ChangedResource(theResource: Handle); inline $A9AA;
procedure WriteResource(theResource: Handle); inline $A9B0;
function  HomeResFile(theResource: Handle): Integer; inline $A9A4;
function  CurResFile: Integer; inline $A994;
function  GetResAttrs(theResource: Handle): Integer; inline $A9A6;
procedure SetResAttrs(theResource: Handle; attrs: Integer);  inline $A9A7;
procedure GetResInfo(theResource:  Handle; var theID: Integer;
             var theType: ResType; var name: Str255); inline $A9A8;
procedure SetResInfo(theResource: Handle; theID: Integer;
                    name: Str255);  inline $A9A9;
procedure AddResource(theResource: Handle; theType: ResType;
                    theID: Integer; name: Str255); inline $A9AB;
procedure RmveResource(theResource: Handle); inline $A9AD;
function SizeResource(theResource: Handle): LongInt; inline $A9A5;
function  ResError: Integer; inline $A9AF;

{ new 128K ROM }
function  Get1IndResource(theType: ResType; index: Integer): Handle; inline $A80E;
function  Count1Types: Integer; inline $A81C;
function  Get1Resource(theType: ResType; theID: Integer): Handle; inline $A81F;
function  Get1NamedResource(theType: ResType; name: Str255): Handle; inline $A820;
procedure Get1IndType (var theType: ResType; index: Integer);  inline $A80F;
function  Unique1ID(theType: ResType): Integer; inline $A810;
function  Count1Resources(theType: ResType): Integer; inline $A80D;

function MaxSizeRsrc(theResource:Handle):LongInt; inline $A821;
function RsrcMapEntry(theResource:Handle):LongInt; inline $A9C5;
function OpenRFPerm(fileName:Str255; VRefNum:Integer; permission: Byte):
                    Integer; inline $A9C4;

{ for Control Manager }
function NewControl(curWindow: windowPtr; boundsRect: Rect; title: Str255;
                    visible: Boolean; value: Integer; min: Integer;
                    max: Integer; contrlProc: Integer; refCon: LongInt):
                    ControlHandle;  inline $A954;
procedure DisposeControl(theControl: ControlHandle); inline $A955;
procedure KillControls(theWindow: WindowPtr); inline $A956;

procedure MoveControl(theControl: ControlHandle; h,v: Integer); inline $A959;
procedure SizeControl(theControl: ControlHandle; w,h: Integer); inline $A95C;
procedure DragControl(theControl: ControlHandle; startPt: Point;
                      bounds: Rect; slopRect: Rect; axis:Integer); inline $A967;
procedure ShowControl(theControl: ControlHandle); inline $A957;
procedure HideControl(theControl: ControlHandle); inline $A958;
procedure SetCTitle(theControl: ControlHandle; title: Str255); inline $A95F;
procedure GetCTitle(theControl: ControlHandle; var title: Str255); inline $A95E;
procedure HiliteControl(theControl: ControlHandle; hiliteState:
                    Integer); inline $A95D;
procedure SetCRefCon(theControl: ControlHandle; data: LongInt); inline $A95B;
function  GetCRefCon(theControl: ControlHandle): LongInt;  inline $A95A;

procedure SetCtlValue(theControl: ControlHandle; theValue: Integer); inline $A963;
function  GetCtlValue(theControl: ControlHandle): Integer;  inline $A960;

function  GetCtlMin(theControl: ControlHandle): Integer; inline $A961;
function  GetCtlMax(theControl: ControlHandle): Integer; inline $A962;
procedure SetCtlMin(theControl: ControlHandle; theValue: Integer); inline $A964;
procedure SetCtlMax(theControl: ControlHandle; theValue: Integer); inline $A965;
```

```
function  GetCtlAction(theControl: ControlHandle): ProcPtr; inline $A96A;
procedure SetCtlAction(theControl: ControlHandle; newProc: ProcPtr); inline $A96B;

function  TestControl(theControl: ControlHandle; thePt: Point):
                      Integer; inline $A966;
function  TrackControl(theControl:ControlHandle; thePt: Point;
                        actionProc:ProcPtr):Integer;  inline $A968;

function  FindControl(thePoint: Point; theWindow: WindowPtr; var theControl:
                       ControlHandle): Integer; inline $A96C;
procedure DrawControls(theWindow: WindowPtr); inline $A969;
function  GetNewControl(controlID: Integer; owner: WindowPtr):
                       ControlHandle; inline $A9BE;

{ new 128K ROM }
procedure UpdtControl(theWindow:WindowPtr; updateRgn:RgnHandle); inline $A953;

{ for Dialog Manager }
procedure InitDialogs(resumeProc: ProcPtr); inline $A97B;
function  GetNewDialog(dialogID: Integer; wStorage: Ptr;
                      behind: WindowPtr): DialogPtr; inline $A97C;
function  NewDialog(wStorage: Ptr; boundsRect: Rect; title: Str255;
                     visible: Boolean; theProc: Integer; behind: WindowPtr;
                     goAwayFlag: Boolean; refCon: LongInt; itmLstHndl: Handle):
                     DialogPtr; inline $A97D;
function  IsDialogEvent(event: EventRecord): Boolean; inline $A97F;
function  DialogSelect( event: EventRecord; var theDialog: DialogPtr;
                        var itemHit: Integer): Boolean; inline $A980;
procedure ModalDialog( filterProc: ProcPtr; var itemHit: Integer); inline $A991;
procedure DrawDialog(theDialog: DialogPtr); inline $A981;
procedure CloseDialog(theDialog: DialogPtr); inline $A982;
procedure DisposDialog(theDialog: DialogPtr); inline $A983;
function  Alert(alertID: Integer; filterProc: ProcPtr): Integer; inline $A985;
function  StopAlert(alertID: Integer; filterProc: ProcPtr): Integer; inline $A986;
function  NoteAlert(alertID: Integer; filterProc: ProcPtr): Integer; inline $A987;
function  CautionAlert(alertID: Integer; filterProc: ProcPtr):
                      Integer; inline $A988;
procedure CouldAlert(alertID: Integer); inline $A989;
procedure FreeAlert(alertID: Integer); inline $A98A;
procedure CouldDialog(DlgID: Integer); inline $A979;
procedure FreeDialog(DlgID: Integer); inline $A97A;
procedure ParamText(cite0, cite1, cite2, cite3:  Str255); inline $A98B;
procedure ErrorSound(sound: ProcPtr); inline $A98C;
procedure GetDItem(theDialog: DialogPtr; itemNo: Integer;
                   var kind: Integer; var item: Handle;
                   var box: Rect); inline $A98D;
procedure SetDItem(dialog: DialogPtr; itemNo: Integer; kind: Integer;
                   item: Handle; box: Rect); inline $A98E;
procedure SetIText(item: Handle; text: Str255); inline $A98F;
procedure GetIText(item: Handle; var text: Str255); inline $A990;
procedure SelIText(theDialog: DialogPtr; itemNo: Integer;
                   startSel, endSel: Integer ); inline $A97E;

{ routines designed only for use in Pascal }
function  GetAlrtStage: Integer; inline $3EB8, $0A9A;
procedure ResetAlrtStage; inline $4278, $0A9A;

procedure DlgCut(theDialog: DialogPtr); external;
procedure DlgPaste(theDialog: DialogPtr); external;
procedure DlgCopy(theDialog: DialogPtr); external;
procedure DlgDelete(theDialog: DialogPtr); external;

procedure SetDAFont(fontNum: Integer); inline $31DF, $0AFA;

{ new 128K ROM }
procedure HideDItem(theDialog:DialogPtr; itemNo:Integer); inline $A827;
```

```
procedure ShowDItem(theDialog:DialogPtr; itemNo:Integer); inline $A828;
procedure UpdtDialog(theDialog:DialogPtr; updateRgn:RgnHandle); inline $A978;
function  FindDItem(theDialog:DialogPtr; thePt:Point):Integer; inline $A984;

{ for Desk Manager }
function  SystemEvent(myEvent: EventRecord): Boolean; inline $A9B2;
procedure SystemClick(theEvent: EventRecord; theWindow: windowPtr); inline $A9B3;
procedure SystemTask; inline $A9B4;
procedure SystemMenu(menuResult: LongInt); inline $A9B5;
function  SystemEdit(editCode: Integer): Boolean; inline $A9C2;
function  OpenDeskAcc(theAcc: Str255): Integer; inline $A9B6;
procedure CloseDeskAcc(refNum: Integer); inline $A9B7;

{ for Menu Manager }
procedure InitMenus; inline $A930;
function  NewMenu(menuID: Integer; menuTitle: Str255): menuHandle; inline $A931;
function  GetMenu(rsrcID: Integer): MenuHandle; inline $A9BF;
procedure DisposeMenu(menu: menuHandle); inline $A932;
procedure AppendMenu(menu: menuHandle; data: str255); inline $A933;

procedure InsertMenu (menu: MenuHandle; beforeId: Integer); inline $A935;
procedure DeleteMenu (menuId: Integer); inline $A936;
procedure DrawMenuBar; inline $A937;
procedure ClearMenuBar; inline $A934;

function  GetMenuBar:Handle; inline $A93B;
function  GetNewMBar(menuBarID:  Integer): Handle; inline $A9C0;
procedure SetMenuBar(menuBar: Handle); inline $A93C;

function  MenuSelect(startPt: Point): LongInt; inline $A93D;
function  MenuKey(ch: CHAR): LongInt; inline $A93E;
procedure HiLiteMenu(menuId: Integer); inline $A938;

procedure SetItem(menu: menuHandle; item: Integer; itemString:
                  Str255); inline $A947;
procedure GetItem(menu: menuHandle; item: Integer;
                 var itemString: Str255); inline $A946;
procedure EnableItem(menu: menuHandle; item: Integer); inline $A939;
procedure DisableItem(menu: menuHandle; item: Integer); inline $A93A;
procedure CheckItem(menu: menuHandle; item: Integer; checked:
                  Boolean); inline $A945;

procedure SetItemIcon(menu: menuHandle; item: Integer; iconNum: Byte); inline $A940;
procedure GetItemIcon(menu: menuHandle; item: Integer;
                      var iconNum: Byte); inline $A93F;
procedure SetItemStyle(menu: menuHandle; item: Integer; styleVal: Style); external;
procedure GetItemStyle(menu: menuHandle; item: Integer;
                       var styleVal: Style); external;
procedure SetItemMark(menu: menuHandle; item: Integer; markChar: CHAR); inline
$A944;
procedure GetItemMark(menu: menuHandle; item: Integer;
                      var markChar: CHAR); inline $A943;
procedure SetMenuFlash(flashCount: Integer); inline $A94A;
procedure FlashMenuBar(menuID: Integer); inline $A94C;

function  GetMHandle(menuID: Integer): menuHandle; inline $A949;
function  CountMItems(menu: menuHandle): Integer; inline $A950;
procedure AddResMenu(menu: menuHandle; theType:ResType); inline $A94D;
procedure InsertResMenu(menu: menuHandle; theType:ResType;
                        afterItem: Integer); inline $A951;
procedure CalcMenuSize(menu:menuHandle); inline $A948;

{ new 128K ROM }
procedure InsMenuItem(theMenu:Menuhandle;
                      itemstring: Str255; afterItem:Integer); inline $A826;
procedure DelMenuItem(theMenu:MenuHandle; item:Integer); inline $A952;
```

```
{ for Scrap Manager }
function  GetScrap( hDest: Handle; what: ResType;
                    var offset: LongInt ): LongInt; inline $A9FD;
function  InfoScrap: pScrapStuff; inline $A9F9;
function  LoadScrap: LongInt; inline $A9FB;
function  PutScrap( length: LongInt; what: ResType;
                    source: Ptr  ): LongInt; inline $A9FE;
function  UnloadScrap: LongInt; inline $A9FA;
function  ZeroScrap: LongInt; inline $A9FC;

{ package manager }
procedure InitAllPacks; inline $A9E6;
procedure InitPack(packID: Integer); inline $A9E5;
```

PackIntf

Packages are sets of data structures and routines that are stored as resources in the SYSTEM file and brought into memory only when needed. They serve as extensions to the Toolbox and Mac OS; the most useful (and most commonly used) is the Standard File Package, which brings up the standard Mac dialog box to open files or select a file name for output. Other packages include Disk Initialization, International Utilities, and Binary-Decimal Conversion.

```
unit PackIntf(10);

interface

uses MemTypes,QuickDraw,OsIntf,ToolIntf;

{ disk initialization package ---------------------------------------- }

procedure DILoad; external;
procedure DIUnLoad; external;
function DIBadMount(where: Point; evtMessage: LongInt): Integer; external;
function DIFormat(drvNum: Integer): OsErr; external;
function DIVerify(drvNum: Integer): OsErr; external;
function DIZero(drvNum: Integer; volName: Str255): OsErr; external;

{ standard file package ---------------------------------------------- }

const

   putDlgID = -3999;     { SFPutFile dialog template ID }

   putSave   = 1;        { save button }
   putCancel = 2;        { cancel button }
   putEject  = 5;        { eject button }
   putDrive  = 6;        { drive button }
   putName   = 7;        { editTExt item for file name }

   getDlgID = -4000;     { SFGetFile dialog template ID }

   getOpen   = 1;        { open button }
   getCancel = 3;        { cancel button }
   getEject  = 5;        { eject button }
   getDrive  = 6;        { drive button }
   getNmList = 7;        { userItem for file name list }
   getScroll = 8;        { userItem for scroll bar }

type

   SFReply = record
               good: Boolean;      { ignore command if False }
               copy: Boolean;      { not used }
               fType: OsType;      { file type or not used }
               vRefNum: Integer;   { volume reference number }
               version: Integer;   { file's version number }
               fName: String[63];  { file name }
             end;                { SFReply }
   SFTypeList = array[0..3] of OSType;

procedure SFPutFile(where: Point; prompt: Str255; origName: Str255;
                    dlgHook: ProcPtr; var reply: SFReply); external;
```

```
procedure SFPPutFile(where: Point; prompt: Str255; origName: Str255;
                     dlgHook: ProcPtr; var reply: SFReply; dlgID: Integer;
                     filterProc: ProcPtr); external;
procedure SFGetFile(where: Point; prompt: Str255; fileFilter: ProcPtr;
                     numTypes: Integer; typeList: SFTypeList; dlgHook: ProcPtr;
                     var reply: SFReply); external;
procedure SFPGetFile(where: Point; prompt: Str255; fileFilter: ProcPtr;
                     numTypes: Integer; typeList: SFTypeList; dlgHook: ProcPtr;
                     var reply: SFReply; dlgID: Integer; filterProc: ProcPtr);
external;

{ international utilities package ---------------------------------- }

const

   { constants for manipulation of international resources }
   { masks used for setting and testing currency format flags }
   currSymLead   =  16;   { set if currency symbol leads, reset if trails }
   currNegSym    =  32; { set if minus sign for negative num, reset if parentheses }
   currTrailingZ =  64;   { set if trailing zero }
   currLeadingZ  = 128;   { set if leading zero }

   { constants specifying absolute value of short date form }
   MDY = 0;              { month,day,year }
   DMY = 1;              { day,month,year }
   YMD = 2;              { year,month,day }

   { masks used for date element format flags }
   dayLdingZ =  32;        { set if leading zero for day }
   mntLdingZ =  64;        { set if leading zero for month }
   century   = 128;        { set if century, reset if no century }

   { masks used for time element format flags }
   secLeadingZ =  32;      { set if leading zero for seconds }
   minLeadingZ =  64;      { set if leading zero for minutes }
   hrLeadingZ  = 128;      { set if leading zero for hours }

   { country codes for version numbers }
   verUS           =  0;
   verFrance       =  1;
   verBritain      =  2;
   verGermany      =  3;
   verItaly        =  4;
   verNetherlands  =  5;
   verBelgiumLux   =  6;
   verSweden       =  7;
   verSpain        =  8;
   verDenmark      =  9;
   verPortugal     = 10;
   verFrCanada     = 11;
   verNorway       = 12;
   verIsrael       = 13;
   verJapan        = 14;
   verAustralia    = 15;
   verArabia       = 16;
   verFinland      = 17;
   verFrSwiss      = 18;
   verGrSwiss      = 19;
   verGreece       = 20;
   verIceland      = 21;
   verMalta        = 22;
   verCyprus       = 23;
   verTurkey       = 24;
   verYugoslavia   = 25;
```

```
type

   intlOHndl = ^intlOPtr;
   intlOPtr = ^intlORec;
   intlORec = packed record
       decimalPt: Char;              { ASCII character for decimal point }
       thousSep: Char;               { ASCII character for thousand separator }
       listSep: Char;                { ASCII character for list separator }
       currSym1: Char;               { ASCII for currency symbol (3 bytes long) }
       currSym2: Char;
       currSym3: Char;
       currFmt: Byte;                { currency format flags }
       dateOrder: Byte;              { short date form - DMY,YMD, or MDY }
       shrtDateFmt: Byte;            { date elements format flag }
       dateSep: Char;                { ASCII for date separator }
       timeCycle: Byte;              { indicates 12 or 24 hr cycle }
       timeFmt: Byte;                { time elements format flags }
       mornStr: packed array[1..4] of Char;
                                     { ASCII for trailing string from 0:00 to 11:59 }
       eveStr: packed array[1..4] of Char;
                                    { ASCII for trailing string from 12:00 to 23:59 }
       timeSep: Char;                { ASCII for the time separator }
       time1Suff: Char;              { suffix string used in 24 hr mode }
       time2Suff: Char;              { 8 characters long }
       time3Suff: Char;
       time4Suff: Char;
       time5Suff: Char;
       time6Suff: Char;
       time7Suff: Char;
       time8Suff: Char;
       metricSys: Byte;              { indicates metric or English system }
       intlOVers: Integer;           { version: high byte = country, low byte = vers }
     end; {intlORec}

   intl1Hndl = ^intl1Ptr;
   intl1Ptr = ^intl1REc;
   intl1Rec = packed record
       days: array[1..7] of String[15];     { length and word for Sun through Mon }
       months: array[1..12] of String[15];  { length and word for Jan to Dec }
       suppressDay: Byte;             { 0 for day of week, 255 for no day of week }
       lngDateFmt: Byte;              { expanded date format 0 or 255 }
       dayLeading0: Byte;             { 255 for leading 0, 0 for no leading 0 }
       abbrLen: Byte;                 { length of abbreviated names in long form }
       st0: packed array[1..4] of Char;     { the string st0 }
       st1: packed array[1..4] of Char;     { the string st1 }
       st2: packed array[1..4] of Char;     { the string st2 }
       st3: packed array[1..4] of Char;     { the string st3 }
       st4: packed array[1..4] of Char;     { the string st4 }
       intl1Vers: Integer;                     { version word }
       localRtn: Integer;                { routine for localizing mag comp; }
                                         { minimal case is $4E75 for RTS, but }
                                         { routine may be longer than one integer }
     end;                                { intl1Rec }


   DateForm = (shortDate, longDate, abbrevDate);


function  IUGetIntl(theID: Integer): Handle; external;
procedure IUSetIntl(refNum: Integer; theID: Integer; intlParam: Handle); external;
procedure IUDateString(dateTime: LongInt; longFlag: DateForm;
                       var result: Str255); external;
procedure IUDatePString(dateTime: LongInt; longFlag: DateForm;
                       var result: Str255; intlParam: Handle); external;
```

```
procedure IUTimeString(dateTime: LongInt; wantSeconds: Boolean;
                       var result: Str255); external;
procedure IUTimePString(dateTime: LongInt; wantSeconds: Boolean;
                       var result: Str255; intlParam: Handle); external;
function  IUMetric: Boolean; external;
function  IUCompString(aStr,bStr: Str255): Integer; external;
function  IUEqualString(aStr,bStr: Str255): Integer; external;
function  IUMagString(aPtr,bPtr: Ptr; aLen,bLen: Integer): Integer; external;
function  IUMagIDString(aPtr,bPtr: Ptr; aLen,bLen: Integer):Integer; external;

{ binary-decimal conversion package ------------------------------- }

procedure StringToNum(theString: Str255; var theNum: LongInt); external;
procedure NumToString(theNum: LongInt; var theString: Str255); external;

{ list manager }
const

  { for list manager }
  { masks for selection flags (selFlags) }
  LOnlyOne     = -128;        { 0 = multiple selections, 1 = one }
  LExtendDrag  =   64;        { 1 = drag select without shift key }
  LNoDisjoint  =   32;        { 1 = turn off selections on click }
  LNoExtend    =   16;        { 1 = don't extend shift selections }
  LNoRect      =    8;        { 1 = don't grow (shift,drag) selection as rect }
  LUseSense    =    4;        { 1 = shift should use sense of start cell }
  LNoNilHilite =    2;        { 1 = don't highlight empty cells }

  { masks for other flags (listFlags) }

  LDoVAutoscroll = 2;         { 1 = allow vertical autoscrolling }
  LDoHAutoscroll = 1;         { 1 = allow horizontal autoscrolling }

  { messages to list definition procedure }

  LInitMsg   = 0;             { tell drawing routines to init themselves }
  LDrawMsg   = 1;             { draw (and de/select) the indicated data }
  LHiliteMsg = 2;             { invert highlight state of specified cell }
  LCloseMsg  = 3;             { shut down, the list is being disposed }

type

  Cell = Point;

  dataArray = packed array [0..32000] of Char;
  dataPtr = ^dataArray;
  dataHandle = ^dataPtr;

  ListPtr = ^ListRec;
  ListHandle = ^ListPtr;
  ListRec =  record
      rView: Rect;                { Rect in which we are viewed }
      port: GrafPtr;              { Grafport that owns us }

      indent: Point;              { Indent pixels in cell }
      cellSize: Point;            { Cell size }

      visible: Rect;              { visible row/column bounds }

      vScroll: ControlHandle;     { vertical scroll bar (or NIL) }
      hScroll: ControlHandle;     { horizontal scroll bar (or NIL) }

      selFlags: SignedByte;       { defines selection characteristics }
      LActive: Boolean;           { active or not }
      LReserved: SignedByte;      { internally used flags }
      listFlags: SignedByte;      { other flags }
```

```
      clikTime: LongInt;            { save time of last click }
      clikLoc: Point;               { save position of last click }
      mouseLoc: Point;              { current mouse position }
      LClikLoop: Ptr;               { routine called repeatedly during ListClick }
      lastClick: Cell;              { the last cell clicked in }

      refCon: LongInt;              { reference value }

      listDefProc: Handle;          { handle to the defProc }
      userHandle: Handle;           { general purpose handle for user }

      dataBounds: Rect;             { total number of rows/columns }
      cells: dataHandle;            { handle to data}

      maxIndex: Integer;            { index past the last element }
      cellArray: array [1..1] of Integer;  { offsets to elements }
  end;

procedure LActivate( act: Boolean; lHandle: ListHandle ); external;
function  LAddColumn( count, colNum: Integer; lHandle: ListHandle ):
                      Integer; external;
function  LAddRow( count, rowNum: Integer; lHandle: ListHandle ):
                    Integer; external;
procedure LAddToCell( dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                      lHandle: ListHandle ); external;
procedure LAutoScroll( lHandle: ListHandle ); external;
procedure LCellSize( cSize: Point; lHandle: ListHandle); external;
function  LClick( pt: Point; modifiers: Integer; lHandle: ListHandle ):
                  Boolean; external;
procedure LClrCell( theCell: Cell; lHandle: ListHandle ); external;
procedure LDelColumn(count, colNum: Integer; lHandle:ListHandle); external;
procedure LDelRow(count, rowNum: Integer; lHandle: ListHandle); external;
procedure LDispose( lHandle: ListHandle ); external;
procedure LDoDraw( drawIt: Boolean; lHandle:ListHandle ); external;
procedure LDraw( theCell: Cell; lHandle: ListHandle ); external;
procedure LFind( var offset, len: Integer; theCell: Cell;
                 lHandle: ListHandle ); external;
procedure LGetCell( dataPtr: Ptr; var dataLen: Integer; theCell: Cell;
                    lHandle: ListHandle ); external;
function  LGetSelect ( next: Boolean; var theCell: Cell;
                       lHandle: ListHandle): Boolean; external;
function  LLastClick ( lHandle: ListHandle ): LongInt; external;
function  LNew( rView, databounds: Rect; cSize: Point; theProc:
                Integer; theWindow: WindowPtr;
                drawIt,hasGrow,scrollHoriz,scrollVert:
                Boolean): ListHandle; external;
function  LNextCell( hNext,vNext: Boolean; var theCell: Cell;
                     lHandle: ListHandle): Boolean; external;
procedure LRect( var cellRect: Rect; theCell: Cell;
                 lHandle: ListHandle ); external;
procedure LScroll( dRows, dCols: Integer; lHandle: ListHandle ); external;
function LSearch(dataPtr: Ptr; dataLen: Integer; SearchProc: Ptr;
                 var theCell: Cell; lHandle: ListHandle): Boolean; external;
procedure LSetCell( dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                    lHandle: ListHandle ); external;
procedure LSetSelect( setIt: Boolean; theCell: Cell;
                      lHandle: ListHandle  ); external;
procedure LSize( listWidth,listHeight: Integer; lHandle: ListHandle); external;
procedure LUpdate( theRgn: RgnHandle; lHandle : ListHandle; external;
```

MacPrint

The *MacPrint* unit provides access to the Macintosh Printing Manager. The Printing Manager is a set of RAM-based data types and routines that allow you to use standard *QuickDraw* routines to print text or graphics on a printer. These provide a device-independent interface to printer drivers, which enable you to print on a specific device (ImageWriter, LaserWriter, and other printers). One (or more) of these printer drivers — usually found in the SYSTEM folder — must be available in order to use this package.

```
unit MacPrint(-11);

interface

uses MemTypes,QuickDraw,OSIntf,ToolIntf;

const
   iPrPgFract  = 120;        { Page scale factor; }
                             { ptPgSize (below) is in units of 1/iPrPgFract }

   iPrPgFst = 1;             { page range constants }
   iPrPgMax = 9999;

   iPrRelease = 3;           { current version number of the code. }
                             { DC 7/23/84 }
   iPfMaxPgs  = 128;         { max number of pages in a print file. }

   { driver constants }
   iPrBitsCtl = 4;           { Bitmap print proc's ctl number }
   lScreenBits= $00000000;   { Bitmap print proc's screen bitmap param }
   lPaintBits = $00000001;   { Bitmap print proc's paint [sq pix] param }
   lHiScreenBits= $00000010; { Bitmap print proc's screen Bitmap param }
   lHiPaintBits = $00000011; { Bitmap print proc's paint [sq pix] param }
   iPrIOCtl   = 5;           { raw byte IO proc's ctl number }
   iPrEvtCtl  = 6;           { PrEvent proc's ctl number }
   lPrEvtAll  = $0002FFFD;   { PrEvent proc's CParam for the entire screen }
   lPrEvtTop  = $0001FFFD;   { PrEvent proc's CParam for the top folder }
   iPrDevCtl  = 7;           { PrDevCtl proc's ctl number }
   lPrReset   = $00010000;   { PrDevCtl proc's CParam for reset }
   lPrPageEnd = $00020000;   { PrDevCtl proc's CParam for end page }
   lPrLineFeed= $00030000;   { PrDevCtl proc's CParam for paper advance }
   lPrLFSixth = $0003FFFF;   { PrDevCtl proc's CParam for 1/6th inch paper advance }
   lPrLFEighth= $0003FFFE;  { PrDevCtl proc's CParam for 1/8th inch paper advance }
   iFMgrCtl   = 8;           { FMgr's Tail-hook Proc's ctl number }
                             { [The Pre-Hook is the status call] }

   { error constants: }
   iMemFullErr = -108;
   iPrAbort    = 128;
   iIOAbort    = -27;

   { PrVars lo mem area: }
   pPrGlobals  = $00000944;
   bDraftLoop  = 0;
   bSpoolLoop  = 1;
   bUser1Loop  = 2;
   bUser2Loop  = 3;

   { currently supported printers: }
   bDevCItoh = 1;  iDevCItoh = $0100;  { CItoh }
```

```
   bDevDaisy = 2;  iDevDaisy = $0200;  { Daisy }
   bDevLaser = 3;  iDevLaser = $0300;  { Laser }

type

   TPRect   =  ^Rect;              { A Rect Ptr }
   TPBitMap =  ^BitMap;            { A BitMap Ptr }

   { NOTE: Changes will also affect: PrEqu, TCiVars & TPfVars }
   TPrVars = record              { 4 longs for printing; }
                                 { see SysEqu for location }
      iPrErr:     Integer;       { current print error; }
                                 { set to iPrAbort to abort printing }
      bDocLoop:   SignedByte;    { Doc style: Draft, Spool, .., and ..}
                                 { currently use low 2 bits; }
                                 { the upper 6 are for flags }
      bUser1:     SignedByte;    { spares used by the print code }
      lUser1:     LongInt;
      lUser2:     LongInt;
      lUser3:     LongInt;
   end;
   TPPrVars = ^TPrVars;

   TPrInfo = record           { print info record: }
                              { the parameters needed for page composition }
      iDev:       Integer;    { font mgr/QuickDraw device code }
      iVRes:      Integer;    { resolution of device, in device coordinates }
      iHRes:      Integer;    { ..note: V before H => compatible with point }
      rPage:      Rect;       { page (printable) rectangle in device coordinates }
   end;
   TPPrInfo = ^TPrInfo;

   { types of paper feeders }
   TFeed = ( feedCut, feedFanfold, feedMechCut, feedOther );

   TPrSt1 = record       { printer style: the printer configuration }
                         { and usage information }
      wDev:   Integer;   { device (driver) number: }
                         { Hi byte=RefNum, Lo byte=variant }
                         { f0 = fHiRes, f1 = fPortrait, f2 = fSqPix, }
                         { f3 = f2xZoom, f4 = fScroll }
      iPageV: Integer;   { paper size in units of 1/iPrPgFract }
      iPageH: Integer;   { NOTE: V before H => compatible with Point }
      bPort:  SignedByte; { IO port number. Refnum? }
      feed:   TFeed;     { paper feeder type }
   end;
   TPPrSt1 = ^TPrSt1;

   { Banding data structures.  Not of general interest to Apps. }
   TScan =                { band scan direction: Top-Bottom, Left-Right, etc. }
      ( scanTB,  scanBT, scanLR, scanRL );

   TPrXInfo = record          { print time extra information }
      iRowBytes:  Integer;      { band's rowBytes }
      iBandV:     Integer;      { size of band, in device coordinates }
      iBandH:     Integer;      { NOTE: V before H => compatible with Point }
      iDevBytes:  Integer;      { size for allocation; may be more than rBounds! }
      iBands:     Integer;      { number of bands per page }

      bPatScale:  SignedByte;   { pattern scaling }
      bULThick:   SignedByte;   { three underscoring parameters }
      bULOffset:  SignedByte;
      bULShadow:  SignedByte;

      scan:       TScan;        { band scan direction }
      bXInfoX:    SignedByte;   { an extra byte }
```

```
   end;
   TPPrXInfo = ^TPrXInfo;

   TPrJob = record                { print job: }
                                    { Print "form" for single print request }
      iFstPage:   Integer;          { page range }
      iLstPage:   Integer;
      iCopies:    Integer;          { number copies }
      bJDocLoop:  SignedByte;       { doc style: Draft, Spool, .., and .. }
      fFromUsr:   Boolean;          { printing from an user's app (not PrApp) flag }
      pIdleProc: ProcPtr;           { Proc called while waiting on IO, etc. }
      pFileName:  StringPtr;        { spool file name: NIL for default. }
      iFileVol:   Integer;          { spool file vol: set to 0 initially }
      bFileVers:  SignedByte;       { spool file version: set to 0 initially }
      bJobX:      SignedByte;       { an extra byte }
   end;
   TPPrJob = ^TPrJob;

   TPrint = record                { The universal 120 byte printing record }
      iPrVersion: Integer;          {  2 }  { printing software version }
      PrInfo:     TPrInfo;          { 14 }  { PrInfo data associated with the current
style }
      rPaper:     Rect;             {  8 }  { paper rectangle [offset from rPage] }
      PrStl:      TPrStl;           {  8 }  { print request's style }
      PrInfoPT:   TPrInfo;          { 14 }  { print time imaging metrics }
      PrXInfo:    TPrXInfo;         { 16 }  { print time (expanded) print info record }
      PrJob:      TPrJob;           { 20 }  { print job request }
                                    { 82 }  { total of above: 120 - 82 = 38 bytes
needed to fill 120 }
      PrintX:     array[1..19] of Integer; { spare to fill to 120 bytes }
   end;
   TPPrint = ^TPrint;
   THPrint = ^TPPrint;

   { Printing Graf Port. All printer imaging, whether spooling, banding, etc.,
   happens "thru" a GrafPort }
   TPrPort =  record               { this is the "PrPeek" record }
      GPort:      GrafPort;          { the Printer's graf port }
      GProcs:     QDProcs;           { ..and its procs }

      lGParam1:   LongInt;           { 16 bytes for private parameter storage }
      lGParam2:   LongInt;
      lGParam3:   LongInt;
      lGParam4:   LongInt;

      fOurPtr:    Boolean;           { whether the PrPort allocation was done by us }
      fOurBits:   Boolean;           { whether the BitMap allocation was done by us }
   end;
   TPPrPort = ^TPrPort;

   TPrStatus = record    { print status: print information during printing }
      iTotPages:  Integer;           { total pages in print file }
      iCurPage:   Integer;           { current page number }
      iTotCopies: Integer;           { total copies requested }
      iCurCopy:   Integer;           { current copy number }
      iTotBands:  Integer;           { total bands per page }
      iCurBand:   Integer;           { current band number }
      fPgDirty:   Boolean;           { True if current page has been written to }
      fImaging:   Boolean;           { set while in band's DrawPic call }
      hPrint:     THPrint;           { handle to the active printer record }
      pPrPort:    TPPrPort;          { Ptr to the active PrPort }
      hPic:       PicHandle;         { handle to the active picture }
   end;
   TPPrStatus = ^TPrStatus;

   { PicFile = a TPfHeader followed by n QuickDraw Pics (whose PicSize is invalid!)}
```

```
   TPfPgDir = record
      iPages:     Integer;
      lPgPos:     array[0..iPfMaxPgs] of LongInt;
   end;
   TPPfPgDir = ^TPfPgDir;
   THPfPgDir = ^TPPfPgDir;

   TPfHeader = record              { print file header }
      Print: TPrint;
      PfPgDir: TPfPgDir;
   end;

   TPPfHeader = ^TPfHeader;
   THPfHeader = ^TPPfHeader;        { NOTE: Type compatible with an hPrint }

   { This is the Printing Dialog Record. Only used by folks appending their own
   dialogs. }
   TPrDlg = record              { Print Dialog: The dialog stream object }
      Dlg:        DialogRecord;  { the dialog window }
      pFltrProc:  ProcPtr;       { the filter proc. }
      pItemProc:  ProcPtr;       { the item evaluating proc. }
      hPrintUsr:  THPrint;       { the user's print record }
      fDoIt:      Boolean;
      fDone:      Boolean;
      lUser1:     LongInt;       { 4 longs for users to hang global data }
      lUser2:     LongInt;
      lUser3:     LongInt;
      lUser4:     LongInt;
   {  ...Plus more stuff needed by the particular printing dialog }
   end;
   TPPrDlg = ^TPrDlg;      { == a dialog ptr }

{  --init--  }
procedure PrOpen; external;
   { Open the .Print driver, get the Current Printer's Rsrc file }
   { name from SysRes, open the resource file, and open the .Print
   { driver living in SysRes. PrOpen MUST be called during init time. }
procedure PrClose; external;
   { Closes JUST the print rsrc file.  Leaves the .Print driver in SysRes open. }

{  --Print Dialogs & Default--  }
procedure PrintDefault  ( hPrint: THPrint ); external;
   { defaults a handle to a default print record. }
   { NOTE: You allocate (or fetch from file's resources..) the handle, }
   { I fill it. Also, I may invoke this at odd times whenever I think }
   { you have an invalid Print record! }

function  PrValidate ( hPrint: THPrint ): Boolean; external;
   { Checks the hPrint.  Fixes it if there has been a change in }
   { SW version or in the current printer. Returns fChanged. }
   { NOTE: Also updates the various parameters within the Print }
   { record to match the current values of the PrStl & PrJob. }
   { It does NOT set the fChanged result if these parameters }
   { changed as a result of this update. }

function  PrStlDialog  ( hPrint: THPrint ): Boolean; external;
function  PrJobDialog  ( hPrint: THPrint ): Boolean; external;
   { The dialog returns the fDoIt flag:
         if PrJobDialog(..) then begin
            PrintMyDoc (..);
            SaveMyStl (..)
         end
               or
         if PrStlDialog(..) then SaveMyStl (..)
```

```
    { NOTE: These may change the hPrint^^ if the version number
            is old or the printer is not the current one.  }

procedure PrJobMerge (hPrintSrc, hPrintDst: THPrint); external;

   { Merges hPrintSrc's PrJob into hPrintDst [Source/Destination].
     Allows one job dialog being applied to several docs [Finder printing] }

{  --The Document printing procs: These spool a print file.--  }

function PrOpenDoc ( hPrint:        THPrint;
                     pPrPort:         TPPrPort;
                     pIOBuf: Ptr ): TPPrPort; external;

   { Set up a graf port for Pic streaming and make it the current }
   { port. Init the print file page directory. Create and open the }
   { print file. hPrint: The print info. pPrPort: the storage to }
   { use for the TPrPort.  If nil we allocate. pIOBuf: an IO buf; }
   { if nil, file sys uses volume buf. returns TPPrPort: The TPPrPort }
   { (graf port) used to spool thru. }

procedure PrCloseDoc ( pPrPort: TPPrPort ); external;
   { Write the print file page directory. Close the print file. }

procedure PrOpenPage ( pPrPort: TPPrPort; pPageFrame: TPRect ); external;

   { If current page is in the range of printed pages: Open a picture }
   { for the page. Otherwise set a null port for absorbing an image. }
   { pPageFrame := PrInfo.rPage, unless you're scaling. Set pPageFrame }
   { to nil unless you want to perform PicScaling on the printer. }
   { [The printing procs will call OpenPicture (pPageFrame^) and }
   { DrawPicture (hPic, rPage); ] NOTE: Use of QuickDraw may now }
   { cause File I/O errors due to our Pic spooling! }

procedure PrClosePage( pPrPort: TPPrPort ); external;
   { Close & kill the page picture. Update the file page directory. }
   { If we allocated the TPrPort then de-allocate it. }

{  --The "Printing Application" proc: Read and band the spooled PicFile.--  }

procedure PrPicFile( hPrint:        THPrint;
                     pPrPort:         TPPrPort;
                     pIOBuf:          Ptr;
                     pDevBuf:         Ptr;
              var PrStatus:       TPrStatus ); external;
   { Read and print the spooled print file. }
   { The idle proc is run during imaging and printing. }

{  --Get/Set the current Print Error--  }
function  PrError: Integer; external;
procedure PrSetError ( iErr: Integer ); external;

{  --The .Print driver calls.--  }
procedure PrDrvrOpen; external;
procedure PrDrvrClose; external;
   { Open/Close the .Print driver in SysRes. Make it purgable or not. }
   { ONLY used by folks doing low-level stuff, not full document printing. }

procedure PrCtlCall (iWhichCtl: Integer; lParam1, lParam2, lParam3: LongInt);
```

```
external;
   { A generalized Control proc for the Printer driver. }
   { The main use is for bitmap printing:
   PrCtlCall (iPrBitsCtl, pBitMap, pPortRect, lControl);
         ==
         procedure PrBits  (  pBitMap:   Ptr;       --QuickDraw bitmap
                              pPortRect: TPRect;      --a portrect.
                                                      * use bounds for whole bitmap *
                              lControl:  LongInt );  --0=>Screen resolution/Portrait
     This dumps a bitmap/portrect to the printer.
     lControl is a device dep param; use 0 for screen res/portrait/etc.
     Each different printer will use lControl parameter differently.
     Thus PrCtlCall (iPrBitsCtl, @MyPort^.ScreenBits, @MyPort^.PortRect.Bounds,0)
     performs a screen dump of just my port's data.

     Two special control calls are included in the driver for screen
     printing from the key board:
           PrCtlCall (iPrEvtCtl, lPrEvtAll, 0, 0);  Prints the screen
           PrCtlCall (iPrEvtCtl, lPrEvtTop, 0, 0);  Prints the top folder
     These are handled by the system for keyboard access but can be
     called by anyone at any time.  They can be very cheap printing for
     ornaments, for example!

     Another useful call is used for sending raw data to the printer:
         PrCtlCall (iPrIOCtl, pBuf, lBufCount, pIdleProc); }

{ --Semiprivate stuff-- }
function  PrStlInit ( hPrint: THPrint ): TPPrDlg; external;
function  PrJobInit ( hPrint: THPrint ): TPPrDlg; external;
function  PrDlgMain ( hPrint: THPrint; pDlgInit: ProcPtr ): Boolean; external;

procedure PrCfgDialog; external;
```

FixMath

The *FixMath* unit is a collection of types and functions that implement fixed-point real numbers. This unit is very useful for applications that require real numbers but don't need the accuracy of floating-point math. Fixed-point operations run much faster than regular floating point, so you can choose precision over increased speed.

```
unit FixMath(-12);

{ These calls support three types of fixed point numbers, each 32 bits long. }
{ The bits are interpreted as shown.  The '-' represents the sign bit. }

 Type    <---------Integer Portion--------> <-------Fractional Portion------>
 LongInt -xxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx.
 Fixed                            -xxxxxxx xxxxxxxx.xxxxxxxx xxxxxxxx
 Fract                                 -x.xxxxxxxx xxxxxxxx xxxxxxxx xxxxxx

{ Type LongInt can represent integers between +/-2147483647.  Type Fixed can }
{ represent fractional quantities between +/-32768, to about 5 digits of }
{ accuracy. Type Fract can represent fractional quantities between +/-2 to }
{ about 9 digits of accuracy.  These numeric representations are useful for }
{ applications that do not require the accuracy of the floating-point routines }
{ and need to run as fast as possible.  The Graf3D three-dimensional }
{ graphics package resides on top of these routines. Although FixMul is in the }
{ file ToolTraps, it is listed below to show how it handles different types. }
{ Additional fixed point routines are described in the Inside Macintosh chapter, }
{ "Toolbox Utilities" }

interface

uses MemTypes;

type Fract = LongInt;

{ These routines are only available on a system with a 128K ROM: }

function Long2Fix(x:longint):Fixed;   inline $A83F;
function Fix2Long(x:fixed):LongInt;   inline $A840;
function Fix2Frac(x:fixed):Fract;     inline $A841;
function Frac2Fix(x:fract):Fixed;     inline $A842;

{ Functions to convert between fixed-point types }

function Fix2X(x:fixed):Extended;     inline $A843;
function X2Fix(x:extended):Fixed;     inline $A844;
function Frac2X(x:fract):Extended;    inline $A845;
function X2Frac(x:extended):Fract;    inline $A846;

{ Functions to convert between fixed, fract, and the extended }
{ floating-point type }

function FixAtan2(x,y:LongInt):Fixed; inline $A818;
{ FixATan2 returns the arctangent of y / x.  Note that FixATan2 effects
  "arctan(type / type) --> Fixed":
               arctan(LongInt / LongInt) --> Fixed
               arctan(Fixed   / Fixed  ) --> Fixed
               arctan(Fract   / Fract  ) --> Fixed }
```

```
{ The following routines are supplied as glue code: }
function FracMul(x, y: Fract): Fract; external;
 { FracMul returns x * y.
   Note that FracMul effects "type * Fract --> type":
        Fract   * Fract         --> Fract
        LongInt * Fract         --> LongInt
        Fract   * LongInt       --> LongInt
        Fixed   * Fract         --> Fixed
        Fract   * Fixed         --> Fixed }

function FixDiv(x, y: Fixed): Fixed;  external;
 { FixDiv returns x / y.
   Note that FixDiv effects "type / type --> Fixed":
        Fixed / Fixed           --> Fixed
        LongInt / LongInt       --> Fixed
        Fract / Fract           --> Fixed
        LongInt / Fixed         --> LongInt
        Fract / Fixed           --> Fract }

function FracDiv(x, y: Fract): Fract; external;
 { FracDiv returns x / y.  Note that FracDiv effects "type / type --> Fract":
        Fract / Fract           --> Fract
        LongInt / LongInt       --> Fract
        Fixed / Fixed           --> Fract
        LongInt / Fract         --> LongInt
        Fixed / Fract           --> Fixed }

function FracSqrt(x: Fract): Fract;   external;
 { FracSqrt returns the square root of x.  Both argument and result }
 { are regarded as unsigned }

function FracCos(x: Fixed): Fract;    external;
function FracSin(x: Fixed): Fract;    external;
 { FracCos and FracSin return the cosine and sine, respectively, }
 { given the argument x in radians }
```

Graf3D

Graf3D is a RAM-based, three-dimensional graphics package that sits on top of *QuickDraw*. It implements 3-D *GrafPorts* and provides a complete set of 3-D operations, including rotation, translation, scaling, and clipping.

```
unit Graf3D(-13);

interface

uses MemTypes,QuickDraw,FixMath;

const radConst = 3754936; {radConst = 57.295780}

type Point3D=record
              x: fixed;
              y: fixed;
              z: fixed;
             end;

     Point2D=record
              x: fixed;
              y: fixed;
             end;

     XfMatrix  = array[0..3,0..3] of fixed;
     Port3DPtr = ^Port3D;
     Port3D    = record
                   GrPort:                  GrafPtr;
                   viewRect:                Rect;
                   xLeft,yTop,xRight,yBottom: fixed;
                   pen,penPrime,eye:        Point3D;
                   hSize,vSize:             fixed;
                   hCenter,vCenter:         fixed;
                   xCotan,yCotan:           fixed;
                   ident:                   Boolean;
                   xForm:                   XfMatrix;
                 end;


var thePort3D: Port3DPtr;

procedure InitGrf3D   (globalPtr: Ptr); external;
procedure Open3DPort  (port: Port3DPtr); external;
procedure SetPort3D   (port: Port3DPtr); external;
procedure GetPort3D   (var port: Port3DPtr); external;

procedure MoveTo2D    (x,y: fixed); external;
procedure MoveTo3D    (x,y,z: fixed); external;
procedure LineTo2D    (x,y: fixed); external;
procedure LineTo3D    (x,y,z: fixed); external;
procedure Move2D      (dx,dy: fixed); external;
procedure Move3D      (dx,dy,dz: fixed); external;
procedure Line2D      (dx,dy: fixed); external;
procedure Line3D      (dx,dy,dz: fixed); external;

procedure ViewPort    (r: Rect); external;
procedure LookAt      (left,top,right,bottom: fixed); external;
procedure ViewAngle   (angle: fixed); external;
procedure Identity;   external;
procedure Scale       (xFactor,yFactor,zFactor: fixed); external;
procedure Translate   (dx,dy,dz: fixed); external;
```

```
procedure Pitch       (xAngle: fixed); external;
procedure Yaw           (yAngle: fixed); external;
procedure Roll        (zAngle: fixed); external;
procedure Skew        (zAngle: fixed); external;
procedure TransForm   (src: Point3D; var dst: Point3D); external;
function Clip3D (src1,src2: Point3D; var dst1,dst2: Point): Boolean;
external;

procedure SetPt3D     (var pt3D: Point3D; x,y,z: fixed); external;
procedure SetPt2D     (var pt2D: Point2D; x,y: fixed); external;
```

AppleTalk

AppleTalk is the Macintosh local-area network; that is, the means by which you connect a group of Macs with printers, disks, other devices, and each other. The *AppleTalk Manager* is used to communicate with devices connected to an *AppleTalk* network. See Chapter 7 for more details on using this unit.

```
unit AppleTalk(-14);

interface

uses Memtypes,QuickDraw,OSIntf;

const

  lapSize = 20;
  ddpSize = 26;
  nbpSize = 26;
  atpSize = 56;

  { error codes }

  ddpSktErr     = -91;
  ddpLenErr     = -92;
  noBridgeErr   = -93;
  LAPProtErr    = -94;
  excessCollsns = -95;

  nbpBuffOvr    = -1024;
  nbpNoConfirm  = -1025;
  nbpConfDiff   = -1026;
  nbpDuplicate  = -1027;
  nbpNotFound   = -1028;
  nbpNISErr     = -1029;

  reqFailed     = -1096;
  tooManyReqs   = -1097;
  tooManySkts   = -1098;
  badATPSkt     = -1099;
  badBuffNum    = -1100;
  noRelErr      = -1101;
  cbNotFound    = -1102;
  noSendResp    = -1103;
  noDataArea    = -1104;
  reqAborted    = -1105;

  buf2SmallErr  = -3101;
  noMPPErr      = -3102;
  ckSumErr      = -3103;
  extractErr    = -3104;
  readQErr      = -3105;
  atpLenErr     = -3106;
  atpBadRsp     = -3107;
  recNotFnd     = -3108;
  sktClosedErr  = -3109;

type

  ABByte = 1..127;

  STR32 = STRING[32];
```

```
ABCallType = (tLAPRead,tLAPWrite,tDDPRead,tDDPWrite,tNBPLookUp,
              tNBPConfirm,tNBPRegister,tATPSndRequest,tATPGetRequest,
              tATPSdRsp,tATPAddRsp,tATPRequest,tATPResponse);

ABProtoType = (lapProto,ddpProto,nbpProto,atpProto);

LAPAdrBlock = packed record
                dstNodeID : Byte;
                srcNodeID : Byte;
                LAPProtType : ABByte;
              end;

AddrBlock = packed record
              aNet : Integer;
              aNode : Byte;
              aSocket : Byte;
             end;

EntityName = record
               objStr : Str32;
               typeStr : Str32;
               zoneStr : Str32;
             end;

EntityPtr = ^EntityName;

RetransType = packed record
                retransInterval : Byte;
                retransCount : Byte;
              end;

BitMapType = packed array [0..7] of Boolean;

BDSElement = record
               BuffSize : Integer;
               BuffPtr : Ptr;
               DataSize : Integer;
               UserBytes : LongInt;
             end;

BDSType = array [0..7] of BDSElement;

BDSPtr = ^BDSType;

ABusRecord = record
               abOpCode : abCallType;
               abResult : Integer;
               abUserReference : LongInt;

               case abProtoType of
                 lapProto:
                   (lapAddress : LAPAdrBlock;
                    lapReqCount : Integer;
                    lapActCount : Integer;
                    lapDataPtr : Ptr;
                    );

                 ddpProto:
                   (ddpType : Byte;
                    ddpSocket : Byte;
                    ddpAddress : AddrBlock;
                    ddpReqCount : Integer;
                    ddpActCount : Integer;
                    ddpDataPtr : Ptr;
                    ddpNodeID : Byte;
                    );
```

```
                    nbpProto:
                      (nbpEntityPtr : EntityPtr;
                       nbpBufPtr : Ptr;
                       nbpBufSize : Integer;
                       nbpDataField : Integer;
                       nbpAddress : AddrBlock;
                       nbpRetransmitInfo : RetransType;
                       );

                    atpProto:
                      (atpSocket : Byte;
                       atpAddress : AddrBlock;
                       atpReqCount : Integer;
                       atpDataPtr : Ptr;
                       atpRspBDSPtr : BDSPtr;
                       atpBitMap : BitMapType;
                       atpTransID : Integer;
                       atpActCount : Integer;
                       atpUserData : LongInt;
                       atpXO : Boolean;
                       atpEOM : Boolean;
                       atpTimeOut : Byte;
                       atpRetries : Byte;
                       atpNumBufs : Byte;
                       atpNumRsp : Byte;
                       atpBDSSize : Byte;
                       atpRspUData : LongInt;
                       atpRspBuf : Ptr;
                       atpRspSize : Integer;
                       );

                end; { record }

  ABRecPtr = ^ABusRecord;
  ABRecHandle = ^ABRecPtr;

function LAPOpenProtocol(theLAPType : ABByte; protoPtr : Ptr) : OSErr; external;
function LAPCloseProtocol(theLAPType : ABByte) : OSErr; external;
function LAPRead(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function LAPWrite(abRecord : ABRecHandle; async : Boolean) : OSErr; external;

function LAPRdCancel(abRecord : ABRecHandle) : OSErr; external;

function DDPOpenSocket(var theSocket : Byte; sktListener : Ptr) : OSErr; external;
function DDPCloseSocket(theSocket : Byte) : OSErr; external;
function DDPRead(abRecord : ABRecHandle; retCksumErrs : Boolean;
function          async : Boolean) : OSErr; external;
function DDPWrite(abRecord : ABRecHandle; doCheckSum : Boolean;
                  async : Boolean) : OSErr; external;

function DDPRdCancel(abRecord : ABRecHandle) : OSErr; external;
function
function NBPLoad : OSErr; external;
function NBPUnLoad : OSErr; external;
function NBPLookUp(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function NBPConfirm(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function NBPRegister(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function NBPRemove(entityName : EntityPtr) : OSErr; external
function NBPExtract(theBuffer : Ptr; numInBuf : Integer; whichOne :Integer;
                    var abEntity : EntityName;
                    var address : AddrBlock) : OSErr; external;

function ATPLoad : OSErr; external;
function ATPUnLoad : OSErr; external;
```

```
function ATPOpenSocket(addrRcvd : AddrBlock; var atpSocket : Byte) : OSErr; external;

function ATPCloseSocket(atpSocket : Byte) : OSErr; external;
function ATPSndRequest(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function ATPGetRequest(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function ATPSndRsp(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function ATPAddRsp(abRecord : ABRecHandle) : OSErr; external;

function ATPRequest(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function ATPResponse(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function ATPReqCancel(abRecord : ABRecHandle; async : Boolean) : OSErr; external;
function ATPRspCancel(abRecord : ABRecHandle; async : Boolean) : OSErr; external;

procedure RemoveHdlBlocks; external;

{ RemoveHdlBlks is a routine that is called automatically at the beginning }
{ of every Pascal call.  It checks for free (disposable) memory blocks that }
{ the interface has allocated and disposes of them.  The memory blocks have }
{ been allocated by the NewHandle call.  Most of these memory blocks are }
{ small (on the order of 20-50 bytes). The user has the option to }
{ make the call whenever s/he wants to.  The general rule is that one memory }
{ block will be allocated every time a network call is made; and it will }
{ not be free until the call completes. }

function GetNodeAddress(var myNode,myNet : Integer) : OSErr; external;

function MPPOpen : OSErr; external;
function MPPClose : OSErr; external;

function IsMPPOpen : Boolean; external;
function IsATPOpen : Boolean; external;
```

SpeechIntf

The *SpeechIntf* unit provides an interface to *MacinTalk*, a speech synthesizer that runs under Mac OS as a driver. In real time, *MacinTalk* converts an ASCII string of phonetic codes into synthetic speech. *MacinTalk* uses a special program, READER, to convert English text into the phonetic codes used by *MacinTalk*. See Chapter 7 for more details.

```
unit SpeechIntf(-15);

interface

uses    MemTypes;

const
  noExcpsFile  = '';              { signals reader to use only basic rules  }
  noReader = 'noReader';          { signals SpeechOn to NOT bring in reader }
  fullUnitT    = -4000;           { error code for driver unit table full   }

type
  SpeechErr     = Integer;
  SpeechRecord  = array [0..99] of Byte;  { Driver parm block, used internally }
  SpeechPointer = ^SpeechRecord;              { pointer to driver parm block      }
  SpeechHandle  = ^SpeechPointer;             { handle to driver parm block       }

  Sex      = (Male, Female);
  FOMode   = (Natural, Robotic, NoChange);
  Language = (xEnglish, French, Spanish, German, Italian);

  VoiceRecord = record
                  theSex:       Sex;
                  theLanguage:  Language;
                  theRate:      Integer;
                  thePitch:     Integer;
                  theMode:      FOMode;
                  theName:      Str255;
                  refCon:       LongInt;
                end;
  VoicePtr = ^VoiceRecord;

function  SpeechOn      (ExcpsFile: Str255;
                         var theSpeech: SpeechHandle): SpeechErr; external;

procedure SpeechOff     (theSpeech: SpeechHandle); external;

procedure SpeechRate    (theSpeech: SpeechHandle; theRate: Integer); external;

procedure SpeechPitch   (theSpeech: SpeechHandle; thePitch: Integer;
                         theMode: FOMode); external;

procedure SpeechSex     (theSpeech: SpeechHandle; theSex: Sex); external;
                          { reserved for future implementation }

function Reader         (theSpeech: SpeechHandle; EnglishInput: Ptr;
                          InputLength: LongInt; PhoneticOutput: Handle):
                          SpeechErr; external;

function MacinTalk      (theSpeech: SpeechHandle; Phonemes: Handle):
                          SpeechErr; external;
```

SCSIIntf

The *SCSIIntf* unit provides access to the Small Computer Standard Interface (SCSI) port found on several models of the Macintosh. It allows you to determine what devices are connected to the SCSI port and to communicate with them.

```
unit SCSIIntf(-16);

interface

uses MemTypes,Quickdraw,OSIntf;

const

  { transfer instruction operation codes }
  scInc    =       1;      { SCINC instruction    }
  scNoInc  =       2;      { SCNOINC instruction  }
  scAdd    =       3;      { SCADD instruction    }
  scMove   =       4;      { SCMOVE instruction   }
  scLoop   =       5;      { SCLOOP instruction   }
  scNOP    =       6;      { SCNOP instruction    }
  scStop   =       7;      { SCSTOP instruction   }
  scComp   =       8;      { SCCOMP instruction   }

  { SCSI manager result codes }
  scBadParmsErr   =       4;      { unrecognized instruction      }
  scCommErr       =       2;      { breakdown in SCSI protocols   }
  scCompareErr    =       6;      { data comparison error in read }
  scPhaseErr      =       5;      { phase error                   }

type

  SCSIInstr = record
                scOpcode:   Integer; { operation code   }
                scParam1:   LongInt; { first parameter  }
                scParam2:   LongInt; { second parameter }
              end;

function SCSIReset:OSErr; external;
function SCSIGet:OSErr; external;
function SCSISelect(targetID: Integer):OSErr; external;
function SCSICmd(buffer:Ptr; count:Integer):OSErr; external;
function SCSIRead(tibPtr:Ptr):OSErr; external;
function SCSIRBlind(tibPtr:Ptr):OSErr; external;
function SCSIWrite(tibPtr:Ptr):OSErr; external;
function SCSIWBlind(tibPtr:Ptr):OSErr; external;
function SCSIComplete(var stat, message:Integer;
                      wait:LongInt):OSErr; external;
function SCSIStat:Integer; external;
```

A P P E N D I X **E**

Macintosh Character Set

Table E-1 shows the decimal and hexadecimal representations of the Macintosh characters. Note, however, that other fonts may produce characters different than those shown in the table.

Table E-1 *The Macintosh Character Set*

DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	NUL	13	0D	CR
1	01	SOH	14	0E	SO
2	02	STX	15	0F	SI
3	03	ETX	16	10	DLE
4	04	EOT	17	11	DC1
5	05	ENQ	18	12	DC2
6	06	ACK	19	13	DC3
7	07	BEL	20	14	DC4
8	08	BS	21	15	NAK
9	09	HT	22	16	SYN
10	0A	LF	23	17	ETB
11	0B	VT	24	18	CAN
12	0C	FF	25	19	EM

Table E-1 The Macintosh Character Set, continued

DEC	HEX	CHAR	DEC	HEX	CHAR
26	1A	SUB	58	3A	:
27	1B	ESC	59	3B	;
28	1C	FS	60	3C	<
29	1D	GS	61	3D	=
30	1E	RS	62	3E	>
31	1F	US	63	3F	?
32	20	SP	64	40	@
33	21	!	65	41	A
34	22	"	66	42	B
35	23	#	67	43	C
36	24	$	68	44	D
37	25	%	69	45	E
38	26	&	70	46	F
39	27	'	71	47	G
40	28	(	72	48	H
41	29	)	73	49	I
42	2A	*	74	4A	J
43	2B	+	75	4B	K
44	2C	,	76	4C	L
45	2D	-	77	4D	M
46	2E	.	78	4E	N
47	2F	/	79	4F	O
48	30	0	80	50	P
49	31	1	81	51	Q
50	32	2	82	52	R
51	33	3	83	53	S
52	34	4	84	54	T
53	35	5	85	55	U
54	36	6	86	56	V
55	37	7	87	57	W
56	38	8	88	58	X
57	39	9	89	59	Y

Table E-1 The Macintosh Character Set, continued

DEC	HEX	CHAR	DEC	HEX	CHAR
90	5A	Z	122	7A	z
91	5B	[	123	7B	{
92	5C	\	124	7C	|
93	5D	]	125	7D	}
94	5E	^	126	7E	~
95	5F	_	127	7F	DEL
96	60	`	128	80	Ä
97	61	a	129	81	Å
98	62	b	130	82	Ç
99	63	c	131	83	É
100	64	d	132	84	Ñ
101	65	e	133	85	Ö
102	66	f	134	86	Ü
103	67	g	135	87	á
104	68	h	136	88	à
105	69	i	137	89	â
106	6A	j	138	8A	ä
107	6B	k	139	8B	ã
108	6C	l	140	8C	å
109	6D	m	141	8D	ç
110	6E	n	142	8E	é
111	6F	o	143	8F	è
112	70	p	144	90	ê
113	71	q	145	91	ë
114	72	r	146	92	í
115	73	s	147	93	ì
116	74	t	148	94	î
117	75	u	149	95	ï
118	76	v	150	96	ñ
119	77	w	151	97	ó
120	78	x	152	98	ò
121	79	y	153	99	ô

*Table E-1 The Macintosh Character Set, **continued***

DEC	HEX	CHAR	DEC	HEX	CHAR
154	9A	ö	186	BA	∫
155	9B	õ	187	BB	ª
156	9C	ú	188	BC	º
157	9D	ù	189	BD	Ω
158	9E	û	190	BE	æ
159	9F	ü	191	BF	ø
160	A0	†	192	C0	¿
161	A1	°	193	C1	¡
162	A2	¢	194	C2	¬
163	A3	£	195	C3	√
164	A4	§	196	C4	*ƒ*
165	A5	•	197	C5	≈
166	A6	¶	198	C6	Δ
167	A7	ß	199	C7	«
168	A8	®	200	C8	»
169	A9	©	201	C9	…
170	AA	™	202	CA	
171	AB	´	203	CB	À
172	AC	¨	204	CC	Ã
173	AD	≠	205	CD	Õ
174	AE	Æ	206	CE	Œ
175	AF	Ø	207	CF	œ
176	B0	∞	208	D0	–
177	B1	±	209	D1	—
178	B2	≤	210	D2	“
179	B3	≥	211	D3	”
180	B4	¥	212	D4	‘
181	B5	µ	213	D5	’
182	B6	∂	214	D6	÷
183	B7	Σ	215	D7	◊
184	B8	Π	216	D8	ÿ
185	B9	π	217	D9	Ÿ

Table E-1 The Macintosh Character Set, continued

DEC	HEX	CHAR	DEC	HEX	CHAR
218	DA	/	237	ED	Ì
219	DB	¤	238	EE	Ó
220	DC	‹	239	EF	Ô
221	DD	›	240	F0	
222	DE	fi	241	F1	Ò
223	DF	fl	242	F2	Ú
224	E0	‡	243	F3	Û
225	E1	·	244	F4	Ù
226	E2	‚	245	F5	ı
227	E3	„	246	F6	ˆ
228	E4	‰	247	F7	˜
229	E5	Â	248	F8	¯
230	E6	Ê	249	F9	˘
231	E7	Á	250	FA	˙
232	E8	Ë	251	FB	˚
233	E9	È	252	FC	¸
234	EA	Í	253	FD	˝
235	EB	Î	254	FE	˛
236	EC	Ï	255	FF	ˇ

APPENDIX F

Turtlegraphics: Mac Graphics Made Easier

If you want to produce graphics on your Macintosh but don't want to take the time to learn the *QuickDraw* routines, Turtlegraphics is the answer. It's an easy graphics program, called *Turtle*, that's included as part of Turbo Pascal.

Turbo Pascal Turtlegraphics is based on a concept devised by S. Papert and his group at the Massachusetts Institute of Technology. To get around the concept of cartesian coordinates, Papert and his colleagues invented the idea of a "turtle" that could "walk" a given distance and turn at specified angles, drawing a line as it walked along. The simple algorithms in this program can create images that are more interesting than those created by algorithms of the same length in cartesian coordinates.

Like the other graphics routines on the Macintosh, *Turtle* operates in the active window. Turtlegraphics, ordinary graphics and even the Turbo Pascal standard text output can be used simultaneously, and can share a common window.

The Turbo Pascal Turtlegraphics routines operate on *turtle coordinates*. The turtle's *Home* position (0,0) in this coordinate system is always in the middle of the active window; positive values stretch to the right (X) and upwards (Y), and negative values stretch to the left (X) and downwards (Y):

The range of coordinates is based on the size of the screen. For a Macintosh and Macintosh+, these are the values: X =0..511 Y = 0..341. However, the actual range is limited to the size of the active window. Coordinates outside the active window are legal, but are ignored. This means that drawings are clipped to the limits of the active window (unless wrap is on).

Following are the 16 Turtlegraphics procedures you can use to create figures.

Back

Syntax: *Back(Dist);*

Moves the turtle backward by the distance given by the integer expression Dist. The turtle moves from its current position in the direction opposite to its current heading and draws a line in the current pen color. If *Dist* is negative, the turtle moves forward.

Clear

Syntax: *Clear;*

Clears the active window and moves the turtle to the *Home* position.

Forwd

Syntax: *Forwd(Dist);*

Moves the turtle forward by the distance given by the integer expression *Dist.* The turtle moves from its current position in the direction that it faces and draws a line in the current pen color. If *Dist* is negative, the turtle moves backwards.

Heading

Syntax: *Heading;*

Returns an integer in the range 0..359 that gives the direction in which the turtle is currently pointing. 0 is upwards, and increasing angles represent headings in a clockwise direction.

Home

Syntax: *Home;*

Puts the turtle at its *Home* position (coordinates 0,0, the middle of the active window) and points it in heading 0 (upwards).

NoWrap

Syntax: *Nowrap;*

Disables the turtle from "wrapping"; that is, reappearing at the opposite side of the active window if it is moved past the window boundary. *NoWrap* is the system's initial value.

PenDown

Syntax: *PenDown;*

Sets the "pen" to the screen so that the turtle draws a line as it moves. This is the initial status of the pen.

PenUp

Syntax: *PenUp;*

Lifts the pen so the turtle moves without drawing a line.

SetHeading

Syntax: *SetHeading(Angle);*

Turns the turtle to the angle specified by the integer expression *Angle*. 0 is upwards, and increasing angles represent clockwise rotation. If *Angle* is not in the range 0..359, it is converted into a number in that range.

Four integer constants are predefined to turn the turtle in the four main directions: *North* = 0 (up), *East* = 90 (right), *South* = 180 (down), and *West* = 270 (left).

SetPosition

Syntax: *SetPosition(X,Y);*

Moves the turtle, without drawing a line, to the location with the coordinates given by the integer expressions *X* and *Y*.

TurnLeft

Syntax: *TurnLeft(Angle);*

Turns the turtle *Angle* degrees from its current direction. Positive angles turn the turtle to the left; negative angles turn it to the right.

TurnRight

Syntax: *TurnRight(Angle);*

Turns the turtle *Angle* degrees from its current direction. Positive angles turn the turtle to the right; negative angles turn it to the left.

When the window is set, the turtle is initialized to its *Home* position and heading north (or 0 degrees).

TurtleDelay

Syntax: *TurtleDelay(Ms);*

Sets a delay in milliseconds between each step of the turtle. Normally, there is no delay.

Wrap

Syntax: *Wrap;*

Makes the turtle reappear at the opposite side of the active window when the turtle exceeds the window boundary. Use *NoWrap* to return to normal.

Xcor

Syntax: *Xcor;*

Returns the integer value of the turtle's current *X* coordinate.

Ycor

Syntax: *Ycor;*

Returns the integer value of the turtle's current *Y* coordinate.

Mac versus IBM Turtlegraphics

There are some differences between Mac Turtlegraphics and IBM Turtlegraphics. The *Home* position (0,0) in Mac turtle graphics is the upper left-hand corner of the window. In IBM turtle graphics, the home position (0,0)is the center of the window. The *ClearScreen* procedure is called *Clear*. The following procedures are not supported:

- *HideTurtle*
- *ShowTurtle*
- *TurtleWindow*
- *TurtleThere*
- *SetPenColor*

The turtle itself is not supported, only its actions.

An Example

Type in the following program. It draws a circle using the *TurnRight* and *Forwd* procedures.

```
program TurtleDraw;
uses MemTypes,Quickdraw,OSIntf,ToolIntf,Turtle;
var
  Angle : Integer;
begin
  PenDown;                                { start drawing }
  for Angle := 0 to 89 do   { for loop to draw circle }
  begin
    TurnRight(4);                  { turn right n degrees }
    Forwd (5);                 { draw 5 pixel line segment }
  end;
  ReadLn;                        {wait for carriage return }
end.                                      { end of TurtleDraw }
```

Glossary

active window: The front-most window on the desktop; the window where the next action specified will take place. The title bar of the active window is highlighted.

Alarm Clock: A desk accessory that displays the current date and time.

Apple menu: The menu on the far left in the menu bar, indicated by an apple symbol.

ASCII: American Standard Code for Information Interchange. ASCII is a standard code for representing characters (letters, numbers, and symbols) as binary numbers.

benchmark: A point of reference used to measure the performance of hardware and/or software.

bitmap: A grid of bits that makes up your Macintosh screen.

buttons: The squares in dialog boxes that you click on to assign, confirm, or cancel an action. See also *mouse button.*

cancel button: A button that appears in dialog boxes. Clicking this button cancels the command.

check box: The small box or circle associated with an option in a dialog box that, when clicked on, adds or removes the option.

click: To position the pointer on something, then press and release the mouse button.

Clipboard: The file that holds what you last cut or copied.

close: To put away a window and call up the icon that represents it.

Close box: The small blank box on the far left side of the title bar of an active window. Clicking on the Close box puts away the window.

⌘ (command key): A key that, when held down while another key is pressed or a mouse action is performed, causes the corresponding command to take effect.

compiler: A program that takes high-level instructions (source code) and converts them into machine code that the computer can read.

Control Panel: A desk accessory that lets you change the speaker volume, start the AppleTalk connection, create a RAM cache, and set other preferences.

cut: To remove something by selecting it and choosing Cut from the Edit menu. What you cut is placed into the Clipboard file.

data fork The part of a file with information that is retrieved via the File Manager.

declare: State a variable's attributes.

desk accessories: Small applications that are available on the desktop from the Apple menu regardless of which application you're using. Turbo Pascal lets you create your own desk accessories.

desktop: The Macintosh working environment: the menu bar and the gray area on the screen.

dialog box: A box that contains a message and requests information from you. Sometimes the message is a warning that you're asking your Macintosh Plus to do something it can't do or that you're about to destroy some of your information. In these cases the message is often accompanied by a beep.

double-click: To position the pointer where you want an action to take place, then press and release the mouse button twice (without moving the mouse).

drag: To position the pointer on something, press and hold the mouse button, move the mouse to a new position, and release the mouse button. This action is used to select several items at once, or to move a file into another file or to a different location on the screen.

folder: A file containing documents, applications, or other folders on the desktop. Folders allow you to organize information under specific headings.

font: A complete set of characters in one typeface. Common fonts include Courier, Helvetica, and Times. Each font family can come in different weights and styles, such as bold and italic.

heap: A portion of memory used by Turbo Pascal (and other compilers) to store pointer variables during program execution. A heap's memory is organized like a stack; that is, from the bottom up.

Hierarchical File System (HFS): A method of using folders to organize documents, applications, and other folders on a disk to keep together related information. Folders (analogous to subdirectories in DOS systems) can be nested inside other folders to create as many levels hierarchy as you need. Opening a folder displays only the information you've put in that folder.

I-beam cursor: A type of pointer used to enter and edit text.

icon: A graphic representation of an object, a concept, or a message. Following are the Turbo Pascal and Mac icons.


Turbo Pascal compiler icon


Icon for Turbo Pascal program compiled to disk


Turbo Pascal source file icon


Program compile-time error icon


Run-time error icon


FONT/DA MOVER icon


RMAKER (resource compiler) icon


UNITMOVER icon


Compiled desk-accessory icon


Compiled unit icon

initialize: To prepare a disk to receive information.

I/O: Input/output. To enter information into a computer is to input; to move the information out (usually to a printer or a plotter) is to output.

JSR: Jump to Subroutine.

machine code: Instructions to the computer in binary code (that is, 0s and 1s); also known as assembly language. A compiler, like Turbo Pascal, takes your high-level instructions and translates them into machine code.

MDS: Macintosh Development System.

menu: A list of commands that appears when you point to and press the menu title in the menu bar. Dragging through the menu and releasing the mouse button while a specific command is highlighted causes that command to be implemented.

menu bar: The horizontal strip at the top of the screen that contains the menu titles.

menu title: A word, phrase, or symbol in the menu bar that designates a menu. Clicking on the menu title causes the title to be highlighted and its menu to appear below it.

mouse: A small device, attached to your computer, that you roll around on a flat surface. The pointer or cursor on the screen echoes the movements of the mouse.

mouse button: The button on the top of the mouse. Generally, pressing and releasing the mouse button initiates some action on whatever the pointer is on, and releasing the button confirms the action.

Option key: A key (like the Shift key) that gives an alternate meaning to the key you press while keeping the Option key pressed down. You use it to type foreign characters or special symbols.

package: A set of data structures and routines stored as resources in the SYSTEM file and brought into memory only as needed.

pointer cursor: A small shape on the screen, most often an arrow pointing up and to the left, that echoes the movement of the mouse.

queue: Line.

resource: A piece of information, stored in a resource file, that can be accessed by its type and ID.

resource fork The part of a file with information used by an application, as well as the application code itself.

run-time error: An error that occurs while a program is executing.

SANE: Standard Apple Numeric Environment library.

scroll: To move a document or directory in its window so that you can see a different part of it. You can also scroll the directory in some dialog boxes.

scroll arrow: An arrow on either end of a scroll bar. Clicking a scroll arrow moves the document or directory one line. Keeping the mouse button pressed on the scroll arrow scrolls the document continuously.

SCSI: Small Computer System Interface, an industry-standard interface for connecting computers with peripheral devices (hard disks, printers, and so on). The Macintosh Plus includes an SCSI port.

select: To designate where the next action will take place. To select, you click or drag across information.

shift-click: A technique that allows you to extend or shorten a selection by holding down the Shift key while you select (or de-select) something related to the current selection. **syntax:** The rules of a programming language that specify how the language symbols can be put together to form meaningful statements. If a program violates the language syntax rules, a syntax error occurs.

title bar: The horizontal bar at the top of a window that shows the name of the window's contents and lets you move the window.

two's-complement arithmetic: A way of representing negative and positive integers, where an integer is negative if its high bit is set.

window: The area that displays information on the desktop. You can open or close a window, move it around on the desktop, and sometimes change its size, edit its contents, and scroll through it.

Index

Borland Software

4585 Scotts Valley Drive, Scotts Valley, CA 95066

Available at better dealers nationwide.
To order by credit card, call (800) 255-8008; CA (800) 742-1133.

SIDEKICK®

Whether you're running WordStar,® Lotus,® dBASE,® or any other program, SideKick puts all these desktop accessories at your fingertips—Instantly!

A full-screen WordStar-like Editor to jot down notes and edit files up to 25 pages long.

A Phone Directory for names, addresses, and telephone numbers. Finding a name or a number is a snap.

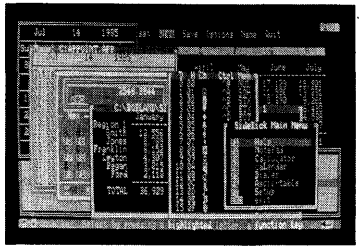
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar from 1901 through 2099.

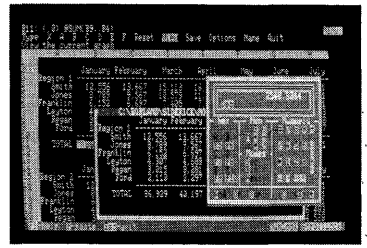
Appointment Calendar to remind you of important meetings and appointments.

A full-featured Calculator ideal for business use. It also performs decimal to hexadecimal to binary conversions.

An ASCII Table for easy reference.



All the SideKick windows stacked up over Lotus 1-2-3.* From bottom to top: SideKick's "Menu Window," ASCII Table, Notepad, Calculator, Appointment Calendar, Monthly Calendar, and Phone Dialer.



Here's SideKick running over Lotus 1-2-3. In the SideKick Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.

The Critics' Choice

"In a simple, beautiful implementation of WordStar's block copy commands, SideKick can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."
—Charles Petzold, PC MAGAZINE

"SideKick deserves a place in every PC."
—Gary Ray, PC WEEK

"SideKick is by far the best we've seen. It is also the least expensive."
—Ron Mansfield, ENTREPRENEUR

"If you use a PC, get SideKick. You'll soon become dependent on it."
—Jerry Pournelle, BYTE

Suggested Retail Price: $84.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr and true compatibles. The IBM PCjr will only accept the SideKick not copy-protected versions. PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive. A Hayes-compatible modem, IBM PCjr internal modem, or AT&T Modem 4000 is required for the autodialer function.

BORLAND INTERNATIONAL

BOR 0060B

SIDEKICK

for the Mac™

SideKick for the Mac brings information management, desktop organization, and telecommunications to your Macintosh.™ Instantly, while running any other program!

A full-screen editor/mini-word processor lets you jot down notes and create or edit files. Your files can also be used by your favorite word processing program, like MacWrite™ or Microsoft® Word.

A complete telecommunications program sends or receives information from any on-line network or electronic bulletin board while using any of your favorite application programs (modem required).

A full-featured financial and scientific calculator sends a paper-tape output to your screen or printer and comes complete with function keys for financial modeling purposes.

A print spooler prints *any* "text only" file while you run other programs.

A versatile calendar lets you view your appointments for a day, a week, or an entire month. You can easily print out your schedule for quick reference.

A convenient "Things-to-Do" file reminds you of important tasks.

A convenient alarm system alerts you to daily engagements.

PhoneLink™ allows you to autodial any phone number, as well as access any long-distance carrier.

A phone log keeps a complete record of all your telephone activities. It even computes the cost of every call.

Area code look-up provides instant access to the state, region, and time zone for all area codes.

An expense account file records your business and travel expenses.

A credit card file keeps track of your credit card balances and credit limits.

A report generator prints out your mailing list labels, phone directory, and weekly calendar in convenient sizes.

A convenient analog clock with a sweeping second-hand can be displayed anywhere on your screen.

On-line help is available for all of the powerful SideKick features.

Best of all, everything runs concurrently

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: 128K RAM and one disk drive. Two disk drives are recommended if you wish to use other application programs. HFS compatible.

BOR 0069C

Traveling SIDEKICK®

The Organizer For The Computer Age!

Traveling SideKick is *BinderWare*,™ both a binder you take with you when you travel and a software program—which includes a Report Generator—that *generates* and *prints out* all the information you'll need to take with you.

Information like your phone list, your client list, your address book, your calendar, and your appointments. The appointment or calendar files you're already using in your SideKick® can automatically be used by your Traveling SideKick. You don't waste time and effort reentering information that's already there.

One keystroke prints out a form like your address book. No need to change printer paper; you simply punch three holes, fold and clip the form into your Traveling SideKick binder, and you're on your way. Because Traveling SideKick is CAD (Computer-Age Designed), you don't fool around with low-tech tools like scissors, tape, or staples. And because Traveling SideKick is electronic, it works this year, next year, and all the "next years" after that. Old-fashioned daytime organizers are history in 365 days.

What's inside Traveling SideKick

What the software program and its Report Generator do for you before you go—and when you get back

Before you go:

- Prints out your calendar, appointments, addresses, phone directory, and whatever other information you need from your data files

When you return:

- Lets you quickly and easily enter all the new names you obtained while you were away into your SideKick data files

It can also:

- Sort your address book by contact, zip code or company name
- Print mailing labels
- Print information selectively
- Search files for existing addresses or calendar engagements

Suggested Retail Price: $69.95

Minimum system configuration: IBM PC, XT, AT, Portable, PCjr, 3270 and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 256K RAM mimimum.

***Special introductory offer**

BORLAND INTERNATIONAL

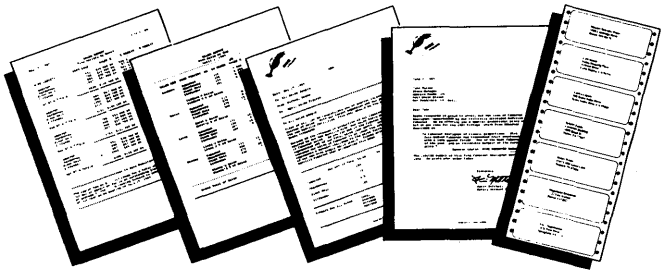
REFLEX:® THE DATABASE MANAGER

The high-performance database manager that's so advanced it's easy to use!

Lets you organize, analyze and report information faster than ever before! If you manage mailing lists, customer files, or even your company's budgets—Reflex is the database manager for you!

Reflex is the acclaimed, high-performance database manager you've been waiting for. Reflex extends database management with business graphics. Because a picture is often worth a 1000 words, Reflex lets you extract critical information buried in mountains of data. With Reflex, when you look, you see.

The ***REPORT VIEW*** allows you to generate everything from mailing labels to sophisticated reports. You can use database files created with Reflex or transferred from Lotus 1-2-3,® dBASE,® PFS: File,® and other applications.



Reflex: the critics' choice

". . . if you use a PC, you should know about Reflex . . . may be the best bargain in software today."
Jerry Pournelle, BYTE

"Everyone agrees that Reflex is the best-looking database they've ever seen."
Adam B. Green, InfoWorld

"The next generation of software has officially arrived." ***Peter Norton, PC Week***

Reflex: don't use your PC without it!

Join hundreds of thousands of enthusiastic Reflex users and experience the power and ease of use of Borland's award-winning Reflex.

Suggested Retail Price $149.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, and true compatibles. 384K RAM minimum. IBM Color Graphics Adapter, Hercules Monochrome Graphics CArd, or equivalent. PC-DOS (MS-DOS) 2.0 or greater. Hard disk and mouse optional. Lotus 1-2-3, dBASE, or PFS: File optional.

BORLAND INTERNATIONAL

REFLEX WORKSHOP™

Includes 22 "instant templates" covering a broad range of business applications (listed below). Also shows you how to customize databases, graphs, crosstabs, and reports. It's an invaluable analytical tool and an important addition to another one of our best sellers, Reflex: The Analyst 1.1.

Fast-start tutorial examples:

Learn Reflex® as you work with practical business applications. The Reflex Workshop Disk supplies databases and reports large enough to illustrate the power and variety of Reflex features. Instructions in each Reflex Workshop chapter take you through a step-by-step analysis of sample data. You then follow simple steps to adapt the files to your own needs.

22 practical business applications:

Workshop's 22 "instant templates" give you a wide range of analytical tools:

Administration

- Scheduling Appointments
- Planning Conference Facilities
- Managing a Project
- Creating a Mailing System
- Managing Employment Applications

Sales and Marketing

- Researching Store Check Inventory
- Tracking Sales Leads
- Summarizing Sales Trends
- Analyzing Trends

Production and Operations

- Summarizing Repair Turnaround
- Tracking Manufacturing Quality Assurance
- Analyzing Product Costs

Accounting and Financial Planning

- Tracking Petty Cash
- Entering Purchase Orders
- Organizing Outgoing Purchase Orders
- Analyzing Accounts Receivable
- Maintaining Letters of Credit
- Reporting Business Expenses
- Managing Debits and Credits
- Examining Leased Inventory Trends
- Tracking Fixed Assets
- Planning Commercial Real Estate Investment

Whether you're a newcomer learning Reflex basics or an experienced "power user" looking for tips, Reflex Workshop will help you quickly become an expert database analyst.

Minimum system configuration: IBM PC, AT, and XT, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 384K RAM minimum. Requires Reflex: The Analyst, and IBM Color Graphics Adapter, Hercules Monochrome Graphics Card or equivalent.

BORLAND INTERNATIONAL

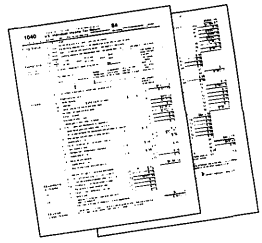
Suggested Retail Price: $69.95 (not copy protected)

REFLEX FOR THE MAC™

The easy-to-use relational database that thinks like a spreadsheet. Reflex for the Mac lets you crunch numbers by entering formulas and link databases by drawing on-screen lines.

5 free ready-to-use templates are included on the examples disk:

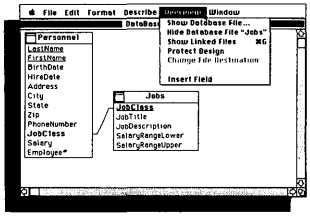
- A sample 1040 tax application with Schedule A, Schedule B, and Schedule D, each contained in a separate report document.
- A portfolio analysis application with linked databases of stock purchases, sales, and dividend payments.
- A checkbook application.



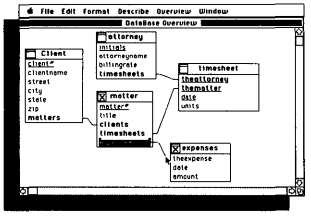
- A client billing application set up for a law office, but easily customized by any professional who bills time.
- A parts explosion application that breaks down an object into its component parts for cost analysis.

Reflex for the Mac accomplishes all of these tasks without programming—using spreadsheet-like formulas. Some other Reflex for the Mac features are:

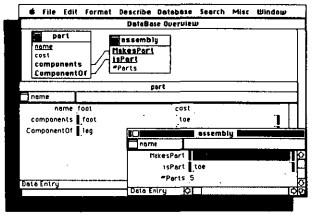
- Visual database design.
- "What you see is what you get" report and form layout with pictures.
- Automatic restructuring of database files when data types are changed, or fields are added and deleted.
- Display formats which include General, Decimal, Scientific, Dollars, Percent.
- Data types which include variable length text, number, integer, automatically incremented sequence number, date, time, and logical.
- Up to 255 fields per record.
- Up to 16 files simultaneously open.
- Up to 16 Mac fonts and styles are selectable for individual fields and labels.



After opening the "Overview" window, you draw link lines between databases directly onto your Macintosh screen.



The link lines you draw establish both visual and electronic relationships between your databases.



You can have multiple windows open simultaneously to view all members of a linked set—which are interactive and truly relational.

Critic's Choice

". . . a powerful relational database . . . uses a visual approach to information management." ***InfoWorld***

". . . gives you a lot of freedom in report design; you can even import graphics." ***A+ Magazine***

". . . bridges the gap between the pretty programs and the power programs." **Stewart Alsop, *PC Letter***

BORLAND INTERNATIONAL

*Suggested Retail Price: $99.95**

*Introductory offer through 1-15-87

Minimum System Requirements: *512K*

BOR 0149

SuperKey®

Increased Productivity for Anyone Using IBM®PCs or Compatibles

SuperKey turns 1,000 keystrokes into 1!

Yes, SuperKey can *record* lengthy keystroke sequences and play them back at the touch of a single key. Instantly. Like magic.

Say, for example, you want to add a column of figures in 1-2-3.® Without SuperKey, you'd have to type 5 keystrokes just to get started: *@ s u m (* . With SuperKey, you can turn those 5 keystrokes into 1.

SuperKey keeps your confidential files—CONFIDENTIAL!

Time after time you've experienced it: anyone can walk up to your PC and read your confidential files (tax returns, business plans, customer lists, personal letters, etc.).

With SuperKey you can encrypt any file, even while running another program. As long as you keep the password secret, only YOU can decode your file correctly. SuperKey also implements the U.S. government Data Encryption Standard (DES).

SuperKey helps protect your capital investment

SuperKey, at your convenience, will make your screen go blank after a predetermined time of screen/keyboard inactivity. You've paid hard-earned money for your PC. SuperKey will protect your monitor's precious phosphor and your investment.

SuperKey protects your work from intruders while you take a break

Now you can lock your keyboard at any time. Prevent anyone from changing hours of work. Type in your secret password and everything comes back to life—just as you left it.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive.

BORLAND INTERNATIONAL

If you use an IBM® PC, you need

TURBO Lightning®

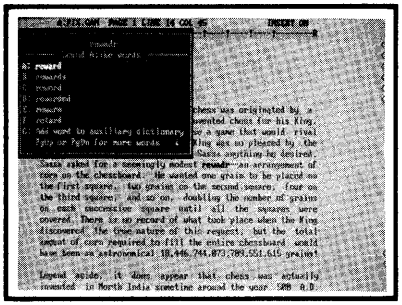
Turbo Lightning teams up with the Random House® Concise Dictionary to check your spelling as you type!

Turbo Lightning, using the 83,000-word Random House Dictionary, checks your spelling *as you type*. If you misspell a word, it alerts you with a beep. At the touch of a key, Turbo Lightning opens a window on top of your application program and suggests the correct spelling. Just press one key and the misspelled word is instantly replaced with the correct word. It's that easy!

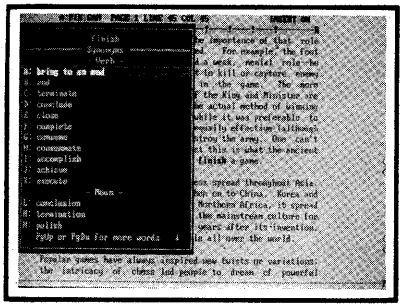
Turbo Lightning works hand-in-hand with the Random House Thesaurus to give you instant access to synonyms

Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once Turbo Lightning opens the Thesaurus window, you see a list of alternate words, organized by parts of speech. You just select the word you want, press ENTER and your new word will instantly replace the original word. Pure magic!

If you ever write a word, think a word, or say a word, you need Turbo Lightning



The Turbo Lightning Dictionary



The Turbo Lightning Thesaurus

Turbo Lightning's intelligence lets you teach it new words. The more you use Turbo Lightning, the smarter it gets

You can also *teach* your new Turbo Lightning your name, business associates' names, street names, addresses, correct capitalizations, and any specialized words you use frequently. Teach Turbo Lightning once, and it knows forever.

Turbo Lightning is the engine that powers Borland's Turbo Lightning Library™

Turbo Lightning brings electronic power to the Random House Dictionary and Random House Thesaurus. They're at your fingertips—even while you're running other programs. Turbo Lightning will also "drive" soon-to-be-released encyclopedias, extended thesauruses, specialized dictionaries, and many other popular reference works. You get a head start with this first volume in the Turbo Lightning Library.

And because Turbo Lightning is a Borland product, you know you can rely on our quality, our 60-day money-back guarantee, and our eminently fair prices.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles with 2 floppy disk drives. PC-DOS (MS-DOS) 2.0 or greater. 256K RAM. Hard disk recommended.

BORLAND INTERNATIONAL

TURBO PASCAL®

VERSION 3.0 with 8087 support and BCD reals

Free MicroCalc Spreadsheet With Commented Source Code!

FEATURES:

One-Step Compile: No hunting & fishing expeditions! Turbo finds the errors, takes you to them, lets you correct them, and instantly recompiles. You're off and running in record time.

Built-in Interactive Editor: WordStar®-like easy editing lets you debug quickly.

Automatic Overlays: Fits big programs into small amounts of memory.

MicroCalc: A sample spreadsheet on your disk with ready-to-compile source code.

IBM® PC Version: Supports Turtle Graphics, color, sound, full tree directories, window routines, input/output redirection, and much more.

THE CRITICS' CHOICE:

"Language deal of the century . . . Turbo Pascal: it introduces a new programming environment and runs like magic."
—Jeff Duntemann, PC Magazine

"Most Pascal compilers barely fit on a disk, but Turbo Pascal packs an editor, compiler, linker, and run-time library into just 39K bytes of random access memory."
—Dave Garland, Popular Computing

"What I think the computer industry is headed for: well-documented, standard, plenty of good features, and a reasonable price."
—Jerry Pournelle, BYTE

LOOK AT TURBO NOW!

- ☑ More than 500,000 users worldwide.
- ☑ Turbo Pascal is the de facto industry standard.
- ☑ Turbo Pascal wins PC MAGAZINE'S award for technical excellence.
- ☑ Turbo Pascal named "Most Significant Product of the Year" by PC WEEK.
- ☑ Turbo Pascal 3.0—the fastest Pascal development environment on the planet, period.

Suggested Retail Price: $99.95; CP/M®-80 version without 8087 and BCD: $69.95

Features for 16-bit Systems: 8087 math co-processor support for intensive calculations. Binary Coded Decimals (BCD): eliminates round-off error! A *must* for any serious business application.

Minimum system configuration: 128K RAM minimum. Includes 8087 & BCD features for 16-bit MS-DOS 2.0 or later and CP/M-86 1.1 or later. CP/M-80 version 2.2 or later 48K RAM minimum (8087 and BCD features not available). 8087 version requires 8087 or 80287 co-processor.

BORLAND INTERNATIONAL

BOR 0061A

TURBO PASCAL TURBO TUTOR®

VERSION 2.0

Learn Pascal From The Folks Who Created The Turbo Pascal® Family

Borland International proudly presents Turbo Tutor, the perfect complement to your Turbo Pascal compiler. Turbo Tutor is really for everyone—even if you've never programmed before.

And if you're already proficient, Turbo Tutor can sharpen up the fine points. The manual and program disk focus on the whole spectrum of Turbo Pascal programming techniques.

- **For the Novice:** It gives you a concise history of Pascal, tells you how to write a simple program, and defines the basic programming terms you need to know.
- **Programmer's Guide:** The heart of Turbo Pascal. The manual covers the fine points of every aspect of Turbo Pascal programming: program structure, data types, control structures, procedures and functions, scalar types, arrays, strings, pointers, sets, files, and records.
- **Advanced Concepts:** If you're an expert, you'll love the sections detailing such topics as linked lists, trees, and graphs. You'll also find sample program examples for PC-DOS and MS-DOS.®

10,000 lines of commented source code, demonstrations of 20 Turbo Pascal features, multiple-choice quizzes, an interactive on-line tutor, and more!

Turbo Tutor may be the only reference work about Pascal and programming you'll ever need!

Suggested Retail Price: $39.95 (not copy protected)

Minimum system configuration: Turbo Pascal 3.0. PC-DOS (MS-DOS) 2.0 or later. 192K RAM minimum (CP/M-80 version 2.2 or later: 64K RAM minimum).

BORLAND INTERNATIONAL

TURBO PASCAL EDITOR TOOLBOX™

It's All You Need To Build Your Own Text Editor Or Word Processor

Build your own lightning-fast editor and incorporate it into your Turbo Pascal® programs. Turbo Editor Toolbox gives you easy-to-install modules. Now you can integrate a fast and powerful editor into your own programs. You get the source code, the manual, and the know-how.

Create your own word processor. We provide all the editing routines. You plug in the features you want. You could build a WordStar®-like editor with pull-down menus like Microsoft's® Word, and make it work as fast as WordPerfect.®

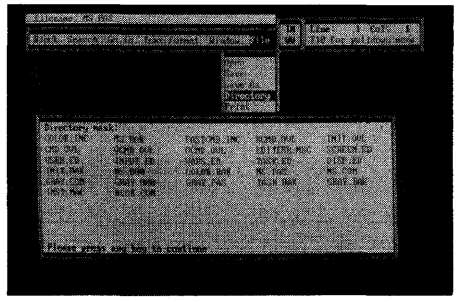
To demonstrate the tremendous power of Turbo Editor Toolbox, we give you the source code for two sample editors:

Simple Editor A complete editor ready to include in your programs. With windows, block commands, and memory-mapped screen routines.

MicroStar A full-blown text editor with a complete pull-down menu user interface, plus a lot more. Modify MicroStar's pull-down menu system and include it in your Turbo Pascal programs.

The Turbo Editor Toolbox gives you all the standard features you would expect to find in any word processor:

- Wordwrap
- UN-delete last line
- Auto-indent
- Find and Find/Replace with options
- Set left and right margin
- Block mark, move, and copy
- Tab, insert and overstrike modes, centering, etc.



MicroStar's pull-down menus.

And Turbo Editor Toolbox has features that word processors selling for several hundred dollars can't begin to match. Just to name a few:

☑ ***RAM-based editor.*** You can edit very large files and yet editing is lightning fast.

☑ ***Memory-mapped screen routines.*** Instant paging, scrolling, and text display.

☑ ***Keyboard installation.*** Change control keys from WordStar-like commands to any that you prefer.

☑ ***Multiple windows.*** See and edit up to eight documents—or up to eight parts of the same document—all at the same time.

☑ ***Multitasking.*** Automatically save your text. Plug in a digital clock, an appointment alarm—see how it's done with MicroStar's "background" printing.

Best of all, **source code is included for everything in the Editor Toolbox.**

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, 3270, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 192K RAM. You must be using Turbo Pascal 3.0 for IBM and compatibles.

BORLAND INTERNATIONAL

TURBO PASCAL GAMEWORKS®

Secrets And Strategies Of The Masters Are Revealed For The First Time

Explore the world of state-of-the-art computer games with Turbo GameWorks. Using easy-to-understand examples, Turbo GameWorks teaches you techniques to quickly create your own computer games using Turbo Pascal.® Or, for instant excitement, play the three great computer games we've included on disk—compiled and ready to run.

TURBO CHESS

Test your chess-playing skills against your computer challenger. With Turbo GameWorks, you're on your way to becoming a master chess player. Explore the complete Turbo Pascal source code and discover the secrets of Turbo Chess.

"What impressed me the most was the fact that with this program you can become a computer chess analyst. You can add new variations to the program at any time and make the program play stronger and stronger chess. There's no limit to the fun and enjoyment of playing Turbo GameWorks Chess, and most important of all, with this chess program there's no limit to how it can help you improve your game."

—George Koltanowski, Dean of American Chess, former President of the United Chess Federation, and syndicated chess columnist.

TURBO BRIDGE

Now play the world's most popular card game—bridge. Play one-on-one with your computer or against up to three other opponents. With Turbo Pascal source code, you can even program your own bidding or scoring conventions.

"There has never been a bridge program written which plays at the expert level, and the ambitious user will enjoy tackling that challenge, with the format already structured in the program. And for the inexperienced player, the bridge program provides an easy-to-follow format that allows the user to start right out playing. The user can 'play bridge' against real competition without having to gather three other people."

—Kit Woolsey, writer of several articles and books on bridge, and twice champion of the Blue Ribbon Pairs.

TURBO GO-MOKU

Prepare for battle when you challenge your computer to a game of Go-Moku—the exciting strategy game also known as Pente.® In this battle of wits, you and the computer take turns placing X's and O's on a grid of 19×19 squares until five pieces are lined up in a row. Vary the game if you like, using the source code available on your disk.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 192K RAM minimum. To edit and compile the Turbo Pascal source code, you must be using Turbo Pascal 3.0 for IBM PCs and compatibles.

BORLAND INTERNATIONAL

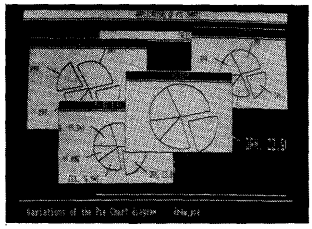
BOR 0065B

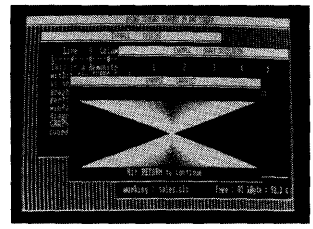
TURBO PASCAL GRAPHIX TOOLBOX®

A Library of Graphics Routines for Use with Turbo Pascal®

High-resolution graphics for your IBM® PC, AT,® XT,® PCjr®, true PC compatibles, and the Heath Zenith Z-100.™ Comes complete with graphics window management.

Even if you're new to Turbo Pascal programming, the Turbo Pascal Graphix Toolbox will get you started right away. It's a collection of tools that will get you right into the fascinating world of high-resolution business graphics, including graphics window management. You get immediate, satisfying results. And we keep Royalty out of American business because you don't pay any—even if you distribute your own compiled programs that include all or part of the Turbo Pascal Graphix Toolbox procedures.





What you get includes:

- Complete commented source code on disk.
- Tools for drawing simple graphics.
- Tools for drawing complex graphics, including curves with optional smoothing.
- Routines that let you store and restore graphic images to and from disk.
- Tools allowing you to send screen images to Epson-compatible printers.
- Full graphics window management.
- Two different font styles for graphic labeling.
- Choice of line-drawing styles.
- Routines that will let you quickly plot functions and model experimental data.
- And much, much more . . .

"While most people only talk about low-cost personal computer software, Borland has been doing something about it. And Borland provides good technical support as part of the price."

John Markov & Paul Freiberger, syndicated columnists.

If you ever plan to create Turbo Pascal programs that make use of business graphics or scientific graphics, you need the Turbo Pascal Graphix Toolbox.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, true compatibles and the Heath Zenith Z-100. Turbo Pascal 3.0 or later. 192K RAM minimum. Two disk drives and an IBM Color Graphics Adapter (CGA), IBM Enhanced Graphics Adapter (EGA), Hercules Graphics Card or compatible.

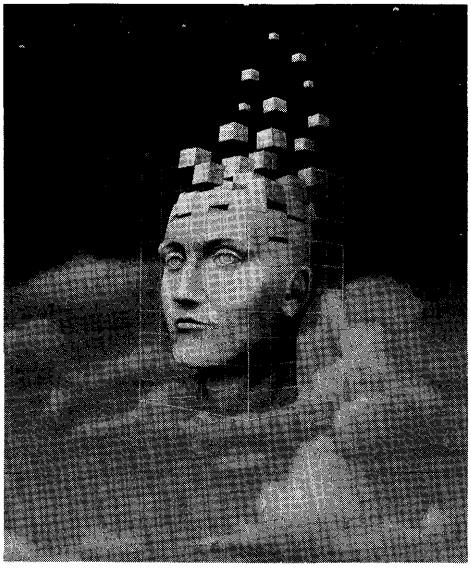
BORLAND INTERNATIONAL

TURBO PROLOG™

the natural language of Artificial Intelligence

Turbo Prolog brings fifth-generation supercomputer power to your IBM®PC!

urbo Prolog takes rogramming into a new, atural, and logical nvironment



/ith ***Turbo Prolog***, ecause of its natural, gical approach, both eople new to programming nd professional programmers an build powerful applica- ons such as expert systems, ustomized knowledge ases, natural language terfaces, and smart formation management systems.

urbo Prolog is a *declarative* language hich uses deductive reasoning to solve rogramming problems.

Turbo Prolog's development system includes:

- ☐ A complete Prolog compiler that is a variation of the Clocksin and Mellish Edinburgh standard Prolog.
- ☐ A full-screen interactive editor.
- ☐ Support for both graphic and text windows.
- ☐ All the tools that let you build your own expert systems and **AI** applications with unprecedented ease.

Turbo Prolog provides a fully integrated programming environment like Borland's Turbo Pascal,® the *de facto* worldwide standard.

You get the complete Turbo Prolog programming system

You get the 200-page manual you're holding, software that includes the lightning-fast ***Turbo Prolog*** six-pass compiler and interactive editor, and the free GeoBase natural query language database, which includes commented source code on disk, ready to compile. (GeoBase is a complete database designed and developed around U.S. geography. You can modify it or use it "as is.")

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 384K RAM minimum.

Suggested Retail Price $99.95 (not copy protected)

BORLAND INTERNATIONAL

How To Buy Borland Software

BORLAND

TURBO PASCAL®

The ultimate Pascal development environment

MACINTOSH™

Borland's new Turbo Pascal for the Mac™ is so incredibly fast that it can compile 1,420 lines of source code in the 7.1 seconds it took you to read this!

And reading the rest of this takes about *5 minutes*, which is plenty of time for Turbo Pascal for the Mac to compile at least *60,000 more lines* of source code!

Turbo Pascal for the Mac does both Windows and "Units"

The *separate* compilation of routines offered by Turbo Pascal for the Mac creates modules called "Units," which can be linked to any Turbo Pascal® program. This "modular pathway" gives you "pieces" which can then be integrated into larger programs. You get a more efficient use of memory and a reduction in the time it takes to develop large programs.

Turbo Pascal for the Mac is so compatible with Lisa® that they should be living together

Routines from Macintosh Programmer's Workshop Pascal and Inside Macintosh can be compiled and run with only the subtlest changes. Turbo Pascal for the Mac is also compatible with the Hierarchical File System of the Macintosh.™

3 MacWinners from Borland!

First there was SideKick for the Mac,™ then Reflex for the Mac,™ and now Turbo Pascal for the Mac™!

The 27-second Guide to Turbo Pascal for the Mac

- ☑ Compilation speed of more than 12,000 lines per minute
- ☑ "Unit" structure lets you create programs in modular form
- ☑ Multiple editing windows—up to 8 at once
- ☑ Compilation options include compiling to disk or memory, or compile and run
- ☑ No need to switch between programs to compile or run a program
- ☑ Streamlined development and debugging
- ☑ Compatibility with Macintosh Programmer's Workshop Pascal (with minimal changes)
- ☑ Compatibility with Hierarchical File System of your Mac
- ☑ Ability to define default volume and folder names used in compiler directives
- ☑ Search and change features in the editor speed up and simplify alteration of routines
- ☑ Ability to use all available Macintosh memory without limit
- ☑ "Units" included to call all the routines provided by Macintosh Toolbox

Minimum system configuration: 256K. One 400K drive.

BORLAND INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066

ISBN 0-87524-154-9