Register C 1071

COMPLETE Pascal ™

Complete Technology, Inc.
2443 S. Colorado Blvd. Suite 221
Denver, CO 80222
303/758-0920

Printed by L&L Productions, Inc.
ISBN: 0-929300-11-4

Table of Contents

Chapter 20 Standard Procedures and Functions

Introduction

Welcome to *Complete Pascal for the Apple IIGS.* Complete Pascal is the next generation of the popular Apple IIGS Pascal programming language TML Pascal II. Complete Pascal implements many new changes and features that make programming the Apple IIGS even easier and faster. The most important change in Complete Pascal is that it fully supports and is designed specifically for the Apple IIGS System Software version 5.0. In particular, Complete Pascal provides complete support for Resource editing and programming with resources.

- *Note: This version of Complete Pascal is designed for single machine users. If you intend to use Complete Pascal on an AppleShare fileserver for a network of Apple IIGS machines, you will need to acquire the special network version of Complete Pascal which fully supports the fileserver and network environment.*

Complete Pascal is designed to meet the needs of the broadest range of Apple IIGS programmers possible. The Complete Pascal language is solidly based upon the American National Standard for the Pascal language with numerous extensions for programmers accustomed to other Pascal implementations. It is a structured, high-level language you can use to write programs for any type or size application.

Complete Pascal implements separately compiled Units, Pascal strings, random disk I/O, and standard subprograms such as MoveLeft, FillChar, etc. that are found in UCSD implementations of Pascal such as Apple Pascal. And, of course, language features from the popular Macintosh version of Complete Pascal such as type casting, bit operations, CYCLE and LEAVE statements, and much more are found in Complete Pascal for the Apple IIGS.

In addition, Complete Pascal implements an extremely powerful Resource Editor which greatly simplifies the process of creating menus, windows, dialogs, etc. for you programs (see Chapter 4).

Using Complete Pascal, you can create five different types of programs:

- Textbook applications.
- Graphics Textbook applications
- Desktop applications
- New Desk Accessories (NDAs)
- Classic Desk Accessories (CDAs)

Textbook applications are the most basic of all Pascal programs. These programs execute in the Apple IIGS text screen and require no specific knowledge of the Apple IIGS Toolbox. See Chapter 6 for more information about Textbook applications.

Graphics Textbook applications are a simple extension of the Textbook applications which execute in the super hires graphics screen. Thus, Graphics Textbook applications can still make simple use of Readln and Writeln routines for I/O, but can also use QuickDraw to add graphics to a program. See Chapter 7 for more information about these types of programs.

Desktop applications are programs like Complete Pascal. These programs make full use of the Apple IIGS Toolbox to create menus, windows, dialogs, etc. Desktop programs are much more difficult to write than Textbook programs, but provide the added benefits of a graphics based application. Chapter 8 discusses the details of creating desktop applications.

Finally, Complete Pascal can be used to create both *New Desk Accessories* (NDAs) and *Classic Desk Accessories* (CDAs). NDAs and CDAs are special types of "mini-applications" which can be run from within other applications. Chapter's 9 and 10 respectively discuss the fundamentals of NDAs and CDAs and teach you how to create both types of programs.

About The Manual

This manual has been designed to help you begin using Complete Pascal quickly and to also serve as a reference manual when your programming level becomes more sophisticated. Each chapter in the manual is individually numbered making it very easy for CTI to update your documentation as future versions of Complete Pascal are released. The manual is divided into four parts: 1) *User's Guide*, 2) *Programming*, 3) *Language Reference*, and 4) *Appendices*. Each of these four parts is described below.

The *User's Guide* introduces you to Complete Pascal and will teach you how to set up your system to use Complete Pascal and how to get the most out of using Complete Pascal. The *User's Guide* is intended to provide a general overview of Complete Pascal's working environment.

Programming delves into the fundamentals of Complete Pascal by teaching you how to write each of the five different types of programs using Complete Pascal.

The *Language Reference* is a complete reference for the Pascal language features implemented by Complete Pascal. Note that this section is a reference and not a tutorial on the Pascal language. If you are not familiar with Pascal, you may need an additional text which teaches the Pascal language.

The *Appendices* summarize the Complete Pascal error messages and I/O results, compiler directives and Apple IIGS Toolbox Units. Additionally, an inside look at Complete Pascal's more advanced features is provided along with a comparison of Complete Pascal with the original TML Pascal.

What You Should Know

Complete Pascal uses the same intuitive, easy-to-use interface popularized by the Apple Macintosh and used in most Apple IIGS applications. If you are already familiar with this interface then you can begin using Complete Pascal immediately. If not, be sure you know the basic techniques of the Apple IIGS before you begin working with Complete Pascal:

- The fundamental mouse techniques of dragging, clicking, and double-clicking
- Pulling down menus and choosing commands

- Working with windows
- Working with icons on a desktop

If you feel unsure about any of these items, the following texts are good references for reading about these basic features of your Apple IIGS.

- *Apple IIGS Owner's Guide* - particularly Chapter 3, "The Mouse and the Keyboard".
- *Your Tour of the Apple IIGS* - the training disk that came with your Apple IIGS computer. This disk includes hands-on experience using the mouse.
- *Apple IIGS System Disk User's Guide* - this manual contains complete information about the Finder.

If you are new to the Pascal programming language, or you would like to review your Pascal programming skills, the following textbooks will offer you an excellent introduction and tutorial:

- *Oh! Pascal!*, Michael Clancy and Doug Cooper, W.W. Norton and Company, 1982.
- *Programming in Pascal*, Peter Grogono, Addison-Wesley, 1978.

If you are looking for technical references to the Pascal programming language you will find the following documents extremely useful:

- *Pascal User Manual and Report*, Kathleen Jensen and Nicklaus Wirth, Springer-Verlag, 1985.
- *American National Standard Pascal Computer Programming Language*, ANSI/IEEE 770X2.97-1983, IEEE/Wiley-Interscience, 1983.

For those experienced Pascal programmers, or those aspiring to write the next great Apple IIGS application, the following texts provide a wealth of information regarding the Apple IIGS Toolbox:

- *Exploring the Apple IIGS*, Gary Little, Addison-Wesley, 1987.
- *Mastering the Apple IIGS Toolbox*, Morgan Davis and Dan Gookin, Compute! Publications, Inc., 1987.
- *Advanced Programming Techniques for the Apple IIGS Toolbox*, Morgan Davis and Dan Gookin, Compute! Publications, Inc., 1987.

  v Note: *Advanced Programming Techniques for the Apple IIGS Toolbox* uses TML Pascal in its discussion of programming the Toolbox.

- *Apple IIGS Technical Reference*, Michael Fischer, Osborne/McGraw-Hill, 1987.

Finally, there is the Apple Programmer's and Developer's Association (APDA). APDA is an Apple Computer membership organization that distributes technical information to programmers and developers. APDA is a great source for Technical Notes, programming utilities, reference books, and information about announced (but unreleased) products.

You will want to obtain technical information about System Software version 5.0.x as it becomes available from APDA. For information about membership and products, contact APDA directly:

Apple Programmer's and Developer's Association (APDA)
Apple Computer, Inc.
20525 Mariani Avenue, MS: 33G
Cupertino, CA 95014-6299
800-282-2732

What You Need

To use Complete Pascal, you will need the following items:

- An Apple IIGS with at least 1024K of Random Access Memory (RAM)
- A color or monochrome monitor
- At least one 800K 3.5" disk drive
- One blank disk for backing up your Complete Pascal Master Disk
- Blank disks for storing the programs you create

The following items are optional for use with Complete Pascal:

- A second 3.5" disk drive or a hard disk drive
- A printer

v **Important: The Complete Pascal Master Disk includes the most recent version of the Finder and GS/OS (version 5.0 and later). If you are currently using a version of GS/OS earlier than version 5.0 and would like to learn more about the capabilities of GS/OS version 5.0, you can order an updated version of the *Apple IIGS System Disk User's Guide* from your Authorized Apple Dealer.**

Register Your Product

Take a minute now to complete and return the Complete Pascal Registration Card to Complete Technology. If you did not receive a registration card with your product, contact Complete Technology's Customer Service Department immediately.

As a registered user of Complete Pascal, you will be notified of enhancements to this product and be afforded discounts on other Complete Technology products.

Contacting Complete Technology

If you have questions concerning your product, a change of address, or any other non-technical question, feel free to call Complete Technology's Customer Service Department. Customer Service is available Monday through Friday between the hours of 9:00 am and 6:00 pm Mountain Time at 303/758-0920.

If you have technical questions concerning Complete Pascal, or any of Complete Technology's other Apple IIGS products, feel free to call specifically between the hours of 2:00 P.M. and 5:00 P.M. Mountain Time, Monday through Friday, at 303/758-0920.

If your prefer to write Complete Technology a letter, please address your letter to the following address:

Complete Technology, Inc.
2443 S. Colorado Blvd. Suite 221
Denver, Colorado 80222

Acknowledgements

All Complete Technology products are trademarks or registered trademarks of Complete Technology, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Published by Complete Technology, Inc.
2443 S. Colorado Blvd. Suite 221
Denver, CO 80222
(303) 758-0920

Complete Pascal

Project Manager:	Vince M. Cooper
Program Authors:	Thomas Leonard Vince M. Cooper Jason Coleman
Manual Authors:	Robert Leonard Thomas Leonard Vince M. Cooper
Manual Copy Editors:	Veronica Sumner Sharon Rains

Special thanks to Lance Taylor-Warren and the staff of L&L Productions for their assistance in the production of this software manual.

Chapter 1

Getting Started

This chapter shows you how to make a backup copy of your Complete Pascal Master Disk and how to operate Complete Pascal from either a floppy disk drive or hard disk drive configuration. Also presented is a discussion of compilers versus interpreters and a list of Complete Pascal's most outstanding new features.

Backing Up the Complete Pascal Master Disk

The first time you use your Complete Pascal package, you will want to make a working copy of your original Complete Pascal Master Disk. Programming a computer naturally lends itself to causing problems if not disasters if programs you write get out of control. Therefore, it is imperative that you use a working copy of the software and store the Master Disk as permanent backup. This process involves only two simple steps:

- Initializing a blank 3.5" floppy disk

- Using the Apple IIGS Finder to create a working copy of your Complete Pascal Master Disk. Therefore allowing you to store your Master Disk as a permanent backup.

  - **Note:** If you are not familiar with these procedures, see your copy of the *Apple IIGS System Disk User's Guide* for a discussion of these topics.

Setting Up Your System

The following three sections describe how you might set up a working environment for using Complete Pascal with either a single 3.5" disk drive system, a dual 3.5" disk drive system, or a hard disk system configuration.

Single Disk Drive System

If you are using a single 3.5" disk drive system, you will find that Complete Pascal can be used exactly as it is shipped on the distribution disk without having to sacrifice any functionality or performance.

The only restriction imposed by a single disk drive system is the size of the programs you develop will be restricted to the available disk space to store them. On your working copy of Complete Pascal, feel free to delete the various folders containing example programs to make room for your new programs. Of course, you do have backups of these folders on your Complete Pascal Master Disk when you decide you need them.

The Examples folder contains those files used in the User's Guide portion of this manual which you should keep on your disk if you intend to follow the Guide's discussions.

Two Disk Drive System

If you have a second disk drive, then you can take advantage of this extra storage capacity for developing larger programs. You may find it easier to keep all of the example programs, as well as any new programs you create, on a separate disk and access them from your second disk drive.

- **Note:** The ToolInterfaces folder should be kept on your working disk in order that all programs compiled using Complete Pascal may share the files contained therein.

Hard Disk Drive System

While a hard disk is not required to use Complete Pascal, you will enjoy the luxury of faster disk access and an extensive amount of disk storage space available for creating large programs.

To use any GS/OS formatted hard disk drive with Complete Pascal, simply copy the entire contents of the Complete Pascal working disk onto your hard disk, with the exception of the SYSTEM folder (you obviously already have the entire GS/OS operating system installed on your hard disk if it is up and running).

v **Important:** Complete Pascal is specifically designed for System Software version 5.0.x, or greater, and GS/OS. Complete Pascal, and programs created with Complete Pascal, cannot run on any version of ProDOS/16 or versions of GS/OS earlier than version 5.0.x. If your hard disk is formatted with any earlier operating system contact your local Apple Authorized Dealer for an immediate upgrade to the most recent GS/OS operating system.

A Note On RAM Disks

Traditionally, Apple II users have found using RAM disks advantageous, and have done so without *stealing* available memory from an application, due to the Apple II's restriction of permitting only 128K of memory or less to a single application.

However, Complete Pascal and the Apple IIGS operate much differently than the earlier Apple II computers. Complete Pascal is a memory resident application. Thus, there is no advantage to storing Complete Pascal on a RAM disk. Further, Complete Pascal maintains, in memory, an entire copy of the file(s) it is editing (including library files, compiled code, etc.) and uses the Apple IIGS Memory Manager to track available memory. Thus, any RAM space you might allocate for a RAM disk would only decrease the amount of memory Complete Pascal has available to it for editing and compiling.

Compiled versus Interpreted Languages

Complete Pascal is a *compiled language*. In this regard, as well as others, it differs from *interpreted languages* such as Apple Computer's AppleSoft BASIC.

A programming language is characterized by its collection of statements, expressions and other components generally known as the syntax, or structure, of the language. While programs written in a computer language are generally understandable to the human reader, they are totally incomprehensible to the computer. Or, in the case of the Apple IIGS, the 65816 microprocessor.

Before a statement written in a computer language can be executed by the computer, it must first be translated into code understood by the computer - *machine language*. Machine language consists of long lists of binary numbers (0's and 1's) that are understood by the computer as a series of off and on states representing operations the computer is capable of performing. Of course, a long string of 0's and 1's is not easily understood or readily comprehended by humans.

A major part of any computer language is its means of translating programs into machine language. In an interpreted language, the translation is done while the program is being executed. This procedure is sometimes denoted as "on the fly". If a statement in the program is executed 100 times, the translation is also performed 100 times by the interpreter. Generally speaking, interpreted languages run slower than compiled languages because of the need for translation to occur during the running of the program.

On the other hand, a compiled language translates a program into machine language prior to actually running the program. Thus, each line in the program is translated only once - during the compilation process. In addition, the compilation process discloses all of the syntax errors before the program is executed. Of course, the compiler cannot find errors in the program's logic such as infinite loops.

Generally speaking, compiled programs run significantly faster than interpreted programs. In addition, compiled programs can run independently of any language processor. That is, compiled Complete Pascal programs can run by themselves under GS/OS without Complete Pascal being available. In contrast, an AppleSoft BASIC program can never be run without having AppleSoft BASIC available to run the program (AppleSoft BASIC programs have to be interpreted every time they are run, and only the AppleSoft BASIC language processor can interpret the code).

Where Now?

That's a good question! At this point you have several options available to you. If you are:

New to programming	Read the entire *User's Guide* portion of this manual. Afterwards, study Chapter's 6 through 10 to learn about the five different types of programs you can create using Complete Pascal. Be certain you have a good Pascal programming textbook available to review the fundamentals of the Pascal programming language. In addition, use this book's *Language Reference* and *Appendices* as references to your work.
Familiar with another Apple IIGS programming language	A quick review of the User's Guide will prove useful. Specifically, Chapter's 4, Resources and Chapter 5, Complete Pascal Menu Reference will provide you with an overview of the key features found in Complete Pascal. Then, depending upon the type of programs you will be developing, review Chapter's 6 through 10. Use this book's *Language Reference* and *Appendices* as references to your work.
Familiar with TML Pascal	Read Appendix E - Complete Pascal versus TML Pascal. You may choose to skip the User's Guide entirely and start the software right away. If so, we urge you to at least review the topic of resources introduced in Chapter 4.

Chapter 2
Using Complete Pascal

With your system properly set up, you are now ready to begin examining the Complete Pascal working environment.

In this chapter you will be introduced to the Complete Pascal editor. Chapter 3 continues this discussion by demonstrating how the Complete Pascal editor can assist you in actually compiling and running programs.

Running Complete Pascal

If you have not already done so, insert your working copy of Complete Pascal into the 3.5" floppy disk drive and turn (boot) your machine on. After the Apple IIGS completes its booting process, you will be presented with the Apple IIGS Finder's Desktop. Figure 2-1 illustrates the desktop's appearance after booting your machine.

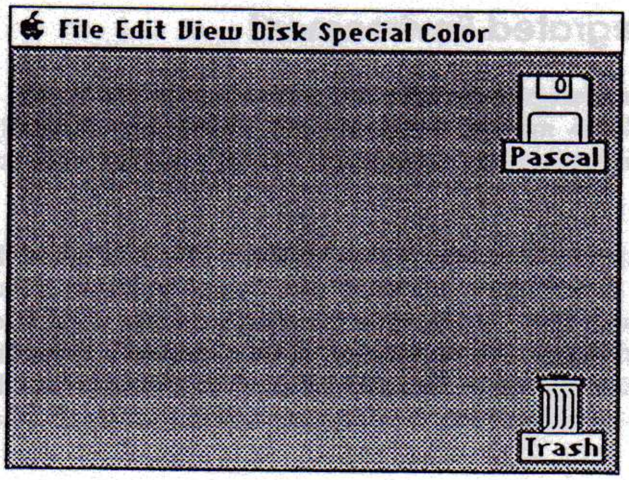


Figure 2-1
Apple IIGS Desktop

- **Note:** Your desktop will appear as shown in Figure 2-1 only if your system configuration consists of a single 3.5" disk drive.

Now, invoke Complete Pascal by clicking the mouse once on the Complete Pascal icon shown on the desktop, then pulling down the Finder's **File** menu, and then selecting the **Open** command (double-clicking the mouse over the Complete Pascal icon accomplishes the same result as selecting the **Open** command).

Opening the Complete Pascal file will automatically load Complete Pascal into your Apple IIGS 's internal memory. Be patient, as the Apple IIGS requires a few moments before it completes the loading process. Figure 2-2 illustrates Complete Pascal's Main Menu after the program has been successfully loaded into memory.

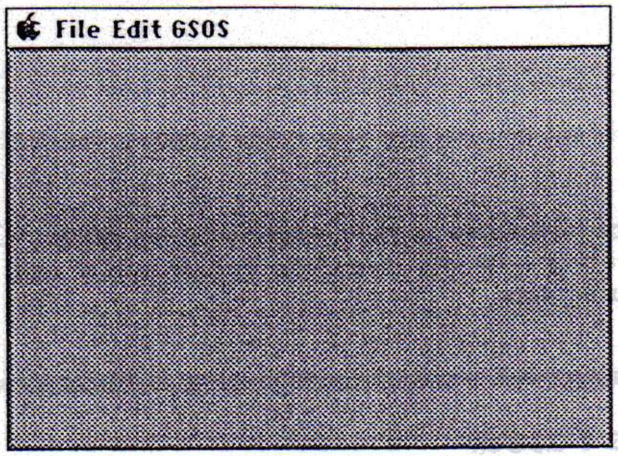


Figure 2-2
Complete Pascal Main Menu

Examining the Integrated Environment

Complete Pascal has been designed to take full advantage of the Apple IIGS desktop interface using the mouse, pull-down menus, windows, etc. This user-friendly environment makes programming easy, as it integrates Complete Pascal's editor and compiler into the same working environment.

You have already examined Complete Pascal's Main Menu (Figure 2-2). The Main Menu remains just as you see it in Figure 2-2 until you choose to begin editing either a program's source code file or resource file. Because a program's source code file is the basis to every program you create, we dedicate the remainder of this chapter's discussion (and Chapter 3) to discussing those features available in Complete Pascal as they pertain to source code files. In doing so, we will reserve our discussion of editing resource files until Chapter 4.

Figure 2-3 graphically illustrates the relationship between editing Complete Pascal source code files versus resource files and the way in which Complete Pascal's Main Menu changes accordingly.

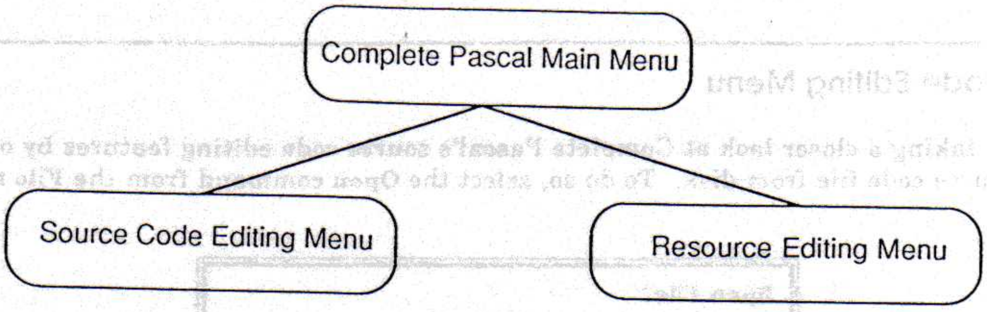


Figure 2-3
Complete Pascal's Two Main Menus

As you can see from the diagram in Figure 2-3, you will be switching between two different versions of Complete Pascal's Main Menu as you move from editing a source code file to a resource file. And vice versa.

Editing Source Code Files

Editing windows are the tools Complete Pascal provides you for entering and modifying program source code. Complete Pascal allows you to have as many program source code files open at the same time as available memory allows. Obviously, the greater amount of RAM you have installed in your IIGS, the more memory space you will have available to create programs and open windows in the editing environment.

As you open additional source code files, Complete Pascal places each file in a different editing window with each window being independent of other open windows. Keep in mind that only one editing window can be *active* at a time. By active, we are referring to the topmost window (if more than one is open) in which all commands issued by the user are performed.

Complete Pascal also implements *dialog boxes* as a means of communicating with the user. Dialog boxes are used in Complete Pascal to provide the user with requested information, or to ask the user for required information before continuing. Dialog boxes usually include OK/Cancel buttons or Yes/No buttons. These buttons allow you to communicate with Complete Pascal to signify when you are finished with the dialog box.

Source Code Editing Menu

Let's begin taking a closer look at Complete Pascal's source code editing features by opening an existing source code file from disk. To do so, select the **Open** command from the **File** menu.

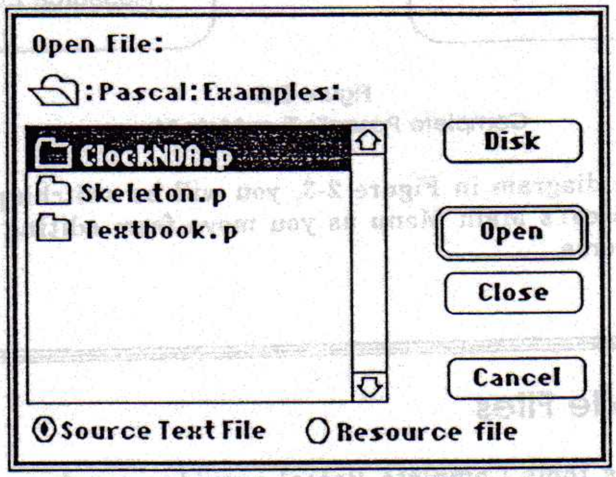
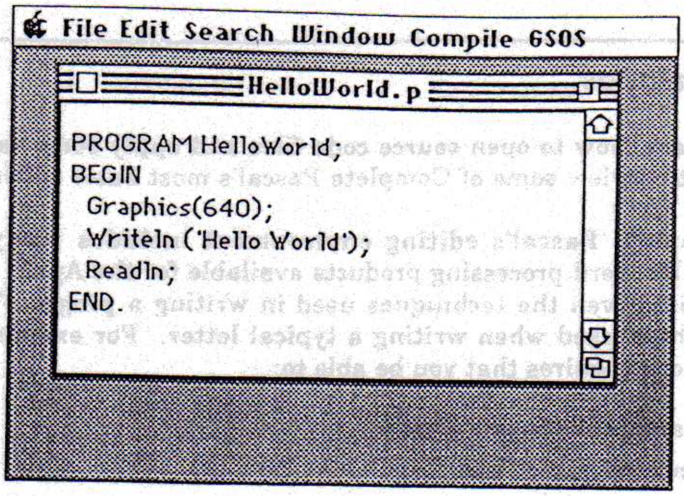


Figure 2-4
Open File Dialog Box

Figure 2-4 illustrates the standard Open File Dialog Box which results from issuing this command. Studying the figure, you will see two buttons located in the bottom of the dialog box which allow you to indicate whether you would like to open either a source code file or resource file. The Open File Dialog Box will always default to open a source code file unless you specify otherwise.

Now, open the Skeleton.p source code file using the Open File Dialog Box (It's located inside the Examples folder on your working disk).

Immediately upon opening the Skeleton.p source code file Complete Pascal's Main Menu changes. As shown in figure 2-5, Complete Pascal's Source Code Editing Main Menu includes seven individual menus. These seven menus are designed to logically organize the several commands available to you in Complete Pascal. Within each menu you will find the various Complete Pascal commands. Using the mouse, pull-down each menu to discover just how easy it is to locate the editing commands available to you in Complete Pascal. Most commands found in the menus may be invoked by typing its corresponding *command-key equivalent* at the keyboard rather than pulling down its menu and selecting the command with the mouse.



```
PROGRAM HelloWorld;
BEGIN
  Graphics(640);
  Writeln('Hello World');
  Readln;
END.
```

Figure 2-5
Source Code Editing Menu

Chapter 5 provides a concise review of every Complete Pascal menu and menu item and should be used as an authoritative reference. However, in addition to Chapter 5, we will be discussing some of these commands throughout the remainder of this chapter and Chapter 4.

File Naming Conventions

When you opened the Skeleton.p source code file you may have noticed a few similarities with some of the other file names on disk. The similarities we are referring to are the naming conventions used to distinguish between a Complete Pascal source code file, a resource file and a stand-alone Complete Pascal application for that matter.

Although you are not required to follow the naming conventions we have adopted, it certainly will make matters more clear for you as you become more involved with using Complete Pascal. Table 2-1 describes the naming conventions Complete Technology has adopted for working with Complete Pascal.

Table 2-1
Naming Conventions

File Type	Example	Description
Source Code File	Skeleton.p	Use a ".P" suffix on all source code filenames
Resource Files	Skeleton.r	Use a ".R" suffix on all resource filenames
Application File	Skeleton	Complete Pascal automatically applies the same name to a stand-alone application, less the suffix, as its origin source code filename.

You should keep in mind that Complete Pascal does not automatically assign the correct suffix to the files you create. Therefore, you should apply the appropriate naming conventions at the time you initially create any new files.

Basic Editing Techniques

Now that you understand how to open source code files and apply some set of standard naming conventions, it's time to review some of Complete Pascal's most basic editing techniques.

As you will see, Complete Pascal's editing environment includes many features similar to some of the more popular word processing products available for the Apple IIGS. This, of course, should be of no surprise given the techniques used in writing a program's source code are not much different than those used when writing a typical letter. For example, as when writing a letter, writing source code requires that you be able to:

- cut, copy, paste and undo ranges of text
- search and replace ranges of text
- apply proper indentation (tabs)
- select a specific font and size to your liking
- scroll through an entire file
- print to an ImageWriter or LaserWriter

On the other hand, Complete Pascal's editing environment offers plenty of commands specific to writing and compiling a program's source code. Some examples include:

- checking a source code file's syntax
- running a program
- creating a stand-alone Apple IIGS application
- invoking GS/OS commands (ie. changing a file's name) from within Complete Pascal
- reporting program error messages

Using the source code file Skeleton.p already open in the Complete Pascal editing environment, you should familiarize yourself with those common Apple IIGS editing features as described above. For example, try using the Complete Pascal editing environment to: copy a range of text and paste the selected text somewhere else in the program; change the font and font size of the entire window; revert all changes you make to the file; and more. Remember, Chapter 5 offers a complete overview of every menu item found in Complete Pascal's editing environment.

After spending a few minutes studying some of the common editing commands on the Skeleton.p program, you will begin to realize just how easy it is to write and edit programs using Complete Pascal's editing environment.

Familiarizing yourself with the common editing commands should be fairly straightforward. Now, if you are ready to learn more about how Complete Pascal's editing environment is ideally suited to assist you with specific editing tasks related to compiling and running programs, let's move on to Chapter 3. On the other hand, if you would like to take a break, simply revert all changes made to the Skeleton.p source code and exit Complete Pascal by issuing the **Quit** command from the **File** menu.

Chapter 3

Creating Programs

In this chapter, the three different compiling options available in Complete Pascal are discussed in order to demonstrate how to create applications and desk accessories with Complete Pascal.

Compiling Alternatives

Complete Pascal offers the programmer three options for compiling programs: **To Memory, To Disk** and **Check Syntax.** The compile commands are found in the **Compile** menu. You can see each of the commands by pressing and holding the mouse button down over the **Compile** menu.

The **To Memory** compile option is likely to be the one you use most often. This command invokes Complete Pascal to compile the source code in the *active* editing window (the topmost window), and then, upon successful completion, executes the program directly within the Apple IIGS's internal memory.

The **To Disk** command is similar to the **To Memory** command except that Complete Pascal does not immediately execute the compiled program, but rather creates a stand-alone GS/OS application file on disk. You will use this command when you have a complete running program free of errors and you wish to execute the program directly from the Finder.

Finally, the **Check Syntax** command allows you to quickly verify the syntax of a Complete Pascal program. This option does not run the selected program nor does it create a disk file. This is the fastest compile option available in Complete Pascal.

When a compile option is invoked by selecting any of the three compile commands, Complete Pascal displays the Compile Progress Dialog Box. This dialog is used to display the compiler's progress during compilation. When the Compile Progress Dialog Box's *indicator bar* reaches the right side of the display, the compile process has been completed.

Testing a Program's Source Code

As already mentioned, the Check Syntax command is the fastest of the three compilation techniques since it does not cause any code to be generated. Instead, this command instructs Complete Pascal to verify that the program in the topmost window was written using valid Pascal key words, statements and functions. It cannot, however, check a program for correct logic. For example, an infinite loop in a program's source code will go undetected by the Check Syntax command.

To study this command, open the Skeleton.p example program provided with Complete Pascal. Pull-down the Compile menu and select the Check Syntax command. The Compile Progress Dialog Box is immediately displayed indicating the compiler's progress as it checks the syntax of the source code.

When the indicator bar inside the Compile Progress Dialog Box reaches the right side of the display, compilation is complete. As you will see, Complete Pascal takes only a brief moment to compile the Skeleton.p program. The reason for this, of course, is that Complete Pascal is a fast compiler. In addition, the program is quite small and the Check Syntax command is the fastest of Complete Pascal's three compile options.

A result of no errors does not necessarily mean a program is completely free of all possible errors. However, using the Check Syntax command will ensure the program does not contain any syntax errors.

It is important you use the Check Syntax command when you are uncertain whether your program will run correctly. Since this command does not run the program after compiling it, you can avoid situations where your program contains *logic* errors which might cause the computer to crash.

If an error is detected in the source code of a program, Complete Pascal will stop the compilation process, return to the Complete Pascal editor, highlight the exact location of the discovered error and then display a descriptive error message. Errors are discussed later in this chapter in the section "Detecting Program Errors".

Running a Program

Once you have determined your program does not contain any syntax errors by issuing the Check Syntax command, the program can then be run. To do this, select the To Memory command from the Compile menu. Upon selecting this command, Complete Pascal again displays the Compile Progress Dialog Box. This time the compiler generates code for the program. If the program does not contain syntax errors the compiled program is immediately run.
To run a compiled program, the Complete Pascal environment temporarily shuts down by hiding its menus, windows, etc. and then transfers control to the compiled program. The compiled program is now in complete control of the computer as it executes. When the program has completed execution, the Complete Pascal environment restores its menus and windows allowing you to continue programming.

Because it is possible the compiled program may contain logic errors causing the machine to crash, Complete Pascal provides a safety feature called *Auto Save*. If this option is turned on, Complete Pascal automatically saves any changes you have made to the program's source code prior to compiling. This feature ensures you will not lose your source code changes in the event of a catastrophic error during your program's execution. The Auto Save option is discussed in more detail in Chapter 5 under the "Preferences..." section.

To compile the Textbook.p program, first, be certain the program is in the frontmost window. The Textbook.p program uses the text screen to read 10 numbers into a binary tree then traverse the tree, printing it out. The program then waits for the Return key to be pressed. After the Return key is pressed, program execution terminates and control is returned to the Complete Pascal environment with the windows restored exactly as you left them.

Creating a Stand-alone Application

As seen above, the compile to memory feature of Complete Pascal is extremely fast and interactive. However, one small problem exists – you must launch Complete Pascal every time you want to run a Complete Pascal program. Thus, the third compilation technique available in Complete Pascal – To Disk. This compile option allows you to create stand-alone GS/OS application that can be run from the Apple IIGS Finder by double-clicking on its icon just as you did the Complete Pascal icon to invoke Complete Pascal. You can even copy the compiled application to another disk and run it from there because Complete Pascal is no longer required after the program is compiled to disk.

Let's compile the Textbook.p program to disk. Again, open the Textbook.p source code in the topmost window (remember the compile commands only work on the active window). Select the To Disk command from the Compile menu to compile the Textbook.p program and create a stand-alone application on disk. You will notice the compilation process takes significantly longer to complete this time.

The reason for this additional amount of time results from the compiled program being written to disk. The name of the resulting application file on disk is Textbook, and it is located in the same folder as the Textbook.p source code file.

- **Important**: The name of a compiled application file is the name of the source code text file with the ".p" suffix removed.

Compiling Units

In addition to compiling *programs*, Complete Pascal can compile Pascal Units. A unit cannot be compiled to create a stand-alone application, but rather serves as a *container* for code. Units are used to split a large program up into smaller more manageable pieces that can be *used* by programs and other units. A unit can contain constant, type, variable, procedure and function declarations.

Because a unit is not capable of being *executed*, the Complete Pascal compiler acts differently when selecting the various compiler commands in the Compile menu. As mentioned above, when a unit is compiled, it does not create an application that can be run. However, Complete Pascal does save the unit's compiled code so that other programs can use it. Thus, when selecting the To Memory command, Complete Pascal compiles the unit but then returns control to the editor instead of transferring control to the compiled code as it would do for a program. Note that Complete Pascal does save the compiled code in memory so that it can later be used by a program.

Units can also be compiled using the To Disk command. When a unit is compiled to disk, it does not create a GS/OS application, but rather, a Complete Pascal ".p.o" file. The ".p.o" file contains the library's compiled source code and symbol table. When another program or unit needs to use a unit that has not been compiled to memory using the To Memory command, Complete Pascal searches for the compiled code on disk in a ".p.o" file.

v **Important:** The name of a compiled unit file is the name of the source code text file with the ".o" suffix added. Thus, the name of a compiled unit ends with the suffix ".p.o".

The Check Syntax command behaves exactly the same for both programs and units. That is, Complete Pascal only verifies that the unit's source code contains legal Pascal statements.

Detecting Errors

So far in this chapter we have discussed how to compile programs using Complete Pascal. However, our discussion has been limited to programs known to be correct. That is, they do not contain any errors. In this section, we will discover how Complete Pascal deals with errors.

First, let's consider the components of the Complete Pascal environment. Complete Pascal is an integrated development tool made up of three separate pieces – the editor, the compiler and the linker. These different pieces work so closely together the user really only perceives them as one in the same. However, knowing how these pieces work together will help you understand the error messages Complete Pascal reports to you.

The editor, of course, is where you spend most of your time. It is responsible for editing windows and most of the commands available in each Complete Pascal menu. The compiler is invoked whenever you select any of the three compile commands. The compiler is responsible for checking if syntax errors exist in your program and then generating code for the program. Finally, the linker component of Complete Pascal is only invoked when you have chosen to compile a program to memory or to disk. The linker is responsible for combining the compiled code with other pieces of code your program needs (i.e. units). It is also responsible for allocating the internal memory a program requires in order to run within the Apple IIGS's memory, and for writing a compiled program to disk.

The editor only reports errors related to the editing environment. For example, it will report an error when you ask it to save a file to a disk which is write protected or to open a file and not enough memory is available to read the file. The compiler only reports errors related to illegal Pascal source code. If you misspell a reserved word or forget to put a comma where one was expected, the compiler reports an error. Finally, the linker reports errors when an attempt to create a final program fails. This might happen if you compile a program to disk and the disk is locked or there is not enough room to fit the compiled program on disk.

When any component of the Complete Pascal environment detects an error it first takes whatever actions necessary to recover without causing any loss of data and then displays the Error Dialog Box with a descriptive error message. In addition to the error message, an *icon* on the left side of the dialog box is also displayed. This icon is used to indicate which component of Complete Pascal detected the error. The icon can usually help you better understand the error message. In addition, if the error is related to a particular part of your program's source code, the editor displays that portion of source code in the editing window and highlights the exact location of the error. Highlighting usually occurs for detected compiler errors.

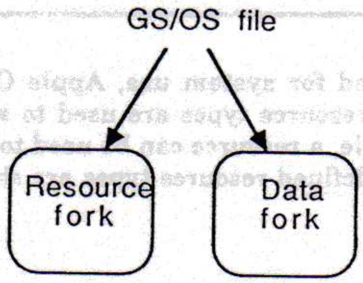
Editor errors are displayed with an upside down yield sign icon, compiler errors are displayed with a green bug icon and the linker reports errors with a chain link icon. In addition to the icon and error message, Complete Pascal will display an error code. If an error code is displayed, it is the error code returned by the Apple IIGS Toolbox or GS/OS that caused Complete Pascal to process the error. Sometimes this error code can help you better understand the circumstance of the error.

Appendix A provides a complete list of the errors detected by Complete Pascal.

Chapter 4

Resources

Perhaps the most exciting addition to the Apple IIGS System Software v5.0 is the *Resource Manager*. The Resource Manager is a special tool that manipulates *resources* stored in the *resource fork* of GS/OS files. Recall that a file stored under the GS/OS operating system can have two forks – the data fork and the resource fork. A file still has only one file name, but each fork can be opened and accessed separately from the other.



The data fork is typically manipulated using GS/OS operating system calls to open, read, write and close the file. The file is simply treated as a collection of bytes on a disk or some other storage device. The organization of data in the data fork is not well defined and is typically very different for each file type. See the *GS/OS Reference* for complete documentation regarding the routines provided by GS/OS to manipulate a data fork.

The resource fork on the other hand, is manipulated using the Resource Manager. The Resource Manager defines the precise structure for the resource fork and provides several routines for accessing and manipulating the information stored in the resource fork. Complete Pascal implements a Resource Editor for graphically creating and editing resources, and is the subject of this chapter. For a complete discussion of the Resource Manager and the format of the resource fork maintained by the Resource Manager see the *Apple IIGS Toolbox Reference Update*.

Resources

A resource is a formatted collection of data, organized in a certain way, which represents a menu or menu bar, a window, alert strings, or any other number of system defined or user defined types of data. The exact structure for each type of resource is not defined by the Resource Manager. The Resource Manager only defines how the resources are stored on the disk, not its contents.

A program identifies a resource by its *resource type* and its *resource ID*. The resource type defines a class or group of resources that share a common format. The resource ID uniquely identifies a specific instance of a resource of a given resource type. Together, the resource type and resource ID completely identify the resource and its format. Note that a resource ID is only

unique among a resource type. Two resources of different resource types can have the same resource ID.
The resource type is a two byte integer number. The following table shows the ranges of resource types.

Table 4-1
Apple Defined Resource Types

Range	Resource Type
$0000	Invalid resource type number (do not use)
$0001 through $7FFF	Available for application use
$8000 through $FFFF	Reserved for system use

Among the resource types reserved for system use, Apple Computer has predefined several resource types. These predefined resource types are used to store the representations of Apple IIGS Toolbox elements. For example, a resource can be used to define the structure and contents of a menu or a window. These predefined resource types are shown in the Table 4-2.

Table 4-2
Apple Defined Resource Types

Conplete	Resource Type Name	Resource Type Number (Hex)
	rIcon	$8001
	rPicture	$8002
Yes	rControlList	$8003
Yes	rControlTemplate	$8004
Yes	rPString	$8006
	rStringList	$8007
Yes	rMenuBar	$8008
Yes	rMenu	$8009
Yes	rMenuItem	$800A
	rTextForLETextBox2	$800B
	rCtlDefProc	$800C
Yes	rWindParam1	$800E
	rWindParam2	$800F
	rWindColor	$8010
Yes	rTextBlock	$8011
	rStyleBlock	$8012
Yes	rToolStartup	$8013
	rResName	$8014
Yes	rAlertString	$8015
Yes	rText	$8016
	rCodeResource	$8017
	rCDEVCode	$8018
	rCDEVFlags	$8019
	rTwoRects	$801A
	rListRef	$801C
Yes	rCString	$801D
Yes	rErrorString	$8020

The resource type name in the table above is given for descriptive purposes. In addition, the Apple IIGS Toolbox interface unit Resources.p uses these same names as constants whose values are those shown in the table. The Complete column indicates if the Resource Editor supports editing of the given resource.

As mentioned above, every resource has a resource type and resource ID. A resource ID is a four byte long integer number. The following table shows the ranges of resource IDs.

Table 4-3
Resource ID Ranges

Range	Resource Type
$00000000	Invalid resource ID number (do not use)
$00000001 through $07FEFFFF	Available for application use
$08000000 through $07FFFFFF	Reserved for system use
$08000000 through $FFFFFFFF	Invalid values (do not use)

When creating new resources, a unique resource ID must be obtained for the resource type to which the resource belongs. The Resource Manager provides the routine UniqueResourceID for this purpose. The resource IDs are very important to remember, because they are required as parameters to several Toolbox routines which load and create menus, windows, etc.

Resource Editing

As mentioned above, Complete Pascal implements a Resource Editor. The Resource Editor is used to graphically create and edit several of the Apple predefined resource types. Using the Resource Editor you can create menus, windows, alerts, strings and much more by simply clicking the mouse. The resources you create can then be incorporated into a program in order to quickly and easily create sophisticated desktop applications.

Resource Document Window

When Complete Pascal opens a resource file, it displays a window like the one shown in Figure 4-1.

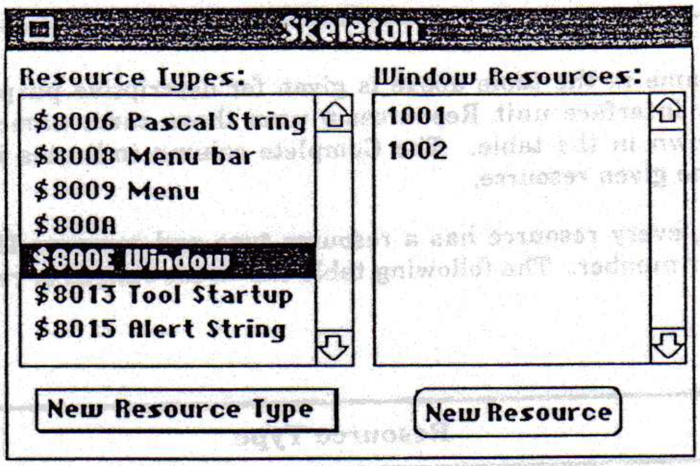


Figure 4-1
Resource Document Window

The window contains two scrollable lists. The list on the left side of the window displays every resource type for which a resource exists in the resource file. The display gives the resource type number and optionally a resource type name. Only resources for which the Resource Editor supports are shown with resource type names. This makes it easy to distinguish which resources can be edited and which cannot.

The list on the right side of the window displays each resource that exists for the selected resource type in the left list. The list displays the resource's ID number. In the figure above, the Window resource type is selected and the file contains two Window resources (1001 and 1002). To edit a particular resource, simply double click the mouse on its resource ID.

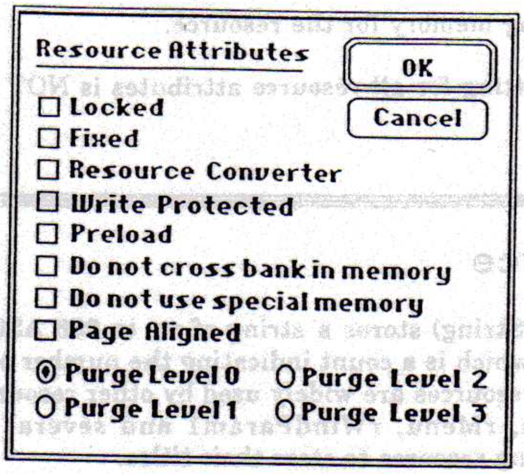
In addition to the two lists, the Resource document window contains a pop-up menu and a button. The New Resource button is used to create new resources of the currently selected resource type. For example, in the window shown above, if the New Resource button is clicked, the Resource Editor will create a new Window resource with a unique resource ID.

The pop-up menu is used to create a new resource when the resource type does not yet exist in the file. For example, in the window shown above, there is no C String resource, so to create a C string resource, click the mouse in the pop-up menu and select the resource name C String.

Resource Attributes

Every resource has a set of *attributes* that define how the resource can be used. The attributes are stored by the Resource Manager for each resource in the *attribute flag word*. In addition, the Resource Manager provides two routines, GetResourceAttr and SetResourceAttr, to read and write a resource's attributes.

The Resource Editor provides an "Attr..." button in every resource editing window. Clicking this button displays the Resource Attributes dialog box with the current settings of that resource's attributes. Clicking the OK button in this dialog will update the resource's attributes to the new settings. Figure 4-2 shows the Resource Attributes dialog.



Resource Attributes

- [ ] Locked
- [ ] Fixed
- [ ] Resource Converter
- [ ] Write Protected
- [ ] Preload
- [ ] Do not cross bank in memory
- [ ] Do not use special memory
- [ ] Page Aligned

[✓ Purge Level 0] Purge Level 1 Purge Level 2 Purge Level 3

Figure 4-2
Resource Attributes Dialog

The meaning of each of the resource attributes is discussed in the following table.

Locked	If this attribute is set then the Resource Manager will call NewHandle to create a *locked* handle when allocating memory for the resource.
Fixed	If this attribute is set then the Resource Manager will call NewHandle to create a *fixed* handle when allocating memory for the resource.
Resource converter	This attribute indicates whether the representation of the resource as stored in the resource fork of a file must be converted to a different representation when read into memory. If the attribute is set, the resource must be converted.
Write protevcted	If this attribute is set, the resource is *write protected.* This means that an application cannot update the contents of the resource in the resource fork of a file.
Preload	If this attribute is set, the Resource Manager will automatically load the resource into memory with its resource file is opened. If a resource is not set to preload then it must be explicitly loaded to memory with the Resource Manager routine LoadResource.
Do not cross bank in memory	If this attribute is set then the Resource Manager will call NewHandle to create a handle which does not cross a bank of memory when allocating storage for the resource.
Do not use special memory	If this attribute is set then the Resource Manager will call NewHandle to create a handle which does not occupy *special* memory when allocating memory for the resource.
Page aligned	If this attribute is set then the Resource Manager will call NewHandle to create a *page aligned* handle when allocating memory for the resource.
Purge Level	The Resource Manager passes the purge level setting to NewHandle when

allocating memory for the resource.

- **Note:** The default setting for all resource attributes is NOT set and each has a default purge level of zero (0).

Pascal String Resource

The Pascal string string (rPString) stores a string of up to 255 ASCII characters. The string begins with an integer byte which is a count indicating the number of characters that follows in the resource. Pascal string resources are widely used by other resource types. For example, the resource types rMenuItem, rMenu, rWindParam1 and several of the rControlTemplate variations reference a rPString resource to store their titles.

Figure 4-3 illustrates the Resource Editor window used to display and edit a rPString resource.

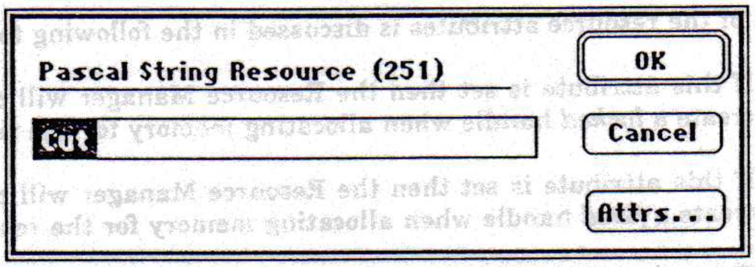


Figure 4-3
Pascal String Resource

C String Resource

The C string (rCString) stores a string of characters which are terminated by a zero byte. There is no restriction on the number of characters in the string resource. rCString resources are not used often in Complete Pascal programs because Pascal provides no natural mechanism to use these types of strings. However, the Resource Editor does support this resource type as shown in Figure 4-4.

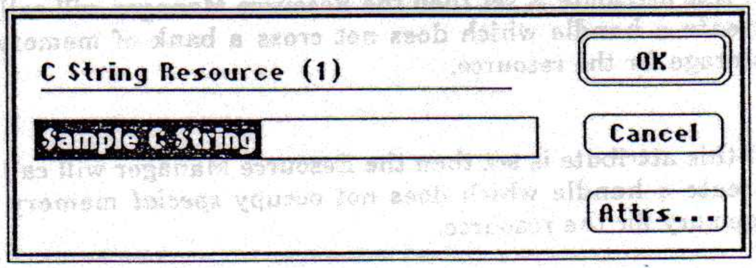


Figure 4-4
C String Resource

Alert String Resource

The Alert string (rAlertString) stores a string of characters which are terminated by a zero byte. The string is used with the AlertWindow function from the Window Manager toolset to display simple alert windows. The alert string resource stores the message to appear in an AlertWindow along with special codes which define the size of the window, whether or not an icon appears in the alert and the buttons.

For complete documentation of the format and structure of the alert string see the Window Manager chapter of the *Apple IIGS Toolbox Reference Update*. Figure 6-5 shows the window used by the Resource Editor for editing an alert string resource.

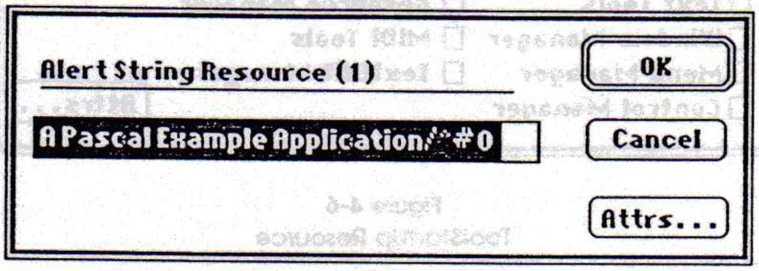
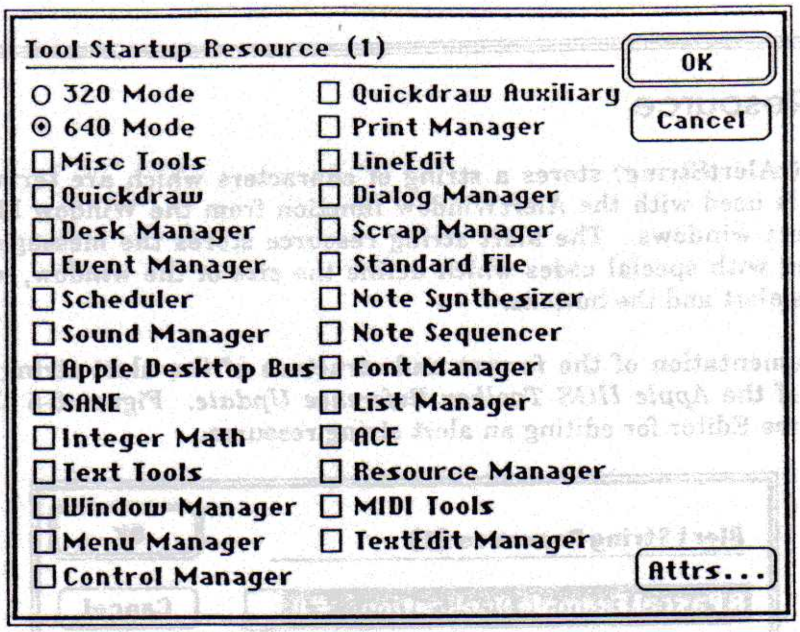


Figure 4-5
Alert String Resource

ToolStartUp Resource

The rToolStartup is used by an application to specify the Apple IIGS Toolbox toolsets which are required by the application and whether the application uses the 320 or 640 mode super hires graphics screen. Figure 4-6 illustrates the Resource Editor dialog for editing a rToolStartUp resource.



Tool Startup Resource (1)

OK

Cancel

( ) 320 Mode

[✓ 640 Mode]

☐ Misc Tools	☐ Quickdraw Auxiliary
☐ Quickdraw	☐ Print Manager
☐ Desk Manager	☐ LineEdit
☐ Event Manager	☐ Dialog Manager
☐ Scheduler	☐ Scrap Manager
☐ Sound Manager	☐ Standard File
☐ Apple Desktop Bus	☐ Note Synthesizer
☐ SANE	☐ Note Sequencer
☐ Integer Math	☐ Font Manager
☐ Text Tools	☐ List Manager
☐ Window Manager	☐ ACE
☐ Menu Manager	☐ Resource Manager
☐ Control Manager	☐ MIDI Tools
	☐ TextEdit Manager

Attrs...

Figure 4-6
ToolStartUp Resource

The rToolStartup resource is used with the new StartUpTools function and the ShutDownTools procedure provided in the Tool Locator toolset. These two routines used together with a rToolStartUp completely implement the work necessary to begin using the Apple IIGS Toolbox. These operations include:

1) Start the Resource Manager,
2) Open an application's resource fork,
3) Allocate the appropriate amount of direct page memory for the toolsets it uses,
4) Then start every toolset used by the application.

Menu Bar Resource

The Menu Bar (rMenuBar) resource is an ordered collection of Menu resources which define a menu bar. The resource is used by the NewMenuBar2 procedure in the Menu Manager toolset to create an application's menu bar. Figure 4-7 illustrates the Resource Editor window used to edit a rMenuBar resource.

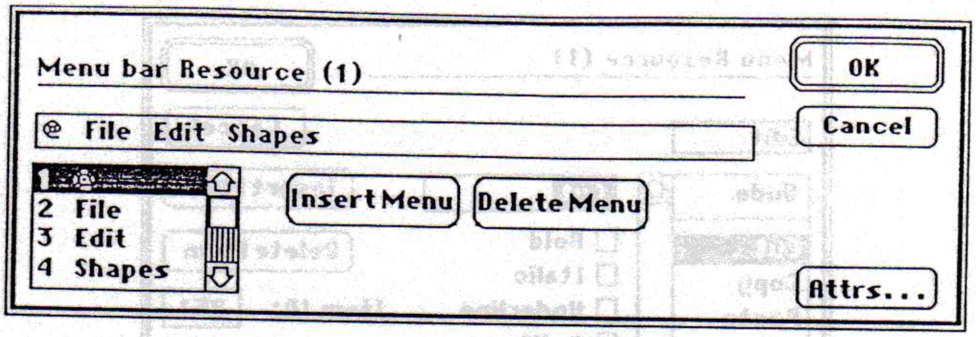


Figure 4-7
Menu Bar Resource

The long rectangle at the top of the dialog represents the menu bar currently defined by the resource. The scrollable list in the bottom left corner of the dialog is a complete list of every menu resource in the open resource file. The buttons Insert Menu and Delete Menu are used to add menus and delete menus from the menu bar. To add a new menu to the menu bar, first select the menu in the menu bar where the newly added menu should be placed after. Then, select the menu to add from the list of available menus, and finally click the Insert Menu button. To delete a menu from the menu bar, select the menu in the menu bar to delete then click in the Delete Menu button.

Menu Resource

The Menu (rMenu) resource is an ordered collection of MenuItem resources which define a menu. Menu resources are typically referenced by Menu Bar resources, but can also be used directly by the Menu Manager toolset routine NewMenu2.

The Menu resource is one of two *super resources* that the Resource Editor supports (the other is the Window resource). A Menu resource as defined by Apple Computer simply stores references to *other* resources. In particular, a Pascal string resource for the menu's title and then an ordered list of references to Menu Item (rMenuItem) resources. Each Menu Item resource then in turn references another Pascal string resource for its title. As you can see a small menu with only six menu items actually consists of 14 different resources. One for the Menu Resource, one for the menu title's Pascal string resource, and six Menu Item resources which reference six Pascal string resources for their titles. Clearly, creating the several menus that a typical application requires would be a very tedious task if each resource had to be individually created and referenced. Therefore, the Resource Editor groups all of these resource editing tasks into a single dialog to make creating and editing Menu resources easy. The Resource Editor provides no direct means of editing a Menu Item resource. Figure 4-8 illustrates the Menu resource editing dialog.

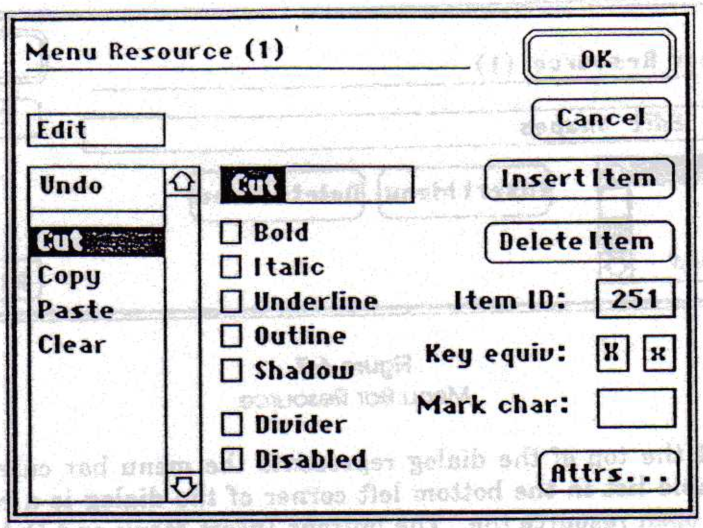


Figure 4-8
Menu Resource

The edit item at the top left of the dialog is the menu's title. The scrollable list in the left side of the dialog is the current list of menu items contained in the menu. To edit a particular menu item, simply click on its name in the list. When a menu item is selected, its name and settings are displayed in the several items in the right side of the dialog.

The Resource Editor allows you to specify any of the five type styles for drawing the menu item's title and whether the menu item has a divider line and/or is disabled. In addition, you can specify a command-key equivalent for the menu item and a mark character. The item ID is the value returned by TaskMaster or MenuSelect in an application when the user selects a menu item. The Resource Editor also uses this number as the resource ID for the Menu Item resource and its title's Pascal string resource.

Window Resource

The Window (rWindParam1) resource stores the necessary information to create a window on the Apple IIGS desktop using the NewWindow2 function in the Window Manager toolset. The Window resource defines the window's location, zoom size, title, frame definition and other attributes. In addition, the resource can reference a list content controls. Content controls can be buttons, check boxes, radio buttons, edit text, scroll bars, pop up menus, etc which appear in the content of a window.

The Window resource is the second of the two *super resources* the Resource Editor supports (the other is the Menu resource). Like the Menu resource, the Window resource can reference many other resources. A window can reference a Pascal string resource for its title, and if the window has content controls, it references a Control List (rControlList) resource. A Control List resource then references several Control Template (rControlTemplate) resources for each button, check box, radio button, edit text item, scroll bar, pop up menu, etc that appears in the content of the window. And further each Control Template may optionally reference a Pascal string resource for its title. Clearly, creating a Window resource would be a very tedious task if each of these items had to be created individually then referenced appropriately. Therefore, the Resource Editor groups all of these resource editing tasks into a single dialog to make creating and editing Window resources easy. The Resource Editor provides no direct means of editing a Control List or Control Template resource. Figure 4-9 illustrates the main Window resource editing dialog.

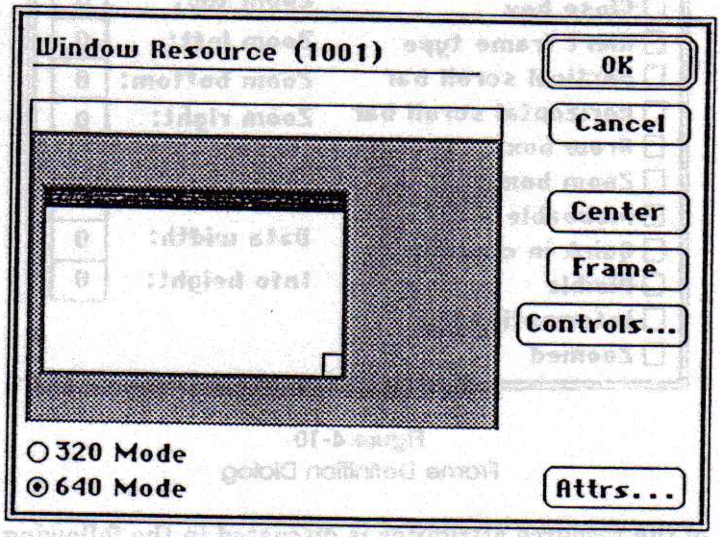
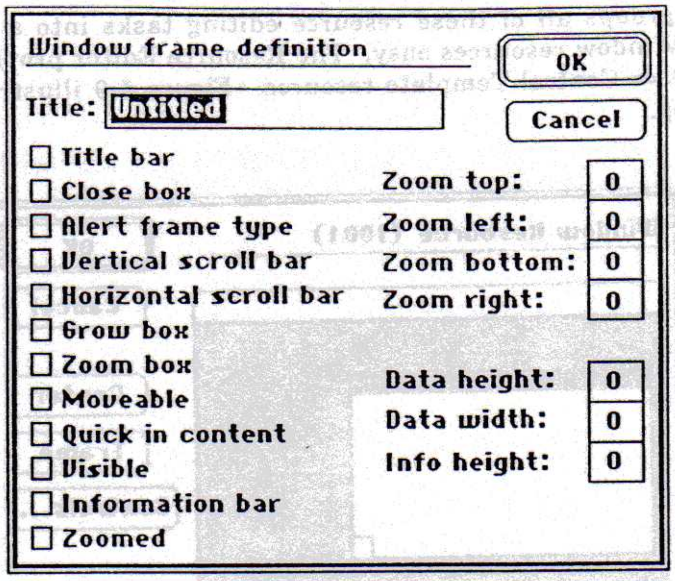


Figure 4-9
Window Resource

The large light blue area in the center of the dialog is used to represent a 50% scale of the Apple IIGS desktop with a window. The window represents the location and size of the window as defined by the Window resource. To change the location of the window, simply click in the content area of the window and drag it. To change the size of the window simply click in the size box in the lower right corner of the window and drag. The Center button can be used to quickly center the location of the window on the desktop. The 320 Mode and 640 Mode radio buttons are used to inform the Resource Editor what graphics screen mode will be used when the window is created. The Resource Editor uses this information to properly scale the window size in the dialog.
The Frame... button is used to invoke the Frame Definition dialog to display and edit the window's frame information. The Controls... button is used to invoke the Content Controls dialog to create and edit the content controls that belong in the window.

Window Frame Definition

The Frame Definition dialog, as shown in Figure 4-10, is used to define the window's frame attributes, zoom size, content data size and information bar height.



Title: Untitled

- [ ] Title bar
- [ ] Close box
- [ ] Alert frame type
- [ ] Vertical scroll bar
- [ ] Horizontal scroll bar
- [ ] Grow box
- [ ] Zoom box
- [ ] Moveable
- [ ] Quick in content
- [ ] Visible
- [ ] Information bar
- [ ] Zoomed

Zoom top: 0
Zoom left: 0
Zoom bottom: 0
Zoom right: 0

Data height: 0
Data width: 0
Info height: 0

Figure 4-10
Frame Definition Dialog

The meaning of each of the resource attributes is discussed in the following table.

Table 4-3
Resource Attribute Meanings

Attribute	Meaning
Title bar	Set if the window has a title bar.
Close box	Set if the window has a close box.
Alert frame type	*Set if the window should be drawn with an alert style frame instead of the* standard document style frame. An alert frame is typically used for modal dialogs.
Vertical scroll bar	Set if the window has a vertical scroll bar.
Horizontal scroll bar	Set if the window has a horizontal scroll bar.
Grow box	Set if the window has a grow box.
Zoom box	Set if the window has a zoom box.
Moveable	Set if the window's title bar is to be treated as the drag region for moving the window on the desktop.
Quick in content	Set if a mouse click in the content region of a window (not in the front) should select the window as well as be treated as click in the content.
Visible	Set if the window is visible when created.

Information bar	Set if the window has an information bar.
Zoomed	Set if the window is initially zoomed.
Zoom rect	Defines the top, left, bottom and right coordinates of the window when zoomed.
Data height/width	Defines the size in pixels of the window's height and width.
Info height	Defines the height of the information bar if the window has one.

- **Note:** The default setting for newly created window's frame definition are as displayed in Figure 6-10.

Window Controls Definition

The Window Controls dialog as shown in Figure 4-11 is used to define the window's content controls. The content controls are the simple button, check box, radio button, scroll bar, size box, static text, picture, icon button, line edit, text edit, pop-up menu, and list. To create a new control simply click the mouse on the control palette in the left side of the dialog box for the control type desired then click the mouse in the window content area in the position where the control should be placed.

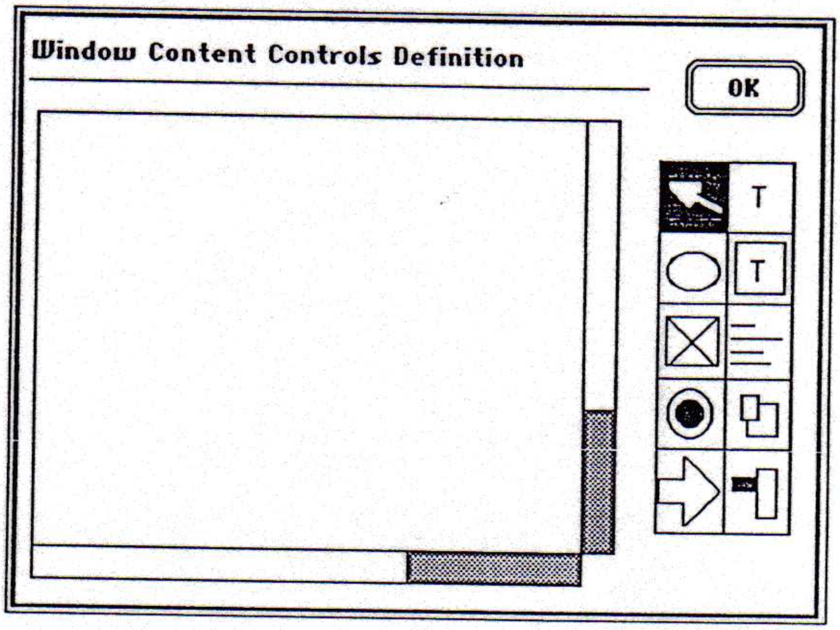


Figure 4-11
Window Content Control Definition Dialog

Chapter 5

Complete Pascal Menu Reference

This chapter provides a complete reference to the commands available and contained in each of Complete Pascal's seven menus. Complete Pascal's seven menus are the **Apple, File, Edit, Search, Windows, Compile,** and **GSOS** menus. Recall that most menu commands can be issued by entering *command-key equivalents* rather than clicking the mouse on the menu and releasing it over the menu command. A discussion of command-key equivalents is provided at the end of this chapter.

The Apple Menu

The **Apple** menu is a standard menu for Apple IIGS desktop applications such as Complete Pascal, and is always the first menu in the menu bar. In Complete Pascal, the **Apple** menu has two parts: the **About Complete Pascal...** command and the list of installed new desk accessories (NDAs) available in Complete Pascal. Because, the list of desk accessories depends upon which desk accessories are installed on your particular system disk, Figure 5-1 may not match your menu exactly.

About Complete Pascal... ?

The **About Complete Pascal...** menu item displays the About Pascal Dialog Box. The dialog contains Complete Technology's' address and phone number. More importantly, the version of Complete Pascal you are using and Complete Technology's' copyright notice also appear in the About Pascal Dialog Box.

Desk Accessories

The desk accessory menu items represent each of the NDAs installed on your system disk. Recall that desk accessories must be properly installed on your bootable system disk to be available. For a desk accessory to be properly installed, it must be in the DESK.ACCS folder which can be found in the SYSTEM folder. Selecting a desk accessory name from the Apple menu will cause that desk accessory's window to be opened on the Complete Pascal desktop.

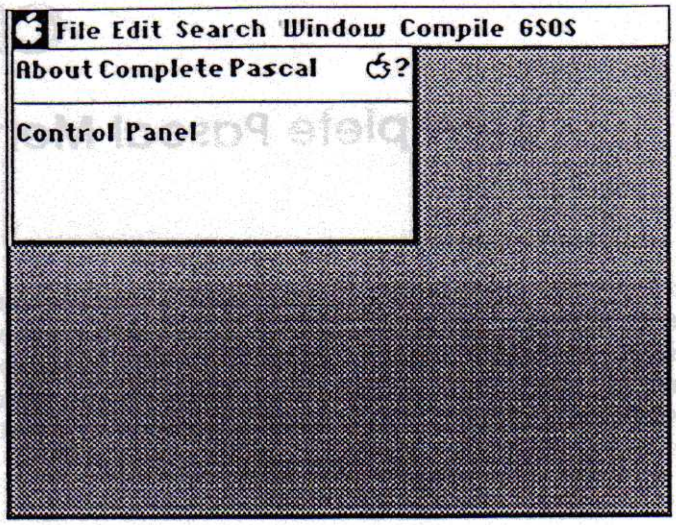


Figure 5-1
Apple Menu

Open "NDA"

The Open "NDA" menu item is only included in the Apple menu when you have compiled the source code of a New Desk Accessory program to memory. The actual name of the menu item will be the word Open followed by the name of the desk accessory that was compiled. The Open "NDA" command works just like any of the other desk accessory menu items, that is, it opens the desk accessory causing it to display its window.

Remove "NDA"

Like the Open "NDA" menu item, the Remove "NDA" menu item is only included in the Apple menu when you have compiled the source code of a New Desk Accessory program to memory. The Remove "NDA" menu item will close the desk accessory if its window is open. It then removes the desk accessory's compiled code from memory and deallocates any memory the desk accessory may have allocated. After the desk accessory has been removed from memory, the Open "NDA" and Remove "NDA" menu items are removed from the Apple menu.

The File Menu

The File menu contains the various file related commands (Figure 5-2) in Complete Pascal. The menu items are grouped into three basic categories: accessing disk files, printing, and exiting Complete Pascal. Following is a description of each menu item contained in the File menu.

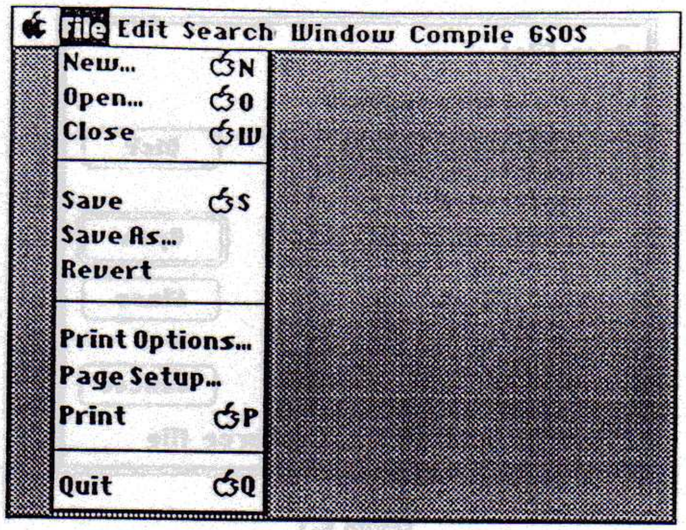


Figure 5-2
File Menu

New

This item opens a new source code text window or resource window. The new window becomes the active window ready for editing. Before the window is created, Complete Pascal asks that you name the file associated with the window and specify whether the window is a text or resource editing window. Complete Pascal allows as many windows to be opened as available memory permits.

Open

The **Open** menu item displays the Open File Dialog Box (Figure 5-3) allowing you to select a file for editing or compiling.

Close

This menu item closes the active (topmost) editing window. If the source code contained in the active window has had changes made to it since last opened, you are prompted to save the changes you have made.

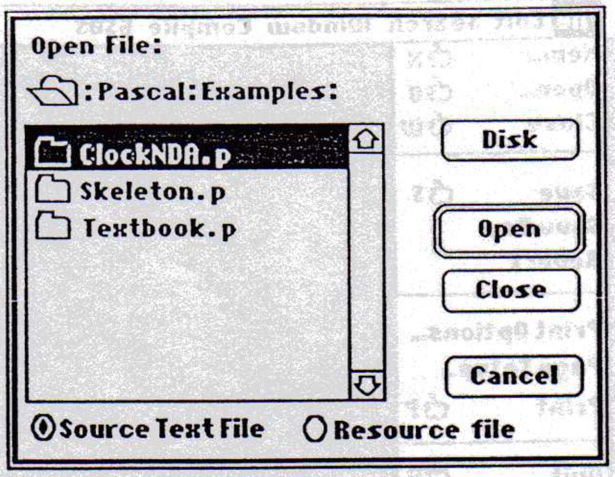


Figure 5-3
Open File Dialog Box

Save

The Save menu item saves the contents of the active window. The original file on disk is overwritten by the contents of the current window.

Save As...

Selecting this menu item allows you to save the contents of the active editing window to a new disk file. To do this, you are again prompted with the Put File Dialog Box to name a file for this window. If the filename you choose already exists in the specified subdirectory, you will be warned of this and asked if you wish to replace the existing file. Save As is not available for resource files.

Revert

This menu item will cause all of the editing changes you have made to be replaced with the most recently saved version of the file. You will be asked to confirm this choice before the operation is performed. Revert is not available for resource files.

Print Options...

This menu item displays the Print Options Dialog Box (Figure 5-4).

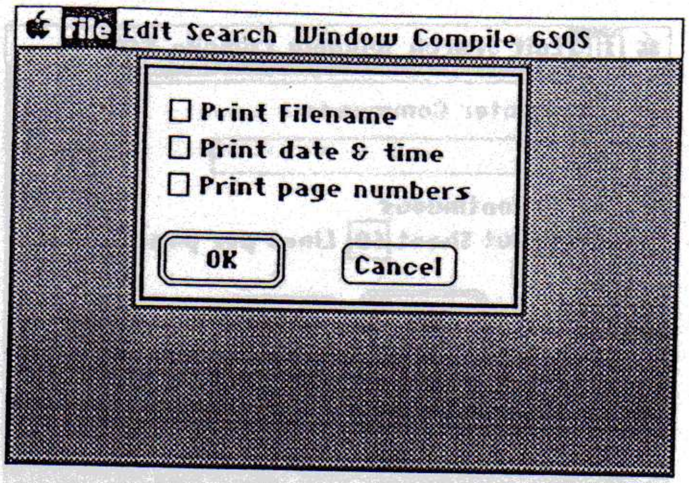


Figure 5-4
Print Options Dialog Box

When Complete Pascal prints a file to the printer, it optionally prints a header across the top of every page. The header can include the name of the file (Print Title), the current date and time (Print Date/Time), and page numbers (Print Page Numbers). If an option is checked, Complete Pascal prints the corresponding information in the header. If none of the options are selected, a header is not printed.

Page Setup

This menu item displays the Page Setup Dialog Box (Figure 5-5).

The Page Setup Dialog Box is used to configure the way Complete Pascal prints a page. There are two options: Continuous and Cut Sheet. If Continuous is selected, a header is only printed on the first page, and no blank lines are printed at the end or beginning of a piece of paper. This option maximizes the number of lines that can be printed on a page. However, if the paper is misaligned, a line of text may print on the perforation in the paper.

If the Cut Sheet option is used, a header is printed at the top of every page, and blank lines are printed at the end and beginning of every page. When this option is selected, the number of lines per page must be set. The default setting is for standard 8 1/2 by 11 inch paper.

Finally, the Page Setup Dialog Box allows you to enter a special character sequence which represents a Printer Command. The character sequence is sent to the printer before printing every file. The Printer Command can be used to instruct a printer to use a special built-in font, font size, page size, etc. In order to send a control character to the printer use the caret character (^) followed by the appropriate letter that defines the control character. For example, ^[sends an ASCII 27 (an escape character).

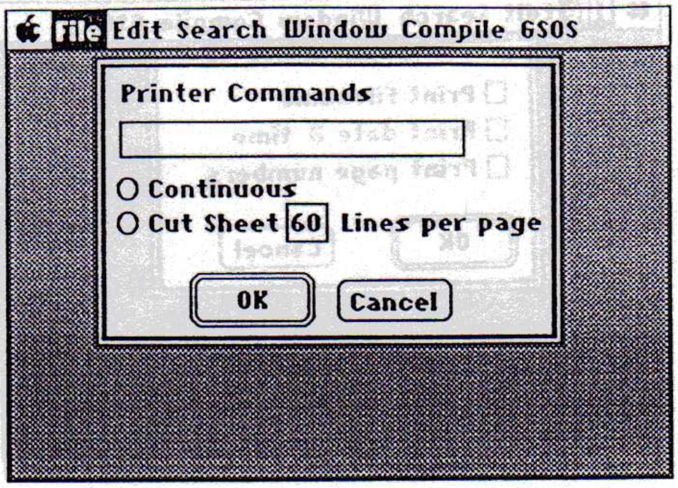


Figure 5-5
Page Setup Dialog Box

Print

The **Print** menu item causes the contents of the active window to be printed to the printer through the currently selected serial port (slot). The text is printed using the built-in font of the printer. Complete Pascal does not use the Apple IIGS Print Manager for printing.

If the Option key is held down when choosing this command, Complete Pascal prints the currently selected text in the active window rather than the entire contents. This is especially useful when editing large files.

Quit

Selecting **Quit** closes all open windows, allowing you to verify whether changes made to each window should be saved, and then exits back to the Apple IIGS Finder.

The Edit Menu

The Edit menu contains several useful editing commands. The menu is in the standard Apple IIGS format thus allowing it to be used with desk accessories. See Figure 5-6.

Undo

The Undo menu item allows you to undo the last editing change made in a source code edit window. Note that after you complete an editing operation and begin a new one, the previous edit operation can not be *undone*. Every new editing operation starts a new undoable operation and the ability to undo past edits are lost.

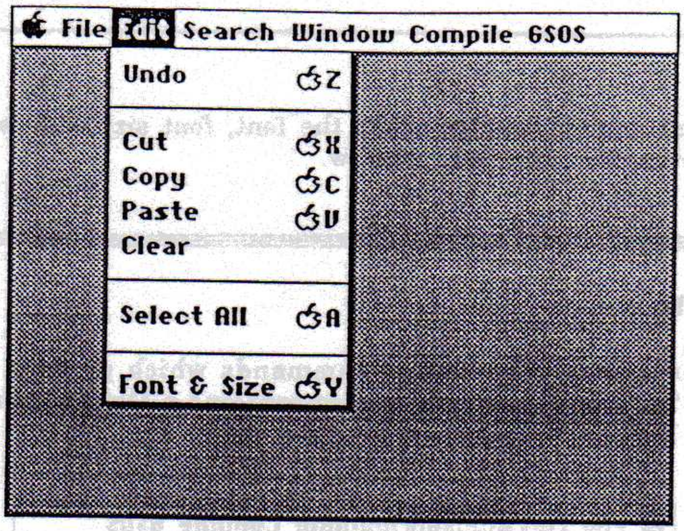


Figure 5-6
Edit Menu

Cut

This command cuts the currently selected text. The operation deletes the selected text from the active window and saves it into the clipboard.

Copy

This command copies the currently selected text, but does not delete it from the active window, and saves it into the clipboard.

Paste

The **Paste** menu item copies the contents of the Complete Pascal clipboard into the active window at the current insertion point. If text is currently selected then it is deleted before the paste is performed.

Clear

The **Clear** menu item deletes the range of selected text from the active window, but does not save it into the clipboard.

Select All

This command selects all text contained in the active window. It is a shortcut for selecting all text rather than moving to the beginning of the text, clicking the mouse, and then moving to the end of the text and shift-clicking.

Set Font & Size

The Font & Size menu item is used to select the font, font size and tab width used by the Complete Pascal Editor for the active edit window.

The Search Menu

The **Search** menu contains a collection of commands which perform search and replace operations (Figure 5-7). The **Search** menu also contains a Goto command that scrolls the window content to the location of the current insertion point.

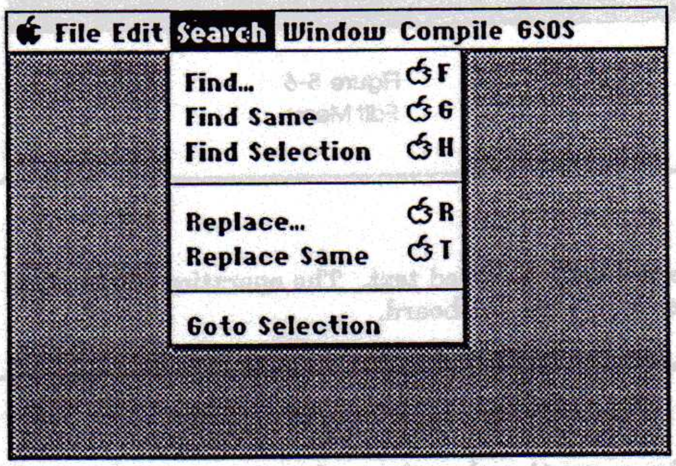


Figure 5-7
Search Menu

Find...

This menu item displays the Complete Pascal Find Dialog Box allowing you to specify a search string. Upon entering a search string and selecting the Find button in this dialog, the search begins from the current insertion point (not the beginning of the file).

Find Next

This command searches forward in the active window, from the current insertion point, for the next occurrence of the **Find...** string specified in the Find Dialog Box. Upon locating the next occurrence, the active window scrolls to display the string. If no occurrence of the string is found an error message is displayed.

Find Selection

This command searches forward in the active window, from the current insertion point, for the next occurrence of the current selection in the window. Upon locating the next occurrence, the active window scrolls to display the string. If no occurrence of the string is found an error message is displayed.

Replace...

This menu item displays the Complete Pascal Change Dialog Box allowing you to specify a search string and a replacement string. Upon entering a search string and selecting the Find button in this dialog, the search begins from the current insertion point (not the beginning of the file). When the string is found is replaced with the substitution string. Upon locating the next occurrence, the active window scrolls to display the string.

Replace Same

This command searches forward in the active window, from the current insertion point, for the next occurrence of the **Replace...** string specified in the Replace Dialog Box. Upon locating the next occurrence,the text string is changed and the active window scrolls to display the string.

Goto Selection

This command scrolls the active window so that the insertion point (or currently selected text) is visible in the window.

The Windows Menu

The **Windows** menu provides several commands to arrange the open windows within the Complete Pascal desktop and obtain information about the windows. The Windows menu also contains the name of every open window on the desktop. Selecting the name of a window in the Windows menu brings that window to the front. Figure 5-8 shows the contents of this menu.

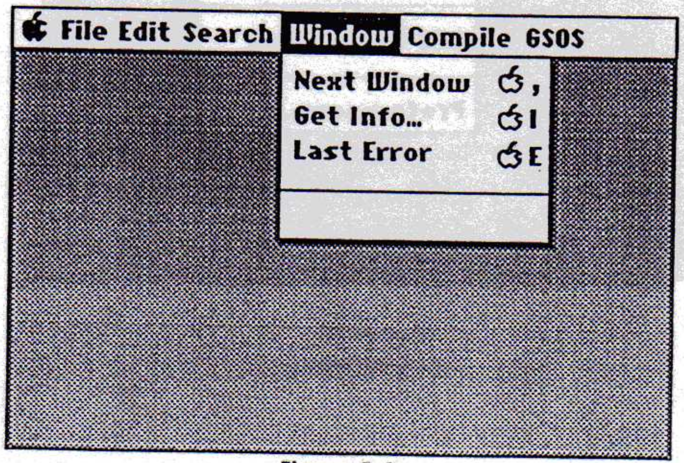


Figure 5-8
Windows Menu

Next Window

The Next Window menu item places the active window in back of all other open windows on the screen and brings the window directly behind the previously active window to the front. This command provides an easy method of switching between windows when it might not be possible to click on a window because it is completely covered by another window.

Get Info

The Get Info command displays a File Information Dialog Box. The dialog box displays the following information about the active editing window: the full pathname for the file associated with the editing window, its size in bytes and the number of lines.

Last Error

This command displays the Complete Pascal Error Dialog Box, displaying the most recently encountered error.

The Compile Menu

The **Compile** menu (Figure 5-9) contains the commands which invoke the Complete Pascal compiler. When invoking the compiler, the contents of the active editing window are compiled. Also included are the Add Resources... command and the **Preferences...** command.

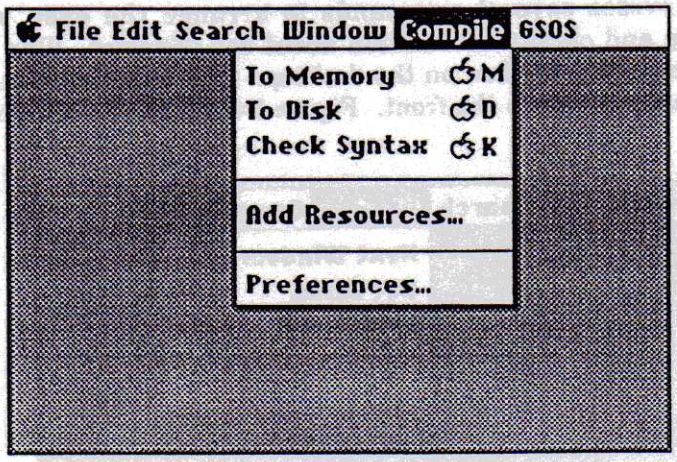


Figure 5-9
Compile Menu

To Memory

This command invokes Complete Pascal to compile the source code contained in the active editing window. If the compilation completes successfully and the active window contains a program which is an application (rather than a unit), the state of Complete Pascal, including all open windows, is saved and control is transferred to the compiled application. Upon quitting the compiled program, you are returned to Complete Pascal with all of your windows intact.

If the contents of the active window is a unit rather than a program then there is no program to run and, therefore, no transfer of control out of Complete Pascal. Instead, the compiled code for the unit is retained in memory so that other units and programs which use the unit will have access to its code.

To Disk

The **To Disk** menu item invokes the Complete Pascal compiler to compile the contents of the active window to disk creating a stand-alone GS/OS application file.

If the source code contained in the active window is a program then Complete Pascal creates an *S16* (filetype $B3) application load file in the same directory as the source code. On the other hand, if the source code is a unit then the unit's symbol table and code are saved to a ".p.o" file in the same directory as the source code.

Check Syntax

The **Check Syntax** command invokes the Complete Pascal compiler only to verify that the source code contained in the active window consists of legal Pascal statements.

Add Resources...

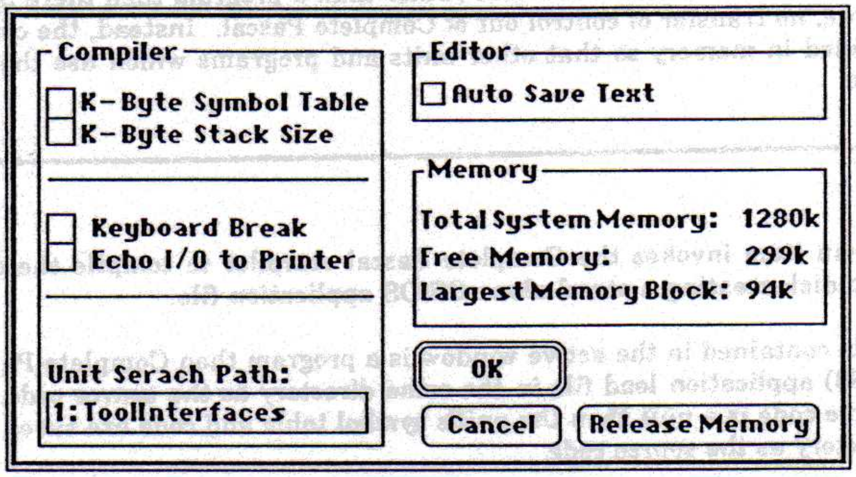
The **Add Resources...** command is used to attach resources created by the Resource Editor to an application. When the menu item is selected it presents an Open File dialog so that you can designate the resources which belong to the application whose source code is in the topmost editing window. When the source is compiled to memory or disk, the Complete Pascal Linker copies the resources in the specified file into the compiled application's resource fork.

Preferences...

Selecting the **Preferences** command displays the Preferences Dialog Box. The Preferences Dialog Box is used to configure the Complete Pascal editor and compiler to your particular needs. The information presented in the dialog is grouped into three major categories: Compiler, Editor, and Memory. In addition, there are three buttons: OK, Cancel and Release Memory. The Preferences Dialog is displayed in Figure 5-10 with its default settings. The next several paragraphs describe each component of the dialog in detail.

Before discussing each component of the Preferences Dialog, an explanation of *edit text items* and *check boxes* is in order. An edit text item is an item contained in the Preferences Dialog which requires input to determine a components value, whereas a check box acts as an on/off switch. These two mechanisms are the means by which you modify each component of the Preferences Dialog Box.

Simply position the cursor over an edit text item, click once and begin typing to enter the value for its component. Check boxes, on the other hand, are modified by positioning the cursor over the check box and clicking the cursor once to toggle between on and off states (a check representing "on").


Compiler
☐ K-Byte Symbol Table
☐ K-Byte Stack Size
☐ Keyboard Break
☐ Echo I/O to Printer
Unit Serach Path: 1:ToolInterfaces

Editor
☐ Auto Save Text

Memory
Total System Memory: 1280k
Free Memory: 299k
Largest Memory Block: 94k

Figure 5-10
Preferences Dialog Box

K-byte Symbol Table This option allows you to specify the amount of memory the Complete Pascal compiler should allocate for a *symbol table.* A symbol table is the data structure the compiler uses to store the declarations of labels, variables, arrays, procedures and functions. For most all programs, the default size of 12K bytes is sufficient. However, larger programs may encounter the Compiler error Symbol Table Space Exhausted. If compiling a program encounters this error, then you should increase the value of this setting. 32K bytes is the largest setting allowed. This setting can also be lowered if you are running short of memory and are compiling small programs. The smallest allowable value is 2K bytes.

K-byte Stack Complete Pascal programs require a data structure known as the *Runtime Stack.* The default value of a 8K byte stack should be sufficient for most Complete Pascal programs. This value can be changed from 1K to 32K bytes. The Stack size may also be changed by using the $StackSize directive.

Keyboard Break The Keyboard Break option is used to implement the Control-C abort mechanism.

If this option is turned on, Complete Pascal generates code between each statement to check if the control-C character has been typed. If this option is turned off, it is impossible to abort the execution of a Complete Pascal program. The only way to do so is to reset the Apple IIGS. If you do not use the Control-C abort mechanism you should turn this option off so that your programs will be smaller and faster.

This option may also be turned off and on using the $KeyboardBreak metastatement. See Appendix B.

Unit Search Path The pathname specified here is where the Complete Pascal compiler searches for unit files which have been specified in a USES clause. The default value for this option is 1:ToolInterfaces: which specifies the ToolInterfaces folder in the same directory as the Complete Pascal compiler. This is the folder which contains all of the Complete Pascal predefined units for accessing the Apple IIGS Toolbox. Recall that Complete Pascal first searches in the current source code folder first for a unit file and then the path specified by this option.

Auto Save Text The Auto Save option allows you to specify whether or not changes to any of the editing windows should be automatically saved before Complete Pascal transfers control to a compiled to memory application. You should select this option if your program is in the early stages of development and might cause the Apple IIGS to crash when run. If this option is on you will never lose any editing changes you have made, but not explicitly saved, however, it does slow down the compile cycle since it must write to the disk.

Total System Memory Obviously, this value can not be changed while Complete Pascal is running. The total system memory is displayed for informational purposes only. The value represents, in kilobytes, the amount of RAM memory installed in your machine.

Free Memory This number indicates how much memory is currently available. This number is important because, it reflects whether or not Complete Pascal has enough memory to open a new program file, compile a program to memory, etc. Because Complete Pascal retains various pieces of information in memory, this number can sometimes be increased by selecting the Release Memory button described below.

Largest Memory Block This value indicates the largest block of memory available for use by Complete Pascal.

OK Button Clicking the mouse in this button (or pressing the Return key) indicates that you want Complete Pascal to accept all the changes to the Preferences dialog you have made. After choosing this button, the dialog disappears and Complete Pascal updates all options and settings.

Cancel Button Clicking the mouse in this button causes Complete Pascal to remove the dialog, and to ignore any changes made to the options and settings and to leave them as they were before the dialog was opened.

Release Memory This button is used to release all memory associated with programs and units that have been compiled to memory or loaded to memory by a USES clause. Selecting this button will usually adjust the Free Memory and Largest Memory Block values. Clicking the mouse in this button does not cause the dialog box to close.

The GSOS Menu

The GSOS menu provides access to three GSOS commands (Figure 5-11).

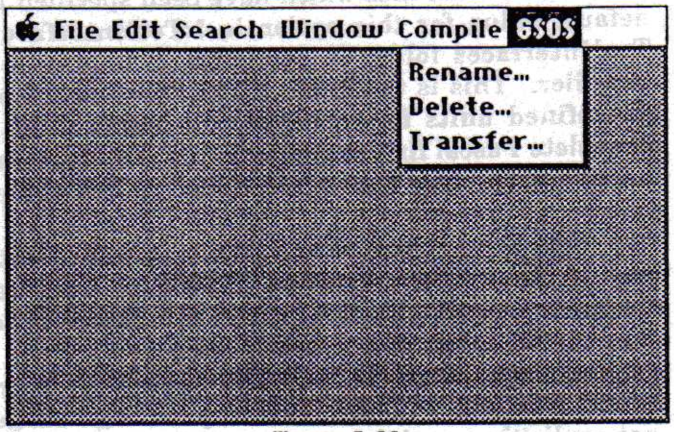


Figure 5-11
GSOS Menu

Rename

The **Rename** command displays the Rename File Dialog Box allowing you to choose the file you would like to rename. After selecting a file, you are prompted to provide the new filename. The new filename must be a legal GS/OS filename otherwise an error results.

Delete

The **Delete** command displays the Delete File Dialog Box allowing you to choose a file you would like to delete. After selecting a file, you are prompted to confirm that, in fact, you would like to delete the file. If you confirm that the file should be deleted, Complete Pascal will permanently delete the file from the disk.

Transfer

The **Transfer** command displays the Transfer Dialog Box allowing you to choose an application you would like to transfer control. Upon selecting an application, Complete Pascal asks you to save any changed files and then automatically quits and invokes the specified application without returning to the Apple IIGS Finder. The only way to return to Complete Pascal is to launch it from the Finder again.

Chapter 6

Textbook Applications

Textbook applications are complete stand-alone, double-clickable applications that can be run from the Apple IIGS Finder or, for that matter, even designated as a startup application. What makes Textbook applications special is that they require no knowledge of the Apple IIGS Toolbox. In addition, Textbook programs represent typical applications one might find in a Pascal textbook. Hence, the name Textbook applications. Textbook applications are the easiest type of programs to create with Complete Pascal.

Complete Pascal actually supports two types of Textbook applications: text screen and graphics screen applications. Text screen applications execute using the 24 row by 80 column text screen display of the Apple IIGS, while graphics screen applications execute in either the 320 or 640 mode super hires graphics screen display. For discussion purposes, text screen Textbook applications are sometimes referred to as simple *Textbook applications,* while graphics screen Textbook applications are sometimes referred to as *Graphics Textbook applications.*

This chapter discusses the fundamentals of setting up a simple text screen application. Chapter 7, on the other hand, provides an explanation of a graphics screen textbook application.

Text Screen Textbook Applications

Text screen applications are the default type of Textbook applications. And, as mentioned above the text screen application executes using the Apple IIGS 24 row by 80 column text screen display. As such, text screen programs do not require any knowledge of the Apple IIGS Toolbox. Additionally, text screen applications cannot support any type of graphic display.

The following "Hello World" program illustrates the basic structure of a text screen Textbook application. When this program is executed it will display the message "Hello World" on the first line of the text screen, wait for the user to press the Return key and then terminate.

```
PROGRAM HelloWorld;
BEGIN
        Writeln('Hello World');
        Readln;
END.
```

The Complete Pascal distribution disk contains a larger text screen application named Textbook.p.

Chapter 7

Graphics Textbook Applications

Chapter 6 introduced the most basic of the five types of programs Complete Pascal can create – the Textbook application. This chapter shows how this very simple type of program can be easily extended to support graphics, hence the name Graphics Textbook applications.

Graphics Screen Textbook Applications

While text screen Textbook applications are simple to write and require absolutely no knowledge of the Apple IIGS Toolbox, they are strictly limited to text input and output only.

In contrast, graphics screen Textbook applications are implemented by Complete Pascal in order to provide for graphics output by your Pascal programs while maintaining the simplicity of text screen Textbook applications. Namely, as little knowledge of the Apple IIGS Toolbox as necessary. Graphics screen Textbook applications are a good place to start programming if you would like to begin implementing limited amounts of graphics rather than having to understand all of the complexities introduced with the Apple IIGS Toolbox.

Graphics screen Textbook applications use the Apple IIGS super hires graphics screen in either 320 or 640 mode. Thus, the Apple IIGS QuickDraw toolset can be used to create text, lines, rectangles, ovals, polygons and any of the other graphic primitives supported by QuickDraw. In addition, the Apple IIGS Event Manager is initialized and the mouse is activated.

The Graphics Procedure

Graphics screen Textbook applications are implemented using the predefined Complete Pascal Graphics procedure (see Chapter 20).

```
      PROCEDURE Graphics(screenMode: Integer);
```

The procedure has one parameter used to indicate whether the 320 or 640 mode super hires screen should be used and should be called at the beginning of a program. The Graphics procedure initializes and starts the Apple IIGS QuickDraw toolset and turns on the super hires graphics screen using the designated mode. The graphics screen is initialized as all white and does not contain a menu bar or any windows.

The following example illustrates how the Hello World program shown in the previous section can be rewritten as a graphics screen application. Note that the only addition is a call to the Graphics procedure.

```
PROGRAM HelloWorld;
BEGIN
        Graphics(640);
        Writeln('Hello World');
        Readln;
END.
```

Of course, this program does not use any of the QuickDraw graphic routines. However, it is important to notice that the standard Pascal `Readln` and `Writeln` routines still operate in the graphics screen. In order to begin adding some degree of graphic operation, the program must use the Types and QuickDraw units (see Chapter 8 and Appendix C).

The following program illustrates how graphics might be added to our simple Hello World program.

```
PROGRAM GraphicHelloWorld;
USES    Types,
        QuickDraw;
VAR     aRect:    Rect;
BEGIN
        SetRect(aRect,10,10,30,40);
        FrameRect(aRect);

        OffsetRect(aRect,25,30);
        FrameOval(aRect);

        OffsetRect(aRect,25,30);
        FrameRRect(aRect,20,20);

        MoveTo(50,30);
        DrawString('Hello World');
END.
```

The Complete Pascal distribution disk contains a larger graphics screen Textbook application named Graphics.p.

Chapter 8

Desktop Applications

Like Textbook applications, *Desktop* applications are complete stand-alone, double-clickable applications that can be run from the Apple IIGS Finder. However, the similarities between the two types of applications stop there. Desktop applications are those which take full advantage of the Apple IIGS Toolbox, use menus, windows, dialogs, the mouse, etc., and implement the Apple Desktop metaphor for their user interface. Desktop applications require at least a basic understanding of how to program with the Apple IIGS Toolbox.

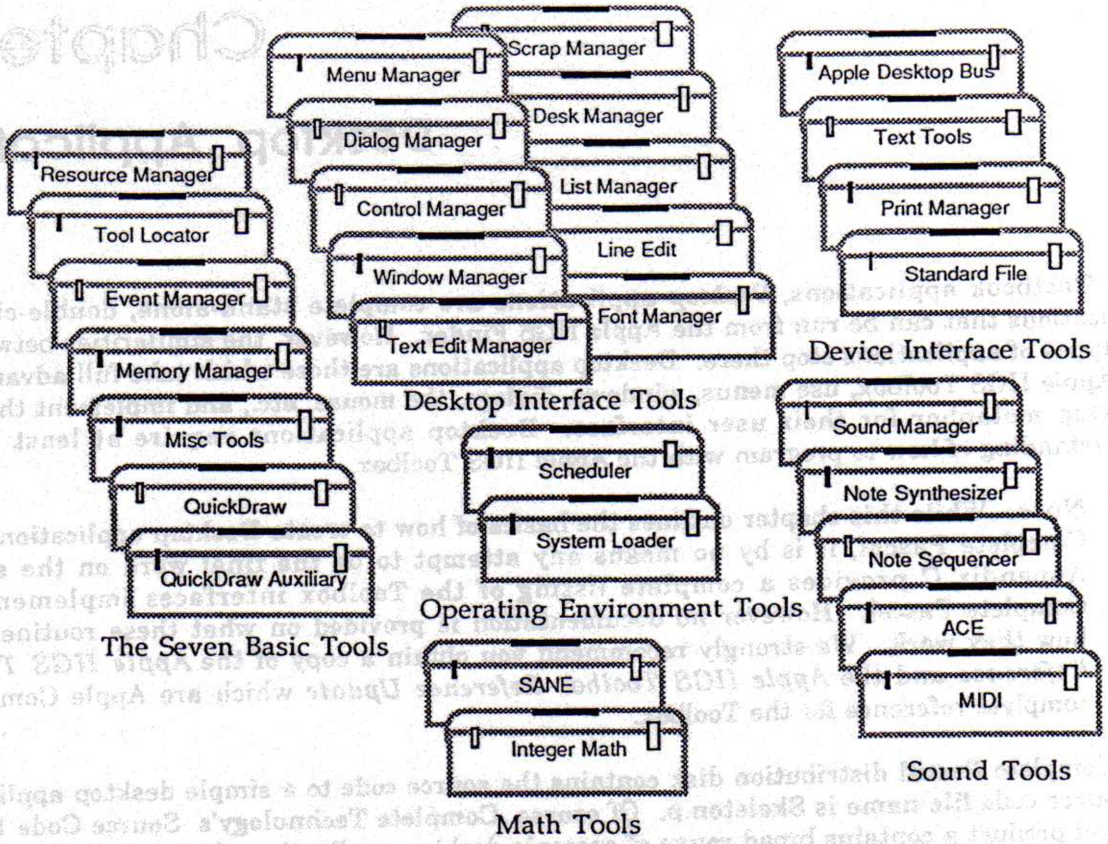
> v **Note:** While this chapter outlines the basics of how to create Desktop applications using Complete Pascal, it is by no means any attempt to be the final word on the subject. Appendix C provides a complete listing of the Toolbox interfaces implemented by Complete Pascal. However no documentation is provided on what these routines do or how they work. We strongly recommend you obtain a copy of the *Apple IIGS Toolbox Reference* and the *Apple IIGS Toolbox Reference Update* which are Apple Computer's complete reference for the Toolbox.

The Complete Pascal distribution disk contains the source code to a simple desktop application. Its source code file name is Skeleton.p. Of course, Complete Technology's Source Code Library - Pascal product a contains broad range of example desktop applications for you to study.

The Apple IIGS Toolbox

The Apple IIGS Toolbox is a large collection of software which gives the Apple IIGS its character. The toolbox is organized into several functional groups called toolsets, and within each toolset are numerous functions. Complete Pascal provides complete support for the Apple IIGS System Software version 5.0 including its new additions to the toolbox.

The toolbox which is part of System Software version 5.0 consists of 30 toolsets. The toolsets may be grouped into six major functional categories: the seven basic tools, the desktop interface tools, the device interface tools, the operating environment tools, the sound tools and the math tools. The following diagram illustrates the functional organization of the toolsets.



The Seven Basic Toolsets

The seven basic toolsets provide the framework upon which all of the other tools are built upon. All of these tools are used in event-driven programs.

Tool Locator	**The Tool Locator is the most important of the Apple IIGS toolsets. The Tool Locator allows you to load toolsets from disk into RAM and is responsible for locating a toolset routine when a program calls a Toolbox procedure or function.**
Memory Manager	**The Memory Manager is the second most important toolset. This tool is entirely responsible for the allocation, deallocation, and repositioning of** memory blocks on the Apple IIGS. The Memory Manager keeps track of how much memory is free and what parts are allocated and to whom. Whenever a program needs memory, it must ask the Memory Manager to allocate it.
Miscellaneous Tools	The Miscellaneous Tools consist mostly of system-level routines that must be available to most other toolsets.
QuickDraw	QuickDraw is the toolset that controls the graphics environment of the Apple IIGS and draws simple objects and text in the Super Hi-Res graphics screen. All other tools which create graphical objects such as the Menu and Window Manager call the QuickDraw toolset.

QuickDraw Auxiliary	This tool contains additional graphics routines which complement the QuickDraw toolset.
Event Manager	The Event Manager is responsible for detecting system events such as mouse-clicks, keystrokes, window updates, etc. It queues the events and then delivers the events to an application as requested.
Resource Manager	The Resource Manager is responsible for implementing access to and manipulation of the *resource fork* of a GS/OS file.

Desktop Interface Tools

The toolsets in this group support the Apple Desktop Interface. The desktop interface is the visual interface between the user of an application and the computer. It includes the menu bar and the blue colored area on the screen. Applications usually have documents on the desktop displayed in windows and perhaps other graphic objects such as icons. Applications implementing the desktop will always use the Menu, Window and Control managers, and usually most of the others as well. New Desk Accessories are supported by the Desk Manager.

Control Manager	The Control Manager consists of all the routines necessary to manipulate controls. Examples of controls include scroll bars, radio buttons, check boxes, etc.
Desk Manager	The Desk Manager is the tool which enables an application to support both classic desk accessories and new desk accessories.
Dialog Manager	The Dialog Manager provides the routines which allow an application to create and use both dialog boxes and alerts as a means of communication between a user and your program.
Font Manager	The Font Manager is the toolset which allows an application to make use of different text fonts, font styles, etc. within QuickDraw.
Line Edit	Line Edit is used to display and edit a line of text on the screen and allow a user to edit the text.
List Manager	The List Manager is used to create, display and allow selection of a variable amount of similar data.
Menu Manager	The Menu Manager controls and maintains the use of pull-down menus and items in the menus.
Scrap Manager	The Scrap Manager implements the desk scrap, which implements the Cut, Copy, and Paste operations of an application.
Text Edit Manager	The Text Edit Manager implements a multi-line text editing tool which supports multiple fonts and font sizes, rulers and more.
Window Manager	The Window Manager creates the desktop environment and is responsible for the creation and manipulation of windows.

Device Interface Tools

The toolsets in this group are used to manage input and output between the computer and peripheral devices and a program.

Apple Desktop Bus	The Apple Desktop Bus is a method and a protocol for connecting input devices, such as keyboards and mice with the Apple IIGS. The routines in this toolset are used to send commands and data between the Apple Desktop Bus Microcontroller and the rest of the system.
Print Manager	The Print Manager allows an application to use QuickDraw routines to print text and graphics to an ImageWriter or LaserWriter.
Standard File	The Standard File toolset implements the standard user interface for specifying a file to be opened or saved.
Text Tools	The Text Tools provide an interface between the Apple II character device drivers, which must be executed in emulation mode, and applications running in native mode.

Operating Environment Tools

The operating environment tools control the interaction between low-level hardware and software functions. While not listed here, the Memory Manager and Miscellaneous Tools toolsets implement similar low-level operations characteristic of the Operating Environment tools, and in many cases interact with these toolsets.

Scheduler	The Scheduler delays the activation of a desk accessory or other system task until the resources which that task/desk accessory requires become available. This avoids potential system crashes when more than one task attempts to use the same resource at the same time.
System Loader	The System Loader is responsible for loading and relocating code for applications and desk accessories to memory.

Sound Tools

The sound tools implement the many sound related capabilities of the Apple IIGS sound hardware. In particular, the ENSONIQ DOC chip. Other toolsets implement MIDI and audio compression and expansion.

Sound Manager	The Sound Manager provides access to the Apple IIGS's sound hardware for creating basic sounds.
Note Synthesizer	The Note Synthesizer is used to create complex musical sounds simulating a variety of instruments using the Apple IIGS's sound hardware.
Note Sequencer	The Note Sequencer is used to string together notes from the Note Synthesizer into sequences, patterns and phrases that make up a song.

ACE	The Audio Compression and Expansion toolset (ACE) implements a collection of routines which compress and expand digital audio data in order to conserve storage requirements.
MIDI	The MIDI toolset implements a Musical Instrument Digital Interface for the Apple IIGS through either of its serial ports and the Apple MIDI adapter.

Math Tools

The math toolsets implement a wide variety of complex mathematical operations for both integer and floating-point calculations.

Integer Math	This toolset consists of a varied collection of operations for integers, long integers and signed fractional numbers. These include multiplication, division, conversions, etc.
SANE	SANE implements the Standard Apple Numeric Environment. It is an extended-precision IEEE 754 conformant implementation of floating point arithmetic and transcendental functions.

How Calling a Tool Routine Works

This section is intended for advanced programmers who want to understand how a tool routine is actually invoked from Pascal. If you are content with the fact that everything works and that the tool routines are essentially additional built-in routines then feel free to skip this section.

Complete Pascal provides for programmer access to the Apple IIGS toolsets via a collection of Pascal units. These units declare all of the procedures, functions, types, constants, etc. for each toolset in order that they can be used from the Pascal language. There is one unit for each toolset. Each of these 30 toolsets are listed together with the Pascal unit which declares its interface in the table below. Appendix C provides a complete source code listing for each of these units.

Table 5-1
Apple IIGS Toolbox

Tool Number	Tool Name	Pascal Unit
1	Tool Locator	Locator.p
2	Memory Manager	Memory.p
3	Miscellaneous Tools	MiscTool.p
4	QuickDraw II	QuickDraw.p
5	Desk Manager	Desk.p
6	Event Manager	Event.p
7	Scheduler	Scheduler.p
8	Sound Manager	Sound.p
9	Apple Desktop Bus	ADB.p
10	SANE	SANE.p
11	Integer Math	IntMath.p

12	Text Tools	TextTool.p
14	Window Manager	Windows.p
15	Menu Manager	Menus.p
16	Control Manager	Controls.p
17	System Loader	Loader.p
18	QuickDraw Auxiliary Routines	QDAux.p
19	Print Manager	Print.p
20	Line Edit	LineEdit.p
21	Dialog Manager	Dialogs.p
22	Scrap Manager	Scrap.p
23	Standard File	StdFile.p
25	Note Synthesizer	NoteSyn.p
26	Note Sequencer	NoteSeq.p
27	Font Manager	Fonts.p
28	List Manager	Lists.p
29	Audio Compression Expansion	ACE.p
30	Resource Manager	Resources.p
32	MIDI	MIDI.p
34	Text Edit Manager	TextEdit.p

Each tool routine in a toolset is declared as either a procedure or function, depending upon whether or not the routine returns a value on the stack, and may have zero or more parameters. The procedure or function declaration is completed with the TOOL directive. The tool directive is a special extension to Complete Pascal for the specific purpose of declaring interfaces to the Toolbox.

The following procedure declaration is taken from the QuickDraw.p unit, and is the interface to the MoveTo procedure in the QuickDraw toolset.

```
        PROCEDURE MoveTo(h,v: Integer);          Tool 4,58;
```

As you can see, the procedure declaration is completed with the tool directive Tool 4,58. The first integer in the tool directive specifies the toolset to which the routine belongs. In this case, it is toolset number 4 which is the QuickDraw toolset. The second integer is the function number of the routine within the toolset. Every routine within a toolset is assigned a unique function number. The MoveTo routine is assigned number 58. Together, these two integers uniquely identify the MoveTo procedure in the entire Apple IIGS Toolbox.

The Apple IIGS defines a consistent mechanism for invoking a Toolbox routine. To invoke a Toolbox routine, space for any function result value must first be reserved on the stack followed by pushing the values of any parameters. Then the 65816 X-register must be loaded with the desired Toolbox routine's function number and toolset number such that X-register = 256 * function number + toolset number. Finally, a jump subroutine long instruction is made to the address $E10000 which then contains a jump into the Tool Locator which finds the code associated with the desired Toolbox routine and passes control to it. Upon returning from the Toolbox routine, all parameters have been removed from the stack leaving the function result value (if any) on the top of the stack. In addition, the 65816 processor's carry flag is set if an error occurred during the execution of the Toolbox routine, and, if this occurs, then the 65816 accumulator register contains an error code.

By using Complete Pascal's tool directive with a procedure or function declaration, the preceding conventions are obeyed. In addition, Complete Pascal will generate a store accumulator instruction to the Pascal global variable _ToolErr so that potential error codes returned by a Toolbox routine can be examined.

Thus, an invocation of MoveTo(16,20) would generate the following 65816 instructions.

```
pea  $0010
pea  $0014
ldx  $3A04               ; 58 * 256 + 4
jsl  $E10000
sta  _ToolErr
```

In order to allow programs written in Complete Pascal to perform error checking on calls to Toolbox routines, Complete Pascal has defined the special function IsToolError which examines the current state of the processor's carry flag. The IsToolError function should only be used IMMEDIATELY after a call to a Toolbox routine to ensure that the state of the processor's carry flag has not been corrupted by any intervening operations.

Thus, a program written in Complete Pascal might use the following code to detect an error which occurs in the Toolbox routine MoveTo.

```
MoveTo(16,20);
if IsToolError then
   svToolErr := _ToolErr;
   Writeln('Error occurred in MoveTo, #',svToolErr);
   end;
```

Note that the value of _ToolErr was first saved to the temporary variable svToolErr before the call to Writeln. This is because Writeln itself makes tool calls that would destroy the value of _ToolErr associated with the error condition returned by MoveTo.

There are at least three cases where the compiler's generation of the STA _ToolErr instruction is not required. These are the following:

- Many Toolbox routines do not return errors (this is the case in the above example).
- An application has otherwise guaranteed that all possible error conditions do not exist.
- An application is not effected if an error occurs, proceed regardless (usually poor programming style, but sometimes appropriate).

If these reasons occur often enough in an application, then the generation of the STA _ToolErr instruction can potentially increase the size of an application's code unnecessarily. To avoid this possibility, Complete Pascal provides the $ToolErrorChk directive to turn off and on the generation of this instruction (see Appendix B).

For example, the following call to the Toolbox routine MoveTo would NOT generate the STA _ToolErr instruction.

```
{$ToolErrorChk-}
MoveTo(16,20);
```

While use of the $ToolErrorChk directive can save a considerable amount of code, the programmer must be very careful of its use in order to avoid erroneously checking the value of _ToolErr when the directive is turned off, and therefore _ToolErr has not been assigned an error code.

Event-Driven Programming

Desktop applications are event-driven programs. That is, the application is driven by events from the user, not the other way around. An application usually has few clues as to when the next event from the user will occur, nor does it know what kind of input it will be. For example, it could be a keyboard event, a mouse event, etc.

An application decides what to do from one moment to the next by repeatedly calling the Toolbox Event Manager function GetNextEvent or the Window Manager function TaskMaster. The program's code which makes these repeated calls to the toolbox is called the *main event loop*. The Toolbox Event Manager will inform the application of which action is to be processed next. Based upon this information, the application can take appropriate action. An example of one possible implementation of the main event loop using TaskMaster is given in Program 5-1. In this example, only a mouse click in the close box of a window, a menu selection and a mouse click in a window's content control are handled by the application. All other event processing is performed by TaskMaster.

Program 8-1
Main Event Loop

```
procedure MainEventLoop;
var      code: Integer;
begin
   gMainEvent.wmTaskMask := $001FFFFF;  { Allow TaskMaster to do
                                               everything. }
   gDone := false;

   repeat
       code := TaskMaster($FFFF,gMainEvent);
       case code of
          wInGoAway:  DoClose;
          wInSpecial,
          wInMenuBar: HandleMenu;
          wInControl: DoControlHit;
       end;
   until gDone;
end; { of MainEventLoop }
```

Clearly, different applications will handle different events in different ways. The following table identifies each of the GetNextEvent and TaskMaster events. For detailed information about events, consult the *Apple IIGS Toolbox Reference*.

Table 8-3
GetNextEvent Event Types

Type	Occurrence
nullEvent	Reported when no other event is available.
mouseDownEvt	Generated when the user presses the mouse button.
mouseUpEvt	Generated when the user releases the mouse button.
keyDownEvt	Generated when the user presses any character key on the keyboard or keypad. The character keys include all keys except the Shift, Caps Lock, Control, Option and Apple keys which are modifier keys.
autoKeyEvt	Generated when the user holds a key down. The auto-key is generated after an initial delay and then at periodic intervals.
updateEvt	This is an internally generated event indicating that the contents of a window need to be updated (redrawn).
activateEvt	This is an internally generated event when a window becomes active or inactive. That is, when a window moves from back to front or from front to back respectively.
switchEvt	Generated when a switch control is pressed.
deskAccEvt	Generated when the Classic Desk Accessory menu is invoked via the Control-Apple-Escape key sequence.
driverEvt	Generated when a device driver performs a PostEvent due to some circumstance, usually when data transmission has occurred or has been interrupted.
app1Evt-app4Evt	There can be four different application defined events generated. The meaning of these events are defined by the application and entered into the event queue using PostEvent.
	TaskMaster Event Types
wInDesk	A mouse-down event occurred in the desktop (not in any window).
wInMenuBar wInSpecial	A mouse-down event occurred in the menu bar and then released over a menu item which was not a desk accessory from the Apple menu or from a menu added by a desk accessory. TaskMaster tracks the mouse until it has been released over a particular menu item, thus selecting it.
wInContent	A mouse-down event occurred in the content region of a window.
wInDrag	A mouse-down event occurred in the drag region of a window.

wInGrow	A mouse-down event occurred in the grow icon of a window.
wInGoAway	A mouse-down event occurred in the close box of a window.
wInZoom	A mouse-down event occurred in the zoom box of a window.
wInInfo	A mouse-down event occurred in the information bar of a window.
wInFrame	A mouse-down event occurred in the frame of a window.
wInactMenu	A menu item was selected that was inactive.
wClosedNDA	A desk accessory was closed.
wCalledSysEdit	System Edit was called.
wTrackZoom	A mouse-down event occurred in the active window's zoom box but was not released in the zoom box.
wHitFrame	A mouse-down event occurred in the window's frame of the active window.
wInControl	A mouse-down event occurred in the content region of a window, but within the bounding rectangle of a content control.

Program Structure

The source code for a desktop application can, of course, be organized according to the programmer's own requirements and desires. However, every desktop application will share the same fundamental main program structure. That is, every desktop application must start up the Toolbox toolsets it requires, initialize any application globals, create the application menus and windows, process events, and finally shutdown the toolset used by the application.

The following code fragment is from the source code of Skeleton.p's main program. An important thing to notice is that the program source code references *resources*. In this particular case, the StartupTools function references a StartStop resource which defines the Toolbox toolsets required by this application. See Chapter 4 for detailed information about resources.

Program 8-2
Skeleton.p Source Code Fragment

```
gMyMemoryID := MMStartUp;

gStartStopRef :=
  StartupTools(gMyMemoryID,RefIsResource,Ref(kStartStopResID));
if _ToolErr = noError then begin
  InitializeGlobals;
  SetUpMenus;
  SetUpWindows;

  InitCursor;
  MainEventLoop;
```

```
    end;

  ShutDownTools(RefIsHandle,gStartStopRef);
```

Before starting out to program your own applications, you should spend some time studying the complete source code and resources of the Skeleton.p application to understand the basic structure of a desktop application.

Adding Resources to an Application

As shown in the example program above, Complete Pascal programs can not use resources. And, Chapter 4 explained in detail how to use the Complete Pascal Resource Editor to create and edit resources. However, in order for an application to use resources, its resources must be copied into the resource fork of the application file itself. The CTI Linker performs this automatically as part of the link process. The resources copied into the resource fork of the application are those found in the resource file specified by the Add Resources... menu item of the Compile menu.

To add resources to a program, bring the window containing the application's main program to the front. Then select the Add Resources... menu item from the Compile menu and select the appropriate resource file. Complete Pascal records the selected file so that each time the application is compiled its resources are copied into the application file.

Definition Procedures (DefProcs)

Often times, the Apple IIGS Toolbox routines must call a procedure which is actually part of your application. These types of procedures (and sometimes functions) are given the name Definition Procedures, or *DefProcs* for short. The reason for the name definition procedure is that these routines are generally used to allow the application to provide a custom definition of some generic operation. For example, there are Menu Definition Procedures which allow an application to provide customized drawing procedures for drawing the representation of menus – perhaps a menu that contains a palette of colors rather than a list of text items (see the Complete Source Code Library - Pascal for an example of this). As you might expect, the Toolbox also allows for definition procedures of windows, controls, lists, etc.

Another component of the Toolbox where an application must use definition procedures (and the most likely), is with the NewWindow2 function in the Window Manager. The NewWindow2 routine contains several parameters, one of which is the address of the window's *content definition procedure.*

The content defProc routine is called by the Window Manager whenever it detects that the content of the window must be updated (redrawn) because a portion of the window's content which was previously hidden has become visible.

As you might expect, Complete Pascal's memory model may not be in place when a definition procedure is called. This can happen because the procedure is being called from the Toolbox, and it is likely that the Toolbox has temporarily changed the state of the 65816 in such a way that effects the way Pascal global variables are addressed.

Typically, global variables are addressed using the 65816's absolute addressing mode versus the less efficient absolute long addressing mode since Complete Pascal ensures that the 65816's

Data Bank Register points to the memory bank containing the program's global variables. However, in the case of definition procedures, Complete Pascal's convention may not be obeyed by a particular Toolbox routine (ie. the Toolbox routine has temporarily changed the value of the Data Bank register). Thus, it is necessary to inform Complete Pascal that a particular procedure is indeed a *definition procedure* and that it will be called by the Toolbox so that it guarantees the data bank register is set to the appropriate data bank when the procedure is called and then restored when the procedure exits. This is accomplished by using the $DefProc compiler directive. For example:

```
{$DefProc}
PROCEDURE WindowContentDraw;
BEGIN
    { redraw the window contents }
END;
```

Large Programs and Segmentation

The Apple IIGS limits the size of a application's code and data segments to 64K bytes. Code segments contain the application's executable code, while data segments contain the storage required for the application's global variables. The reason for this size restriction is that a segment must not cross the boundaries of a bank of memory. On the Apple IIGS, a bank of memory is 64K bytes. Thus, in order to develop applications which have more than 64K bytes of code or 64K bytes of data, the program must be segmented. Normally, Complete Pascal creates one code segment and one data segment for an application. To obtain more than one segment, the compiler's $CSeg and $DSeg directives must be used.

Code segments are named so that the CTI Linker can organize the different pieces of code together based on their code segment names. The default code segment name is "main". In order to change the name of the current code segment, the Complete Pascal {$CSeg *segname* } compiler directive is used. When a {$CSeg *segname* } directive appears in a program or unit, the code for all subsequent procedures and functions is placed in the new code segment. To restore code segmentation back to the default segment, merely place the {$CSeg main } directive in your program.

Data segments are named just as code segments are so that the CTI Linker can organize the different pieces of data together based on their data segment names. The default data segment name is ~global. In order to change the name of the current data segment, the Complete Pascal {$DSeg *segname* } compiler directive is used. When a {$DSeg *segname* } directive appears in a program or unit, the data for all subsequent global variable declarations is placed in the new data segment. To restore data segmentation back to the default segment, simply place the {$DSeg ~global } directive in your program.

Unless a program absolutely requires a large amount of global storage, the {$DSeg *segname* } should not be used. The reason for this is that all global storage allocated outside of the ~global data segment is addressed using less efficient addressing modes than data allocated in the ~global data segment.

For more information regarding the use of the {$CSeg *segname*} and {$DSeg *segname*} directive see Appendices B and D.

Chapter 9

New Desk Accessories

A New Desk Accessory (NDA) is a "mini-application" which is accessed from the Apple menu and executes from within the event-driven environment of desktop applications. There are actually two types of desk accessories – Classic Desk Accessories and New Desk Accessories. This chapter discusses only New Desk Accessories. Chapter 10 addresses Classic Desk Accessories.

When a New Desk Accessory is selected from the Apple menu it typically creates a window on the desktop and may also add a menu to the menu bar. An NDA relies upon the application in which it is operating to call the appropriate Desk Manager routines (or TaskMaster) in order to support the NDA. NDAs require this cooperation with desktop applications because NDAs are not stand-alone programs. Instead they are a collection of procedures which are called on certain occasions.

The Complete Pascal distribution disk contains the source code for an example NDA – ClockNDA.p. In addition, the Complete Source Code Library - Pascal product contains more example NDA source code.

Getting Started

As mentioned above, NDAs operate within the desktop environment of Apple IIGS applications. As such NDAs assume that an application which supports NDAs will have loaded and started at least the following Apple IIGS toolsets:

- QuickDraw
- Event Manager
- Window Manager
- Menu Manager
- Control Manager
- Scrap Manager
- LineEdit
- Dialog Manager

An NDA is allowed to call any toolbox function contained in these toolsets. If an NDA must call routines from toolsets other than those listed above, the NDA itself must ensure the required toolset is loaded and properly initialized.

Program Structure

The structure of a new desk accessory's source code is quite different from that of a normal Apple IIGS program. In particular, an NDA does not have a main program, but instead contains four special routines – DAOpen, DAClose, DAAction, and DAInit – which are called directly by the Apple IIGS Desk Manager at particular and well defined times. In a sense, an NDA has four "main programs" which work together and communicate by setting global variables. In addition, an NDA does not have a MainEventLoop procedure since the application in which the NDA runs detects the events and passes them onto the NDA when appropriate. However, the most noticeable distinction between the source code structure of an application and a desk accessory is that a desk accessory is implemented as a unit rather than a program. This is because there is no main program in an NDA.

Note that a NDA must have the four required routines, and they must be spelled exactly as defined – DAOpen, DAClose, DAAction, and DAInit. If you fail to provide these routines, Complete Pascal returns an error when you attempt to compile the NDA.

In addition to the four special routines every NDA must have, three additional pieces of information must also be provided to the Complete Pascal compiler in order for it to properly create the NDA. These are the *service period*, the *event mask*, and its *menu name*. This information is specified in Complete Pascal with the $NDA compiler directive.

```
{$NDA servicePeriod eventMask menuName  }
```

The service period defines how often the NDA should be "called" with the DARun action code (see below) in order to service the NDA's functionality. A period of 1 is 1/60th of a second, a period of 2 is 2/60ths of a second (or 1/30th of a second), a period of 60 is 60/60ths of a second (or 1 second), etc. A period of -1 (or $FFFF) is never. For example, if a NDA displays the current time, then it would specify a service period of 60 so that it could update its display every second.

The event mask defines which events should be handled by the desk accessory. These values are a subset of those used by Apple IIGS applications using GetNextEvent or TaskMaster, and are listed below for reference from the Events unit. Of the six listed below, the update and activate events are always passed to the desk accessory regardless of the event mask, however, the remaining four event types must be specified explicitly. If all events should be handled by the desk accessory then an event mask of -1 (or $FFFF) should be specified.

```
CONST      mDownMask    = $0002;
           mUpMask      = $0004;
           keyDownMask  = $0008;
           autoKeyMask  = $0020;
           updateMask   = $0040;
           activeMask   = $0100;
           EveryEvent   = $FFFF
```

Finally, the menu name is the name for the desk accessory which should appear in the Apple menu of an application supporting its desk accessories.

As mentioned above, this information is specified with the compiler's $NDA directive. This directive must appear as the first line of the program before the reserved word UNIT. For example, the following directive specifies a service period of every 60 ticks (1 second), that all events should be handled by the desk accessory and that the menu name for the desk accessory is "Clock".

```
{$NDA 60 -1 Clock }
UNIT ClockNDA;
```

Note that the name of the unit, ClockNDA, is not the name used by Complete Pascal for the name of the desk accessory which appears in the Apple menu. The name used is the one specified in the $NDA compiler directive.

Thus, we arrive at the basic structure for a new desk accessory written in Complete Pascal.

```
{$NDA 60 -1 Clock }
UNIT ClockNDA;

INTERFACE

FUNCTION  DAOpen: WindowPtr;
PROCEDURE DAClose;
PROCEDURE DAAction(Code: Integer; Param: LongInt);
PROCEDURE DAInit(Code: Integer);

IMPLEMENTATION

FUNCTION DAOpen: WindowPtr;
BEGIN
   { Code for DAOpen }
END;

PROCEDURE DAClose;
BEGIN
   { Code for DAClose }
END;

PROCEDURE DAAction(Code: Integer; Param: LongInt);
BEGIN
   { Code for DAAction }
END;

PROCEDURE DAInit(Code: Integer);
BEGIN
   { Code for DAInit }
END;

END.
```

The following four sections define each of the required desk accessory routines and outlines each of their responsibilities.

The DAInit Procedure

The DAInit procedure is the very first and very last NDA procedure to be called by the Apple IIGS Desk Manager. The DAInit procedure is first called when an application calls the DeskStartUp procedure. At this time the NDA should perform any necessary initialization which is required. The DAInit procedure is last called when the DeskShutDown procedure is called by the application, thus the NDA should perform any termination operations that are necessary. Essentially, life begins and ends for a NDA with the DAInit procedure.

The DAInit procedure has one integer parameter, Code, which indicates under which circumstance the routine is being called. If Code = 0 then DAInit is being called due to a DeskShutDown call, otherwise the call is due to a DeskStartUp call. The following example of a DAInit procedure illustrates its basic structure.

```
PROCEDURE DAInit(Code: Integer);
{  The variable myWindOpen is global }
BEGIN
   if Code = 0 then begin
     { A DeskShutDown Call, check that the DA window is closed }
     end
  else begin
     { a DeskStartUp Call, init the myWindOpen flag }
     myWindOpen := false
     end
END;
```

Since most NDA's have a window that is displayed when the desk accessory is opened, the NDA must keep track of whether its window is open or not. The easiest way to do this is using a global windOpen variable which holds the value true when the window is open and the value false when the window is closed. Thus, when DAInit is called because of a DeskStartUp it should set its global windOpen variable to false.

Note that it is possible for the DAInit procedure to be called to terminate itself due to a DeskShutDown call when its window is still open. This can happen if a user quits an application with a desk accessory still open on the desktop. Thus, it is imperative that your DAInit procedure check to be sure the window has been closed before allowing termination.

The DAOpen Function

The DAOpen function is called in order to open the desk accessory, which for most NDA's is the time to create and display its window. A NDA is opened by a user by selecting its name from the Apple menu. When a NDA is selected from the Apple menu, the application calls the Desk Manager routine OpenNDA either explicitly or from TaskMaster.

If the NDA's window is not yet open when DAOpen is called then it should create the window and specify it as a system window. A window is made a system window by calling the Window Manager SetSysWindow routine. If DAOpen is called and its window is already created then the DAOpen routine should ensure that its window is topmost. To make its window topmost simply call the Window Manager routine SelectWindow. (DAOpen can be called when its window is open and when the user selects its name more than once from the Apple menu.) In either case, DAOpen should return, as its function result value, a pointer to its window.

Note that the NDA is told via the DAAction procedure (below) when to redraw the contents of the NDA's window.

The following is a source code fragment showing the basic structure of the DAOpen function.

```
FUNCTION DAOpen: WindowPtr;
```

```
{ The variables myWindOpen, myWindPtr, and myWind are globals }
BEGIN
    if myWindOpen then
        SelectWindow(myWindPtr)
    else begin
        myWindOpen := true;
        { set up myWind for creating the window }
        myWindPtr := NewWindow2(...);
        SetSysWindow(myWindPtr);
        end;

    DAOpen := myWindPtr;
END;
```

The DAClose Procedure

The DAClose procedure is called in order to close the NDA's window. This procedure is typically called when the user presses the mouse button in the close box of the NDA's window. In order to prevent errors in situations when DAClose is called and the NDA's window is not open the procedure should always check to be sure the window is open first.

```
PROCEDURE DAClose;
{  The variables myWindOpen and myWindPtr are globals }
BEGIN
    if myWindOpen then begin
        CloseWindow(myWindPtr);
        myWindOpen := false;
        end;
END;
```

The DAAction Procedure

The DAAction procedure is the routine which does all the work associated with the desk accessory between the time that it has been opened until it is closed. The DAAction procedure has two parameters – Code which indicates what type of action to perform and Param whose meaning depends upon the Code parameter. There are nine potential values for the Code parameter, each of which must be implemented by the DAAction procedure. These operations are summarized in the following table together with the meaning of the Param parameter in each case.

Table 9-1
Apple IIGS Toolbox

Action	Description
DAEvent	An event relevant to the desk accessory has occurred. Param points to the EventRecord describing the event. The DAEvent action indicates that the application detected an event which is

associated with the desk accessory. The event could be a mouse-down in the window's content, an update, an activate, etc. When this action occurs, the NDA should respond to each event type that the NDA supports.

DARun	The number of ticks specified as the period for the NDA has elapsed since the last DARun call. Param has no meaning.
DACursor	This code is passed to a desk accessory if it is the frontmost window each time SystemTask is called. The purpose is to allow the desk accessory to change the cursor when it is over the NDA's window. Param has no meaning.
DAMenu	This is passed to a desk accessory if an item from a system menu is selected. LoWrd(Param) is the Menu ID and HiWrd(Param) is the Item ID.
DAUndo DACut DACopy DAPaste DAClear	Each of these codes are passed to a desk accessory if the application determines that the user has selected one of the standard edit commands from the Edit menu. The DAAction procedure should assign the value of 1 in the Code parameter if the action was handled, otherwise a value of 0 should be assigned.

The following source code fragment demonstrates the basic structure of a DAAction procedure.

```
PROCEDURE DAAction(Code: Integer; Param: Longint);
{  The variable myWindPtr is globals }
VAR   currPort: GrafPortPtr;
BEGIN
   case Code of
      DAEvent: begin
         case EventRecordPtr(param)^.what of
            mouseDownEvt: ;
            mouseUpEvt:   ;
            keyDownEvt:   ;
            autoKeyEvt:   ;
            updateEvt:    ;
            activateEvt:  ;
         end;
         end;
      DARun: begin
         currPort := GetPort;
         SetPort(myWindPtr);
         SetPort(currPort);
         end;
      DACursor: begin
         { code to update the cursor }
         end;
      DAMenu: begin
         { code to respond to a menu selection }
         end;
      DAUndo: begin
```

```
            { code to perform an Undo for the DA }
            Code := 1;
            end;
        DACut: begin
            { code to perform an Undo for the DA }
            Code := 1;
            end;
        DACopy: begin
            { code to perform an Copy for the DA }
            Code := 1;
            end;
        DAPaste: begin
            { code to perform an Paste for the DA }
            Code := 1;
            end;
        DAClear: begin
            { code to perform an Clear for the DA }
            Code := 1;
            end;
        end;
    END;
```

Of particular interest in the DAAction procedure above is the code for the DARun action. Most actions are only reported to the NDA when its window is topmost. However, this is not the case for the DARun action. The DAAction procedure will always be called with the DARun action code whenever the service period time has elapsed, regardless of whether or not its window is topmost. Therefore, before the NDA executes any code which draws into its window it is absolutely necessary to ensure that the window is the current grafport and to restore the previous grafport before leaving the DAAction procedure.

Compiling a Desk Accessory

Desk accessories are compiled in the same way as applications are compiled. That is, you may choose the To Memory, To Disk or Check Syntax commands from the Complete Pascal Compile menu described in Chapter 3. When selecting the Check Syntax menu command, the compiling results are the same as an application. However, when choosing To Memory or To Disk, the compiler behaves differently for NDAs than for applications.

When choosing the To Memory command the Complete Pascal compiler is invoked to compile and link the desk accessory. However, when the compiler successfully completes the compilation it does not shut down the Complete Pascal environment and transfer control to the desk accessory as it does for an application. But rather, the desk accessory is installed into Complete Pascal's Apple menu below the installed desk accessories.

Note that three items are actually added to Complete Pascal's Apple menu. The first is a dividing line which separates the new items from the existing items. The second item, “Open NDA”, opens the desk accessory just as it would if the desk accessory had been installed. You should select this item to open and test the desk accessory. The last item, “Remove NDA”, removes the desk accessory from the Apple menu and then purges its compiled code from memory. You should select this item when you have finished testing the desk accessory and want to free up the memory it is using.

Complete Pascal ensures that a desk accessory which is compiled to memory is operated on in the exact same way as a normal installed desk accessory. The DAInit procedure is called immediately after the desk accessory is compiled with Code <> 0 in order to emulate the call which occurs when DeskStartUp is called for other desk accessories, and with Code = 0 in order to emulate the DeskShutDown call. While the the desk accessory is open, the service period is honored and all events related to the desk accessory are properly passed to it.

When choosing the To Disk compile option, Complete Pascal creates an Apple IIGS load file for the desk accessory which is in the proper format for a NDA rather than an application. In particular, the file is created with the GS/OS filetype of $B8 rather than $B3 for applications. When compiling a desk accessory to disk, it is necessary to Install the desk accessory before it can be used. This process is described in the following section.

Installing a Desk Accessory

Once you have successfully created and tested a desk accessory and would like for it to appear in the Apple menu of all desktop applications you must install it. For an NDA to be installed, it must reside in the SYSTEM:DESK.ACCS: directory of the boot disk so that desktop applications supporting desk accessories may access it.

Installing a NDA consist of a simple three step process as outlined below.

- Choose the To Disk option from the Complete Pascal's Compile menu. This will compile the NDA to disk as a GS/OS load file having the proper file type of $B8 (that is, it will not be an application you can run from the Apple IIGS Program Launcher, but an NDA load file).

- The Apple IIGS Desk Manager requires that all desk accessories be placed in the special system directory SYSTEM:DESK.ACCS:. Thus, it is necessary to copy the desk accessory load file into this directory. To do this it will be necessary to leave the Complete Pascal environment and use the Finder to copy the file into this directory.

- Finally, the Apple IIGS must be rebooted. During the boot process of the Apple IIGS, the special directory SYSTEM:DESK.ACCS: is searched for all currently installed desk accessories. Since this process is only done at boot time it is necessary to reboot the machine in order for it to recognize the new desk accessory.

Chapter 10

Classic Desk Accessories

A Classic Desk Accessories (CDA) is a "mini-application" accessible from the Apple IIGS Classic Desk Accessory menu. A Classic Desk Accessory executes in a non-desktop, non-event based environment. Unlike New Desk Accessories, a CDA takes full control of the machine during what is basically an interrupt state while the CDA is in use. Classic desk accessories are invoked by pressing the Open-Apple, Control and Escape keys simultaneously.

Complete Pascal provides direct support for implementing Classic Desk Accessories in Pascal. In addition, the Complete Pascal Master Disk contains the source code for an example CDA titled SHRDump.p.

Program Structure

The structure of a classic desk accessory's source code is quite different than that of a normal Apple IIGS program (ie: Textbook and/or Desktop applications). In particular, a CDA does not have a main program, but instead contains two special routines – StartUpCDA and ShutDownCDA – which are called directly by the Apple IIGS Desk Manager at particular and well defined times. In a sense, a CDA has two "main programs" which work together and communicate by setting global variables.

Note that a CDA must have the two required routines identified above, and they must be spelled exactly as defined – "StartUpCDA" and "ShutDownCDA". If you fail to provide these routines, Complete Pascal will report an error when you attempt to compile the CDA.

In addition to the two special routines that every CDA must have, the program must use the compiler's $CDA directive to indicate that the source code does in fact implement a Classic Desk Accessory. The $CDA directive has one argument which is the name of the CDA as it should appear in the Classic Desk Accessory menu. The format of the $CDA directive follows:

{$CDA *menuName* }

The location of the $CDA directive is also important. The $CDA directive must appear before the reserved word UNIT for the directive to have any effect. An error will result if the directive appears after the reserved word UNIT. For example:

```
{$CDA SHRDump }
UNIT MySHRDump;
```

Remember, the name of the program (MySHRDump) is NOT the name used by Complete Pascal for the name of the desk accessory which appears in the Classic Desk Accessory menu. The name used is the one specified in the $CDA compiler directive.

Thus, we arrive at the basic structure for a classic desk accessory written in Complete Pascal.

```
{$CDA SHRDump }
UNIT MySHRDump;

INTERFACE

PROCEDURE StartUpCDA;
PROCEDURE ShutDownCDA;

IMPLEMENTATION

PROCEDURE StartUpCDA;
BEGIN
    { Code for StartUpCDA }
END;

PROCEDURE ShutDownCDA;
BEGIN
    { Code for ShutDownCDA }
END;

END.
```

The following two sections define each of the required classic desk accessory routines and outlines each of the routine's respective responsibilities.

The StartUpCDA Procedure

The StartUpCDA procedure has no parameters and is not required to perform any specific operation or have any specific structure. The StartUpCDA procedure actually implements the entire functionality of the CDA. The Desk Manager calls this procedure when the CDA's name is chosen from the Classic Desk Accessory menu. Unlike desktop applications or NDAs, a CDA is not event-driven. Thus, the procedure is given complete control of the Apple IIGS to perform its operations until it is complete.

```
PROCEDURE StartUpCDA;
BEGIN
  { Perform any and all actions for the CDA }
END;
```

The ShutDownCDA Function

The ShutDownCDA procedure, like the StartUpCDA procedure, has no parameters and is not required to perform any specific operation or have any specific structure. The ShutDownCDA procedure is called when a call to the Desk Manager DeskShutDown procedure has been made by an application or when the Apple IIGS switches between GS/OS and ProDos8.

The ShutDownCDA procedure gives the CDA an opportunity to terminate any "tasks" the StartUpCDA procedure my have initiated. Since most CDAs are very modal (ie. they take control of the machine, perform their operation, and then quit), the ShutDownCDA procedure is rarely used by a CDA.

```
PROCEDURE ShutDownCDA;
BEGIN
   { Perform any terminating actions for the CDA }
END;
```

Compiling a Desk Accessory

Desk accessories are compiled using the To Disk or Check Syntax commands in the Compile menu. However the To Memory command cannot be used. The To Memory command cannot be used because Complete Pascal has no way to automatically install a CDA after it is compiled.

When choosing the To Disk compile option, Complete Pascal creates an Apple IIGS load file for the desk accessory which is in the proper format for a CDA rather than an application. In particular, the file is created with the GS/OS filetype of $B9 rather than $B3 for applications. When compiling a desk accessory to disk, it is necessary to Install the desk accessory before it can be used. Installed CDA's is described in the following section.

Installing a Desk Accessory

Once you have successfully created and tested a desk accessory and would like for it to appear in the Apple menu of all desktop applications you must install it. For a CDA to be installed, it must reside in the SYSTEM:DESK.ACCS: directory of the boot disk so that desktop applications supporting desk accessories may access it.

Installing a CDA consist of a simple three step process as outlined below.

- Choose the To Disk option from Complete Pascal's Compile menu. This will compile the CDA to disk as a GS/OS load file having the proper file type of $B9 (that is, it will not be an application you can run from the Apple IIGS Program Launcher, but a CDA load file).
- The Apple IIGS Desk Manager requires that all desk accessories be placed in the special system directory SYSTEM:DESK.ACCS:. Thus, it is necessary to copy the desk accessory load file into this directory. To do this it will be necessary to exit the Complete Pascal environment and use the Finder to copy the file into this directory.
- Finally, the Apple IIGS must be rebooted. During the boot process of the Apple IIGS, the special directory SYSTEM:DESK.ACCS: is searched for all currently installed desk accessories. Since this process is only done at boot time it is necessary to reboot the machine in order for it to recognize the classic desk accessory.

Chapter 11

Tokens

Lexical *tokens* are the smallest units of text in a Pascal program. Tokens in Pascal are classified into *special-symbols, identifiers, directives, unsigned-numbers, labels* and *character-strings*. Aside from character-strings, the representation of any letter (upper-case, lower-case, font, etc.) is insignificant to the meaning of a program.

The text of a Pascal program consists of tokens and *separators*, where a separator is either a *blank* (the space or tab characters) or a *comment*. Two adjacent tokens must be separated by one or more separators if each token is an identifier, number, or word-symbol. A word-symbol is a special case of a special-symbol.

Special Symbols

Special-symbols are tokens having special meanings and are used to delimit the syntactic units of the language.

The following single characters are special-symbols:

```
+  -  *  /  =  <  >  [  ]  .  ,  (  )  :  ;  ^  @  {  }
```

The following *character-pairs* are special-symbols:

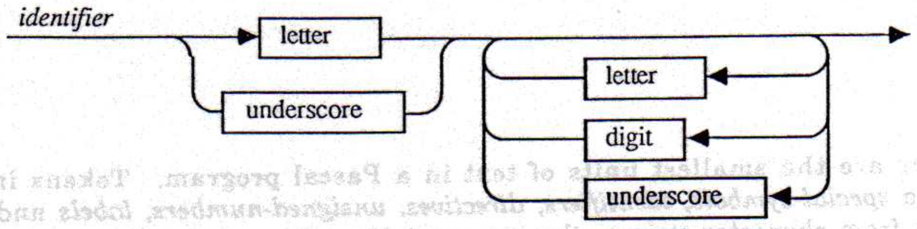
```
<>  <=  >=  :=  ..  (*  *)
```

The following *word-symbols* (or *reserved-words*) are special-symbols:

```
and       else            interface   procedure   unit
array     end             label       program     until
begin     file            mod         record      uses
body      for             nil         repeat      var
case      function        not         set         while
const     goto            of          string      with
div       if              or          then
do        implementation  otherwise   to
downto    in              packed      type
```

Identifiers

Identifiers are names used to denote constants, types, variables, procedures, functions, *programs, units* and *fields* in records. An identifier can be of any length so long as it fits on a single line, however, only the first 255 characters are significant. Corresponding upper- and lower-case letters are equivalent in identifiers. No identifier can have the same spelling as a word-symbol. Identifiers consist of letters, digits and underscore characters, except for the first character of the identifier, which must be either a letter or underscore character.



Examples of identifiers in Complete Pascal:

```
I   MaxInt   Writeln   A_very_long_identifier   _DataInit
```

Directives

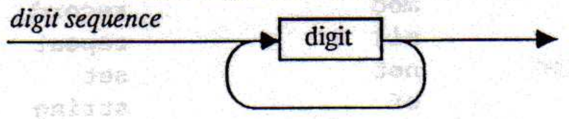
Directives are identifiers that have special meanings in the context of a procedure declaration or function declaration. They can otherwise be used as identifiers in all other contexts. The directives available in Complete Pascal are:

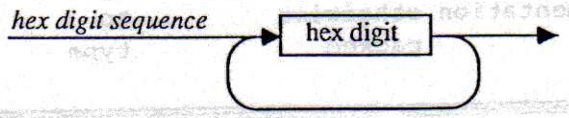
```
EXTERNAL   FORWARD   INLINE   TOOL
```

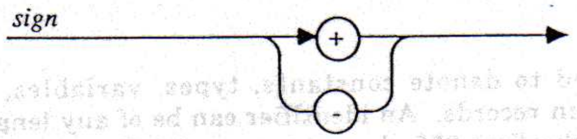
Further information on the EXTERNAL, FORWARD, INLINE and TOOL directives may be found in Chapter 17.

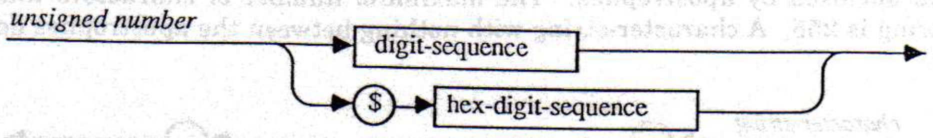
Numbers

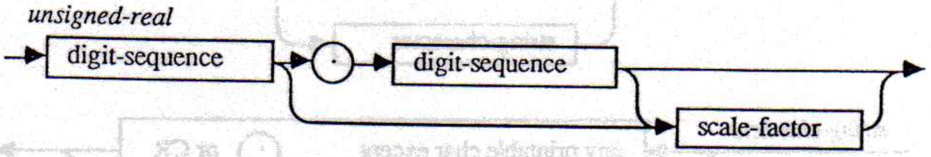
Numbers are unsigned-integers in decimal or hexadecimal (hexadecimal integers have the $ character as a prefix) notation representing constants of the data types Integer and LongInt. Unsigned-reals in decimal notation represent constants of the data type Extended. The letter 'E' or 'e' preceding a scale factor means *times ten to the power of.*

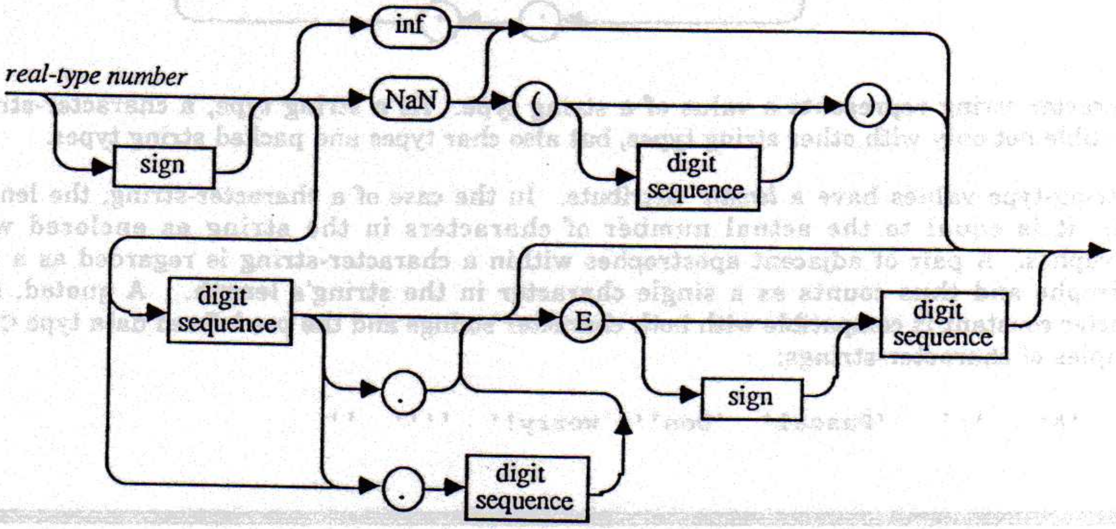












Examples of Numbers:

```
1    +100    -0.1    $A05D    5.329E4    NaN(1)    Inf
```

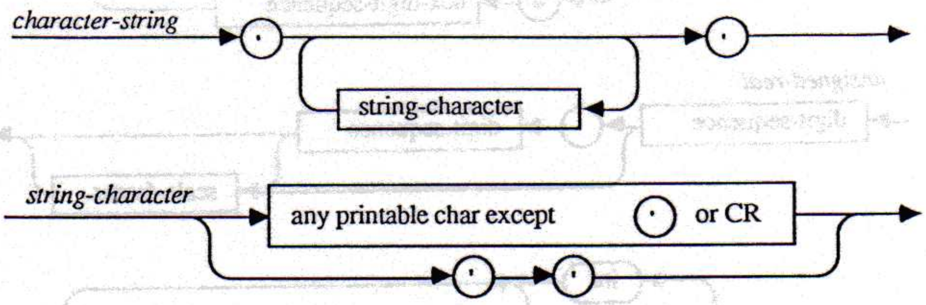
Unless explicitly assigned to a variable of another type, any number written with a decimal point or exponent is stored as an extended real number. All other numbers are written in the smallest type possible (`Integer` or `LongInt`). For further information on numeric types, see Chapter 13.

Labels

A *label* is a one to four digit-sequence whose value may be integer in the range 0..9999. Leading zeros in a label are insignificant, e.g. the labels *1* and *0001* are considered equivalent. Labels are used with Goto statements, described in Chapter 16.

Character-Strings

A *character-string* is a sequence of zero or more printing characters all on the same line in a program and enclosed by apostrophes. The maximum number of characters that can be in a character-string is 255. A character-string with nothing between the apostrophes denotes a *null-string* value.



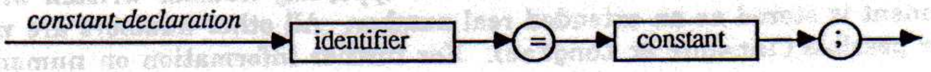
A character-string represents a value of a string type. As a string type, a character-string is compatible not only with other string types, but also char types and packed string types.

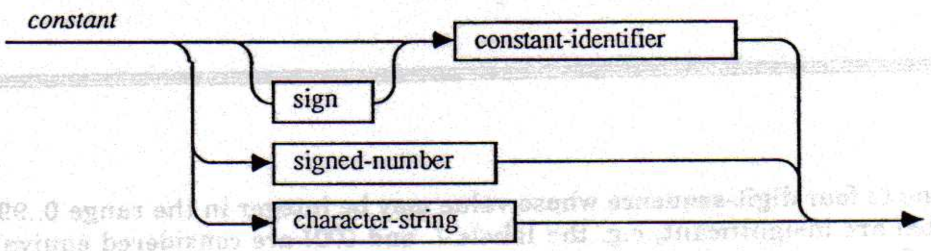
All string-type values have a *length* attribute. In the case of a character-string, the length is fixed; it is equal to the actual number of characters in the string as enclosed within apostrophes. A pair of adjacent apostrophes within a character-string is regarded as a single apostrophe and thus counts as a single character in the string's length. A quoted, single character constant is compatible with both character strings and the predefined data type `Char`. Examples of character-strings:

```
'A'   ';'   'Pascal'  'Don''t worry!'   ''''   ''
```

Constant Declarations

A *constant-declaration* defines 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 signed-number may be an integer or real number.

Comments and Compiler Directives

The constructs:

```
{ any text not containing right-brace }
(* any text not containing star-right-parenthesis *)
```

are called *comments* and are ignored by the compiler.

The substitution of a blank for a comment or a comment for a blank does not alter the meaning of the program. That is, a comment, as a separator, may appear anywhere in a program where a blank may appear.

Comments of the form {...} may be nested within comments of the form (* ... *), and vice versa, however, no other nesting of comments is available. The occurrence of the special symbol } within a {...} comment, or the special symbol *) within a (* ... *) comment always terminates the comment.
A compiler directive is a comment that contains a $ (dollar-sign) character immediately after the { or (* that begins a comment. The $ character is then followed by one or more letters which represent a specific compiler directive. Compiler directives serve to affect the behavior of the compiler. Each of the compiler directives and their affects are described in Appendix B.

Examples of compiler directives:

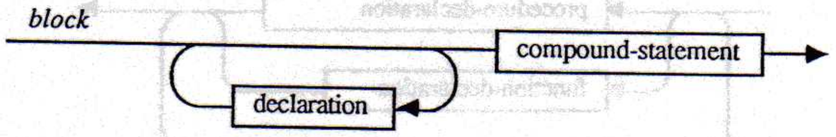
```
{$J+} (*$CSeg Printing *)
```

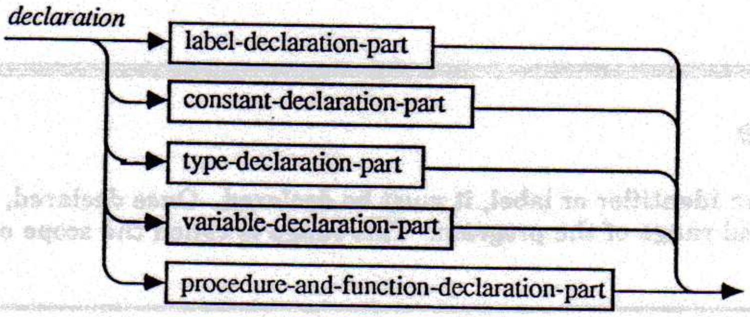
Chapter 12

Blocks, Scope and Activations

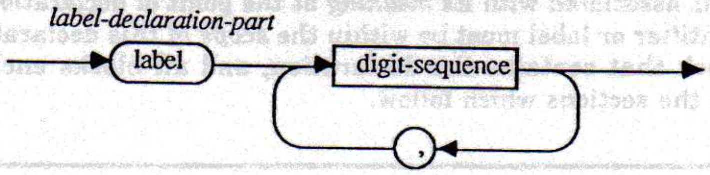
Definition of a Block

The *block* is the fundamental construct for Pascal source code. A block consists of a *declaration part* and a *statement part*. The declaration part consists of zero or more declarations which may appear in any order. The statement part is a compound statement and follows the declarations. Every block is part of a procedure declaration, a function declaration, a program, or a unit. All identifiers and labels that are declared in the declaration part of a block are *local* to that block. The program block contains all other blocks; therefore, declarations in the program block are termed *global*.

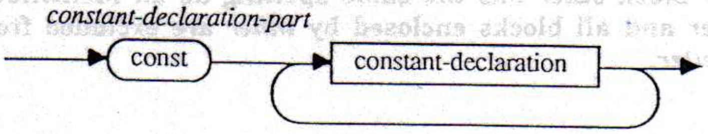



declaration
label-declaration-part
constant-declaration-part
type-declaration-part
variable-declaration-part
procedure-and-function-declaration-part

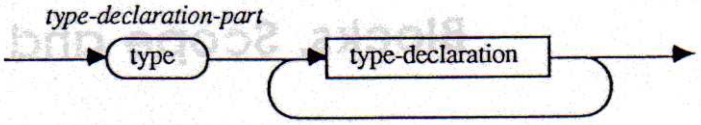
The *label declaration part* declares labels that mark statements in the corresponding statement part. Each label must mark exactly one statement in the statement part.



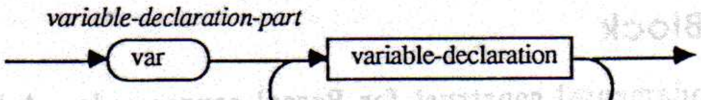
The *constant declaration part* contains *constant declarations* local to the block. See Chapter 11 for more details.



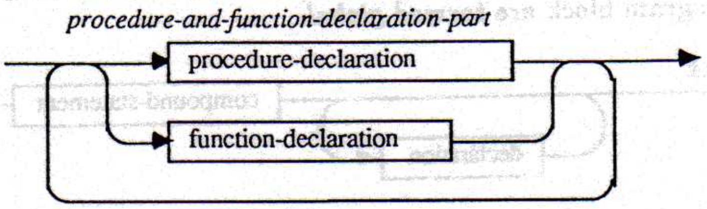
The *type declaration part* contains *type declarations* (see Chapter 13) local to the block.



The *variable declaration part* contains *variable declarations* (see Chapter 14) local to the block.



The *procedure and function declaration part* contains all *procedure* and *function* declarations local to the block (see Chapter 17).



Rules of Scope

Prior to any use of an identifier or label, it must be declared. Once declared, however, its use is valid within a defined range of the program. This range is called the scope of the declaration.

Scope of a Declaration

The appearance of an identifier or label in a declaration defines the identifier or label. That is, the identifier or label is associated with its *meaning* at the point of declaration. All other applied occurrences of the identifier or label must be within the *scope* of this declaration. The scope of a declaration is the block that contains the declaration, and all blocks enclosed by that block except as explained in the sections which follow.

Redeclaration in an Enclosed Block

Suppose that *outer* is a block, and that *inner* is another block declared within *outer*. If an identifier declared in block *outer* has the same spelling as an identifier declared in block *inner*, then block *inner* and all blocks enclosed by *inner* are excluded from the scope of the declaration in block *outer*.

Position of Declaration Within Its Block

The declaration of an identifier or label must precede all applied occurrences of that identifier or label in the program text. That is, identifiers and labels cannot be used until they are declared.

There is one exception to this rule. In a type declaration, the domain type of a pointer type can be an identifier that has not yet been declared. However, the forward referenced identifier must be declared somewhere in the same declaration part as the pointer type.

Redeclaration Within a Block

An identifier or label cannot be declared more than *once* within a block, unless it is declared within a contained block, or if it appears in the field-list of a record declaration.

A record field identifier is declared within a record type. It is meaningful only in combination with a reference to a variable of that record type. Therefore, a field identifier can be declared within the same block as another identifier with the same spelling, as long as it has not been declared previously in the same field-list. An identifier that has been declared can be used again as a field identifier in the same block.

Identifiers of Standard Objects

Complete Pascal provides a set of standard (predeclared) constants, types, procedures, and functions that behave as if they were declared in a block that contains the entire program. Their scope is the entire program or unit (See Chapter 20).

Scope of Unit Interface Identifiers

The identifiers declared within the unit interface part of a unit are provided to a program or unit which specifies the unit name in a Uses clause. These identifiers act as if they were declared in the same block where the Uses clause appears.

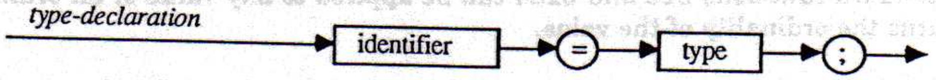
Activations

The execution of a block is referred to as an *activation* of a block. At any given time, a block can have zero or more activations. If a block is not currently being executed, then it has zero activations. If a block is being executed, then there is at least one activation. When a block has more than one activation, it is said to be *recursive*.

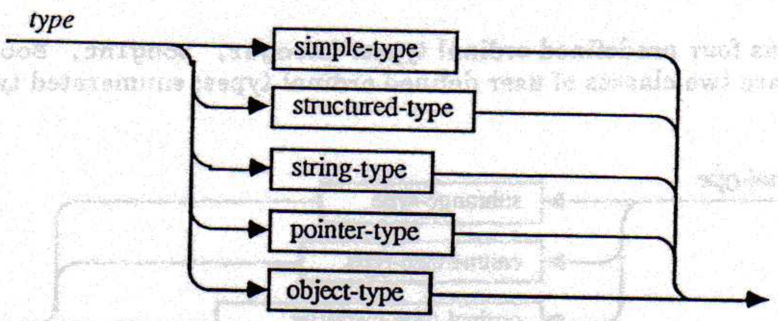
Chapter 13

Types

When a variable is declared, its *type* must be given. The type of a variable determines the set of values that the variable can assume and the operations that can be performed upon it. A type declaration introduces an identifier to denote a type.



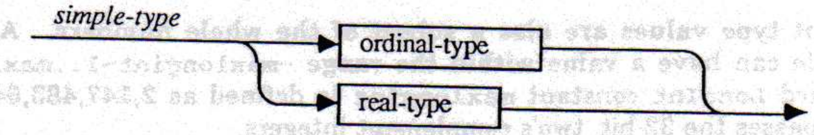
Several predefined types exist in Complete Pascal and are discussed in detail below. There also exists a capability to create user-defined types. User-defined types are also discussed in detail in this chapter.



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.

Simple Types

All the simple types define ordered sets of values.



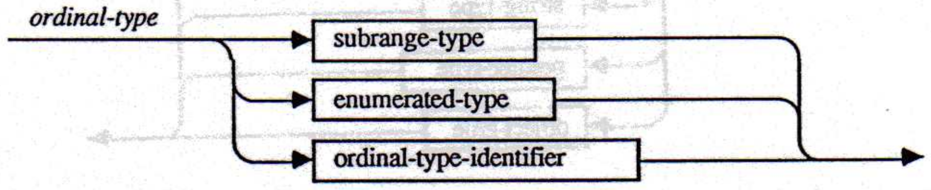
An integer type identifier is one of the standard identifiers `Integer` or `LongInt`. A real type identifier is one of the standard identifiers `Real`, `Single`, `Double`, `Comp` or `Extended`. The type identifier `Comp` may also be spelled as `Computational`. See Chapter 11 on how to denote constant integer and real type values.

Ordinal Types

Ordinal types are the subset of the simple types that have the following special characteristics:

- The possible values of an ordinal type are an ordered set and every value has an ordinality which is an integral value. Except for integer types, the first value of every ordinal type has ordinality 0, the next has ordinality 1, etc. For integer types, the ordinality of a value is the value itself. Every value of an ordinal type except the first has a predecessor based on the ordering of the type, and every value of an ordinal type except the last has a successor based on the ordering of the type.
- The standard functions `ord` and `ord4` can be applied to any value of an ordinal type, and it returns the ordinality of the value.
- The standard function `pred` can be applied to any value of an ordinal type, and it returns the ordinality of the predecessor of the value.
- The standard function `succ` can be applied to any value of an ordinal type, and it returns the ordinality of the successor of the value.

Complete Pascal has four predefined ordinal types: `Integer`, `LongInt`, `Boolean`, and `Char`. In addition, there are two classes of user defined ordinal types: enumerated types and subrange types.



Each class of ordinal types is discussed below.

Standard Ordinal Types

`Integer`	Integer type values are a subset of the whole numbers. An integer type variable can have a value within the range `-maxint-1..maxint`, 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.
`LongInt`	LongInt type values are also a subset of the whole numbers. A LongInt type variable can have a value within the range `-maxlongint-1..maxlongint`. The standard `LongInt` constant `maxlongint` is defined as 2,147,483,647. The range encompasses the 32-bit, two's complement integers.

Arithmetic operations with integer type operands use `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 first converted to `LongInt`, 32-bit precision is used for the operator, and the result is of type `LongInt`.

- The expression on the right side of an assignment statement is evaluated independently of the left side.

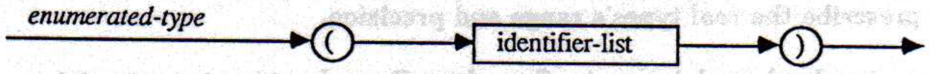
An `Integer` value may be explicitly converted to a `LongInt` by using the standard function `ord4` described in Chapter 20.

`Boolean` Boolean type values are denoted by the predefined constant identifiers `false` and `true`, where `ord(false)=0`, and `ord(true)=1`. Values of type `Boolean` are required by the Pascal If statement, Repeat statement, and While statement.

`Char` The `Char` type has a set of values that are the ASCII characters. The function call `Ord(Ch)`, where `Ch` is a `Char` value, returns the ordinality of `Ch`. A string constant of length 1 may be used to denote a constant `Char` value. Any value of type `Char` may be generated via the standard function `Chr`.

Enumerated Types

An *enumerated* type defines an ordered set of values by enumerating a collection of identifiers that denote these values. The ordering of these values is determined by the sequence in which the identifiers are listed. That is, for two enumeration identifiers *x* and *y*, if *x* precedes *y* then the ordinal value of *x* is less than the ordinal value of *y*.



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. The type of this constant is the enumerated type in which it is declared. The ordinality of an enumerated constant is its position in the identifier list, where the ordinality of the first enumerated constant in the list is always 0.

Examples of enumerated types:

```
suit  = ( club, diamond, heart, spade )
color = ( red, yellow, green, blue )
```

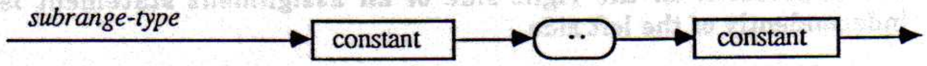
Given these declarations, `yellow` is an enumerated constant of type `color` with ordinality 1, `spade` is an enumerated constant of type `suit` with ordinality 3, and so on. For the above definitions of `suit` and `color`, the following relationships hold:

```
ord(club)     < ord(diamond)
ord(green)    > ord(red)
```

```
pred(diamond)  = club
succ(green)    = blue
```

Subrange Types

A *subrange* type defines a subset of the values of some ordinal type called the host type. The definition of a subrange type specifies the smallest and the largest value in the subrange.



Both constants in a subrange type must be of the same ordinal type. Subrange types of the form *a..b* require that *a* is less than or equal to *b*.

A variable of subrange type possesses all the properties of variables of the host type, with the restriction that its value must always be one of the values in the range defined by the subrange type.

Examples of subrange types:

```
1..100
-128..127
spade..heart
```

Real Types

The *real* types have sets of values that are subsets of the real numbers, which can be represented in floating point notation using a fixed number of digits. In general, a floating point notation of a value *n* is comprised of a set of three values *m, b,* and *e* such that $m * b^e = n$, 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 types's range and precision.

There are four standard real types in Complete Pascal: `Single`, `Double`, `Comp` and `Extended`. In addition, the standard identifier `Real` is defined to be equivalent to the type `Single`. The real types differ in the range and precision of values they can represent.

Table 13-1
Real Types

Type Identifier	Memory Size	Magnitude
`Real, Single`	4 bytes	approx 1.4E-45 to 3.4E38
`Double`	8 bytes	approx 5.0E-324 to 1.7E308
`Extended`	10 bytes	approx 1.9E-4951 to 1.1E4932
`Comp, Computational`	8 bytes	approx -9.2E18 to 9.2E18

Real type variables may have the possible values:

- Finite values (a subset of the mathematical real numbers). Note that the value zero has a sign associated with it (i.e. zero may be either positive or negative).

- Infinite values, +INF and -INF, which result from overflow of a real number storage type and division of a finite number by zero.

- NaNs (Not a Number) represent the results of operations which have no mathematical interpretation, such as the result of multiplying ±∞ by zero is a NaN. NaNs are represented in textual format as `NaN(x)`, where x is an integer that defines the source of the NaN.

As defined in Table 13-1, the four real types differ in the range and precision of values and storage space required.

- `Single` (or `Real`) types require 4 bytes of storage and their magnitude ranges from approximately 1.401298464E-45 to 30402823466E38. Precision of *Single* values is 7 or 8 digits.

- `Double` types require 8 bytes of storage and their magnitude ranges from approximately 5.0E-324 to 1.7E4932. Precision of *Double* values is 15 or 16 digits.

- `Extended` types require 10 bytes of storage and their magnitude ranges from approximately 1.9E-4951 to 1.1E4932. Precision of *Extended* values is 19 or 20 digits.

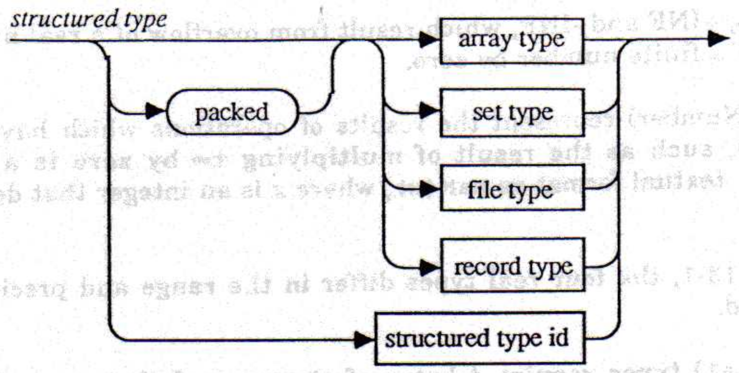
- `Comp`, or `Computational`, types require 8 bytes of storage and their range of values is approximately -9.2E18 to +9.2E18. The exact range is $-2^{63}+1$ to $2^{63}-1$ (-2^{63} is treated as a NaN). `Comp` type variables are used for fixed-point values, where the decimal point is placed by the application. Although `Comp` types may store only whole numbers, they should not be considered extensions of integer types since `Comp` values are converted to `Extended` before being used in calculations.

Real types provided in Complete Pascal are implemented using the Apple IIGS *Standard Apple Numeric Environment (SANE)* package. Real type operations result in code which calls SANE. Additional, real type operations are available in the interface file SANE.p. For complete information about SANE, see the *Apple Numerics Manual.*

❖ **Note:** All real values are converted to the type `Extended` by the compiler before calculations are performed so that maximum accuracy can be obtained. Thus, calculations on data stored as the type `Extended` result in faster and more compact code than calculations on data stored in other representations. The smaller representations should be used when data storage space is more critical than execution speed.

Structured Types

A *structured-type,* characterized by its structuring method and by its component-type(s), holds more than one value. The type of a component may itself be structured and there is no inherent limit on the number of levels to which types can be structured.



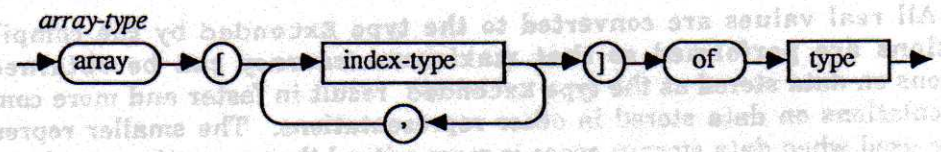
The use of the word PACKED in the declaration of a structured type indicates that storage organization of all values of that type should be compressed to economize storage, even if this causes the access of the component of a variable of this type to be less efficient.

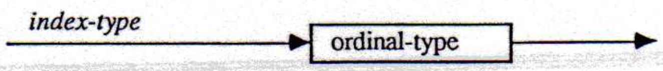
Structured types may contain other structured types in its definition. However, use of the word PACKED only affects the representation of one level of the structured type in which it occurs. Although the word PACKED can be used when declaring a structured type, PACKED only affects the storage of record and array types. Note that you cannot use components of packed variables as actual variable parameters to procedures and functions.

❖ **Note:** Complete Pascal only supports packing to byte boundaries. Bit level packing is not implemented. For more information regarding storage allocation and data representation see Appendix D, Inside Complete Pascal.

Array Types

An *array* type defines a structured type which has a fixed number of components that are all of the same type.





The type that follows the word of is the component type of the array. The number of elements is determined by one or more *index types*, one for each dimension of the array. The index type must be an ordinal type. There is no inherent limit on the number of dimensions an array type can have and index types of multi-dimensional arrays need not be of the same type.

❖ **Note:** Complete Pascal restricts the size of an array variable to 32,767 bytes of storage.

An array type of the form

```
packed array [1..n] of Char
```

is referred to as 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).

Examples of array types:

```
array [1..100] of Real
packed array [color] of Boolean
array [Boolean] of Integer
array[low..high] of Boolean
```

If the component type of an array is also an array, then the resulting type can be considered either an array of arrays or as a single multi-dimensional array. For example,

```
array [Boolean] of array [0 .. MaxSize] of Real
```

is equivalent to

```
array [Boolean , 0 .. MaxSize] of Real
```

A component of an array may be accessed by giving the index(es) of the component inside brackets immediately following the array identifier. Multiple sets of brackets may be used. For example, given the declaration

```
var anArray: array [1 .. MaxLength , 1 .. MaxWidth] of Real;
```

the expressions

```
anArray[1,1]
anArray[1][1]
```

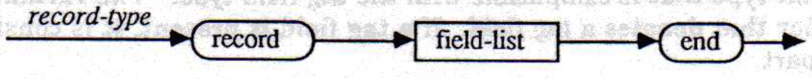
both access the same component, the first element of the first subarray of the array `anArray`. Note that

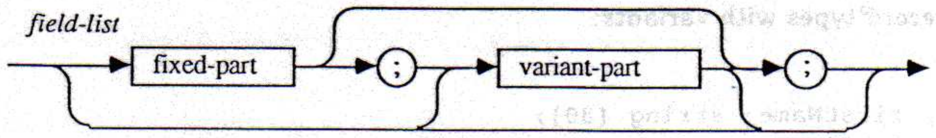
```
anArray[2]
```

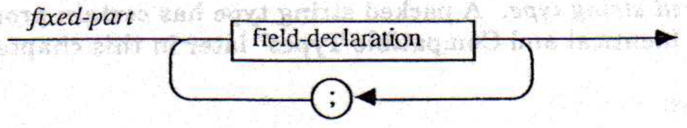
accesses the entire second subarray.

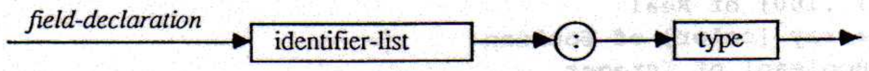
Record Types

A *record* type consists of a fixed collection of components called fields, each of which may be a different type. For each component, the record type specifies the type of the field and an identifier that names it.









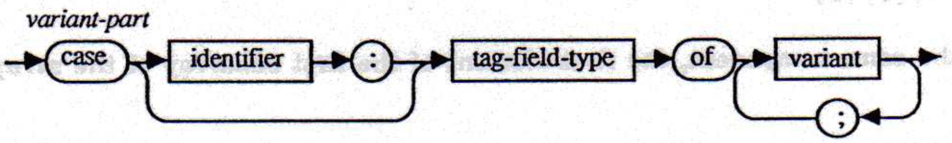
The fixed part of a record type specifies a field list that is always accessible in a variable of the record type, giving an identifier and a type for each field. Each of these fields contains data that is always accessed in the same way.

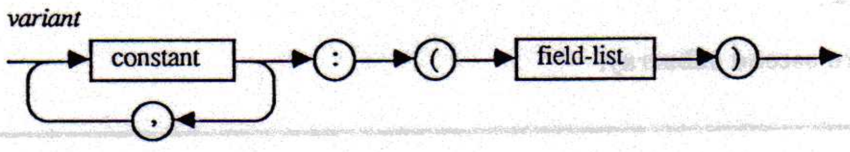
Example of a record type

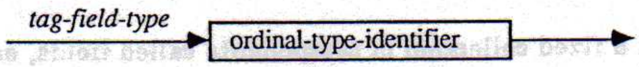
```
    record
       year:    Integer;
       month:   1..12;
       day:     1..31;
    end
```

A variant part consists of several alternative field lists which are allocated in the same memory space of a record variable, thus allowing data in this space to be accessed in more than one way. Each of the lists of fields is called a *variant*. The variants "overlay" each other in memory, and all fields of all variants are accessible at all times.

❖ **Note:** Complete Pascal restricts the size of a record variable to 32,767 bytes of storage.







Each variant is introduced by one or more constants. All of the constants must be distinct and must be of an ordinal type that is compatible with the tag field type. The variant part allows for an optional identifier that denotes a *tag field*. If a tag field is present, it is considered a field of the previous fixed part.

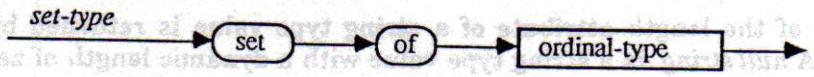
Examples of record types with variants:

```
  record
      name, firstName: string [80];
      age:  0..99;
      case married: Boolean of
         true:  (spousesName: string [80]);
```

```
        false: ()
    end

    record
        x,y: Real;
        case kind: figure of
            rectangle: (height,width:  Real);
            triangle:  (side1,side2,angle:  Real);
            circle:    (radius:  Real);
    end
```

Set Types

A *set* type has a range of values that is the powerset of some ordinal type, called the base type. Each possible value of a set type is a subset of the possible values of the base type.



❖ Note: Complete Pascal restricts the base type to not more than 256 possible values. If the base type is a subrange of *integer*, it must be in the limits 0..255. For more information regarding storage allocation and data representation see Appendix D.

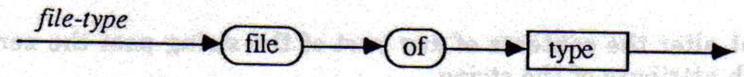
Every set type can hold the value `[ ]`, called the *empty set.*

Example of set type:

```
    Set of Char
    Set of 0 .. 31
    set of (red, green, blue)
```

File Types

A *file* type is a structured type consisting of a linear sequence of components that are all of one type, the component type. The component type may be any type that is not a file type or a structured type that contains a file type component. The number of components is not fixed by the file type declaration.



The standard file type `Text` denotes a special packed file of characters organized into lines. Files of type `Text` are supported by special I/O procedures discussed in Chapter 19.

❖ Note: Due to the representations of types in Complete Pascal, a *file of Char* accesses file components which are 16-bit words, whereas a *packed file of Char (or Text)* accesses file components which are 8-bit bytes. For more information regarding storage allocation and data representation see Appendix D.

Example of file type:

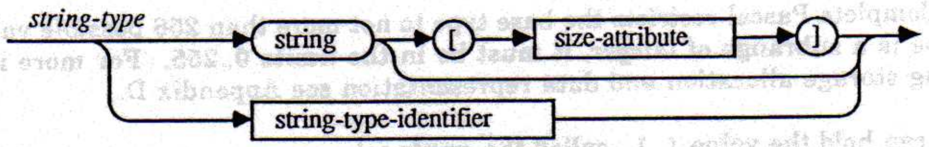
```
IntFile = file of Integer
```

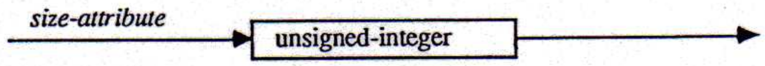
Complete Pascal allows file variables to be passed to procedures and functions only as variable parameters.

Chapter 19 presents a detailed discussion of files and operations which may be performed on the different types of files possible.

String Types

A *string* type value is a sequence of characters with a dynamic length attribute and a constant size attribute from 1 to 255. The constant size is a maximum limit on the length of any value of this type. If an explicit size attribute is not given for a string type, then it is given a size of 255 by default.
The current value of the length attribute of a string type value is returned by the standard function `Length`. A *null string* is a string type value with a dynamic length of zero.





The ordering relationship between any two string values is determined by lexical comparison based on the ordering relationship between character values in corresponding positions in the two strings. When the two strings are of unequal lengths, each character in the longer string that does not correspond to a character in the shorter one compares "higher"; thus the string value 'attribute' is greater than the value 'at'. Two strings must always have the same lengths to be equal.

A string is stored as a one-byte length field followed by the characters in the string. Therefore, the length of the string can be changed by reassigning the zeroeth character as follows:

```
aString[0]  := chr(5);
```

Note that this does not alter the contents of any part of the string past the zeroeth character, it only changes the length attribute of the string.

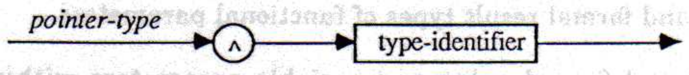
Operators applicable to strings are specified in Chapter 15. Standard procedures and functions for manipulating strings are described in Chapter 20.

Example of string types:

```
string[50]
string[255]
string
```

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 heap for a dynamic variable and stores the address of that area of memory in the pointer value. The `@` operator directs the pointer variable to the memory area containing any existing variable. The `Pointer` function points the pointer variable to a specific memory address. The `Pointer` function and the `@` operator avoid Complete Pascal's type-checking safeguards and should be used with caution. `New` and `Pointer` are discussed in Chapter 20.

Chapter 14 discusses accessing a variable pointed to by pointers.

The predeclared constant identifier `Nil` represents a pointer valued constant that is a possible value of every pointer type. Conceptually, `Nil` is a pointer that does not point to anything.

Example of pointer types:

```
^LongInt;
^Char;
^String[32];
```

Identical and Compatible Types

Two types may or may not be *identical*, and identity is required in some contexts. Other times, even if not identical, two types need only be *compatible*, and other times *assignment compatibility* is required.

Type Identity

There are three levels of type compatibility in Pascal:

- Two types may be the same. Two types are the same when they are declared using the same type identifier or when their definitions can be traced back to the same type identifier.
- Two types may be compatible.
- Two types may be assignment compatible.

Compatibility and assignment compatibility are discussed below.

Identical types are required only in the following contexts:

- Between actual and formal variable parameters.
- Between actual and formal result types of functional parameters.
- Between actual and formal value and variable parameters within parameter lists of procedural or functional parameters
- When a one-dimensional `PACKED ARRAY OF Char` is being compared with another via a relational operator.

Parameters are discussed in Chapter 17.

Assignment compatibility is usually required in other contexts, although simple compatibility is occasionally sufficient.

Compatibility of Types

Compatibility is required in most contexts where two or more entities are used together (i.e. in expressions, and `FOR` statement control variables and their initial and final values, etc.). Specific instances where type compatibility is required are noted elsewhere in this manual.

Two types are *compatible* if any of the following are true:

- Both are identical.
- One is a subrange of the other.
- Both are subranges of identical types.
- Both types are set types with compatible base types.
- Both are string types.
- Both are of type packed string type and have the same number of components.

Assignment Compatibility

Assignment compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment statement) or implicitly (as in passing value parameters). A value of type t_2 is assignment compatible with a type t_1 if any of the following are true:

- t_1 and t_2 are identical types and neither is a file type nor a structured type that contains a file type component.
- t_1 is a real type and t_2 is an integer type.

- t_1 and t_2 are compatible ordinal types, and the value of type t_2 is within the range of possible values of t_1.
- t_1 and t_2 are compatible set types, and all the members of the value of type t_2 are within the range of possible values of the base type of t_1.
- t1 and t2 are both of type `PACKED ARRAY OF Char`.
- t_1 is a string type or a Char type and t_2 is a string type of a quoted character constant.
- t_1 is a packed string type with *n* components and the value of type t_2 is a string type of a quoted character constant and has a length of *n*. This is not true, however, if n=1, because a string constant of length 1 is a quoted character constant.

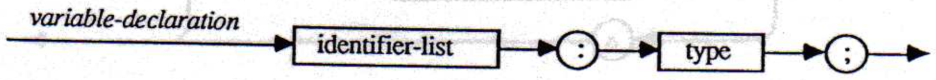
It is an error if assignment compatibility is required and none of the above is true.

Chapter 14

Variables

Variable Declarations

A *variable declaration* is used to allocate and associate a piece of storage with a particular type. A variable is an entity in which value(s) are stored. Each identifier in an identifier-list of a variable declaration denotes a distinct variable possessing the type of the variable declaration.



The occurrence of an identifier within the identifier list of a variable declaration declares it as 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 creates a new variable using the same identifier, without affecting the value of the original variable.

All variables have undefined values with the start of activation of its containing block. The main program block is activated when the program is executed. The procedure and function blocks are activated each time the procedure or function is called.

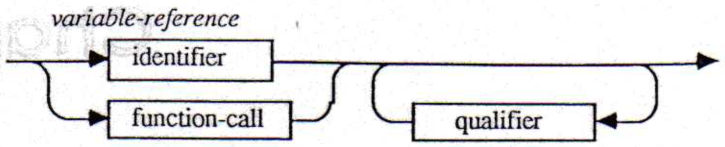
Variables declared in procedure or function blocks can be no larger than 32K bytes.

Examples of variable declarations:

```
x,y,z:          real;
c:              color;
p1,p2:          person;
today:          date;
operator:       (plus, minus, times);
digit:          0..9;
coord:          polar;
done,error:     boolean
```

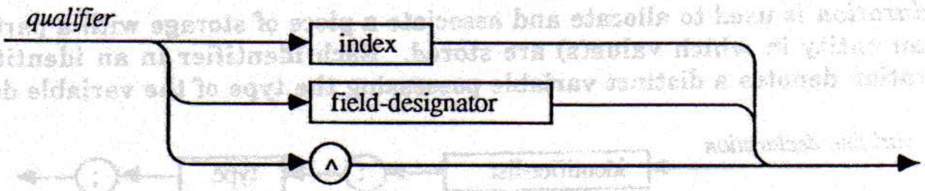
Variable References

A *variable reference* denotes either an entire variable, a component of a structured or string type variable, a dynamic-variable pointed to by a pointer type variable, or a variable reached through a function call. Syntax for the various kinds of qualifiers used for variable access is given here.



Qualifiers

A variable reference is a variable identifier followed by zero or more *qualifiers* which modify the meaning of the variable reference.



There can be zero or more qualifiers following a variable identifier or function call, depending on the levels of structure in the variable and which particular level to be accessed. For example, given the following declaration

```
    var aMultiDimArray: array [1..100] of array[1..100] of Integer
```

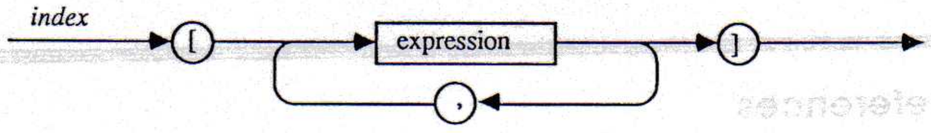
the following references are all valid expressions.

```
    aMultiDimArray
    aMultiDimArray[1]
    aMultiDimArray[1, 1]
```

The first example accesses the entire array, the second example the entire first subarray. The third example access just the first component of the first subarray.

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 qualifier 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 qualifier that specifies the character position.



Examples of indexed arrays:

```
    m[i, j]
    a[i + j]
```

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index types in the array declaration. The index expression must be assignment compatible with the corresponding index type.

When indexing a multi-dimension array, multiple indexes or multiple expressions within an index can be used interchangeably. For example,

```
MyMatrix [i] [j]
```

has the same meaning as

```
MyMatrix [i,j]
```

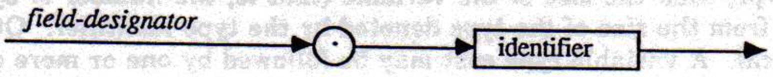
A string variable can be indexed with a single index expression, whose value must be within the range *0..n*, where *n* is the declared size of the string. Indexing a string accesses one character of the string value. The first character of a string variable (index 0) contains the dynamic length of the string.

In general, a value cannot be assigned to an individual character position in a string unless a character previously occupied that position. That is, if the dynamic length of the string is less than the individual character position being manipulated, the operation will leave the string unchanged. Values may not be assigned to character positions beyond the current length of the string.

Predefined procedures for string manipulation are described in Chapter 20.

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 that specifies the field.



Examples of field designators:

```
today.year
p2^.pregnant
```

It is an error to access a variant component of a record that is not active. See Chapter 13 for a discussion on variant records.

The record variable identifier and period (.) may be omitted inside a `with` statement that lists the record variable identifier. See Chapter 16 for a description of the `with` statement.

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.

Dynamic variables and pointer values that point to them are created by the standard procedure `New`. Additionally, the @ operator and the standard procedure `Pointer` may be used to create pointer values that are not in fact pointers to dynamic variables, but are treated as such.

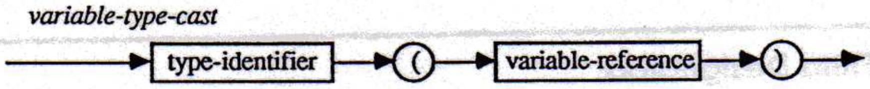
The constant `Nil` does not point to any variable. It is an error if you access a dynamic variable when the pointer's value is `Nil` or undefined.

Examples of references to dynamic variables:

```
p1^
p1^.sibling^
```

Variable Type Casts

A variable reference of one type can be changed into a variable reference of another type through a mechanism called 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. If the variable and type identifier are both of an ordinal type, then the size of the variable (that is, the number of bytes of storage it occupies) may differ from the size of the type denoted by the type identifier. Otherwise, the two sizes must be identical. A variable type cast may be followed by one or more qualifiers just as a variable reference.

Complete Pascal does not support the type cast of set variables.

Examples of variable type casts:

```
type Point = record
                v,h: integer;
             end;
var p: Point;
    l: LongInt;
begin
   p := Point (l);
   l := LongInt (p);
   LongInt (p) := LongInt (p) + $00020002;
end;
```

Chapter 15

Expressions

Expressions denote values. The simplest expression is merely a variable reference, however, most expressions consist of *operators* and *operands*. Most Pascal operators are *binary*, that is, they have two operands. The remaining operators are *unary* and have only one operand. When more than one operator appears in an expression, precedence rules are applied to determine which operands are associated with which operators. For example, the expression:

```
a+b*c
```

can be interpreted as either (a+b)*c or a+(b*c). The precedence rules make the interpretation unambiguous:

- When an operand appears between two operators of different precedence, it is bound to the operator with the higher precedence.
- When an operand is written between two operators of the same precedence, it is bound to the operator to the left.
- A parenthesized expression is always evaluated before it is applied as an operand.

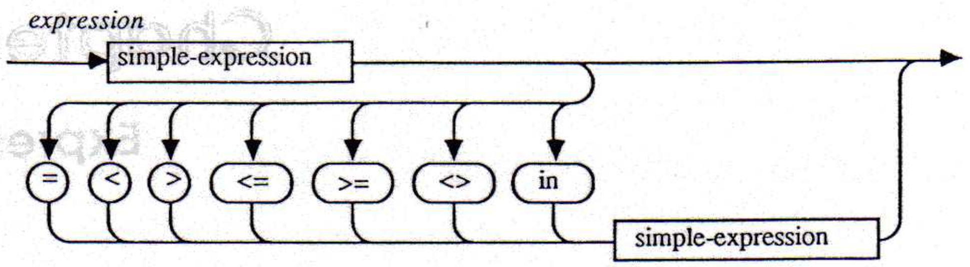
The precedence of the Complete Pascal operators are given in Table 15-1.

Table 15-1
Precedence of Operators

Operators	Precedence	Category
@, NOT	highest	exponent and unary operators
*, /, DIV, MOD, AND	second	"multiplying" operators
+, -, OR	third	"adding" operators and sign
=, <>, <, >, <=, >=, IN	lowest	relational operators

Thus, a+b*c is interpreted as a+(b*c), since * has a higher precedence than +. Note that a+b-c is interpreted as (a+b)-c, since + and - have the same precedence.
The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple expressions.

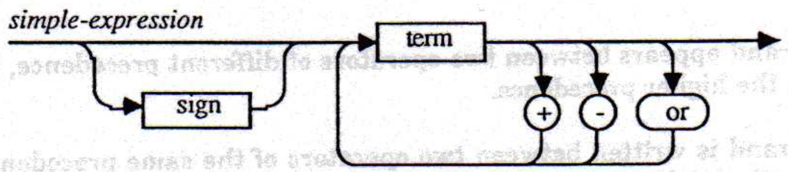
The syntax of an expression consists of *relational* operators applied to simple expressions:



Example of expressions:

```
x  =  1.5
c in huel
done <> error
p <= q
```

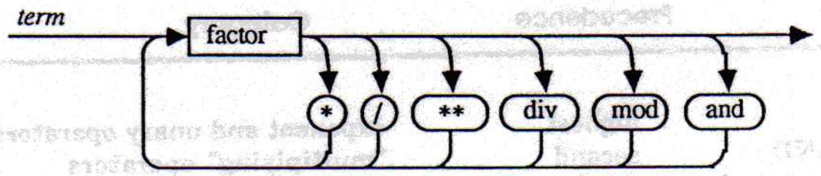
The syntax of a simple expression consists of *adding* operators and *signs* applied to terms:



Examples of simple expressions:

```
x+y
-x
hue1 + hue2
b or c
```

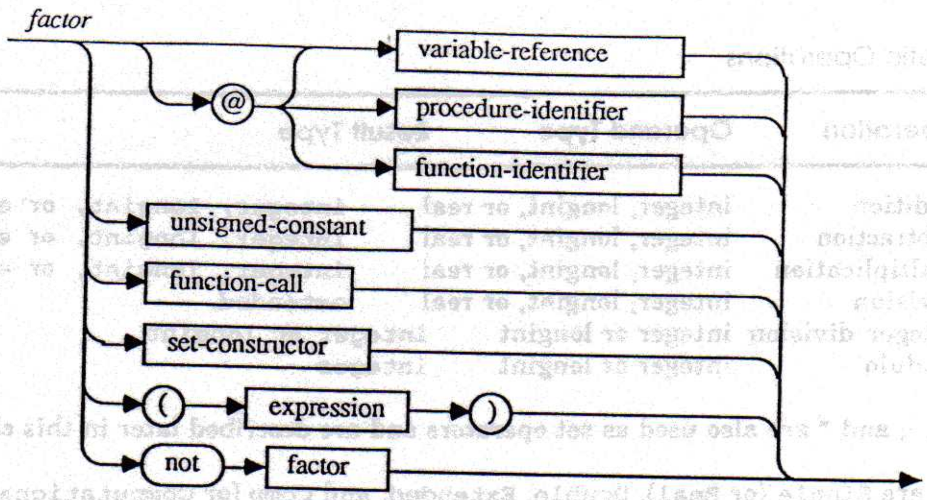
The syntax of a term consists of *multiplying* operators applied to factors:



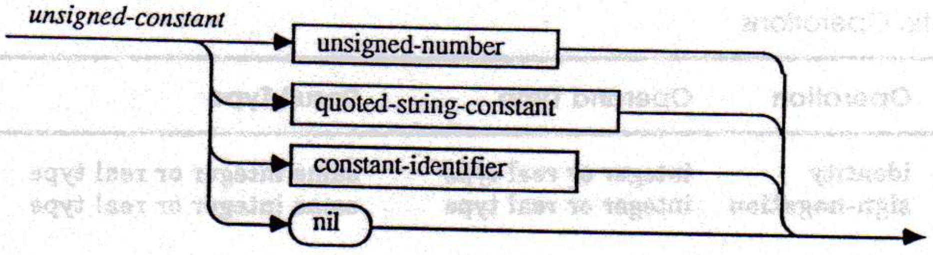
Examples of terms:

```
x * y
e / (1 - e )
done and error
```

The syntax for a factor consists of the following basic constructs:



An unsigned constant has the following syntax:



Examples of factors:

`x`	{variable reference}
`@x`	{pointer to a variable}
`15`	{unsigned constant}
`'hello world'`	{unsigned constant}
`(x+y+z)`	{sub expression}
`sin (x/2)`	{function call}
`not q`	{negation of a Boolean}
`['A'..'F','a'..'f']`	{set construction}

Operators

The operators are classified as arithmetic operators, boolean operators, set operators, relational operators, and the @ operator.

Arithmetic Operators

The following two tables show the types of operands and results for the binary and unary arithmetic operators respectively.

Table 15-2
Binary Arithmetic Operations

Operator	Operation	Operand Type	Result Type
+	addition	integer, longint, or real	integer, longint, or extended
-	subtraction	integer, longint, or real	integer, longint, or extended
*	multiplication	integer, longint, or real	integer, longint, or extended
/	division	integer, longint, or real	extended
div	integer division	integer or longint	integer or longint
mod	modulo	integer or longint	integer

The symbols +, -, and * are also used as set operators and are described later in this chapter.

The real types are `Single` (or `Real`), `Double`, `Extended`, and `Comp` (or `Computational`) and are discussed in Chapter 13.

Table 15-3
Unary Arithmetic Operations

Operator	Operation	Operand Type	Result Type
+	identity	integer or real type	same integer or real type
-	sign-negation	integer or real type	same integer or real type

If both operands of the +, -, *, `div`, or `mod` operators are of the same integer type (`Integer` or `LongInt`), the result is always of the same integer type. If one of the operands is type `LongInt` and the other is type `Integer`, then the integer operand is first converted to `LongInt` and the result type is `LongInt`. In either case, the resultant value is determined by the normal mathematical rules for integer arithmetic. It is an error if the value of the result is outside the range `-maxint-1..maxint` or `-maxlongint-1..maxlongint`. for `Integer` and `LongInt` result types respectively.

❖ **Note:** The range of any operator is not infinite. Given an expression as follows:

```
(a+b)-c
```

where a, b, and c are all integer values, an overflow error will result if the intermediate result of a+b yields a result greater than `maxint`.

If one of the operands of the +, -, or * operators is of any real type, the result is always of type `Extended`, and has a value that is an approximation of the normal mathematical result. The result of the / operator is always type `Extended`.

If the operand of the identity or sign negation operator is of an integer type, the result is always of the same integer type and the absolute value of the result is always identical to the absolute value of the operand.

If the operand of the identity or sign negation operator is of any real type, the result is always of type `extended` and the absolute value of the result is always identical to the absolute value of the operand.

Boolean Operators

The types of operands and results for Boolean operations are shown in the following table.

Table 15-4
Boolean Operations

Operator	Operation	Operand Type	Result Type
or	disjunction	boolean	boolean
and	conjunction	boolean	boolean
not	negation	boolean	boolean

The result of a boolean operation is determined by the normal rules of boolean logic, e.g. *a and b* evaluates to *true* if and only if both *a* and *b* are true.

Set Operators

The types of operands and results for set operations are shown in Table 15-5.

Table 15-5
Set Operations

Operator	Operation	Operand Type
+	union	compatible set types
-	difference	compatible set types
*	intersection	compatible set types

The results of the set operations are determined by the normal rules of set logic. For example:

- An ordinal value *c* is in the set a+b if and only if *c* is in a *or* in b.
- An ordinal value *c* is in the set a-b if and only if *c* is in a *and not* in b.
- An ordinal value *c* is in the set a*b if and only if *c* is in a *and* in b.

Relational Operators

The types of operands and results for relational operations are shown in the following table.

Table 15-6
Relational Operations

Operator	Operation	Operand Type	Result Type
=	equal to	compatible set, simple, or pointer types	`boolean`
<>	not equal to	compatible set, simple, or pointer types	`boolean`
<	less than	compatible simple types	`boolean`
>	greater than	compatible simple types	`boolean`
<=	less than or equal to	compatible simple types	`boolean`
>=	greater than or equal to		compatible simple types
`boolean`			
<=	subset of	compatible set types	`boolean`
>=	superset of	compatible set types	`boolean`
`in`	member of	left operand: any scalar type T right operand: type SET OF T	`boolean`

Comparing Ordinals

When the operands of =, <>, <, >, >=, or <= are of an ordinal type, they must be of compatible types unless one of the operands is a real type. In this case, the other operand is allowed to be an integer type. The result is the mathematical relation of their ordinalities. When comparing real types, the results may not be as expected since the representation of a real value is only an approximation.

❖ **Note:** Because of extensions provided for use with the Standard Apple Numeric Environment (SANE), the result of a comparison can be unordered. An unordered result occurs from a comparison involving a NaN (Not a Number). One important effect is that NOT (a<b) is true if either a is greater than b or a and b are unordered. Use of the `Relation` function, which is included in the SANE library, may be used to test for an unordered comparison.

Comparing Strings

When the relational operators =, <>, <, >, <=, or >= are used to compare strings, they are compared according to their lexicographic ordering. Note that any two string values can be compared since all string values are compatible. Additionally, a `Char` value is compatible with a string type value, and when the two are compared, the `Char` value is treated as a string type value with length one. 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 Arrays of Char

The relational operators =, <>, <, >, <=, and >= can also be used to compare two values of a packed string type if both have the same number of components. If that number is *n*, then the result is the same as if the values were string type with each having a *length* of *n*. See Chapter 13 for more details on packed arrays of char.

Comparing Sets

If a and b are set operands then

- a=b is *true* if, and only if, every member of *a* is a member of *b* and every member of *b* is a member of *a*; otherwise, a<>b.
- a<=b is *true* if, and only if, every member of *a* is also a member of *b*.
- a>=b is *true* if, and only if, every member of *b* is also a member of *a*.

Thus, a=b, and a<>b denote the equivalence and non-equivalence of the sets *a* and *b* respectively, and a<=b and a>=b denote the inclusion of *a* in *b* and the inclusion of *b* in *a* respectively.

Comparing Pointers

The relational operators = and <> may be applied to compatible pointer type operands. Two pointers are equal if and only if they denote the same object.

Testing Set Membership

The in operator returns true if the value of the ordinal type operand is a member of the set type operand; otherwise it yields the value false. The type of the left operand must be compatible with the base type of the right operand.

The @ Operator

A pointer value that points to a variable, procedure, or function can be created with the @ operator. The operand and result types are shown in Table 15-7.

@ is a unary operator taking a single variable reference or a procedure or function identifier as its operand and computing the value of its pointer. If the operand to the @ operator is a variable reference, then the pointer value is the address in memory where the variable is stored. If the operand to the @ operator is a procedure or function identifier, then the pointer value is the procedure or function's *entry point*. The type of the value is equivalent to the anonymous pointer type of the pointer constant *nil*, i.e. it can be assigned to any pointer variable.

Table 15-7
Pointer Operations

Operator	Operation	Operand Type	Result Type
@	pointer formation	variable, parameter, procedure, or function	pointer

The @ operator is a unary operator taking a single variable, parameter, procedure, or function as its operand and computing the value of its pointer. The type of the value is equivalent to the type of NIL and consequently can be assigned to any pointer variable. The pointer type is discussed in Chapter 13.

@ with a Variable

Using the @ operator with ordinary variables (not parameters) is straightforward. For example, given the declarations

```
    type twochar = packed array [0..1] of Char;
    var  int: Integer;
         twocharptr: ^twochar;
```

the statement

```
    twocharptr := @int
```

causes twocharptr to point to the variable int. Since the types Integer and twochar have the same storage requirements, the value of int, when accessed via twocharptr, is reinterpreted as type twochar.

@ with a Value Parameter

When @ is applied to a formal value parameter, the result is a pointer to the stack location containing the actual value. Let aParam be a formal variable parameter in a procedure, actParam be the variable passed to the procedure as aParam's actual parameter, and aPtr is a pointer variable. If the procedure then executes the statement

```
    aPtr := @aParam
```

then aPtr is a pointer to actParam on the stack and aPtr^ denotes the value of actParam.

@ with a Variable Parameter

When @ is applied to a formal variable parameter, the result is a pointer to the actual parameter. In this case, the pointer is simply taken from the stack. Let aParam be a formal variable parameter in a procedure, actParam be the variable passed to the procedure as aParam's actual parameter, and aPtr is a pointer variable. If the procedure then executes the statement

```
    aPtr := @aParam
```

then aPtr is a pointer to actParam and aPtr^ denotes the contents of actParam.

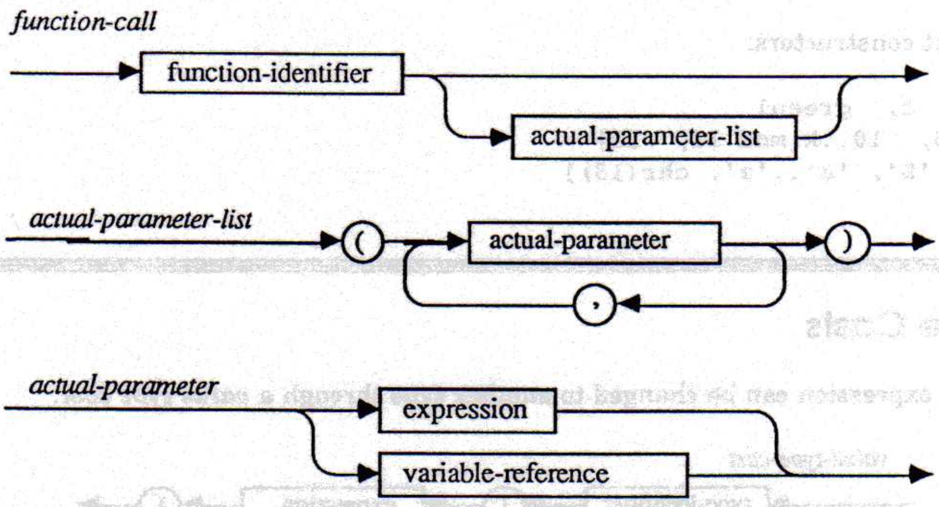
@ with a Procedure or Function

When the @ operator is applied to a procedure or function, the result will be a pointer to its entry point. There is no mechanism in Pascal for using such a pointer. The only use for a procedure pointer is to pass it to an Apple IIGS Toolbox procedure which the Toolbox will use to call the designated function with an assembly language JSL instruction. Procedure pointers are commonly used with the Toolbox to implement *filter procedures* and *definition procedures.*

❖ **Note:** The @ operator should only be used in conjunction with procedures and functions declared in the declaration part of the program or unit (global declarations) when the resulting pointer value is passed to a Apple IIGS Toolbox routine. Procedures and functions declared in the declaration part of another procedure or function (nested declarations) have a different calling convention than those in the declaration part of the program which is not compatible with the Apple IIGS Toolbox routines.

Function Call

A *function call* specifies the activation of the block associated with the function identifier. The result returned by the function activation is subsequently used as an expression value. If the function has any formal parameters, then the function designator must contain a corresponding list of actual parameters. Each actual parameter is substituted for the corresponding formal parameter.



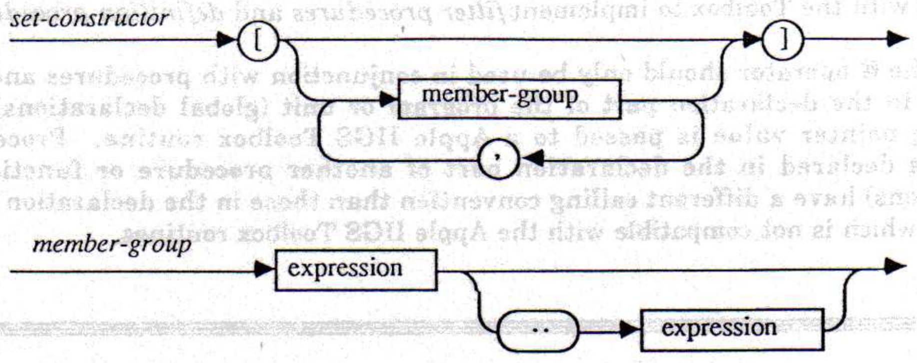
Examples of functions calls:

```
sum(a, 63)
sin(x+y)
eof(f)
ord(f^)
```

See Chapter 17 for a description of the procedure call statement.

Set Constructors

A *set constructor* denotes a value of a set type, 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 to every set type. Any member group *x..y* denotes as set members all values in the range *x..y*. If the value of *x* is greater than the value of *y*, then *x..y* denotes no members and *[x..y]* denotes the empty set.

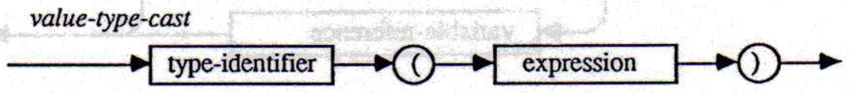
All expression values in the member groups of a particular set constructor must be of compatible ordinal types. If *a* is the smallest ordinal value in the resulting set, and if *b* is the largest ordinal value in the resulting set, then the base type of the resulting set is *a..b*.

Examples of set constructors:

```
[red,  c,  green]
[1,  5,  10..k mod 12,  23]
['A'..'Z', 'a'..'z', chr(13)]
```

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 pointer type. The result of the type cast is of the specified type, and its ordinal value is obtained by converting the expression. The syntax of a value type cast is almost identical to that of a variable type cast. However, value type casts operate on values, not variables, and can therefore not participate in variable references. That is, a value type cast may not have qualifiers appear on the left side of an assignment statement, or as an actual parameter where the formal parameter is declared as a VAR parameter.

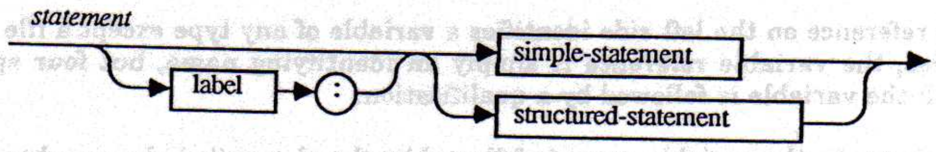
Examples of value type casts:

```
Integer('c')
Ptr($89F2)
Boolean(0)
```

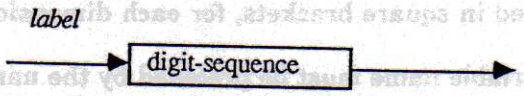
Chapter 16

Statements

Statements describe algorithmic actions that can be executed. There are two classes of statements – simple statements and structured statements. Statements may be prefixed by a label and a labeled statement can be referenced by goto statements.

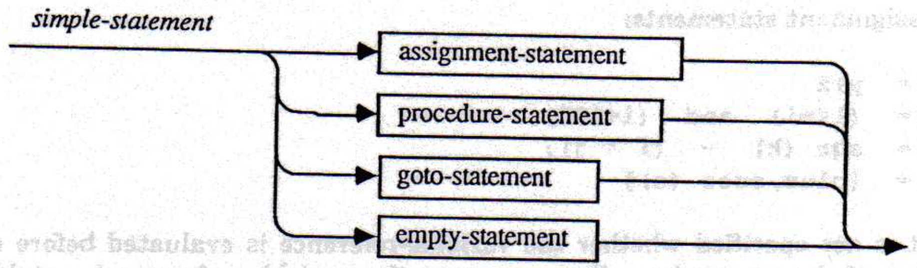


A label is a non-negative integer constant, and must first be declared in a label declaration. See Chapter 11 for more information on labels.



Simple Statements

A *simple* statement is a statement that does not contain any other statements. The *empty* statement is a simple statement which contains no symbols and denotes no action.



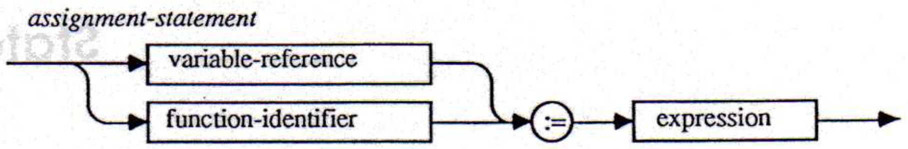
empty-statement

Assignment Statement

The *assignment* statement can be used to perform either of two actions:

- To replace the current value of a variable by a new value as specified by an expression.

- To specify an expression whose value is to be returned by a function.



The symbol := can be read as "set to." The expression must be assignment compatible with the type of the variable or the result type of the function. The function identifier must be the function identifier of the enclosing function block.

The variable reference on the left side identifies a variable of any type except a file type. With most variables, the variable reference is simply an identifying name, but four special cases exist in which the variable is followed by a qualification:

- string element — the variable name is followed by the element's index number enclosed in brackets.
- array element — the variable is identified by the array name followed by an index value, enclosed in square brackets, for each dimension of the array.
- record field — the variable name must be preceded by the name of its containing record and a period. The exception to this rule is if the statement is within the scope of a WITH statement for the associated record.
- dynamic variable — the variable reference is identified by the name of its pointer followed by a caret.

In addition, a variable reference on the left side may be type cast. Refer to Chapter 13 for more information on type compatibility and syntax for variable references and type casts.

Examples of assignment statements:

```
x     :=  y+z
p     :=  (1<=i)  and  (i<100);
i     :=  sqr  (k)   -   (i * j);
huel  :=  [blue,succ  (c)]
```

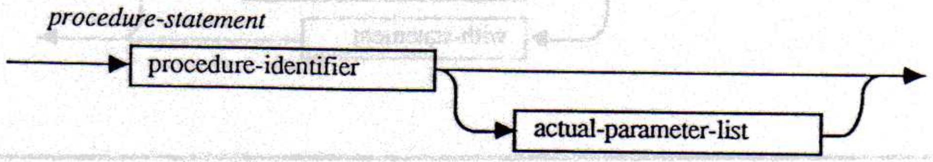
❖ **Note:** It is not specified whether the variable-reference is evaluated before or after the evaluation of the expression. However, once the variable-reference is established, it is not altered by side-effects of the remaining execution of the assignment-statement. Thus, the outcome of

```
A[i] := Some_function(i);
```

depends on whether `Some_function` modifies `i` and, if so, whether `Some_function` is evaluated before or after `A[i]`. A program is erroneous if it relies upon a particular order of evaluation.

Procedure Statement

A *procedure* statement specifies the activation of the procedure block denoted by the procedure identifier. If the procedure has any formal parameters, then the procedure statement must contain a matching list of actual parameters. Each actual parameter is substituted for the corresponding formal parameter as part of the procedure call.



The identifier used to activate a procedure must be identical to the identifier used in the procedure or function declaration. The parameters in the procedure or function declaration are called *formal parameters*; those in the calling statement are called *actual parameters.* The values of the actual parameter list are said to be passed to the formal parameters as part of the call. The number of parameters in the formal parameter list must be equivalent and compatible with the parameters in the actual parameter list. Three exceptions exist for actual and formal parameter compatibility:

- Subrange types are equivalent to their base types.
- A formal parameter of type `Longint` will accept an actual parameter of type `Integer`.
- Formal parameters preceded by `univ` accept any actual parameter that occupies the same space in memory. See Chapter 19 for a complete discussion on `univ`.

The actual parameters specified in any procedure or function call must also follow the following rules:

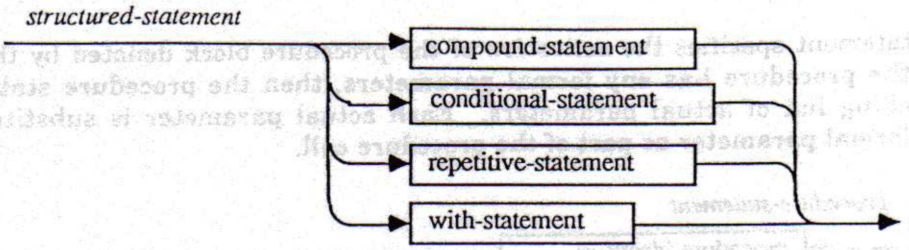
- Actual variable parameters, as opposed to value parameters, must be variables. Actual variable parameters cannot be constants, expressions, or elements of packed variables.
- The value of any actual string variable may be passed to any formal variable string parameter, regardless of length.
- If the value of an actual parameter exceeds the range of a formal parameter, an execution error will result.

Examples of procedure statements:

```
PrintHeading;
Transpose (a,n,m);
Find (name, address)
```

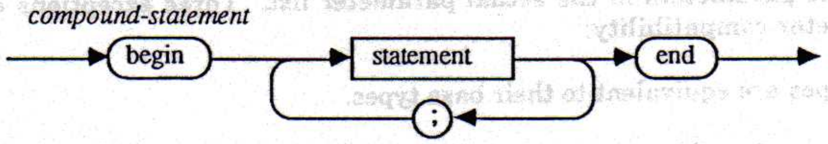
Structured Statements

Structured statements are made up of other statements that are to be executed either conditionally (conditional statements), repeatedly (repetitive statements), or in sequence (compound statement or with statement).



Compound Statements

The *compound* statement specifies the execution of a sequence of statements in the order in which they are written. The compound statement is treated as one statement in contexts where only a statement is allowed.



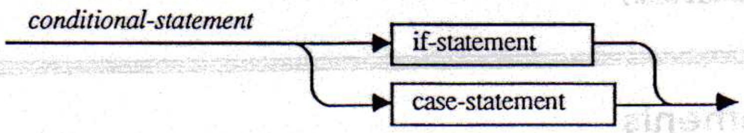
The body of every Pascal procedure, function, and main program consists of a single compound statement. To create a single compound statement out of a sequence of individual statements, preface the sequence with BEGIN and terminate it with END, separating the internal statements with semicolons. Compound statements may be nested within other compound statements. In this case, each END associated with a compound statement is paired with the nearest preceding BEGIN.

Example of a compound statement:

```
      begin
           z   :=  x;
           x   :=  y;
           y   :=  z
      end
```

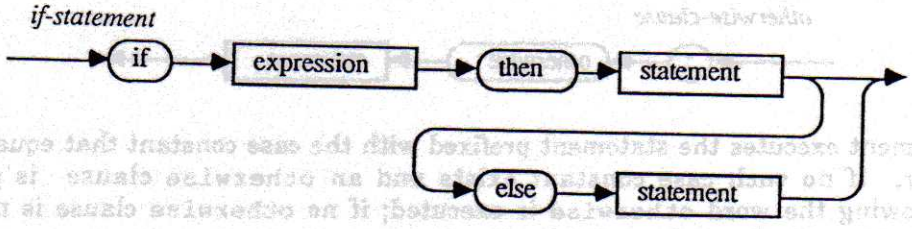
Conditional Statements

A *conditional* statement selects one or none of its component statements for execution.



If Statements

The *if* statement executes a single controlled statement (which may be a compound statement as defined above) if a `Boolean` expression is true. An optional `else` clause may be added that executes another statement if the `Boolean` expression is false. The statement associated with the `else` clause may also be a compound statement.



The expression between the `if` and `then`, usually formed out of relational and logical operators, must have a `Boolean` type.

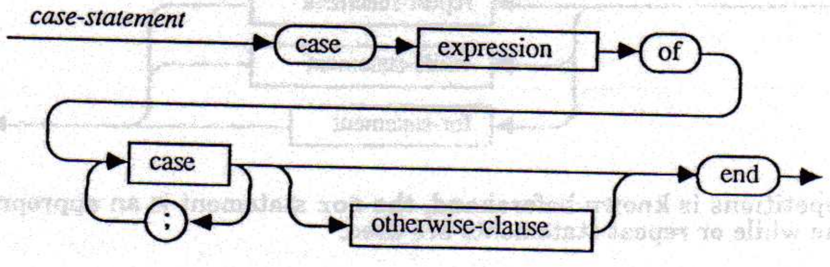
Nesting of `if` statements is allowed. In this case, each `else` is always associated with the nearest `if` statement that is not already associated with an `else`.

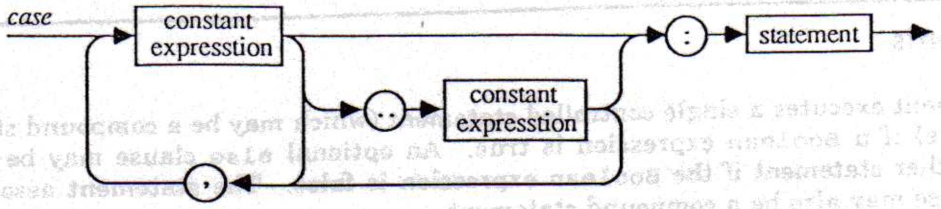
Examples of if statements:

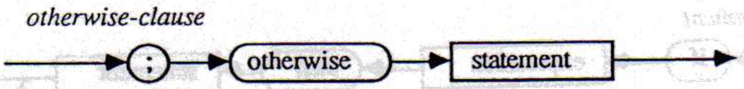
```
if x  <  1.5 then
     z  :=  x+y
else
     z  :=  1.5

if p1 <> nil then
     p1 := p1^.father;
```

Case Statements

The *case* statement consists of an expression (the selector) and a list of statements. Each statement is prefixed with one or more constants (called case constants), or with the reserved word `otherwise`. All the *case* constants must be distinct and must be of an ordinal type that is compatible with the type of the selector expression.







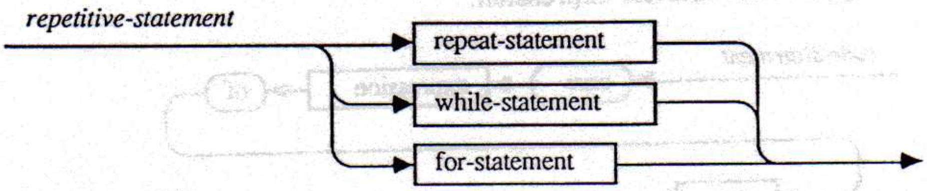
The case statement executes the statement prefixed with the case constant that equals the value of the selector. If no such case constant exists and an `otherwise` clause is present, the statement following the word `otherwise` is executed; if no `otherwise` clause is present then execution continues with the statement following the case statement.

Any of the controlled statements in the case clause or the default statement following otherwise may be either single statements or compound statements as defined above.
Examples of case statements:

```
case operator of
   plus : x   :=  x+y;
   minus: x   :=  x-y;
   times: x   :=  x*y
end;

case i of
   1   : x := sin(x);
   2   : x := cos(x);
   3,4,5 : x := exp(x);
   otherwise  x   :=  ln(x)
end
```

Repetitive Statements

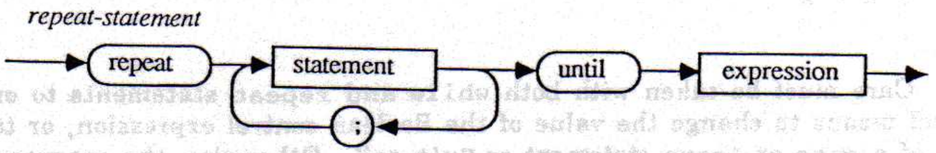
Repetitive statements specify a group of statements to be executed repeatedly.



If the number of repetitions is known beforehand, the `for` statement is an appropriate statement to use, otherwise the while or repeat statements are used.

Repeat Statements

A *repeat* statement contains an expression that controls the repeated execution of a sequence of statements contained within the repeat statement.



The expression must yield a result of the standard type `Boolean`. The statements between the symbols `repeat` and `until` are repeatedly executed in sequence until, at the end of a sequence, the expression yields the value `true`. The sequence of statements is executed at least once, because the expression is evaluated *after* the execution of each sequence.

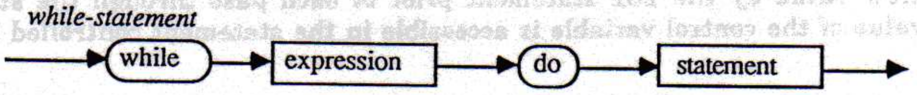
`Repeat` and `until` create their own compound statement out of the statements they control; there is no need to use `begin` and `end` as required for other compound statements.

Examples of repeat statements:

```
repeat
   k := i  mod  j;
   i := j;
   j := k
until j = 0

repeat
   process(f^);
   Read(f)
until eof(f)
```

While Statements

A *while* statement contains an expression that controls the repeated execution of a statement.



The expression must yield a result of the standard type `Boolean`. The expression is evaluated before each pass of the contained statement is executed. The contained statement is repeatedly executed as long as the expression yields the value true. If the expression yields false at the beginning, the statement is not executed.

The statement controlled by the `while` statement may be either a single statement or a compound statement as defined earlier in this chapter.

Examples of while statements:

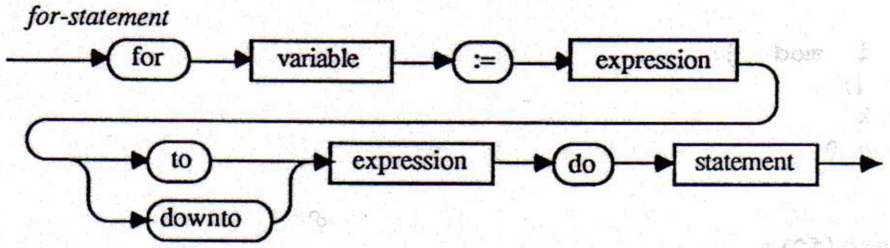
```
while a[i] <> x do
    i := i+1

while  i > 0  do begin
    if odd[i] then
       z := z * x;
```

```
      i := i div 2;
      x := x * x
   end
```

❖ **Note:** Care must be taken with both `while` and `repeat` statements to ensure some practical means to change the value of the Boolean control expression, or to escape by means of a `goto` or `Leave` statement or `Exit` call. Otherwise, the program will never exit the loop and therefore never terminate. `Leave` and `Exit` are defined later in this chapter.

For Statements

The *for* statement causes a statement to be repeatedly executed while a progression of values is assigned to a variable called the control variable.



The first expression following the := is called the *initial-value*. The second expression after the := is called the *final-value*. The control variable, initial-value, and final-value must all be compatible. See Chapter 13 for information on type compatibility.

The control variable is a scalar type (e.g. `Integer`, `Char`, `Boolean`, subrange, or user-defined). Therefore, it cannot be an array or string element, a record field, or a dynamic variable. The control variable must be declared in the block that contains the `for` statement. It is assigned a new value by the `for` statement prior to each pass through the statement it controls. The value of the control variable is accessible in the statement controlled by the `for` statement.

If the token after the first expression is the keyword `to`, the control variable is to be incremented prior to each pass through the statement. Execution of the `for` statement continues until the control variable obtains a greater value than the *final-value*. If the token after the first expression is the keyword `downto`, the control variable is decremented prior to each pass through the `for` statement. Execution of the `for` statement continues until the control variable obtains a lesser value than the *final-value*.

Below are several rules to follow when writing `for` statements:

- The control variable must be a simple variable declared in the local scope.
- If the control variable is a subrange type or user-defined scalar, it must be capable of accepting all values from the *initial-value* to the *final-value*, inclusively.
- The control variable must not be modified within the statement controlled by the for statement.
- The control variable may not be included in either the initial-value or final-value expressions.

- The control variable may be unspecified upon termination of the `for` statement.
- The initial-value and final-value expressions are evaluated just once, prior to the first pass. Changing the value of these expressions within the for statement will not alter the behavior of the `for` statement.
- If the initial-value and final-value are equal, the `for` statement will execute exactly once.

Examples of for statements:

```
for i := 2 to 63 do
   if a[i] > max then
      max  :=  a[ i ]
for C := red to blue do Check(C);
```

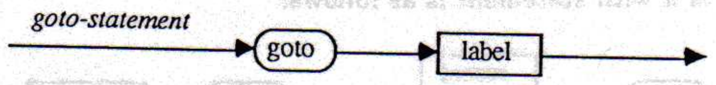
Control Statements

The repetition statements, conditional statements, assignment statements, and procedure and function calls are sufficient to handle almost all programming jobs. Situations may arise, however, that may demand immediate transfer or suspension of program execution. For such cases, Complete Pascal provides the following tools:

- the Goto statement to transfer control directly to any other program statement within the same block.
- the Cycle statement to force an immediate reiteration of a repetitive statement.
- the Leave statement to cancel the enclosing repetitive statement.
- the Halt statement to immediately stop program execution.

Goto Statement

The *goto* statement transfers program control to the statement immediately following the specified label.



The following rules must be observed when using a goto statement:

- The label referenced by a goto statement must be in the same block as the goto statement, or within a block which encloses the block containing the goto statement.
- Jumping into a structured statement from outside that structured statement can have undefined effects and is illegal. However, Complete Pascal does not detect the occurrence of such a goto statement.

See Chapter 13 for more information on labels.

Cycle Statement

The *Cycle* statement passes program control to the end of the statement controlled by the enclosing while, repeat, or for statement. Use of the *Cycle* statement outside of a while, repeat, or for statement will result in an error.

Example of the cycle statement:

```
for i:=1 to 100 do begin
   if a[i] <= 0 then cycle;
   f(a[i]);
end;
```

❖ **Note:** The word `cycle` is not a reserved word. If `cycle` is redefined, cycle statements may not be used within the scope of that definition.

Leave Statement

The *Leave* statement terminates the enclosing while, repeat, or for statement and passes control to the statement immediately following the enclosing repetition statement. Use of the *Leave* statement outside of a while, repeat, or for statement will result in an error.

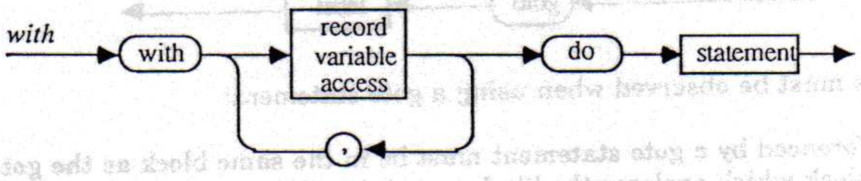
Example of the leave statement:

```
while i < 63 do begin
   if a[i] = x then leave;
   i:=i + 1;
end;
```

❖ **Note:** **The word `leave` is not a reserved word. If `leave` is redefined, leave statements may not be used within the scope of that definition.**

With Statement

The *with* statement is a shorthand method for specifying the fields of a record. Within a with statement, the fields of one or more record variables can be referenced using only their field identifiers. The syntax of a with statement is as follows:



The occurrence of a variable in the with statement must denote a record variable. Within a with statement, each variable reference is first checked to see if it can be interpreted as a field of the record variable. If so, it is always interpreted as such, even if a variable with the same name is accessible.

The following rules govern the use of with statements:

- When listing a record that is a field of another record, list the containing record earlier in the list or list that field in explicit form.
- With statements may be nested. When with statements are nested, the enclosing *with* statements remain valid within the nested with statements.
- When several records have fields of the same name, with accesses the field of that record name last in the list of the with statement.
- Where a record field identifier is the same as a variable or other identifier declared outside the record, with accesses the field.

Examples 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
```

When more than one record variable reference appears in a with statement as follows:

```
with var1, var2, ... varn do
   statement
```

it is considered equivalent to the following sequence of nested with statements:

```
with var1 do
   with var2 do
     ....
       with varn  do
          statement
```

Thus, if var_n in the above statements is a field of both var_1 and var_2, it is interpreted to mean $\mathrm{var}_2.\mathrm{var}_n$ and not $\mathrm{var}_1.\mathrm{var}_n$.

Null Statements

Null statements are statements that do not contain anything. Unnecessary semicolons are interpreted by the Pascal compiler as null statements. The result of unnecessary semicolons is two statements where only one was intended. Most of the time, this is harmless, but it occasionally causes an error when only one statement is allowed.

Chapter 17

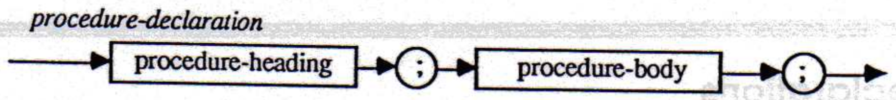
Procedures and Functions

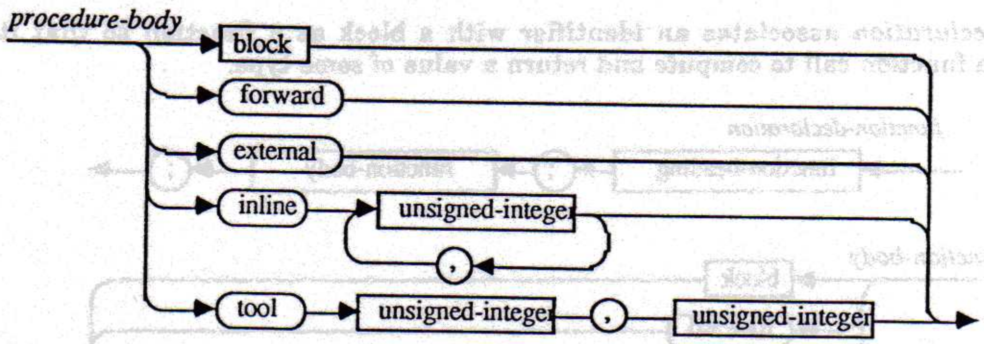
Procedures and *functions* allow you to nest additional blocks in the main program block or define blocks within a unit. Procedures and functions are also known together as *subprograms.* Each procedure or function has a heading followed by a block or a special directive. A procedure is activated by a procedure statement, and a function is activated by the evaluation of an expression that contains a function call.

This chapter describes the different types of procedures and functions and their parameters.

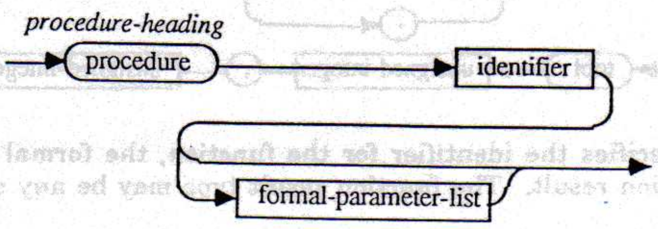
Procedure Declarations

A *procedure declaration* associates an identifier with a block as a procedure so that it can be activated by a procedure statement.





The procedure heading specifies the identifier for the procedure, and the formal parameters, if any.



The syntax for a formal parameter list appears later in this chapter.

A procedure is activated by a *procedure* statement, which gives the procedure's identifier and any actual parameters required by the procedure. The statements to be executed upon activation of the procedure are specified by 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 it is executing.

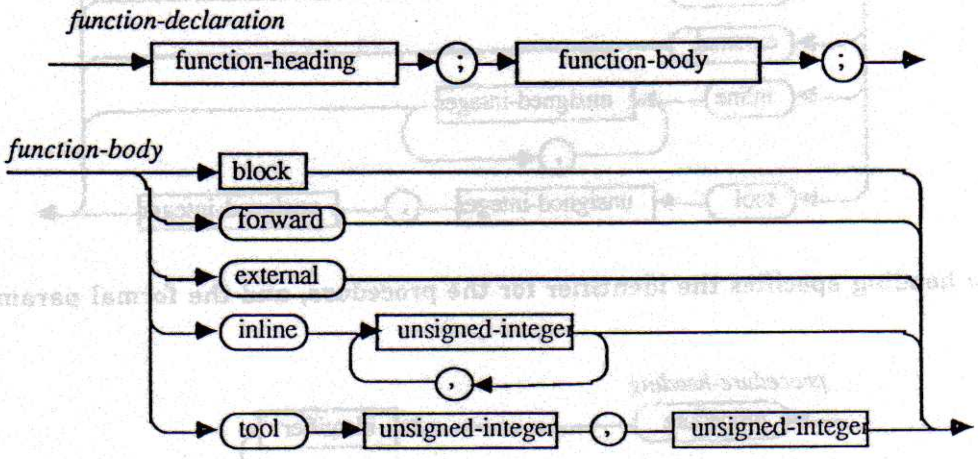
Example of a procedure declaration:

```
procedure Num2String (N: Integer; var S: string);
var V: Integer;
begin
  V := Abs (N);
  S := '';
  repeat
    S := Concat(Chr (V mod 10 + Ord ('0')),S);
    V := V div 10;
  until V =  0;
  if V < 0 then S := Concat('-',S);
end;
```

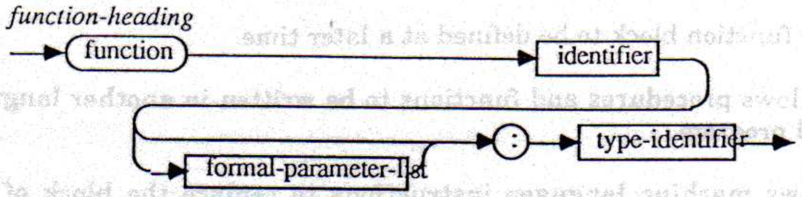
Instead of the block in a procedure or function declaration, a FORWARD, EXTERNAL, INLINE or TOOL directive may be given in its place. See the "Procedure and Function Directives" section in this chapter for more details concerning directives.

Function Declarations

A *function declaration* associates an identifier with a block as a function so that it can be activated by a function call to compute and return a value of some type.



The function heading specifies the identifier for the function, the formal parameters, if any, and the type of the function result. The function result type may be any simple or structured type.



A function is activated by the evaluation of a function call, which gives the function's identifier and any actual parameters required by the function. The function call appears as an operand in an expression. The expression is evaluated by executing the function and, in effect, replacing the function call with the value returned by the function.

The statements to be executed upon activation of the function are specified by the statement part of the function's block. The block should normally 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 exists but is not executed, the value returned by the function is undefined.

If the function's identifier is used in a function call within the function's block, the function is executed *recursively*.

Example of a function declaration:

```
function Num2String (N: Integer): string;
var V: Integer;
    S: String;
begin
   V := Abs (N);
   S := '';
   repeat
      S := Concat(Chr (V mod 10 + Ord ('0')),S);
      V := V div 10;
   until V =  0;
   if V < 0 then S := Concat('-',S);
   Num2String:=S;
end;
```

A function can be declared FORWARD, EXTERNAL, INLINE or TOOL in same manner as a procedure as described above.

❖ **Note:** If the return value of the function is a record type or a pointer to a record type, it cannot be used in the list of a with statement to assign values to the fields of the record. The compiler will interpret use of the function's identifier in the with statement as a function call.

Procedure and Function Directives

The block part of a procedure or function may be replaced with one of three directives:

- FORWARD allows the procedure or function to be declared immediately while allowing the procedure or function block to be defined at a later time.

- EXTERNAL allows procedures and functions to be written in another language and linked to the Pascal program.
- INLINE allows machine language instructions to replace the block of a procedure or function.
- TOOL designates a procedure or function as an Apple IIGS Toolbox routine.

Forward Directives

A procedure declaration that has the directive FORWARD instead of a block is called a *forward declaration*. Somewhere after the forward declaration, the procedure is actually defined by a *defining declaration* – a procedure declaration that uses the same procedure identifier and includes a block. The defining declaration may repeat the formal parameter list, but if the formal parameter list is repeated it must be identical to the forward declaration. The forward declaration and the defining declaration must be in the same declaration part, but need not be contiguous. That is, other procedures, functions, types, variables, etc. can be declared between them and can call the procedure that has been declared forward. Forward declarations allow mutual recursion.

The forward declaration and the defining declaration constitute a complete declaration of the procedure. The procedure is considered to be declared at the place of the forward declaration. Forward procedures and functions may not be written in the interface part of a unit.

Example of a forward declaration:

```
procedure Proc2 (m,n: integer); forward;

procedure Proc1 (x,y: real);
begin
   ...
   Proc2 (4,5);
end;

procedure  Proc2 (m,n: integer);
begin
   ...
   Proc1 (8.3,2.4);
end;
```

External Directives

A procedure declaration, whose body is declared EXTERNAL, defines the Pascal interface to routines assembled or compiled in a language other than Complete Pascal. The external code for the procedure must be available at link time.
Examples of an external declaration:

```
procedure GotoXY(x,y: Integer);  External;
```

In this example, GotoXY is an external procedure that must be linked to the host program prior to execution.

It is the responsibility of the programmer to ensure the external procedures and functions are compatible with their companion declarations in the Pascal program. The linker does not check for compatibility.

Inline Directives

The INLINE directive allows you to write machine code in place of a procedure's block. The code may only consist of a sequence of integer constants which each represent a single byte of 65816 machine code. When the procedure is called, the compiler generates the machine code specified by the INLINE directive. If the procedure has any parameters, they are pushed onto the stack before the code is generated.

The inline directive is intended for writing small routines. For example, the following procedure would clear the 65816 Interrupt Disable flag by generating the CLI instruction.

Example of an inline declaration:

```
    procedure  GenCLI; inline $58;
```

Tool Directives

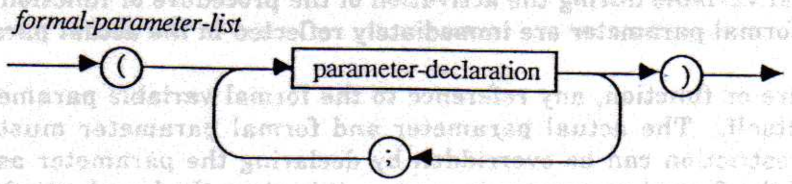
The TOOL directive is used to define the body of a procedure to be one of the Apple IIGS Toolbox routines. The Apple IIGS Toolbox is divided into several *toolsets* and then into individual *tool routines.* Each toolset is identified by a unique *tool number,* and each tool routine within a toolset is assigned a unique *function number.* Using this special method of describing an Apple IIGS Toolbox routine provides Complete Pascal with the information it needs to generate a call into the Toolbox. For example, the MoveTo procedure in the QuickDraw toolset (tool number 4) is assigned the function number 58. Thus, the tool declaration is as follows.

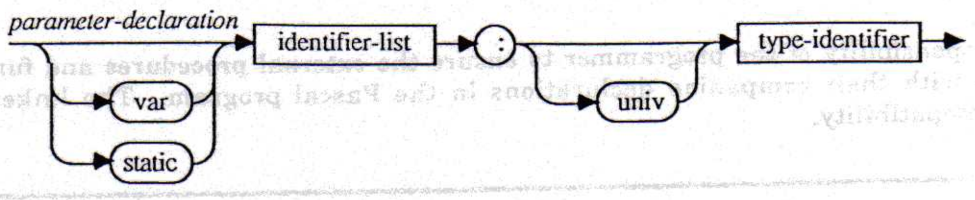
Example of a tool declaration:

```
    procedure  MoveTo(h,v: Integer);          Tool 4,58;
```

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.





Procedure and function declarations may have any or all of three kinds of formal parameters:

- *variable*: a group of parameters preceded by the keyword `var`.
- *value:* a group of parameters without being preceded by the keyword `var`.
- *static*: a group of parameters preceded by the word `static`.

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 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 a file type or of any structured type containing 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 static size attribute of 255.

❖ **Important:** If the size of the formal parameter (in bytes of storage) is greater than 4 bytes, then the actual parameter is passed by address and then the value of the actual parameter is copied into local variable space. Therefore, assignments to the formal parameter do not affect the value of the actual parameter.

The actual parameter and formal parameter must be assignment compatible. This restriction can be overridden by declaring the parameter as *Univ*. Univ is described below.

Variable Parameters

A variable parameter is used when the value of a parameter must be passed back 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 parameter are immediately reflected in the actual parameter.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The actual parameter and formal parameter must be assignment compatible. This restriction can be overridden by declaring the parameter as `univ`, which is described below. If the formal parameter is `string`, it is given the length attribute 255, and the actual variable parameter must be a string type with a length of 255.

File types can only be passed as variable parameters.

Components of variables of any packed structured type cannot be used as actual variable parameters.

> ❖ **Note:** If accessing an actual variable involves indexing an array, finding the identified variable of a pointer, or finding the field of a record, the action is executed prior to the activation of the procedure or function. Caution should be taken in accessing variables found in a relocatable block on the heap. Compaction of the heap can cause the original object to be moved, possibly leading to unpredictable results.

Static Parameters

Static parameters are a special *extension* to Complete Pascal for the Apple IIGS. They have been added for the specific purpose of obtaining improved code generation. Static parameters are treated *exactly* like value parameters described above except for the restriction that a static formal parameter should not be assigned a new value within the procedure or function.

Value parameters whose formal type requires more than 4 bytes of storage are passed by address and then copied into the local storage for the formal parameter so that assigning new values to the formal value parameter does not affect the actual parameter (see Appendix D). However, there are cases when the formal parameter is only *read* from and never written to. In these cases, it is not necessary to copy the actual parameter value into local storage for the formal parameter, the formal parameter may access the actual parameter directly.

Static parameters reduce the amount of stack space required by an application, and reduce execution time by not having to copy the value of an actual parameter into local storage for the formal parameter.

> ❖ **Warning:** Complete Pascal does NOT check that a static parameter is never written to. It is the responsibility of the programmer to ensure the correct usage of static parameters.

UNIV Parameter Types

When the word `UNIV` appears before the type identifier in the formal parameter list, the restriction that the actual parameter and formal parameter must be assignment compatible in the case of value parameters, and identical in the case of variable parameters is not enforced. When `UNIV` is used, the actual parameter may be of any type so long as the number of bytes required to store a value of the actual parameter's type is the same as that of the formal parameter.

Here is an example of a UNIV parameter:

```
    TYPE Ptr = ^Char;
    VAR  aLong: LongInt;
         aPtr:  Ptr;

       procedure aProc(p: univ Ptr);
       begin
       end;

    begin
       aProc (aLong);
       aProc (aPtr);
    end.
```

Parameter List Compatibility

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or functional parameters.

Two formal parameter lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if one of the following is true:

- They are both value parameters of identical type.
- They are both variable parameters of identical type.
- The formal parameter has UNIV before its type, and the actual parameter is a value or variable of the same size. The parameters must still both be value parameters, or both be variable parameters.

Chapter 18

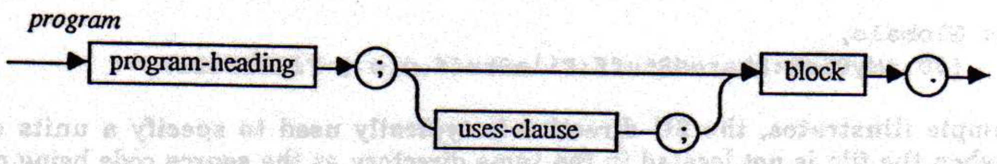
Programs and Units

Complete Pascal provides two basic constructs which are the fundamental units of a piece of Pascal source code. These are *programs* and *units*. The main difference between a program and a unit, is that a program represents a complete application which can be compiled and executed. A unit, however, can not be executed by itself, it is merely a construct in which *parts* of a program can be defined and compiled independently of a program. Programs and units are independently compiled. Their object files are combined by the Complete Pascal linker to form a single executable file.

Part I of this manual provides information about how to compile and link programs and units. In addition, Part II provides detailed information about how to create each of the types of programs supported by Complete Pascal.

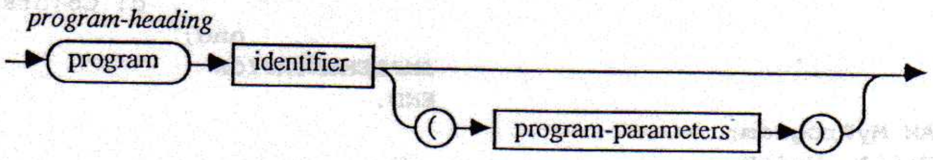
Programs

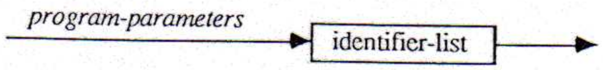
A Pascal *program* has the form of a procedure declaration except for its heading and an optional *USES clause.*



The occurrence of an identifier immediately after the word `program` in the program heading declares it as the program's identifier. *Program parameters*, as described by Jensen and Wirth and the ANS Standard have no meaning to Complete Pascal.

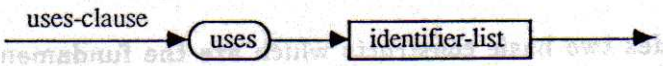
- ❖ Note: TML Pascal versions 1.x, associated a special meaning with the presence of the file parameters Input and Output in the program heading. They were used to signal to the compiler that it should create the *Plain Vanilla* operating environment. Complete Pascal NO LONGER associates any special meaning with the file parameters Input and Output. To create the *Plain Vanilla* (now called *Textbook Graphics*) operating environment, call the procedure `Graphics`.





Uses Clause

The *USES clause* is used to identify those units which are required by a program or unit in order to compile successfully.



When Complete Pascal encounters an identifier (the name of a unit) in a USES clause, it must find the unit's compiled object code file which contains the unit's object code and symbol table. To do this, the string ".p.o" is appended to the unit name to form a file name. For example, the following statement

```
uses Globals, FileStuff;
```

causes the files Globals.p.o and FileStuff.p.o to be opened.

The uses clause in the main program lists all units required by the program. Such units include those used directly by the main program and indirectly by units used by the main program.

It is possible that the unit's compiled object code file name may not match the unit's name. In this case, the compiler's $U directive must be used to specify the filename which contains the unit. The $U directive must appear immediately before the unit name in the USES clause. For example, the USES clause given above might be rewritten as

```
uses Globals,
    {$U :MyDisk:SharedStuff:FileStuff.p.o } FileStuff;
```

As the example illustrates, the $U directive is typically used to specify a units complete pathname when the file is not located in the same directory as the source code being compiled. The $U directive is fully documented in Appendix B.

When a unit named in a USES clause uses other units itself, the names of those units must also appear in the uses clause, and they must appear *before* the unit is named.
Consider the following example:

```
UNIT UnitA;                              UNIT UnitB;
INTERFACE                                USES UnitA;
  type Colors=(red,white,blue);          INTERFACE
IMPLEMENTATION                                 type Rec = record
END.                                                     i: Integer;
                                                         c: Colors;
                                                     end;
                                         IMPLEMENTATION
                                         END.

PROGRAM MyProgram;
USES UnitA, UnitB;
VAR  aRec: Rec;
BEGIN
END.
```

In this example, the program `MyProgram` declares a variable `aRec` of type `Rec`, which is declared in unit `UnitB`. Therefore, a USES clause is used to name `UnitB`. However, `UnitB` has a uses clause which names `UnitA`. Thus, the uses clause in the program `MyProgram` must name `UnitA` in its uses clause, and further it must appear before `UnitB`.

If a unit has been recompiled, then all units which use it must also be recompiled. This is required so that all dependent units do not attempt to reference unit declarations that might have changed or no longer exist. For instance, in the example above, if `UnitB` is recompiled, then `MyProgram` must also be recompiled, but `UnitA` need not be recompiled. And if `UnitA` is recompiled then both `UnitB` and `MyProgram` must be recompiled.

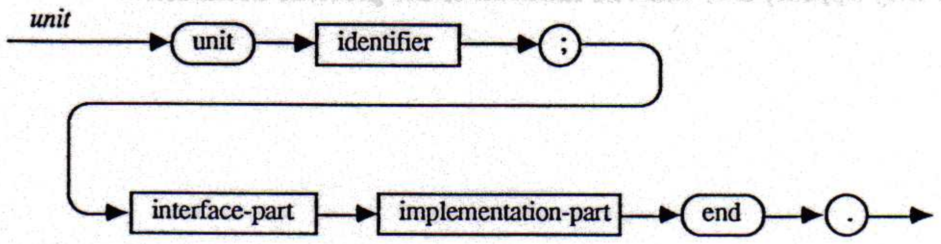
Units

Units are the basis for modular programming in Complete Pascal. Units are compiled separately from one another and should be used to organize large programs into logically related parts. Dividing a program into several units also reduces the amount of time necessary to recompile a piece of code.

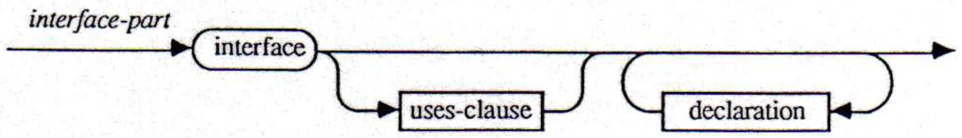
There are several reasons for using units in Pascal programming:

- They help modularize large programs.
- They make common declarations and blocks easily available to many different programs.
- They can be used to make sections of source text private from the rest of the program.

The syntax for units follows:



The identifier following the reserved word unit is the unit identifier. This is the name that other units as well as the main program use in a USES clause to specify that the unit's declarations should be made available.

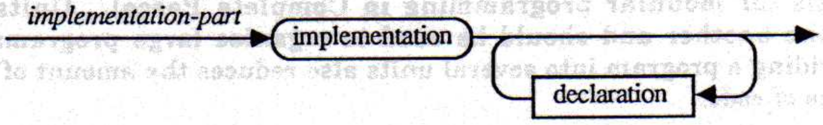


The *interface* part of a unit declares constants, types, variables, procedures, and functions that are *public*. That is, the declarations made in the interface part are available to other units or programs that name the unit in a USES clause. In other words, the scope of the public declarations is the entire program or unit that uses the unit. The program or unit that uses a unit can access the public declarations just as if they had been declared in its own block.

Label declarations are not permitted in the interface part of a unit. Procedures and functions in the interface part are declared by giving only the procedure or function name, the formal parameters (if any), and the result type (if a function). No code block is given for a procedure or function declaration in the interface part. Instead, the procedure or function heading is repeated in the implementation part where its code block is declared. Procedure and function declarations in the interface part behave as if the `FORWARD` directive had been specified. However, procedure and function declarations *may* have the `EXTERNAL`, `INLINE` and `TOOL` directives in the interface part since they will not have a code block in the implementation part.

Variables, procedures and functions which appear in the interface part are termed *global*. The entire unit is within the scope of the block in which the uses clause that references the unit appears.

The interface part may contain a uses clause, so any unit can use another unit.



The *implementation* part, which follows the last declaration of the interface part, declares any constants, types, variables, procedures, or functions that are *private*, that is, not available to the program or unit which uses it. Private procedures and functions are declared like procedures and functions in programs, with a procedure or function heading and a body.

All public procedures and functions declared in the interface part are redeclared in the implementation part. The only exception are thos procedures and functions declared using the EXTERNAL, INLINE and TOOL directives. Formal parameters and result types may be omitted, but if they appear, they must be identical to the previous declaration.

Chapter 19

Input and Output

This chapter describes the standard input and output (I/O) procedures and functions provided by the Complete Pascal compiler for the manipulation of files. The predefined procedures and functions which do not affect I/O are described in Chapter 20.

File I/O in Complete Pascal

Complete Pascal provides two ways to implement input and output:

- I/O routines in the Apple IIGS Toolbox.
- I/O procedures and functions built into Complete Pascal.

The most direct method of accessing the Apple IIGS screen, keyboard, and mouse is through the Apple IIGS Toolbox routines for QuickDraw and the Event Manager. The GS/OS v5.0 File System calls also provide complete access to the Apple IIGS operating system for manipulating files. Appendix C contains a list of all interface files to the Apple IIGS Toolbox.

Complete Pascal's built-in routines provide the easiest way to access the contents of files and perform I/O operations with external devices. The remainder of this chapter discusses each of these routines in detail.

Pascal Files

In Complete Pascal, files are accessed using a *file variable.* A file variable is simply a variable which is declared to be some file type (see Chapter 13). Complete Pascal supports two different types of files:

- Text files
- Typed files (also called Structured files)

Text files are declared with the predefined type `Text`. Text files store data as sequences of character organized into *lines.* Complete Pascal provides several special I/O procedures and functions which operate on lines of text.

Complete Pascal predefines two Text files for every program: `Input` and `Output`. Input is defined to be a read-only file, reading data from the keyboard. Output is defined as a write-only file, writing data to the screen (text screen or graphics screen).

Typed files consists of a sequence of components. Whereas a component in a text file is a single character, the component in a typed file can be of any type other than a file type or a structured type containing a file type. A single occurrence of a component is called a *logical record*.

Examples of files:

```
      var aFile: text;                { Example of a text file  }
      var aFile: file of integer;     { Example of a typed file }
```

File variables refer to files consisting of a sequence of components. For text files, that component is always of type `Char`. For typed files, the component can be any Pascal type except for a file type or a structured type containing a file. In any case, the component is considered a logical record. Files may have any number of logical records, but only one is accessible at a time with the file variable. The position of the accessible logical record with respect to the beginning of a file is called the *current file position*.

Prior to using a file variable, it must be associated with a file. Making this association is called *opening* a file. There are three procedures that are used to open a file: `Reset`, `Rewrite`, and `Open`. Each procedure is defined in detail later in the chapter.

❖ **Note:** The file variables `Input` and `Output` are predefined and are opened automatically at the start of the program.

To open a file, its external name must be given. This name can be any valid Apple IIGS (GS/OS v5.0) file pathname or device name.

Standard Procedures and Functions for All Files

The procedures and functions described in this section may be used with any text or typed file.

The Reset Procedure

Syntax: `Reset ( f [, title ] )`
Reset opens an existing file for input or "rewinds" an open file by repositioning the current file position to the zero component. The file is opened for sequential read access only. When an already existing file is reset, its contents are not erased. `f` is a file variable of any file type. `title` is an optional string expression.

If `title` is provided in the parameter list, then `Reset` attempts to open an already existing file with the name `title` and then associates the file variable `f` with the external file. If `title` is not a valid GS/OS file pathname or device name, or if the file cannot be opened, an error is returned in `IOResult`.

If `title` is not provided in the parameter list, then `f` must already be associated with an open file. In this case, `Reset` repositions the file position to the zero component of the file. If f is not already associated with a file, then an error is returned in `IOResult`.

The Rewrite Procedure

Syntax: `Rewrite ( f [, title ] )`

Rewrite opens an existing file or creates a new file for output or "rewinds" an open file by repositioning the current file position to the zero component. The file is opened for sequential write access only. When an already existing file is opened with rewrite, its contents are erased. `f` is a file variable of any file type. `title` is an optional string type expression.

If `title` is provided in the parameter list, then `Rewrite` creates and opens a new external file with the name `title` and then associates the file variable `f` with the external file. If the file already exists, it is opened and its entire contents erased.

If `title` is not provided in the parameter list, then `f` must already be associated with an open file. In this case, `Rewrite` repositions the file position to the zero component of the file. If `f` is not already associated with a file, then an error is returned in `IOResult`.

The Open Procedure

Syntax: `Open ( f, title )`

Open opens an existing file or creates a new file. The file is opened for random read and write access. When an already existing file is opened, its contents are not erased. `f` is a file variable of any file type. `title` is an optional string type expression.

If `title` is not a valid GS/OS file pathname or device name, or if the file cannot be opened, an error is returned in `IOResult`.

The Close Procedure

Syntax: `Close ( f )`

Close closes the open file. `f` is a file variable of any file type. The association between `f` and its external file is broken and the file system marks the external file "closed".

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, otherwise it returns `false`.

The Seek Procedure

Syntax: `Seek ( f, n )`

Changes the current file position to the file component `n` and reads the new current logical record into the file window variable. `f` is a file variable, and `n` is an expression of type `LongInt`. For text files, the logical record size is one byte. The number of the first file component is zero. If the value of `n` is greater than the number of components in the file, then the current file position is moved to the end of the file, and `Eof(f)` is true.

The Erase Procedure

Syntax: `Erase ( title )`

Erases an external file. `title` is a string type expression. The external file with the name `title` is deleted from its external storage device.

The IOResult Function

Syntax: `IOResult`
Result type: `Integer`

Returns an integer value that is the status of the last I/O operation performed. A value of zero indicates successful completion of the last I/O operation, while a non-zero indicates an error. Note that `IOResult` returns the status of the *last* I/O operation performed. Thus, the following two statements do not provide the results one might expect.

```
        Reset(f,'myfile');
        Writeln('IOResult for Reset = ',IOResult);
```

The call to the `IOResult` function in the Writeln parameter list actually returns the status of the Writeln operation for the string `'IOresult for Reset = '` since that was the most recent `I/O` operation, and not the call to `Reset`. Instead the previous two statements should be rewritten as:

```
        Reset(f,'myfile');
        svIOResult := IOResult;
        Writeln('IOResult for Reset = ', svIOResult );
```

The FilePos Function

Syntax: `FilePos ( f )`
Result type: `LongInt`

Returns the current file position of the opened file `f`. The first logical record in a file is called the zeroeth position. With typed files, a logical record is an occurrence of the component type. With text files, the component type is one byte. `f` is a file variable reference and the file referred to by f must be open.

The Rename Procedure

Syntax: `Rename(OldName, NewName)`

Renames an exiting external file. `OldName` and `NewName` are string type expressions. The external file named `OldName` is renamed to `NewName`. If a file with the name `OldName` can not be found then an error is returned in `IOResult`.

Standard Procedures for Typed-Files

The procedures discussed in this section are used to randomly access logical records of typed files. The component type of typed files may be any type other than file type or structured type containing a file type.

The Read Procedure With Typed-Files

Syntax: `Read ( f,` v_1 `[ ,` v_2 `, ... ,` v_n `] )`

Reads a file component into a variable. `f` is a file variable, and each parameter `v` is a variable of the same type as the component type of the file `f`. For each parameter v, the file component at the current file position is read into v and the file position advances to the next file component. If an attempt is made to read past the end of file, then an error is returned by `IOResult`.

The `Read` procedure is also used with text files as described below. With text files, the file variable `f` is an optional parameter since, if it is omitted, the read statement would then read from standard input (i.e. the keyboard) which is defined as type text. When the read procedure is used with typed-files, the file reference variable is required.

The Write Procedure With Typed-Files

Syntax: `Write ( f,` v_1 `[ ,` v_2 `, ... ,` v_n `] )`

Writes each variable v into a file component. `f` is a file variable, and each parameter `v` is a variable of the same type as the component type of the file `f`. For each parameter v the value of v is written to the file component at the current file position and the file position is advanced to the next file component. If the current file position is at the end of the file, then the file is expanded to include the new file component.

The write procedure is also used with text files as described below. With text files, the file variable f is an optional parameter since, if it is omitted, the write statement would then write to standard output (i.e. the screen) which is defined as type text. When the write procedure is used with typed-files, the file reference variable is required.

Standard Procedures and Functions for Text Files

Text files are distinguished from all other types of files by the fact that they are organized into a collection of lines terminated by the carriage return character. Text files are unique from files defined as `file of Char` since the former is organized into lines while the latter may not.

No procedure or function defined in this section requires an explicit file variable parameter. If no file variable parameter is given, either the predefined file `Input` or `Output` will be assumed based on the type of operation performed by the procedure or function. If the procedure or function is performing an input operation, the predefined file `Input` is assumed, otherwise `Output` is assumed.

The Read Procedure With Text Files

Syntax: `Read ( [ f, ] v`$_1$` [ , v`$_2$` , ... , v`$_n$` ] )`

Reads one or more values from a text file into the corresponding parameters `v`$_i$. `f`, if specified, is a variable of type `Text`. If `f` is omitted, the standard file `Input` is assumed, which is associated with the Apple IIGS keyboard. Each `v` is a variable of an `Integer`, `Longint`, `Real`, `Char`, or `String` type.

Read 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 the read was performed, then the value `Chr(0)` is returned. If Eoln(f) was true before the read was performed, then the value `Chr(13)` is returned. The next read will start with the next character in the file.

Read an Integer or LongInt type variable. With an `Integer` or `Longint` type variable, `Read` expects a sequence of characters which form a signed whole number. All spaces, tabs, and end of lines are skipped until the beginning of the numeric string is found. Then all characters which are not a space, tab or end of line are assumed to be part of the numeric string. The string is then interpreted as a numeric value. If any characters in the string do not represent a signed whole number, then an error is returned by `IOResult`. The next read will start with the character which terminated the numeric string.

Read a Real type variable. With a `Real` type variable, `Read` expects a sequence of characters which form a signed floating point number. All spaces, tabs, and end of lines are skipped until the beginning of the numeric string is found. Then all characters which are not a space, tab or end of line are assumed to be part of the numeric string. The string is then interpreted as a floating point value. If any characters in the string do not represent a real number, then an error is returned by `IOResult`. The next read will start with the character which terminated the numeric string.

Read a String type variable. With a `String` type variable, `Read` reads all characters into the string variable up to, *but not including,* the next end of line character. The next read will start with the end of line character which terminated the read. Note that successive reads of a string type will not read successive lines from the file since a read of a string type variable *never* advances past an end of line character.

The Readln Procedure

Syntax: `Readln ( [ f, ] v1 [ , v2 , ... , vn ] )`

This procedure is an extension of the Read procedure. After performing the same operations as Read would perform for the parameter list, Readln skips to the beginning of the next line of the Input file by skipping all characters in the Input file until an end of line character is found and then reading that end of line character. If no other lines exist, eof(f) becomes true. Again, if f is omitted, then the standard file Input is assumed.

The Write Procedure With Text Files

Syntax: `Write ( [ f, ] v1 [ , v2 , ... , vn ] )`

Writes one or more values to a file of type Text. If f is omitted, the standard file Output is assumed which is associated with the Apple IIGS screen. Each v is an expression of an Integer, Longint, Real, Char, Boolean or String type.

Each v is known as a *write-parameter*. Each write-parameter has the form

```
    OutExpr [  :  MinWidth [  :  DecPlaces  ]  ]
```

where OutExpr is an output expression of an allowable type. MinWidth and DecPlaces are expressions with integer-type values.

MinWidth specifies and *minimum* field width. MinWidth must be greater than or equal to zero. Exactly MinWidth characters are written (using leading spaces if necessary), except when OutExpr has a value that must be represented in more than MinWidth characters; in this 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* value. It can be specified only if OutExpr has a real-type value, and if MinWidth is also specified. If specified, it must be greater than zero. If DecPlaces is not specified, a floating-point representation is written.

The Writeln Procedure

Syntax: `Writeln ( [ f, ] v1 [ , v2 , ... , vn ] )`

This procedure is an extension to the Write procedure. After performing the same operations as Write would perform for the parameter list, Writeln writes the end of line character to the file (a carriage return).

The Eoln Function

Syntax: `Eoln [ ( f ) ]`
Result Type: `Boolean`
Returns the end of line status of a file. `f` must be declared as a file of type `Text`. `Eoln(f)` returns `true` if the character at the current file position is the end of line character or if `Eof(f)` is true, otherwise it returns `false`.

The Page Procedure

Syntax: `Page [ (f) ]`

Writes the form feed character to a text file. `f` must be declared as a file of type `Text`. If `f` is omitted then the standard file `Output` is assumed.

Disk Files and Complete Pascal

When specifying an external file to any of the standard Complete Pascal procedures, the file's pathname must be given. A pathname consists of a file name optionally preceded by the file's volume name and zero or more directory names. The volume name, directory names, and file name are separated by colons (:). For example,

```
      :MyVolume:MyDir1: ... :MyDirN:MyFile
```

In addition, the old style ProDOS/16 pathname syntax may also be used. ProDOS/16 uses the slash (/) separator rather than the colon (:) seperator. For example,

```
      /MyVolume/MyDir1/ ... /MyDirN/MyFile
```

For complete information about GS/OS and pathnames see the *GS/OS Reference.*

Devices and Complete Pascal

In addition to external disk files, Complete Pascal supports a set of devices for input and output. These devices are any legal GS/OS device such as the keyboard, the display, and the printer. The keyboard and display devices are automatically available when the program begins execution with the standard file variables `Input` and `Output` respectively.

The printer is also available as a text device, but must be explicitly opened using the Rewrite procedure with the device name "`.PRINTER`". For example,

```
      PROGRAM TestPrinter;
      var f: Text;
      begin
         Rewrite(f,'.PRINTER');
      for i := 1 to 10 do
         Writeln(f,'Hello printer');
      Close(f);
   end;
```

The device names ".PRINTER" and ".CONSOLE" are predefined by GS/OS. Other device names may be created for the modem port or other slots. For more information about device names see the *GS/OS Reference*.

Chapter 20

Standard Procedures and Functions

This chapter describes all the standard, predeclared procedures and functions provided in Complete Pascal, except for the standard Input/Output procedures and functions which are documented in Chapter 19.

Standard procedures and functions are predeclared. Since predeclared entities act as if they were declared in a block surrounding the program or unit, no conflict arises from a declaration that redeclares the same identifier within the program except that it hides the predeclared procedure or function. However, predeclared procedures and functions may not be used as actual parameters for procedures and functions.

The Graphics Procedure

Syntax: `Graphics (screenMode: Integer);`

The `Graphics` procedure is used to initialize the Complete Pascal Textbook Graphics programming environment. This procedure should be called as the very first statement in the main body of a program. The procedure initializes the Apple IIGS Toolbox QuickDraw and Event Manager toolsets and places the screen in Super Hi-Res 320 or 640 mode depending upon the value of the screenMode parameter. The screen can be used for standard input and output with the Readln, Writeln and other I/O routines. QuickDraw graphics can also be done in this screen.

The `Graphics` procedure is provided in Complete Pascal in order to make programming graphics as simple and easy possible. The Complete Pascal Textbook Graphics programming model is fully described in Chapter 7 of this manual.

The Flow of Control Procedures

The procedures in this section allow for immediate branching of control in a program.

The Exit Procedure

Syntax: `Exit ( id )`

The `Exit` procedure causes execution of a particular block to terminate immediately, where a block can be either a procedure or function, or the entire program. Essentially, it is equivalent to a `goto` statement to a label at the very end of the block identified by `id`.

The Halt Procedure

Syntax: `Halt`

The `Halt` procedure causes execution of a program to terminate immediately.

The Cycle Procedure

Syntax: `Cycle`

The `Cycle` procedure causes the execution of the body of a loop to skip to the end of the loop and continue execution of the next iteration of the loop. The `Cycle` procedure is only meaningful in a for loop, a while loop, and a repeat loop. If it appears outside of the context of these statements, it has no affect. See Chapter 16 for more details.

The Leave Procedure

Syntax: `Leave`

The `Leave` procedure causes the execution of the body of the loop in which it occurs to terminate and continue execution with the first statement after the loop. The `Leave` procedure is only meaningful in a for loop, a while loop, and a repeat loop. If it appears outside of the context of these statements, it has no affect. See Chapter 16 for more details.

Dynamic Allocation Procedures and Functions

These procedures are used to manage the heap, a memory area that is unallocated when a program begins execution. The heap used by the dynamic allocation procedures is the Apple IIGS heap, and the routines are implemented using the Apple IIGS Memory Manager. See the Memory Manager chapter of the *Apple IIGS Toolbox Reference* for details regarding memory management on the Apple IIGS.

The New Procedure

Syntax: `New( p )`

New(p) creates a new variable of the base type of p, and makes p reference it. p can be a pointer variable of any type. The value of p is referenced as p^.

`New` actually calls the Memory Manager routine `NewHandle` to allocate a region of memory and returns a pointer to the allocated block of memory.

An error if the heap does not contain enough free space to create the new variable. In this case, p is set to `nil` and the `HeapResult` function will be set to indicate the error.

The Dispose Procedure

Syntax: `Dispose( p )`

`Dispose(p)` destroys the dynamic variable referenced by `p` and returns its memory region to the heap. `p` must be a variable that was previously assigned by the `New` procedure or was assigned a meaningful value by an assignment statement. The value of `p` then becomes undefined and it is an error to subsequently make reference to any values previously associated with `p`.

Transfer Functions

Transfer functions are used to transfer a value from one type to another. Note that the standard procedures Pack and Unpack as defined by the Pascal Standard are not implemented in Complete Pascal.

The Trunc Function

Syntax: `Trunc(x)`
Result Type: `LongInt`

`Trunc(x)` **returns a `LongInt` result that is the value of the real type variable `x` truncated to the nearest whole number that is between 0 and `x` inclusive. It is an error if the result of this rounding is outside the range `-maxlongint-1..maxlongint`.**

The Round Function

Syntax: `Round(x)`
Result Type: `LongInt`

`Round(x)` **returns a `LongInt` result that is the value of the real type variable `x` rounded to the nearest whole number. If `x` is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude. It is an error if the result of this rounding is outside the range `-maxlongint-1...maxlongint`.**

The Ord4 Function

Syntax: `Ord4(x)`
Result Type: `LongInt`

`Ord4(x)` 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: the *anonymous* pointer type

`Pointer` converts an `Integer` or `Longint` value to a `Pointer` type. The return value of `Pointer(x)` is a pointer to the physical address denoted by the value of `x`. This pointer is of the same type as `nil` in that it is assignment compatible with any pointer type. The value of `Pointer(0)` is `Nil`.

Arithmetic Procedures and Functions

Arithmetic procedures and functions perform numeric operations on real or integer type values. The implementation of these routines can be generated directly by the compiler, or achieved by using the SANE routines built into the Apple IIGS Toolbox.

The Inc Procedure

Syntax: `Inc(x)`
Increments the `Integer` or `LongInt` type variable `x` by one (1).

The Dec Procedure

Syntax: `Dec(x)`

Decrements the `Integer` or `LongInt` type variable `x` by the value one `(1)`.

The Abs Function

Syntax: `Abs(x)`
Result Type: same type as parameter.

Returns the absolute value of `x`; i.e. if `x` is negative, `-x` is returned; otherwise `x` is returned. `x` is an integer or real type argument.

The Sqrt Function

Syntax: `Sqrt(x)`
Result Type: `Extended`

If `x` is non-negative, `Sqrt` returns an `Extended` value which is the square root of `x`. However, if `x` is negative, a diagnostic NaN (Not a Number) is produced and the invalid operation signal is set. See *Apple Numerics Manual* for more information on NaNs.

The Odd Function

Syntax: `Odd(x)`
Result Type: `Boolean`

Returns `True` if `x` is odd, i.e. not divisible by 2 without a remainder. If `x` is even, it returns `False`. `x` is an expression of an ordinal type.

The Sin Function

Syntax: `Sin(x)`
Result Type: `Extended`

Returns the trigonometric sine of x. `X` is any real type expression and is assumed to represent an angle in radians. If x is infinite, a diagnostic NaN is produced and the invalid operation signal is set.

The Cos Function

Syntax: `Cos(x)`
Result Type: `Extended`

Returns the trigonometric cosine of x. `X` is any real type expression and is assumed to represent an angle in radians. If x is infinite, a diagnostic NaN is produced and the invalid operation signal is set.

The Exp Function

Syntax: `Exp(x)`
Result Type: `Extended`

Returns the value of e^x, where e is the base of the natural logarithms. If floating-point overflow occurs, the result is +inf. `X` is any real type expression.

The Ln Function

Syntax: `Ln(x)`
Result Type: `Extended`

`Ln(x)` returns the natural logarithm ($\log_e$) of x. `X` is any real type expression. If x is negative, a diagnostic NaN is produced and the invalid operation signal is set.

The Arctan Function

Syntax: `Arctan(x)`
Result Type: `Extended`

Returns the principle value, in radians, of the arctangent of `x`. `x` is any real type expression. All numeric values of `x` are valid, including ±Inf.

Ordinal Functions

The ordinal functions in this section operate on the ordinal value of scalar and pointer types. Refer to Chapter 13 for more information on scalar and pointer types.

The Ord Function

Syntax: `Ord(x)`
Result Type: `Integer` or `LongInt`

`Ord` returns the ordinal number of a `scalar` or `pointer` type value. If x is of type `Integer` or `LongInt`, the result type is the same as x. If x is a `pointer` type, the result is the corresponding address of the dynamic variable pointed to by x, of type `LongInt`. If x is of an ordinal type, the result is of type `Integer` and the value is the ordinality of `x` . The standard procedure `Ord4` should be used if the result type `LongInt` is desired, regardless of the type of x.

The Chr Function

Syntax: `Chr(x)`
Result Type: `Char`

Returns the `Char` value whose ordinal number is x. For any `Char` value ch, the following is always true: `chr(ord(ch)) = ch.`

The Succ Function

Syntax: `Succ(x)`
Result Type: same as parameter

Returns the successor of `x`. It is an error if `x` is the last value in the type of `x`, i.e. it has no successor.

The Pred Function

Syntax: `Pred(x)`
Result Type: same as parameter

`Pred(x)` returns the successor of `x`. It is an error if `x` is the first value in the type of `x`, i.e. it has no predecessor.

String Procedures and Functions

The string procedures and functions do not accept as parameters packed array of character types, but rather only string types.

The Length Function

Syntax: `Length(str)`
Result Type: `Integer`

Returns the dynamic length of the string, `str`.

The Pos Function

Syntax: `Pos(substr, str)`
Result Type: `Integer`

`Pos(substr,str)` searches for `substr` within `str`, and returns an `Integer` value that is the index of the first character of `substr` within `str`. If `substr` is not found, `Pos(substr,str)` returns zero.

The Concat Function

Syntax: Concat(str$_1$ [, str$_2$, ...str$_n$])
Result Type: anonymous string type

The `Concat` function concatenates all the parameters in the order in which they are written, and returns the concatenated string. Note that the number of characters in the result cannot exceed 255.

The Copy Function

Syntax: `Copy(source, index, count)`
Result Type: String type

The `Copy` function returns a string containing `count` characters from the string `source`, beginning at `source[index]`.

The Delete Procedure

Syntax: `Delete( dest, index, count )`

The `Delete` procedure removes count characters from the value of the string `dest`, beginning at `dest[index]`.

The Insert Procedure

Syntax: `Insert( source, dest, index )`

The `Insert` procedure inserts the string `source` into the string `dest`. The first character of `source` becomes `dest[index]`.

Logical Bit Procedures and Functions

This section describes a set of procedures and functions for bit manipulations. These routines correspond to a set of essentially identical machine instructions of the 65816.

The BAnd Function

Syntax: `BAnd(arg1,arg2)`
Result Type: `Integer or LongInt`

`BAnd` returns the logical AND of its two arguments. `arg1` and `arg2` are both expressions of a `scalar` type.

The BOr Function

Syntax: `BOr(arg1,arg2)`
Result Type: `Integer or LongInt`

`BOr` returns the logical OR of its two arguments. `arg1` and `arg2` are both expressions of a scalar type.

The BXor Function

Syntax: `BXor(arg1,arg2)`
Result Type: `Integer or LongInt`

`BXor` returns the logical exclusive OR of its two arguments. `arg1` and `arg2` are both expressions of a scalar type.

The BNot Function

Syntax: `BNot(arg1)`
Result Type: `Integer or LongInt`

`BNot` returns the logical negation (one's complement) of its argument. `arg1` is an expression of a scalar type.

The BSL Function

Syntax: `BSL(arg)`
Result Type: `Integer or LongInt`

`BSL` shifts left the bits of `arg` by one bit. `arg` is an expression of a scalar type. A zero is shifted into the low-order bit vacated by the shift operation.

The BSR Function

Syntax: `BSR(arg)`
Result Type: `Integer or LongInt`

`BSR` shifts right the bits of `arg` by one bit. `arg` is an expression of a scalar type. A zero is shifted into the high-order bit vacated by the shift operation.

The BRotL Function

Syntax: `BRotL(arg)`
Result Type: `Integer or LongInt`

`BRotL` rotates left the bits of `arg` by one bit. `arg` is an expression of a scalar type. Bits are shifted out of the low-order position and back into the high-order position.

The BRotR Function

Syntax: `BRotR(arg)`
Result Type: `Integer or LongInt`

`BRotR` rotates right the bits of `arg` by one bit. `arg` is an expression of a scalar type. Bits are shifted out of the high-order position and back into the low-order position.

The HiWrd Function

Syntax: `HiWrd(arg)`
Result Type: `Integer`

`HiWrd` returns the high order word of the scalar or pointer value `arg`, that is, bits 31-16 of a

LongInt. If arg is not a 32-bit value, HiWrd returns zero. When the argument is a simple variable or array access, no code is generated by this function because the argument is simply addressed and used as an Integer.

The LoWrd Function

Syntax: LoWrd(arg)
Result Type: Integer

LoWrd returns the low order word of the scalar or pointer value arg, that is, bits 15-0 of a LongInt. When the argument is a simple variable or array access, no code is generated by this function because the argument is simply addressed and used as an Integer.

Miscellaneous Procedures and Functions

This section describes byte-oriented procedures and functions as well as routines that operate on packed character arrays.

The byte-oriented routines allow a program to treat any variable simply as a sequence of bytes. No regard is given to data types. The byte-oriented routines discussed in this section are MoveLeft, MoveRight, and SizeOf.

The packed character array routines discussed in this section are ScanEq, ScanNE, and FillChar. Parameters to these routines cannot be subscripted and the routines always begin with the first character of the array.

The SizeOf Function

Syntax: SizeOf(id)
Result Type: LongInt

Returns the number of bytes occupied by the variable or type id. The value of SizeOf is determined by the Complete Pascal compiler, which treats it as a constant at compile time.

The Card Function

Syntax: Card(s)
Result Type: Integer

Counts the number of elements in the set s and returns an Integer value which is the cardinality of the set, that is, the number of members in the set.

The MoveLeft Procedure

Syntax: `MoveLeft(source, dest, count)`

`MoveLeft` copies a block of `count` contiguous bytes of storage from `source` to `dest` beginning at the lowest memory address of the blocks (the first byte of `source` and `dest`). `Source` and `dest` are variable references of any type other than a file type or a structured type containing a file type. `Count` is an `Integer` expression and is not range checked. When `source` and `dest` overlap, you should use this procedure if `source` is at the higher memory address.

The MoveRight Procedure

Syntax: `MoveRight(source, dest, count)`

`MoveRight` copies a block of `count` contiguous bytes of storage from `source` to `dest` beginning at the highest memory address of the blocks (the last byte of `source` and `dest`). `Source` and `dest` are variable references of any type other than a file type or a structured type containing a file type. `Count` is an integer expression and is not range checked. When `source` and `dest` overlap, you should use this procedure if `source` is at the lower memory address.

The FillChar Procedure

Syntax: `FillChar(dest, count, ch)`

`FillChar` fills a block of `count` contiguous characters of storage with the specified character `ch` beginning at the address of `dest`. `Dest` is a variable reference of type `Packed Array of Char`. `Count` is an integer expression and is not range checked. `Ch` is a value of a character type.

The ScanEq Function

Syntax: `ScanEq(limit, ch, source)`
Result: `Integer`

`ScanEq` scans a block of memory beginning at `source` for the first occurrence of the value `ch`. The scan proceeds until the value `ch` is found, or until `limit` bytes of memory have been scanned. If `ch` is not found within `limit` bytes of memory from the beginning of `source`, the value returned is equal to `limit`. Otherwise, the value returned is the number of bytes scanned before the value `ch` was found.

`limit` is an integer expression truncated to 16 bits and is not range checked. `Ch` is a character type. `Source` is a variable parameter with a value of type `Packed Array of Char`.

The ScanNe Function

Syntax: `ScanNe(limit, ch, source)`
Result: `Integer`

`ScanNe` operates the same as `ScanEq` except that it scans for the first character *not equal* to `ch`.

Apple IIGS Toolbox Error Handling

The Apple IIGS Toolbox defines a convention for reporting errors that may have occurred in the execution of a toolbox routine. If an error is detected during the execution of a toolbox routine, then upon exiting the tool call and returning to the application, the 65816 carry flag is set and the accumulator contains an error code describing the error that was detected. Complete Pascal provides a mechanism to obtain this information in a Pascal program.

The IsToolError Function

Syntax: `IsToolError`
Result Type: `Boolean`

Returns *true* if the last Apple IIGS Toolbox routine returned an error, otherwise *false*. IsToolError tests the carry flag of the 65816 processor to determine if an error exists. The function must be called *immediately* after a tool call, before any other operation is performed that might affect the 65816 carry flag. In the case that the tool call is a function and the function appears in an expression, the result of IsToolError may be incorrect since the evaluation of the expression may have affected the carry flag. In this case, a program should test the value of the variable _ToolErr described below.

The _ToolErr Variable

Syntax: `_ToolErr`
Type: `Integer`

The _ToolErr variable contains the error code returned by the last call to an Apple IIGS Toolbox routine. A non-zero value indicates an error. The compiler generates code which stores the value of the accumulator into the variable _ToolErr immediately after the tool call returns, before any other operation is performed that might destroy the value.

Example usage of IsToolError and _ToolErr:

```
h := NewHandle(100,myMemoryID,0,Ptr(0));
if IsToolError then begin
     theErr := _ToolErr;
     Writeln('Error allocating memory:',theErr);
     end;
```

Note that _ToolErr was saved to a temporary variable before calling the standard procedure Writeln. This is necessary because the implementation of the Writeln procedure calls several Apple IIGS Toolbox routines which would overwrite the value of _ToolErr and cause an erroneous error number to be output.

Many of the Apple IIGS Toolbox routines are defined to never return a non-zero error code because no error is possible. Complete Pascal, however, does not know this and still generates the appropriate code to save the tool error value into _ToolErr. If an application would like to avoid having this code generated, it may use the {$ToolErrorCheck-} directive to turn off this code generation. For more information, see Appendix B.

❖ **Note:** TML Pascal versions 1.x defined the variable ToolErrorNum to have the same meaning as _ToolErr. The variable ToolErrorNum may still be used, however, the new standardized name is _ToolErr, and is the preferred name.

Appendix A

Error Messages

This appendix lists all of the Complete Pascal editor, compiler and linker errors as well as GS/OS and IOResult error codes. Explanatory notes follow some of the error messages to help clarify the message, and in some cases additional notes appear explaining how to solve the error.

Some messages contain the special character '^' which is substituted by Complete Pascal with an identifier, label, or some other value to help make the error message as meaningful as possible.

Editor Errors

- When Complete Pascal detects that you are running dangerously low on memory. In order to avoid the potential loss of data you are recommended to free some memory by closing a document window. You may also choose the *Release Memory* option from the Preferences dialog.

  "Memory is getting low. Close a document window."

- You are not allowed to open the same file more than once.

  "Can't open that file. The file already open in another window."

- An error occurred while reading the document from disk. This could happen if the file is damaged or the disk has been removed from the disk drive.

  "Error reading file."

- This error is reported when Complete Pascal is unable to save the contents of a document window to disk. This is usually because the disk is locked, removed from the disk drive or the disk is full.

  "Error saving file."

- This error is reported after you have chosen to delete a disk file using the **Delete...** command from the GSOS menu and the disk has been removed from the disk drive or the disk is locked.
  "Error deleting file."

- This error is reported after you have chosen to rename a disk file using the **Rename...** command from the GSOS menu and you have specified an illegal filename, the disk has been removed from the disk drive or the disk is locked.
  "Error renaming file."

- When an operation has failed due to the lack of available memory.

  "Insufficient memory to complete that operation."

Compiler Errors

Lexical Errors

- A string constant literal is missing its closing quote.

  "String constant must not exceed source line."

- The syntax for a numeric literal value is incorrect.

  "Error in numeric literal."

- An illegal character has been detected in the soure file.

  "Illegal character in input."

- The end of file was reached before the program or unit was correctly terminated with a period.

  "Incomplete program."

- The Complete Pascal compiler skipped to the end of file searching for the end of a comment. If you open a comment with either { or (* then it must be terminated with } or *) respectively. See Chapter 11.

  "End of file encountered while reading a comment."

Syntax Errors

These error messages indicate that a program contains illegal Pascal syntax. While the error message indicates the symbol that it is expecting when it detected the error, it is usually possible that other symbols could also repair the syntax error. If you are unfamiliar with Pascal syntax then you should study Chapters 11 through 20.

"Identifier expected."
"Unexpected symbol."
"Integer constant expected."
"Error in statement."
"Error in expression."
" 'BEGIN' expected."
" 'DO' expected."
" 'END' expected."
" 'IMPLEMENTATION' expected."

" 'INTERFACE' expected."
" 'OF' expected."
" 'PROGRAM' or 'UNIT' expected."
" 'THEN' expected."
" 'TO' or 'DOWNTO' expected."
" 'UNTIL' expected."
" ')' expected."
" ':' expected."
" '(' expected."
" '[' expected."
" ']' expected."
" ';' expected."
" '=' expected."
" ',' expected."
" '..' expected."
" '.' expected."
" ':=' expected."

Semantic Errors

- When the same identifier has been declared more than once in the current block.

  "Duplicate identifier."

- In an array declaration, the lower bound is declared to be greater than the upper bound.

  "Low bound exceeds high bound."
- In the following contexts, the specified type must be a scalar or a subrange type such as *Integer, Char, Boolean,* etc. The type is also not allowed to be a real type.

  "Identifier is not of appropriate class."
  "Identifier not declared."
  "Sign not allowed."
  "Incompatible subrange types."
  "File not allowed here."

  "Tagfield type must be scalar or subrange."
  "Index type must be scalar or subrange."
  "Base type must be scalar or subrange."

- When the type of an expression passed as a parameter to one of Complete Pascal's predefined procedures or functions is incorrect. See Chapters 19 and 20.

  "Error in type of standard subprogram parameter."

- When a parameter list for a function or procedure declared FORWARD is repeated, or in a unit interface which does not match the current declaration.

  "Repitition of parameter list is not identical to previous declaration."

- File parameters must always be specified as VAR.

  "File value parameter not allowed."

- Complete Pascal does not permit the use of expressions whose type is *LongInt* for case expressions, for loop control variables or indexing arrays.

  "LongInt case/control variable/index expression are not implemented."

- The following semantic errors are self-explanatory.

  "Missing result type in function declaration."
  "Fixed point formatting allowed only for real types."
  "Number of parameters does not agree with declaration."
  "Actual parameter may not be PACKED for VAR formal parameter."
  "Operands are not assignment compatible."
  "Tests on equality allowed only."
  "Strict inclusion not allowed."
  "File comparison not allowed."
  "Illegal type of operand(s)."
  "Type of operand must be Boolean."
  "Set element type must be scalar or subrange."
  "Set element types not compatible."
  "Type of variable is not array."
  "Index type is not compatible with declaration."
  "Type of variable is not record."
  "Type of variable must be pointer."
  "Illegal parameter substitution."
  "Illegal type of loop control variable."
  "Boolean expression expected."
  "Assignment of files not allowed."
  "Label type incompatible with selecting expression."
  "Subrange bounds must be scalar."
  "No such field in this record."
  "Actual parameter must be a variable."
  "Control variable must not be declared on intermediate level."
  "Multidefined case label."
  "Again forward declared."
  "Multidefined label."
  "Multideclared label."
  "Undeclared label."
  "Error in base set."
  "Illegal function result assignment."
  "Must EXIT to an enclosing subprogram."
  "Control variable must not be formal."
  "Assignment to control variable is not allowed."
  "Forward referenced type "^" not completed in previous."
  "Forward declared subprogram "^" not completed in previous."
  "Label "^" was declared but not defined in previous block."
  "Size of string must be between 1 and 255."
  "@ is not allowed for expressions or INLINE and TOOL subprograms."
  "Type cast to a different size is not allowed."
  "Too many nested scopes of identifiers."
  "Too many nested procedures and/or functions."

"Index expression out of bounds."
"Implementation restriction."

Unit Errors

- When a unit named in a uses clause uses another unit itself which does not appear in the current uses clause.

  "The unit "^" require is required to USE this unit."

- When you specify the same unit name more than once in a uses clause. Also, when a unit named in a uses clause has the same name as the unit or program it is within.

  "Repitition of unit not allowed."

- When a unit named in this unit's uses clause has been recompiled. To correct the error, you must recompile the units named in a unit's uses clause.

  "This unit must be recompiled."

- When the ".p.o" file for the named unit cannot be found in either the *current unit prefix* or the *Unit Search Path* specified in the Preferences dialog.

  "Unable to find/open unit's symbol file."

- When the compiler is unable to create, open or write to the unit's ".p.o" file. The disk may be locked, removed from the disk drive or full.

  "Unable to write unit symbol file for this unit."

- Whenever you receive a new version of Complete Pascal you must recompile all of your units.

  "Unit must be recompiled with current version of compiler."

- When the number of declarations in this unit has exhausted the available memory allocated for the unit's symbol table. You should adjust the symbol table size in the Preferences dialog.

  "Symbol table space exhausted."

Linker Errors

- Either of the following two errors may occur when you have specified the same segment name in a *{$DSeg segname }* and *{$CSeg segname }* compiler directive.

  "Out of Memory. "
  "Segment "^" specifed as both CODE and DATA."

- A CODE or DATA segment became larger than 64K bytes. You must resegment your program so that the segment does not exceed this limit. See Chapter 16.

"Segment "^" too large."

- An externally defined label cannot be found by the linker. You should recheck the spelling of the symbol to make sure it is correct.

  "Unresolved linker reference to symbol "^"."

- After a Compile To Disk completes successfully, the linker attempts to write the application load file to disk. This error is reported if this file cannot be created and/or opened. This can happen if the disk is locked or has been removed from the disk drive.

  "Unable to create/open application file."

- This error is reported when Complete Pascal was able to create and/or open the output application file, but encountered an error during writing. This is usually caused because of a locked disk or a disk becoming too full to write the entire file to disk.

  "Error in writing to application file."

GS/OS Error Codes

This section lists the possible result codes of the standard function *IOResult* which reports the success of an I/O operation. The codes correspond to those returned by GS/OS, except for result codes -1, -2, and -3, which are generated by the Complete Pascal runtime routines for Pascal specific errors. The GS/OS error codes are provided here for reference. For complete documentation regarding these error codes consult Apple Computer's *GS/OS Reference* manual.

General Errors

$00	No error
$01	Invalid GS/OS call number
$04	Parameter count out of range
$07	GS/OS is busy

Device Call Errors

$10	Device not found
$11	Invalid device number
$20	Invalid request
$21	Invalid control or status code
$22	Bad call parameter
$23	Character device not open
$24	Character device already open
$25	Interrupt table full
$26	Resources not available
$27	I/O error
$28	No device connected
$29	Driver is busy
$2B	Device is write protected
$2C	Invalid byte count
$2D	Invalid block address
$2E	Disk has been switched
$2F	Device off line or no media present

File Call Errors

$40	Invalid pathname syntax
$43	Invalid reference number
$44	Subdirectory does not exist
$45	Volume not found
$46	File not found
$47	Create or rename with existing name
$48	Volume full error
$49	Volume directory full
$4A	Version error
$4B	Unsupported storage type
$4C	End-of-file encountered
$4D	Position out of range
$4E	Access not allowed
$4F	Buffer too small
$50	File is already open
$51	Directory error
$52	Unknown volume type
$53	Parameter out of range
$54	Out of memory
$57	Duplicate volume name
$58	Not a block device
$59	Specified level outside legal range
$5A	Block number too large
$5B	Invalid path names for ChangePath
$5C	Not an executable file
$5D	Operating system not supported
$5F	Too many applications on stack
$60	Data unavailable
$61	End of directory has been reached
$62	Invalid FST call class
$63	File does not contain required resource

Complete Pascal Specific Errors

-1	Textfile is not open for reading.
-2	Textfile is not open for writing.
-3	Numeric string conversion error in textfile.

Appendix B

Compiler Directives

Complete Pascal provides for several directives (or options) which affect the operation of the compiler and/or the code generated by the compiler. These compiler directives are written within the Pascal comment delimiters {...} or (*...*). A directive always begins with the symbol '$' and must appear immediately inside the opening comment delimiter and is followed by a letter (case insensitive) which designates the particular directive.

There are two types of directives: a switch directive and a parameter directive. A switch directive turns on or off a particular compiler feature by specifying'+' or '-' ,respectively, immediately after the directive letter. A parameter directive has one or more string arguments such as filenames or segment names. A string argument is terminated by a blank, an asterisk, or a right brace. If a string argument must contain one of these characters, then the string should be enclosed in single quotes.

Examples of compiler direcitves:

```
        (*$LongGlobals+ *)
        {$CSeg NewSeg }
```

The following sections describe each of the compiler directives available in Complete Pascal.

Classic Desk Accessory

{$CDA *menuName* }

The *CDA* directive is used to inform the compiler that a program implements a *Classic Desk Accessory* rather than a GS/OS application. The structure of a desk accessory program is somewhat different than an application. In particular, Complete Pascal must generate a special header which contains the CDA's name in the Classic Desk Accessory menu.

For complete information about writing Classic Desk Accessories in Complete Pascal see Chapter 10.

Because the compiler must generate special code for desk accessories before any code in a program, the option MUST appear before the reserved word UNIT in your source code for it to have any affect. Consider the following source code fragment:

```
{$CDA SHRDump }
UNIT MySHRDump;
    ...
end.
```

Code Segment

{$CSeg *segname* }

Default: {$CSeg main }

The *CSeg* option directs the compiler as to which code segment all subsequent subprograms should be allocated. The default code segment has the special reserved name *main*. For other code segment names, any string of characters is allowable so long as it does not contain a space. See Chapter 8 for more information regarding the use of code segments.

Definition Procedure

{$DefProc}

The $DefProc directive is used to inform the Complete Pascal compiler that the next procedure or function which appears in the source code implements an Apple IIGS Toolbox *definition procedure*. A definition procedure is implement exactly like any othe procedure or function except that the compiler generates slightly different code. In particular, the compiler generates code which sets the 65816 *data bank register* equal to the memory bank containing the Pascal global variables. When the procedure exits, code is generated which restores the data bank register.

Data Segment

{$DSeg *segname* }

Default: {$DSeg ~global }

The *DSeg* option directs the compiler as to which data segment all subsequent global variable declarations should be allocated. The default data segment has the special reserved name *~global*, for other data segment names, any string of characters is allowable so long as it does not contain a space, although conventions usually have the name begin with the tilde (~) character. Remeber that the *~global* data segment is the special segment in which the compiler uses the more efficient absolute addressing rather than absolute long addressing. See Chapter 8 for more information regarding the use of data segments.

External Referenced Variable

{$J+} or {$J-}

Default: {$J-}

The *External Referenced Variable* directive informs Complete Pascal that subsequent global variable declarations should not have storage allocated. Rather, the global variable declaration is treated as an external reference to a global variable declared elsewhere.

Typically, this directive is used for Pascal to access global storage declared in 65816 assembly language. However, it may be used with any language compatible with Complete Pascal linking conventions.

Consider the following source code fragment:

```
VAR   GlobVar1: integer;

      {J+ }
      GlobVar2: integer;
      {J- }

      GlobVar3: integer;
```

Long Globals

{$LongGlobals+ } or {$LongGlobals- }

Default: {$LongGlobals- }

This option directs the compiler to either turn on (+) or off (-) the generation of absolute long addresses for global variables in the *~global* data segment. Normally, the compiler generates code which sets the 65816 Data Bank register to the memory bank containing the global variables allocated in the *~global* data segment. However, there are several occasions where a program can not rely on this assumption. In order to guarantee that the compiler generates code which correctly addresses a program's global variables in the *~global* data segment under the conditions stated above, this option should be turned on, thus forcing absolute long addressing for all global variables.

New Desk Accessory

{$NDA *period eventMask menuName* }

The *NDA* directive is used to inform the compiler that a program is actually a *New Desk Accessory* rather than a GS/OS application. The structure of a desk accessory program is somewhat different than an application. In particular, Complete Pascal must generate a special header which contains the *period*, in 60ths of a second, in which the desk accessory needs periodic servicing, an *event mask* which describes what kinds of events the desk accessory must act on, and the *name* for the desk accessory that should appear in an application's *Apple Menu.*

For complete information about writing New Desk accessories in Complete Pascal see Chapter 9.

Because the compiler must generate special code for desk accessories before any code in a program, the option MUST appear before the reserved word UNIT in your source code for it to have any affect. Consider the following source code fragment:

```
{$NDA 60 -1 CTIClock }
UNIT CTIClock;
    ...
end.
```

Stack Size

{$StackSize *numbytes* }

Default: {$StackSize 8096 }

The *StackSize* directive is used to inform Complete Pascal as to how much space (in bytes of memory) should be allocated for the application's runtime stack. The runtime stack is used to store the return addresses of subprogram calls made during execution of a program and for a subprogram's local variables. Thus, the use of local variables in your program directly affects the runtime stack size your program requires.

The default runtime stack size is 8K, or 8096 bytes. If a program requires more or less storage, then this option should be used. However, at least 1K or 1024 bytes, and no more than 40K or 40960 bytes, may be requested. However, Complete Pascal does not check the value specified in the directive. See Appendix D for more information regarding the runtime stack.

Note that this option MUST appear before the reserved word PROGRAM in your source code for it to have any affect. Consider the following source code fragment:

```
{StackSize 10240 }
PROGRAM myProg;
begin
    ...
end.
```

Unit Symbol File Search Prefix

{$U *GSOS prefix* }

Default: {$U 0: }

This option allows an application to specify any legal GS/OS prefix for the purpose of searching for unit symbol files (".p.o" files). The Complete Pascal compiler does not recompile the interface part of units specified in a USES clause, but rather loads a precompiled symbol table of the declarations in a unit from a ".p.o" file. To search for these files, Complete Pascal maintains a current *unit prefix* used to create the full pathname of a ".p.o" file. The default prefix is "0:" which is the GS/OS prefix for the current directory. This *unit prefix* can be changed to any legal prefix using this comiler directive. For example,

```
USES  Types,
      {$U :CTI:MYSTUFF: } HandyRoutines;
```

Note that if a ".p.o" file cannot be found using the compiler's *unit prefix* the Complete Pascal compiler will also attempt to find the file by using the *Unit Search Path* specified in the compiler's Prefrences Dialog (See Chapter 3). If the required ".p.o" file cannot be found using either prefix then an error is reported.

Tool ErrorNum Check

{$ToolErrorChk+} or {$ToolErrorChk-}

Default: {$ToolErrorChk+}

This directive allows an application to control the automatic generation of error checking code for Apple IIGS Toolbox calls. As discussed in Chapter 20, Complete Pascal generates a STA _ToolErr instruction after every call to a Toolbox routine so that the special Complete Pascal global variable *_ToolErr* always contains the error code of the most recently called Toolbox routine. A non-zero *_ToolErr* indicates an error occured during the exectuion of the last Toolbox routine, and the value of *_ToolErr* is an error code that can be used to determine the cause of the error.

In many cases, an application does not need to check the value of *_ToolErr* after Toolbox calls, and would rather not have the STA _ToolErr instruction generated in order to decrease the code size of an application. To achieve this, the *$ToolErrorChk* directive is turned off.

Appendix C

Apple IIGS Toolbox Units

The Apple IIGS Toolbox is the large collection of sophisticated software which is part of every Apple IIGS. The Toolbox implements the QuickDraw Super Hi-res Graphics engine as well as the Apple Desktop Interface which includes windows, menus, dialogs, controls and much more.

As discussed in Chapter 8, Complete Pascal provides access to the Apple IIGS Toolbox with a collection of predefined Pascal Units. This appendix lists the source code to each of these predefined units so that you can use them in your programs. The following units are provided in this appendix:

- ACE
- Controls
- Dialogs
- Fonts
- IntMath
- Lists
- Locator
- Menus
- MiscTool
- NoteSyn
- QDAux
- Resources
- Scheduler
- Sound
- TextEdit
- Types
- ADB
- Desk
- Events
- GSOS
- LineEdit
- Loader
- Memory
- MIDI
- NoteSeq
- Print
- QuickDraw
- SANE
- Scrap
- StdFile
- TextTool
- Windows

In addition, the Index to this manual alphabetizes each of the Toolbox procedures and functions referenced in this Appendix.

ACE

```
{********************************************
; File: ACE.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT ACE;

INTERFACE
```

```
USES Types;

CONST  aceNoError                = $0;          {Error - }
       aceIsActive               = $1D01;       {Error - }
       aceBadDP                  = $1D02;       {Error - }
       aceNotActive              = $1D03;       {Error - }
       aceNoSuchParam            = $1D04;       {Error - }
       aceBadMethod              = $1D05;       {Error - }
       aceBadSrc                 = $1D06;       {Error - }
       aceBadDest                = $1D07;       {Error - }
       aceDatatOverlap           = $1D08;       {Error - }
       aceNotImplemented         = $1DFF;       {Error - }

PROCEDURE ACEBootInit;
PROCEDURE ACEStartUp
              (dPageAddr:    Integer);
PROCEDURE ACEShutDown;
FUNCTION  ACEVersion:        Integer;
PROCEDURE ACEReset;
FUNCTION  ACEStatus:         Boolean;
FUNCTION  ACEInfo
              (infoItemCode: Integer): Longint;
PROCEDURE ACECompBegin;
PROCEDURE ACECompress
              (src:          Handle;
               srcOffset:    Longint;
               dest:         Handle;
               destOffset:   Longint;
               nBlks:        Integer;
               method:       Integer);
PROCEDURE ACEExpand
              (src:          Handle;
               srcOffset:    Longint;
               dest:         Handle;
               destOffset:   Longint;
               nBlks:        Integer;
               method:       Integer);
PROCEDURE ACEExpBegin;

IMPLEMENTATION
END.
```

ADB

```
{********************************************
; File: ADB.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT ADB;
```

```
INTERFACE
USES Types;

CONST  cmndIncomplete       = $0910;  {error - Command not completed. }
       cantSync             = $0911;  {error - Can't synchronize }
       adbBusy              = $0982;  {error - Busy (command pending) }
       devNotAtAddr         = $0983;  {error - Device not present at address }
       srqListFull          = $0984;  {error - List full }
       readModes            = $000A;  {ReadKeyMicroData - }
       readConfig           = $000B;  {ReadKeyMicroData - }
       readADBError         = $000C;  {ReadKeyMicroData - }
       readVersionNum       = $000D;  {ReadKeyMicroData - }
       readAvailCharSet     = $000E;  {ReadKeyMicroData - }
       readAvailLayout      = $000F;  {ReadKeyMicroData - }
       readMicroMem         = $0009;  {ReadKeyMicroMem - }
       abort                = $0001;  {SendInfo - command }
       resetKbd             = $0002;  {SendInfo - command }
       flushKbd             = $0003;  {SendInfo - command }
       setModes             = $0004;  {SendInfo - 2nd param is pointer to mode byte }
       clearModes           = $0005;  {SendInfo - 2nd param is pointer to mode Byte }
       setConfig            = $0006;  {SendInfo - 2nd param is pointer to SetConfigRec }
       synch                = $0007;  {SendInfo - 2nd param is pointer to SynchRec }
       writeMicroMem        = $0008;  {SendInfo - 2nd param is pointer to MicroControlMemRec}
       resetSys             = $0010;  {SendInfo - command }
       keyCode              = $0011;  {SendInfo - 2nd param is pointer to key code byte. }
       resetADB             = $0040;  {SendInfo - command }
       transmitADBBytes     = $0047;  {SendInfo - add number of bytes to this }
       enableSRQ            = $0050;  {SendInfo - command - ADB address in low nibble}
       flushADBDevBuf       = $0060;  {SendInfo - command - ADB address in low nibble}
       disableSRQ           = $0070;  {SendInfo - command - ADB address in low nibble}
       transmit2ADBBytes    = $0080;  {SendInfo - add ADB address to this}
       listen               = $0080;  {SendInfo - adbCommand = listen + (16 * reg) +
                                      (adb address) }
       talk                 = $00C0;  {SendInfo - adbCommand = talk + ( 16 * reg) +
                                      (adb address) }

TYPE   ReadConfigRecPtr     = ^ReadConfigRec;
       ReadConfigRec        =
              PACKED RECORD
                     rcRepeatDelay:       Byte;  { Output Byte: Repeat / Delay }
                     rcLayoutOrLang:      Byte;  { Output Byte: Layout / Language }
                     rcADBAddr            Byte;  { Output Byte: ADB address -
                                                 keyboard and mouse }
              END;

       SetConfigRecPtr      = ^SetConfigRec;
       SetConfigRec         =
              PACKED RECORD
                     scADBAddr:           Byte; { keyboard and mouse }
                     scLayoutOrLang:      Byte;
                     scRepeatDelay:       Byte;
              END;

       SynchRecPtr          = ^SynchRec;
       SynchRec             =
              PACKED RECORD
                     synchMode: Byte;
                     synchKybdMouseAddr:  Byte;
```

```
                synchLayoutOrLang:    Byte;
                synchRepeatDelay:     Byte;
            END;

    ScaleRecPtr         = ^ScaleRec;
    ScaleRec            =
            RECORD
                xDivide:              Integer;
                yDivide:              Integer;
                xOffset:              Integer;
                yOffset:              Integer;
                xMultiply:            Integer;
                yMultiply:            Integer;
            END;

PROCEDURE ADBBootInit;
PROCEDURE ADBStartUp;
PROCEDURE ADBShutDown;
FUNCTION  ADBVersion:           Integer;
PROCEDURE ADBReset;
FUNCTION  ADBStatus:            Boolean;
PROCEDURE AbsOff;
PROCEDURE AbsOn;
PROCEDURE AsyncADBReceive
           (compPtr:            Ptr);
PROCEDURE ClearSRQTable;
PROCEDURE GetAbsScale
           (VAR dataInPtr: ScaleRec);
FUNCTION  ReadAbs:              Integer;
PROCEDURE ReadKeyMicroData
           (dataLength:         Integer;
            dataPtr:            Ptr;
            adbCommand:         Integer);
PROCEDURE ReadKeyMicroMem
           (dataLength:         Integer;
           dataPtr:             Ptr;
           adbCommand:          Integer);
PROCEDURE SendInfo
           (dataLength:         Integer;
            dataPtr:            Ptr;
            adbCommand:         Integer);
PROCEDURE SetAbsScale
           (dataOutPtr:         ScaleRec);
PROCEDURE SRQPoll
           (compPtr:            Ptr;
            adbRegAddr:         Integer);
PROCEDURE SRQRemove
           (adbRegAddr:         Integer);
PROCEDURE SyncADBReceive
           (inputWord:          Integer;
            compPtr:            Ptr;
            adbCommand:         Integer);

IMPLEMENTATION
END.
```

Controls

```
{********************************************
; File: Controls.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Controls;

INTERFACE
USES Types, QuickDraw, Events;

CONST   wmNotStartedUp          = $1001;  {error - Window manager was not started first. }
        noConstraint            = $0000;  {Axis Parameter - No constraint on movement. }
        hAxisOnly               = $0001;  {Axis Parameter - Horizontal axis only. }
        vAxisOnly               = $0002;  {Axis Parameter - Vertical axis only. }
        simpRound               = $0000;  {CtlFlag - Simple button flag }
        upFlag                  = $0001;  {CtlFlag - Scroll bar flag. }
        boldButton              = $0001;  {CtlFlag - Bold round cornered outlined button.}
        simpBRound              = $0001;  {CtlFlag - Simple button flag }
        downFlag                = $0002;  {CtlFlag - Scroll bar flag. }
        simpSquare              = $0002;  {CtlFlag - Simple button flag }
        simpDropSquare          = $0003;  {CtlFlag - Simple button flag }
        leftFlag                = $0004;  {CtlFlag - Scroll bar flag. }
        rightFlag               = $0008;  {CtlFlag - Scroll bar flag. }
        dirScroll               = $0010;  {CtlFlag - Scroll bar flag. }
        horScroll               = $0010;  {CtlFlag - Scroll bar flag. }
        family                  = $007F;  {CtlFlag - Mask for radio button family number }
        ctlInVis                = $0080;  {CtlFlag - invisible mask for any type of control }
        inListBox               = $88;    {CtlFlag - }
        simpleProc              = $00000000;   {CtlProc - }
        checkProc               = $02000000;   {CtlProc - }
        radioProc               = $04000000;   {CtlProc - }
        scrollProc              = $06000000;   {CtlProc - }
        growProc                = $08000000;   {CtlProc - }
        drawCtl                 = $0000;  {DefProc - Draw control command. }
        calcCRect               = $0001;  {DefProc - Compute drag RECT command. }
        testCtl                 = $0002;  {DefProc - Hit test command. }
        initCtl                 = $0003;  {DefProc - Initialize command. }
        dispCtl                 = $0004;  {DefProc - Dispose command. }
        posCtl                  = $0005;  {DefProc - Move indicator command. }
        thumbCtl                = $0006;  {DefProc - Compute drag parameters command. }
        dragCtl                 = $0007;  {DefProc - Drag command. }
        autoTrack               = $0008;  {DefProc - Action command. }
        newValue                = $0009;  {DefProc - Set new value command. }
        setParams               = $000A;  {DefProc - Set new parameters command. }
        moveCtl                 = $000B;  {DefProc - Move command. }
        recSize                 = $000C;  {DefProc - Return record size command. }
        noHilite                = $0000;  {hiliteState - Param to HiliteControl }
        inactiveHilite          = $00FF;  {hiliteState - Param to HiliteControl }
        noPart                  = $0000;  {PartCode - }
```

```
simpleButton          = $0002;    {PartCode - }
checkBox              = $0003;    {PartCode - }
radioButton           = $0004;    {PartCode - }
upArrow               = $0005;    {PartCode - }
downArrow             = $0006;    {PartCode - }
pageUp                = $0007;    {PartCode - }
pageDown              = $0008;    {PartCode - }
growBox               = $000A;    {PartCode - }
thumb                 = $0081;    {PartCode - }
fCtlTarget            = $8000;    {CtlRec.ctlMoreFlags - is current target of
                                  typing commands }
fCtlCanBeTarget       = $4000;    {CtlRec.ctlMoreFlags - can be made the target
                                  control }
fCtlWantEvents        = $2000;    {CtlRec.ctlMoreFlags - control can be called
                                  view SendEventToCtl }
fCtlProcRefNotPtr     = $1000;    {CtlRec.ctlMoreFlags - set = ID of defproc,
                                  clear = pointer to defproc}
fCtlTellAboutSize     = $0800;    {CtlRec.ctlMoreFlags - set if ctl needs
                                  notification when size of owning window changes}
fCtlIsMultiPar        = $0400;    {CtlRec.ctlMoreFlags - set if ctl needs
                                  notification to be hidden }
titleIsPtr            = $00;      {Ctl Verb - }
titleIsHandle         = $01;      {Ctl Verb - }
titleIsResource       = $02;      {Ctl Verb - }
colorTableIsPtr       = $00;      {Ctl Verb - }
colorTableIsHandle    = $04;      {Ctl Verb - }
colorTableIsResource  = $08;      {Ctl Verb - }
ctlHandleEvent        = $0D;      {DefProc message - }
ctlChangeTarget       = $0E;      {DefProc message - }
ctlChangeBounds       = $0F;      {DefProc message - }
ctlWindChangeSize     = $10;      {DefProc message - }
ctlHandleTab          = $11;      {DefProc message - }
ctlHideCtl            = $12;      {DefProc message - }
singlePtr             = $0000;    {InputVerb - }
singleHandle          = $0001;    {InputVerb - }
singleResource        = $0002;    {InputVerb - }
ptrToPtr              = $0003;    {InputVerb - }
ptrToHandle           = $0004;    {InputVerb - }
ptrToResource         = $0005;    {InputVerb - }
handleToPtr           = $0006;    {InputVerb - }
handleToHandle        = $0007;    {InputVerb - }
handleToResource      = $0008;    {InputVerb - }
resourceToResource    = $0009;    {InputVerb - }
simpleButtonControl   = $80000000;     {ProcRefs - }
checkControl          = $82000000;     {ProcRefs - }
radioControl          = $84000000;     {ProcRefs - }
scrollBarControl      = $86000000;     {ProcRefs - }
growControl           = $88000000;     {ProcRefs - }
statTextControl       = $81000000;     {ProcRefs - }
editLineControl       = $83000000;     {ProcRefs - }
editTextControl       = $85000000;     {ProcRefs - }
popUpControl          = $87000000;     {ProcRefs - }
listControl           = $89000000;     {ProcRefs - }
iconButtonControl     = $07FF0001;     {ProcRefs - }
pictureControl        = $8D000000;     {ProcRefs - }
noCtlError            = $1004;    {Error - no controls in window }
noSuperCtlError       = $1005;    {Error - no super controls in window}
noCtlTargetError      = $1006;    {Error - no target super control }
```

```
        notSuperCtlError     = $1007;  {Error - action can only be done on super control }
        canNotBeTargetErro   = $1008;  {Error - conrol cannot be made target }
        noSuchIDError        = $1009;  {Error - specified ID cannot be found }
        tooFewParmsError     = $100A;  {Error - not enough params specified}
        noCtlToBeTargetError = $100B;  {Error - NextCtl call, no ctl could be target }
        noWind_Err           = $100C;  {Error - there is no front window }

TYPE    {$IFC UNDEFINED WindowPtr }
        WindowPtr              = GrafPortPtr;
        {$SETC WindowPtr := 0 }
        {$ENDC}

        BarColorsHndl          = ^BarColorsPtr;
        BarColorsPtr           = ^BarColors;
        BarColors              =
           RECORD
                barOutline: Integer;   { color for outlining bar, arrows, and thumb }
                barNorArrow:Integer;   { color of arrows when not highlighted }
                barSelArrow:Integer;   { color of arrows when highlighted }
                barArrowBack:Integer;  { color of arrow box's background }
                barNorThumb:Integer;   { color of thumb's background when not highlighted }
                barSelThumb:Integer;   { color of thumb's background when highlighted}
                barPageRgn: Integer;   { color and pattern page region: high byte - 1
                                       = dither, 0 = solid }
                barInactive:Integer;   { color of scroll bar's interior when inactive}
           END;

        BoxColorsHndl              = ^BoxColorsPtr;
        BoxColorsPtr               = ^BoxColors;
        BoxColors                  =
           RECORD
                boxReserved:Integer;   { reserved }
                boxNor:     Integer;   { color of box when not checked }
                boxSel:     Integer;   { color of box when checked }
                boxTitle:   Integer;   { color of check box's title }
           END;

        BttnColorsHndl             = ^BttnColorsPtr;
        BttnColorsPtr              = ^BttnColors;
        BttnColors                 =
           RECORD
                bttnOutline:Integer;   { color of outline }
                bttnNorBack:Integer;   { color of background when not selected }
                bttnSelBack:Integer;   { color of background when selected }
                bttnNorText:Integer;   { color of title's text when not selected }
                bttnSelText:Integer;   { color of title's text when selected }
           END;

        CtlRecHndlPtr              = ^CtlRecHndl;
        CtlRecHndl                 = ^CtlRecPtr;
        CtlRecPtr                  = ^CtlRec;
        CtlRec                     =
           PACKED RECORD
                ctlNext:    CtlRecHndl;    { Handle of next control. }
                ctlOwner:   WindowPtr;     { Pointer to control's window. }
                ctlRect:    Rect;          { Enclosing rectangle. }
                ctlFlag:    Byte;          { Bit flags. }
                ctlHilite:  Byte;          { Highlighted part. }
```

```
            ctlValue:    Integer;     { Control's value. }
            ctlProc:     LongProcPtr;  { Control's definition procedure. }
            ctlAction:   LongProcPtr;  { Control's action procedure. }
            ctlData:     Longint;     { Reserved for CtrlProc's use. }
            ctlRefCon:   Longint;     { Reserved for application's use. }
            ctlColor:    Ptr;         { Pointer to appropriate color table. }
            ctlReserved: PACKED ARRAY [1..16] OF Byte;
                                      { Reserved for future expansion }
            ctlID:       Longint;
            ctlMoreFlags:Integer;
            ctlVersion:  Integer;
        END;

    LimitBlkHndl            = ^LimitBlkPtr;
    LimitBlkPtr             = ^LimitBlk;
    LimitBlk                =
        RECORD
            boundRect:  Rect;    { Drag bounds. }
            slopRect:   Rect;    { Cursor bounds. }
            axisParam:  Integer; { Movement constrains. }
            dragPatt:   Ptr;     { Pointer to 32 byte Pattern for drag outline.}
        END;

    RadioColorsHndl         = ^RadioColorsPtr;
    RadioColorsPtr          = ^RadioColors;
    RadioColors             =
        RECORD
            radReserved:Integer;   { reserved }
            radNor:     Integer;   { color of radio button when off }
            radSel:     Integer;   { color of radio button when on }
            radTitle:   Integer;   { color of radio button's title text}
        END;

PROCEDURE CtlBootInit;
PROCEDURE CtlStartUp
            (userID:            Integer;
             dPageAddr:         Integer);
PROCEDURE CtlShutDown;
FUNCTION  ctlVersion:           Integer;
PROCEDURE CtlReset;
FUNCTION  CtlStatus:            Boolean;
PROCEDURE CtlNewRes;
PROCEDURE DisposeControl
            (theControlHandle:  CtlRecHndl);
PROCEDURE DragControl
            (startX:            Integer;
             startY:            Integer;
             limitRectPtr:      Rect;
             slopRectPtr:       Rect;
             dragFlag:          Integer;
             theControlHandle:  CtlRecHndl);
FUNCTION  DragRect
            (actionProcPtr:     VoidProcPtr;
             dragPatternPtr:    Pattern;
             startX:            Integer;
             startY:            Integer;
             dragRectPtr:       Rect;
             limitRectPtr:      Rect;
```

```
           slopRectPtr:        Rect;
           dragFlag:           Integer): Longint;
PROCEDURE DrawControls
          (theWindowPtr:       WindowPtr);
PROCEDURE DrawOneCtl
          (theControlHandle:   CtlRecHndl);
PROCEDURE EraseControl
          (theControlHandle:   CtlRecHndl);
FUNCTION  FindControl
          (VAR foundCtl:       CtlRecHndl;
           pointX:             Integer;
           pointY:             Integer;
           theWindowPtr:       WindowPtr): Integer;
FUNCTION  GetCtlAction
          (theControlHandle:   CtlRecHndl): LongProcPtr;
FUNCTION  GetCtlDPage: Integer;
FUNCTION  GetCtlParams
          (theControlHandle:   CtlRecHndl): Longint;
FUNCTION  GetCtlRefCon
          (theControlHandle:   CtlRecHndl): Longint;
FUNCTION  GetCtlTitle
          (theControlHandle:   CtlRecHndl): Ptr;
FUNCTION  GetCtlValue
          (theControlHandle:   CtlRecHndl): Integer;
FUNCTION  GrowSize: Longint;
PROCEDURE HideControl
          (theControlHandle:   CtlRecHndl);
PROCEDURE HiliteControl
          (hiliteState:        Integer;
           theControlHandle:   CtlRecHndl);
PROCEDURE KillControls
          (theWindowPtr:       WindowPtr);
PROCEDURE MoveControl
          (newX:               Integer;
           newY:               Integer;
           theControlHandle:   CtlRecHndl);
FUNCTION  NewControl
          (theWindowPtr:       WindowPtr;
           boundsRectPtr:      Rect;
           titlePtr:           Ptr;
           flag:               Integer;
           value:              Integer;
           param1:             Integer;
           param2:             Integer;
           defProcPtr:         LongProcPtr;
           refCon:             Longint;
           __colorTablePtr:    Ptr):   CtlRecHndl;
PROCEDURE SetCtlAction
          (newActionPtr:       LongProcPtr;
           theControlHandle:   CtlRecHndl);
FUNCTION  SetCtlIcons
          (newFontHandle:      FontHndl): FontHndl;
PROCEDURE SetCtlParams
          (param2:             Integer;
           param1:             Integer;
           theControlHandle:   CtlRecHndl);
PROCEDURE SetCtlRefCon
          (newRefCon:          Longint;
                theControlHandle:  CtlRecHndl);
PROCEDURE SetCtlTitle
               (title:             Str255;
                theControlHandle:  CtlRecHndl);
PROCEDURE SetCtlValue
               (curValue:          Integer;
                theControlHandle:  CtlRecHndl);
PROCEDURE ShowControl
               (theControlHandle:  CtlRecHndl);
FUNCTION  TestControl
               (pointX:            Integer;
                pointY:            Integer;
                theControlHandle:  CtlRecHndl): Integer;
FUNCTION  TrackControl
               (startX:            Integer;
                startY:            Integer;
                actionProcPtr:     LongProcPtr;
                theControlHndl:    CtlRecHndl): Integer;
FUNCTION  NewControl2
               (ownerPtr:          WindowPtr;
                inputDesc:         RefDescriptor;
                inputRef:          Ref): CtlRecHndl;
FUNCTION  FindTargetCtl:           CtlRecHndl;
FUNCTION  MakeNextCtlTarget:       CtlRecHndl;
PROCEDURE MakeThisCtlTarget
               (targetCtlRecHndl:  CtlRecHndl);
PROCEDURE CallCtlDefProc
               (__ctlRecHndl:      CtlRecHndl;
               defProcMessage:     Integer;
               __param:            Longint);
PROCEDURE NotifyControls
               (__mask:            Integer;
                message:           Integer;
                __param:           Longint;
                window:            WindowPtr);
FUNCTION  SendEventToCtl
               (targetOnlyFlag:    Integer;
                __WindowPtr:       WindowPtr;
                extendedTaskRecPtr Ptr): Boolean;
FUNCTION  GetCtlID
               (theCtlHandle:      CtlRecHndl): Longint;
PROCEDURE SetCtlID
               (newID:             Longint;
                theCtlHandle:      CtlRecHndl);
FUNCTION  GetCtlMoreFlags
               (theCtlHandle:      CtlRecHndl): Longint;
FUNCTION  SetCtlMoreFlags
               (newID:             Longint;
                theCtlHandle:      CtlRecHndl): Longint;
FUNCTION  GetCtlHandleFromID
               (__WindowPtr:       WindowPtr;
                ControlID:         Longint): CtlRecHndl;
PROCEDURE SetCtlParamPtr
               (SubArrayPtr: Ptr);
FUNCTION  GetCtlParamPtr: Ptr;
FUNCTION  CMLoadResource
               (__ResType:         Integer;
                __ResID:           Longint): Handle;
```

```
PROCEDURE CMReleaseResource
                (__ResType:          Integer;
                 __ResID:            Longint);
PROCEDURE InvalCtls
                (__WindowPtr:        Longint);

IMPLEMENTATION
END.
```

Desk

```
{*********************************************
; File: Desk.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
*********************************************}

UNIT Desk;

INTERFACE
USES Types, Events;

CONST  daNotFound              = $0510;   {error - desk accessory not found }
       notSysWindow            = $0511;   {error - not the system window }
       eventAction             = $0001;   {NDA action code - }
       runAction               = $0002;   {NDA action code - }
       undoAction              = $0005;   {NDA action code - }
       cutAction               = $0006;   {NDA action code - }
       copyAction              = $0007;   {NDA action code - }
       pasteAction             = $0008;   {NDA action code - }
       clearAction             = $0009;   {NDA action code - }
       cursorAction            = $0003;   {NDAaction code - }
       undo                    = $0001;   {System Edit - edit type }
       cut                     = $0002;   {System Edit - edit type }
       copy                    = $0003;   {System Edit - edit type }
       paste                   = $0004;   {System Edit - edit type }
       clear                   = $0005;   {System Edit - edit type }

PROCEDURE DeskBootInit;
PROCEDURE DeskStartUp;
PROCEDURE DeskShutDown;
FUNCTION  DeskVersion:               Integer;
PROCEDURE DeskReset;
FUNCTION  DeskStatus:                Boolean;
PROCEDURE ChooseCDA;
PROCEDURE CloseAllNDAs;
PROCEDURE CloseNDA
                (refNum:             Integer);
PROCEDURE CloseNDAByWinPtr
                (theWindowPtr:       GrafPortPtr);
```

```
PROCEDURE FixAppleMenu
              (startingID:        Integer);
FUNCTION  GetDAStrPtr:            Ptr;
FUNCTION  GetNumNDAs:             Integer;
PROCEDURE InstallCDA
              (idHandle:          Handle);
PROCEDURE InstallNDA
              (idHandle:          Handle);
FUNCTION  OpenNDA
              (idNum:             Integer): Integer;
PROCEDURE RestAll;
PROCEDURE RestScrn;
PROCEDURE SaveAll;
PROCEDURE SaveScrn;
PROCEDURE SetDAStrPtr
              (altDispHandle:     Handle;
               stringTablePtr:    Ptr);
PROCEDURE SystemClick
              (eventRecPtr:       EventRecord;
               theWindowPtr:      GrafPortPtr;
               findWndwResult:    Integer);
FUNCTION  SystemEdit
              (editType: Integer):Boolean;
FUNCTION  SystemEvent
              (eventWhat:         Integer;
               eventMessage:      Longint;
               eventWhen:         Longint;
               eventWhere:        Point;
               eventMods:         Integer): Boolean;
PROCEDURE SystemTask;
PROCEDURE AddToRunQ
              (headerPtr:         Ptr);
PROCEDURE RemoveFromRunQ
              (headerPtr:         Ptr);
PROCEDURE RemoveCDA
              (idHandle:          Handle);
PROCEDURE RemoveNDA
              (idHandle:          Handle);

IMPLEMENTATION
END.
```

Dialogs

```
{********************************************
; File: Dialogs.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Dialogs;

INTERFACE
USES Types, QuickDraw, Events, Controls, Windows, LineEdit;

CONST  badItemType          = $150A;   {error - }
       newItemFailed        = $150B;   {error - }
       itemNotFound         = $150C;   {error - }
       notModalDialog       = $150D;   {error - }
       getInitView          = $0001;   {Command - }
       getInitTotal         = $0002;   {Command - }
       getInitValue         = $0003;   {Command - }
       scrollLineUp         = $0004;   {Command - }
       scrollLineDown       = $0005;   {Command - }
       scrollPageUp         = $0006;   {Command - }
       scrollPageDown       = $0007;   {Command - }
       scrollThumb          = $0008;   {Command - }
       buttonItem           = $000A;   {Item Type - }
       checkItem            = $000B;   {Item Type - }
       radioItem            = $000C;   {Item Type - }
       scrollBarItem        = $000D;   {Item Type - }
       userCtlItem          = $000E;   {Item Type - }
       statText             = $000F;   {Item Type - }
       longStatText         = $0010;   {Item Type - }
       editLine             = $0011;   {Item Type - }
       iconItem             = $0012;   {Item Type - }
       picItem              = $0013;   {Item Type - }
       userItem             = $0014;   {Item Type - }
       userCtlItem2         = $0015;   {Item Type - }
       longStatText2        = $0016;   {Item Type - }
       itemDisable          = $8000;   {Item Type - }
       minItemType          = $000A;   {Item Type Range - }
       maxItemType          = $0016;   {Item Type Range - }
       ok                   = $0001;   {ItemID - }
       cancel               = $0002;   {ItemID - }
       inButton             = $0002;   {ModalDialog2 - Part code }
       inCheckBox           = $0003;   {ModalDialog2 - Part code }
       inRadioButton        = $0004;   {ModalDialog2 - Part code }
       inUpArrow            = $0005;   {ModalDialog2 - Part code }
       inDownArrow          = $0006;   {ModalDialog2 - Part code }
       inPageUp             = $0007;   {ModalDialog2 - Part code }
       inPageDown           = $0008;   {ModalDialog2 - Part code }
       inStatText           = $0009;   {ModalDialog2 - Part code }
       inGrow               = $000A;   {ModalDialog2 - Part code }
```

```
      inEditLine                = $000B;   {ModalDialog2 - Part code }
      inUserItem                = $000C;   {ModalDialog2 - Part code }
      inLongStatText            = $000D;   {ModalDialog2 - Part code }
      inIconItem                = $000E;   {ModalDialog2 - Part code }
      inLongStatText2           = $000F;   {ModalDialog2 - }
      inThumb                   = $0081;   {ModalDialog2 - Part code }
      okDefault                 = $0000;   {Stage Bit Vector - }
      cancelDefault             = $0040;   {Stage Bit Vector - }
      alertDrawn                = $0080;   {Stage Bit Vector - }

      {$IFC UNDEFINED atItemListLength } {AlertTemplate - Default number of Item
                                          Templates }
      atItemListLength          = $0005;

      {$SETC atItemListLength := 0}
      {$ENDC}
      {$IFC UNDEFINED dtItemListLength }
                                         {DialogTemplate - Default number of Item
                                          Templates }
      dtItemListLength          = $0008;
      {$SETC dtItemListLength := 0}
      {$ENDC}

TYPE  DialogPtr                 = WindowPtr;
      ItemTempHndl              = ^ItemTempPtr;
      ItemTempPtr               = ^ItemTemplate;
      ItemTemplate              =
              RECORD
                      itemID:              Integer;
                      itemRect:            Rect;
                      itemType:            Integer;
                      itemDescr:           Ptr;
                      itemValue:           Integer;
                      itemFlag:            Integer;
                      itemColor:           Ptr; { pointer to appropriate type of color
                                                 table }
              END;

      AlertTempHndl             = ^AlertTempPtr;
      AlertTempPtr              = ^AlertTemplate;
      AlertTemplate             =
              RECORD
                      atBoundsRect: Rect;
                      atAlertID:           Integer;
                      atStage1:            Byte;
                      atStage2:            Byte;
                      atStage3:            Byte;
                      atStage4:            Byte;
                      atItemList:          ARRAY [1..atItemListLength] OF ItemTempPtr;
                                                 { Null terminated array }
              END;

      DlgTempHndl               = ^DlgTempPtr;
      DlgTempPtr                = ^DialogTemplate;
      DialogTemplate            =
              RECORD
                      dtBoundsRect: Rect;
                      dtVisible:           Boolean;
```

```
                dtRefCon:                 Longint;
                dtItemList:               ARRAY [1..dtItemListLength] OF ItemTempPtr;
                                                { Null terminated array }
            END;

        IconRecordHndl          = ^IconRecordPtr;
        IconRecordPtr           = ^IconRecord;
        IconRecord              =
            RECORD
                iconRect:       Rect;
                iconImage:      PACKED ARRAY [1..1] OF Byte;
            END;

        UserCtlItemPBHndl       = ^UserCtlItemPBPtr;
        UserCtlItemPBPtr        = ^UserCtlItemPB;
        UserCtlItemPB           =
            RECORD
                defProcParm:    Longint; { ? should this be a LongProcPtr? }
                titleParm:              Ptr;
                param2:                 Integer;
                param1:                 Integer;
            END;

    PROCEDURE DialogBootInit;
    PROCEDURE DialogStartUp
                (userID:            Integer);
    PROCEDURE DialogShutDown;
    FUNCTION  DialogVersion:        Integer;
    PROCEDURE DialogReset;
    FUNCTION  DialogStatus:         Boolean;
    FUNCTION  Alert
                (alertTemplatePtr:  AlertTemplate;
                 filterProcPtr:     WordProcPtr): Integer;
    FUNCTION  CautionAlert
                (alertTemplatePtr:  AlertTemplate;
                 filterProcPtr:     WordProcPtr): Integer;
    PROCEDURE CloseDialog
                (theDialogPtr:      DialogPtr);
    FUNCTION  DefaultFilter
                (theDialogPtr:      DialogPtr;
                 theEventPtr:       EventRecord;
                 itemHitPtr:        IntPtr): Boolean;
    FUNCTION  DialogSelect
                (theEventPtr:       EventRecord;
                 VAR resultPtr:     WindowPtr;
                 VAR itemHitPtr:    Integer): Boolean;
    PROCEDURE DisableDItem
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer);
    PROCEDURE DlgCopy
                (theDialogPtr:      DialogPtr);
    PROCEDURE DlgCut
                (theDialogPtr:      DialogPtr);
    PROCEDURE DlgDelete
                (theDialogPtr:      DialogPtr);
    PROCEDURE DlgPaste
                (theDialogPtr:      DialogPtr);
    PROCEDURE DrawDialog
```

```
               (theDialogPtr:       DialogPtr);
PROCEDURE EnableDItem
               (theDialogPtr:       DialogPtr;
                itemID:             Integer);
PROCEDURE ErrorSound
               (soundProcPtr:       VoidProcPtr);
FUNCTION  FindDItem
               (theDialogPtr:       DialogPtr;
                thePoint:           Point): Integer;
FUNCTION  GetAlertStage:            Integer;
FUNCTION  GetControlDItem
               (theDialogPtr:       DialogPtr;
                itemID:             Integer): CtlRecHndl;
FUNCTION  GetDefButton
               (theDialogPtr:       DialogPtr): Integer;
PROCEDURE GetDItemBox
               (theDialogPtr:       DialogPtr;
                itemID:             Integer;
                itemBoxPtr:         Rect);
FUNCTION  GetDItemType
               (theDialogPtr:       DialogPtr;
                itemID:             Integer): Integer;
FUNCTION  GetDItemValue
               (theDialogPtr:       DialogPtr;
                itemID:             Integer): Integer;
FUNCTION  GetFirstDItem
               (theDialogPtr:       DialogPtr): Integer;
PROCEDURE GetIText
               (theDialogPtr:       DialogPtr;
                itemID:             Integer;
                VAR text:           Str255);
PROCEDURE GetNewDItem
               (theDialogPtr:       DialogPtr;
                itemTemplatePtr:    ItemTemplate);
FUNCTION  GetNewModalDialog
               (dialogTemplatePtr: DlgTempPtr): DialogPtr;
FUNCTION  GetNextDItem
               (theDialogPtr:       DialogPtr;
                itemID:             Integer): Integer;
PROCEDURE HideDItem
               (theDialogPtr:       DialogPtr;
                itemID:             Integer);
FUNCTION  IsDialogEvent
               (theEventPtr:        EventRecord): Boolean;
FUNCTION  ModalDialog
               (filterProcPtr:      WordProcPtr): Integer;
FUNCTION  ModalDialog2
               (filterProcPtr:      WordProcPtr): Longint;
PROCEDURE NewDItem
               (theDialogPtr:       DialogPtr;
                itemID:             Integer;
                itemRectPtr:        Rect;
                itemType:           Integer;
                itemDescr:          Ptr;
                itemValue:          Integer;
                itemFlag:           Integer;
                itemColorPtr:       Ptr);
FUNCTION  NewModalDialog
```

```
                (dBoundsRectPtr:    Rect;
                 dVisibleFlag:      Boolean;
                 dRefCon:           Longint): DialogPtr;
FUNCTION  NewModelessDialog
                (dBoundsRectPtr:    Rect;
                 dTitle:            Str255;
                 dBehindPtr:        DialogPtr;
                 dFlag:             Integer;
                 dRefCon:           Longint;
                 dFullSizePtr:      Rect): DialogPtr;
FUNCTION NoteAlert
                (alertTemplatePtr:  AlertTempPtr;
                 filterProcPtr:     WordProcPtr):   Integer;
PROCEDURE ParamText
                (param0:            Str255;
                 param1:            Str255;
                 param2:            Str255;
                 param3:            Str255);
PROCEDURE RemoveDItem
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer);
PROCEDURE ResetAlertStage;
PROCEDURE SelIText
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer;
                 startSel:          Integer;
                 endSel:            Integer);
PROCEDURE SelectIText
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer;
                 startSel:          Integer;
                 endSel:            Integer);
PROCEDURE SetDAFont
                (fontHandle:        FontHndl);
PROCEDURE SetDefButton
                (defButtonID:       Integer;
                 theDialogPtr:      DialogPtr);
PROCEDURE SetDItemBox
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer;
                 itemBoxPtr:        Rect);
PROCEDURE SetDItemType
                (itemType:          Integer;
                 theDialogPtr:      DialogPtr;
                 itemID:            Integer);
PROCEDURE SetDItemValue
                (itemValue:         Integer;
                 theDialogPtr:      DialogPtr;
                 itemID:            Integer);
PROCEDURE SetIText
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer;
                 theString:         Str255);
PROCEDURE ShowDItem
                (theDialogPtr:      DialogPtr;
                 itemID:            Integer);
FUNCTION  StopAlert
                (alertTemplatePtr:  AlertTempPtr;
                filterProcPtr:      WordProcPtr): Integer;
PROCEDURE UpdateDialog
               (theDialogPtr:       DialogPtr;
                updateRgnHandle:    RgnHandle);

IMPLEMENTATION
END.
```

Events

```
{********************************************
; File: Events.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Events;

INTERFACE
USES Types;

CONST   emDupStrtUpErr          = $0601;   {error - duplicate EMStartup Call }
        emResetErr              = $0602;   {error - can't reset error the Event Manager }
        emNotActErr             = $0603;   {error - event manager not active }
        emBadEvtCodeErr         = $0604;   {error - illegal event code }
        emBadBttnNoErr          = $0605;   {error - illegal button number }
        emQSiz2LrgErr           = $0606;   {error - queue size too large }
        emNoMemQueueErr         = $0607;   {error - not enough memory for queue }
        emBadEvtQErr            = $0681;   {error - fatal sys error - event queue damaged }
        emBadQHndlErr           = $0682;   {error - fatal sys error - queue handle damaged }
        nullEvt                 = $0000;   {Event Code - }
        mouseDownEvt            = $0001;   {Event Code - }
        mouseUpEvt              = $0002;   {Event Code - }
        keyDownEvt              = $0003;   {Event Code - }
        autoKeyEvt              = $0005;   {Event Code - }
        updateEvt               = $0006;   {Event Code - }
        activateEvt             = $0008;   {Event Code - }
        switchEvt               = $0009;   {Event Code - }
        deskAccEvt              = $000A;   {Event Code - }
        driverEvt               = $000B;   {Event Code - }
        app1Evt                 = $000C;   {Event Code - }
        app2Evt                 = $000D;   {Event Code - }
        app3Evt                 = $000E;   {Event Code - }
        app4Evt                 = $000F;   {Event Code - }
        mDownMask               = $0002;   {Event Masks - }
        mUpMask                 = $0004;   {Event Masks - }
        keyDownMask             = $0008;   {Event Masks - }
        autoKeyMask             = $0020;   {Event Masks - }
        updateMask              = $0040;   {Event Masks - }
        activeMask              = $0100;   {Event Masks - }
        switchMask              = $0200;   {Event Masks - }
        deskAccMask             = $0400;   {Event Masks - }
        driverMask              = $0800;   {Event Masks - }
        app1Mask                = $1000;   {Event Masks - }
        app2Mask                = $2000;   {Event Masks - }
        app3Mask                = $4000;   {Event Masks - }
        app4Mask                = $8000;   {Event Masks - }
        everyEvent              = $FFFF;   {Event Masks - }
        jcTickCount             = $00;     {Journal Code - TickCount call }
        jcGetMouse              = $01;     {Journal Code - GetMouse call }
        jcButton                    = $02;      {Journal Code - Button call }
        jcEvent                     = $04;      {Journal Code - GetNextEvent and EventAvail calls}
        activeFlag                  = $0001;    {Modifier Flags - set if window being activated }
        changeFlag                  = $0002; {Modifier Flags - set if active wind changed state }
        btn1State                   = $0040;    {Modifier Flags - set if button 1 up }
        btn0State                   = $0080;    {Modifier Flags - set if button 0 up }
        appleKey                    = $0100;    {Modifier Flags - set if Apple key down }
        shiftKey                    = $0200;    {Modifier Flags - set if shift key down }
        capsLock                    = $0400;    {Modifier Flags - set if caps lock key down}
        optionKey                   = $0800;    {Modifier Flags - set if option key down }
        controlKey                  = $1000;    {Modifier Flags - set if Control key down }
        keyPad                      = $2000;    {Modifier Flags - set if keypress from keypad

TYPE    EventRecordHndl             = ^EventRecordPtr;
        EventRecordPtr              = ^EventRecord;
        EventRecord                 =
               RECORD
                      CASE Integer OF
                             0:     (what: Integer;         { event code }
                                    message: Longint;       { event message }
                                    when:  Longint;         { ticks since startup }
                                    where:  Point;          { mouse location }
                                    modifiers: Integer;     { modifier flags }
                                    );
                             1:     (wmhat:                 Integer;
                                    wmMessage:              Longint;
                                    wmWhen:                 Longint;
                                    wmWhere:                Point;
                                    wmModifiers:            Integer;
                                    wmTaskData:             Longint; {TaskMaster return value.}
                                    wmTaskMask:             Longint; {TaskMaster feature mask.}
                                    wmLastClickTick:        Longint;
                                    wmClickCount:           Integer;
                                    wmTaskData2:            Longint;
                                    wmTaskData3:            Longint;
                                    wmTaskData4:            Longint;
                                    wmLastClickPt:          Point; );
                      END;

PROCEDURE EMBootInit;
PROCEDURE EMStartUp
              (dPageAddr:    Integer;
               queueSize:    Integer;
               xMinClamp:    Integer;
               xMaxClamp:    Integer;
               yMinClamp:    Integer;
               yMaxClamp:    Integer;
               userID:       Integer);
PROCEDURE EMShutDown;
FUNCTION  EMVersion:         Integer;
PROCEDURE EMReset;
FUNCTION  EMStatus:          Boolean;
FUNCTION  Button
              (buttonNum:    Integer): Boolean;
FUNCTION  DoWindows:         Integer;
FUNCTION  EventAvail
              (eventMask:    Integer;
               VAR eventPtr: EventRecord): Boolean;
```

```
PROCEDURE FakeMouse
              (changedFlag: Integer;
               modLatch:    Integer;
               xPos:        Integer;
               yPos:        Integer;
               ButtonStatus: Integer);
FUNCTION  FlushEvents
              (eventMask:   Integer;
               stopMask:    Integer): Integer;
FUNCTION  GetCaretTime:     Longint;
FUNCTION  GetDblTime:       Longint;
PROCEDURE GetMouse
              (VAR mouseLocPtr: Point);
FUNCTION  GetNextEvent
              (eventMask:   Integer;
               VAR eventPtr: EventRecord): Boolean;
FUNCTION  GetOSEvent
              (eventMask:   Integer;
               VAR eventPtr: EventRecord): Boolean;
FUNCTION  OSEventAvail
              (eventMask:   Integer;
               VAR eventPtr: EventRecord): Boolean;
FUNCTION  PostEvent
              (eventCode:   Integer;
               eventMsg:    Longint): Integer;
PROCEDURE SetEventMask
              (sysEventMask: Integer);
PROCEDURE SetSwitch;
FUNCTION  StillDown
              (buttonNum:   Integer): Boolean;
FUNCTION  WaitMouseUp
              (buttonNum:   Integer): Boolean;
FUNCTION  TickCount:        Longint;
FUNCTION  GetKeyTranslation: Integer;
PROCEDURE SetKeyTranslation
              (kTransID:    Integer);
PROCEDURE SetAutoKeyLimit
              (newLimit:    Integer);

IMPLEMENTATION
END.
```

Fonts

```
{********************************************
; File: Fonts.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Fonts;

INTERFACE
USES Types, QuickDraw;

CONST   fmDupStartUpErr        = $1B01;   {error - duplicate FMStartUp call }
        fmResetErr             = $1B02;   {error - can't reset the Font Manager }
        fmNotActiveErr         = $1B03;   {error - Font Manager not active }
        fmFamNotFndErr         = $1B04;   {error - family not found }
        fmFontNtFndErr         = $1B05;   {error - font not found }
        fmFontMemErr           = $1B06;   {error - font not in memory }
        fmSysFontErr           = $1B07;   {error - system font cannot be purgeable }
        fmBadFamNumErr         = $1B08;   {error - illegal family number }
        fmBadSizeErr           = $1B09;   {error - illegal size }
        fmBadNameErr           = $1B0A;   {error - illegal name length }
        fmMenuErr              = $1B0B;   {error - fix font menu never called }
        fmScaleSizeErr         = $1B0C;   {error - scaled size of font exeeds limits}
        newYork                = $0002;   {Family Number - }
        geneva                 = $0003;   {Family Number - }
        monaco                 = $0004;   {Family Number - }
        venice                 = $0005;   {Family Number - }
        london                 = $0006;   {Family Number - }
        athens                 = $0007;   {Family Number - }
        sanFran                = $0008;   {Family Number - }
        toronto                = $0009;   {Family Number - }
        cairo                  = $000B;   {Family Number - }
        losAngeles             = $000C;   {Family Number - }
        times                  = $0014;   {Family Number - }
        helvetica              = $0015;   {Family Number - }
        courier                = $0016;   {Family Number - }
        symbol                 = $0017;   {Family Number - }
        taliesin               = $0018;   {Family Number - }
        shaston                = $FFFE;   {Family Number - }
        baseOnlyBit            = $0020;   {FamSpecBits - }
        notBaseBit             = $0020;   {FamStatBits - }
        memOnlyBit             = $0001;   {FontSpecBits - }
        realOnlyBit            = $0002;   {FontSpecBits - }
        anyFamBit              = $0004;   {FontSpecBits - }
        anyStyleBit            = $0008;   {FontSpecBits - }
        anySizeBit             = $0010;   {FontSpecBits - }
        memBit                 = $0001;   {FontStatBits - }
        unrealBit              = $0002;   {FontStatBits - }
        apFamBit               = $0004;   {FontStatBits - }
        apVarBit               = $0008;   {FontStatBits - }
        purgeBit                        = $0010;   {FontStatBits - }
        notDiskBit                      = $0020;   {FontStatBits - }
        notFoundBit                     = $8000;   {FontStatBits - }
        dontScaleBit                    = $0001;   {Scale Word - }

   TYPE FontStatRecHndl                 = ^FontStatRecPtr;
        FontStatRecPtr                  = ^FontStatRec;
        FontStatRec                     =
              RECORD
                     resultID:      FontID;
                     resultStats:   Integer;
              END;

   PROCEDURE FMBootInit;
   PROCEDURE FMStartUp
              (userID:              Integer;
               dPageAddr:           Integer);
   PROCEDURE FMShutDown;
   FUNCTION  FMVersion:             Integer;
   PROCEDURE FMReset;
   FUNCTION  FMStatus:              Boolean;
   PROCEDURE AddFamily
              (famNum:              Integer;
               famName:             Str255);
   PROCEDURE AddFontVar
              (fontHandle:          FontHndl;
               newSpecs:            Integer);
   FUNCTION  ChooseFont
              (currentID:           FontID;
               famSpecs:            Integer): FontID;
   FUNCTION  CountFamilies
              (famSpecs:            Integer): Integer;
   FUNCTION  CountFonts
              (desiredID:           FontID;
               fontSpecs:           Integer): Integer;
   FUNCTION  FamNum2ItemID
              (famNum:              Integer): Integer;
   FUNCTION  FindFamily
              (famSpecs:            Integer;
               positionNum:         Integer;
               famName:             Str255): Integer;
   PROCEDURE FindFontStats
              (desiredID:           FontID;
              fontSpecs:            Integer;
              positionNum:          Integer;
              VAR resultPtr:        FontStatRec);
   PROCEDURE FixFontMenu
              (menuID:              Integer;
               startingID:          Integer;
               famSpecs:            Integer);
   FUNCTION  FMGetCurFID:           FontID;
   FUNCTION  FMGetSysFID:           FontID;
   PROCEDURE FMSetSysFont
              (newFontID:           FontID);
   FUNCTION  GetFamInfo
              (famNum:              Integer;
               famName:             Str255): Integer;
   FUNCTION  GetFamNum
```

```
                (famName:                Str255): Integer;
PROCEDURE InstallFont
                (desiredID:              FontID;
                 scaleWord:              Integer);
PROCEDURE InstallWithStats
                (desiredID:              FontID;
                 scaleWord:              Integer;
                 resultPtr:              Ptr);
FUNCTION  ItemID2FamNum
                (itemID:                 Integer): Integer;
PROCEDURE LoadFont
                (desiredID:              FontID;
                 fontSpecs:              Integer;
                 positionNum:            Integer;
                 VAR resultPtr:          FontStatRec);
PROCEDURE LoadSysFont;
PROCEDURE SetPurgeStat
                (theFontID:              FontID;
                 purgeStat:              Integer);

IMPLEMENTATION
END.
```

GSOS

```
{*********************************************
; File: GSOS.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
*********************************************}

UNIT GSOS;

INTERFACE
USES Types;

CONST  readEnable              = $0001;   {access - read enable bit: CreateRec,
                                          OpenRec access and requestAccess fields }
       writeEnable             = $0002;   {access - write enable bit: CreateRec,
                                          OpenRec access and requestAccess fields }
       fileInvisible           = $0004;   {access - Invisible bit }
       backupNeeded            = $0020;   {access - backup needed bit: CreateRec,
                                          OpenRec access field. (Must be 0 in
                                          requestAccess field ) }
       renameEnable            = $0040;   {access - rename enable bit: CreateRec,
                                          OpenRec access and requestAccess fields }
       destroyEnable           = $0080;   {access - destroy enable bit: CreateRec,
                                          OpenRec access and requestAccess fields }
       startPlus               = $0000;   {base - > setMark = displacement }
       eofMinus                = $0001;   {base - > setMark = eof - displacement }
       markPlus                = $0002;   {base - > setMark = mark + displacement }
```

```
markMinus               = $0003;   {base - > setMark = mark - displacement }
cacheOff                = $0000;'  {cachePriority - do not cache blocks
                                   invloved in this read }
cacheOn                 = $0001;   {cachePriority - cache blocks invloved in
                                   this read if possible }
badSystemCall           = $0001;   {error - bad system call number }
invalidPcount           = $0004;   {error - invalid parameter count }
gsosActive              = $07;     {error - GS/OS already active }

{$IFC UNDEFINED devNotFound }      {error - device not found }
devNotFound             = $10;
                                   {$SETC devNotFound := 0}
                                   {$ENDC}

invalidDevNum           = $11;     {error - invalid device number }
drvrBadReq              = $20;     {error - bad request or command }
drvrBadCode             = $0021;   {error - bad control or status code }
drvrBadParm             = $0022;   {error - bad call parameter }
drvrNotOpen             = $0023;   {error - character device not open }
drvrPriorOpen           = $0024;   {error - character device already open }
irqTableFull            = $0025;   {error - interrupt table full }
drvrNoResrc             = $0026;   {error - resources not available }
drvrIOError             = $0027;   {error - I/O error }
drvrNoDevice            = $0028;   {error - device not connected }
drvrBusy                = $0029;   {error - call aborted, driver is busy }
drvrWrtProt             = $002B;   {error - device is write protected }
drvrBadCount            = $002C;   {error - invalid byte count }
drvrBadBlock            = $002D;   {error - invalid block address }
drvrDiskSwitch          = $002E;   {error - disk has been switched }
drvrOffLine             = $002F;   {error - device off line/ no media present}
badPathSyntax           = $0040;   {error - invalid pathname syntax }
invalidRefNum           = $0043;   {error - invalid reference number }

{$IFC UNDEFINED pathNotFound }     {error - subdirectory does not exist }
pathNotFound            = $44;
{$SETC pathNotFound := 0}
{$ENDC}

volNotFound             = $0045;   {error - volume not found }

{$IFC UNDEFINED fileNotFound }     {error - file not found }
fileNotFound            = $0046;
{$SETC fileNotFound := 0}
{$ENDC}

dupPathname             = $0047;   {error - create or rename with existing
                                   name }
volumeFull              = $0048;   {error - volume full error }
volDirFull              = $0049;   {error - volume directory full }
badFileFormat           = $004A;   {error - version error (incompatible file
                                   format) }

{$IFC UNDEFINED badStoreType }     {error - unsupported (or incorrect) storage
                                   type }
badStoreType            = $004B;
                                   {$SETC badStoreType := 0}
                                   {$ENDC}
```

```
{$IFC UNDEFINED eofEncountered }        {error - end-of-file encountered }
eofEncountered              = $004C;
                                        {$SETC eofEncountered := 0}
                                        {$ENDC}

outOfRange                  = $004D;    {error - position out of range }
invalidAccess               = $004E;    {error - access not allowed }
buffTooSmall                = $004F;    {error - buffer too small }
fileBusy                    = $0050;    {error - file is already open }
dirError                    = $0051;    {error - directory error }
unknownVol                  = $0052;    {error - unknown volume type }

{$IFC UNDEFINED paramRangeErr }         {error - parameter out of range }
paramRangeErr               = $0053;
                                        {$SETC paramRangeErr := 0}
                                        {$ENDC}

outOfMem                    = $0054;    {error - out of memory }
dupVolume                   = $0057;    {error - duplicate volume name }
notBlockDev                 = $0058;    {error - not a block device }

{$IFC UNDEFINED invalidLevel }          {error - specifield level outside legal
                                        range }
invalidLevel                = $0059;
                                        {$SETC invalidLevel := 0}
                                        {$ENDC}

damagedBitMap               = $005A;    {error - block number too large }
badPathNames                = $005B;    {error - invalid pathnames for ChangePath }
notSystemFile               = $005C;    {error - not an executable file }
osUnsupported               = $005D;    {error - Operating System not supported }

{$IFC UNDEFINED stackOverflow }         {error - too many applications on stack }
stackOverflow               = $005F;
                                        {$SETC stackOverflow := 0}
                                        {$ENDC}

dataUnavail                 = $0060;    {error - Data unavailable }
endOfDir                    = $0061;    {error - end of directory has been reached
                                        }
invalidClass                = $0062;    {error - invalid FST call class }
resForkNotFound             = $0063;    {error - file does not contain required
                                        resource }
invalidFSTID                = $0064;    {error - error - FST ID is invalid }
proDOSFSID                  = $0001;    {fileSysID - ProDOS/SOS }
dos33FSID                   = $0002;    {fileSysID - DOS 3.3 }
dos32FSID                   = $0003;    {fileSysID - DOS 3.2 }
dos31FSID                   = $0003;    {fileSysID - DOS 3.1 }
appleIIPascalFSID           = $0004;    {fileSysID - Apple II Pascal }
mfsFSID                     = $0005;    {fileSysID - Macintosh (flat file system) }
hfsFSID                     = $0006;    {fileSysID - Macintosh (hierarchical file system)}
lisaFSID                    = $0007;    {fileSysID - Lisa file system }
appleCPMFSID                = $0008;    {fileSysID - Apple CP/M }
charFSTFSID                 = $0009;    {fileSysID - Character FST }
msDOSFSID                   = $000A;    {fileSysID - MS/DOS }
highSierraFSID              = $000B;    {fileSysID - High Sierra }
iso9660FSID                 = $000C;    {fileSysID - ISO 9660 }
appleShareFSID              = $000D;    {fileSysID - ISO 9660 }
```

```
      characterFST             = $4000;   {FSTInfo.attributes - character FST }
      ucFST                    = $8000;   {FSTInfo.attributes - SCM should upper case
                                          pathnames before passing them to the FST }
      onStack                  = $8000;   {QuitRec.flags - place state information
                                          about quitting program on the quit return stack }
      restartable              = $4000;   {QuitRec.flags - the quitting program is
                                          capable of being restarted from its dormant
                                          memory }
      seedling                 = $0001;   {storageType - standard file with seedling
                                          structure }
      standardFile             = $0001;   {storageType - standard file type (no
                                          resource fork) }
      sapling                  = $0002;   {storageType - standard file with sapling
                                          structure }
      tree                     = $0003;   {storageType - standard file with tree
                                          structure }
      pascalRegion             = $0004;   {storageType - UCSD Pascal region on a
                                          partitioned disk }
      extendedFile             = $0005;   {storageType - extended file type (with
                                          resource fork) }
      directoryFile            = $000D;   {storageType - volume directory or
                                          subdirectory  file }
      minorRelNumMask          = $00FF;   {version - minor release number }
      majorRelNumMask          = $7F00;   {version - major release number }
      finalRelNumMask          = $8000;   {version - final release number }
      isFileExtended           = $8000;   {GetDirEntryGS - }

TYPE  GSString255Hndl          = ^GSString255Ptr;
      GSString255Ptr           = ^GSString255;
      GSString255              =
            RECORD
                  length:      Integer; {Number of Chars in text field}
                  text:        PACKED ARRAY [1..255] OF CHAR;
            END;

      GSString255HndlPtr       = ^GSString255Hndl;

      GSString32Hndl           = ^GSString32Ptr;
      GSString32Ptr            = ^GSString32;
      GSString32               =
            RECORD
                  length:      Integer; {Number of characters in text field }
                  text:        PACKED ARRAY [1..32] OF CHAR;
            END;

      ResultBuf255Hndl         = ^ResultBuf255Ptr;
      ResultBuf255Ptr          = ^ResultBuf255;
      ResultBuf255             =
            RECORD
                  bufSize:     Integer;
                  bufString:   GSString255;
            END;

      ResultBuf255HndlPtr      = ^ResultBuf255Hndl;

      ResultBuf32Hndl          = ^ResultBuf32Ptr;
      ResultBuf32Ptr           = ^ResultBuf32;
```

```
ResultBuf32                    =
        RECORD
                bufSize:             Integer;
                bufString:           GSString32;
        END;

ChangePathRecPtrGS             = ^ChangePathRecGS;
ChangePathRecGS                =
        RECORD
                pCount:              Integer;
                pathname:            GSString255Ptr;
                newPathname:         GSString255Ptr;
        END;

CreateRecPtrGS                 = ^CreateRecGS;
CreateRecGS                    =
        RECORD
                pCount:         Integer;
                pathname:            GSString255Ptr;
                access:         Integer;
                fileType:            Integer;
                auxType:             Longint;
                storageType:         Integer;
                eof:                 Longint;
                resourceEOF:         Longint;
        END;

DAccessRecPtrGS                = ^DAccessRecGS;
DAccessRecGS                   =
        RECORD
                pCount:              Integer;
                devNum:              Integer;
                code:                Integer;
                list:                Ptr;
                requestCount:        Longint;
                transferCount: Longint;
        END;

DevNumRecPtrGS                 = ^DevNumRecGS;
DevNumRecGS                    =
        RECORD
                pCount:              Integer;
                devName:             GSString255Ptr;
                devNum:              Integer;
        END;

DInfoRecPtrGS                  = ^DInfoRecGS;
DInfoRecGS                     =
        RECORD
                pCount:              Integer; { minimum = 2 }
                devNum:              Integer;
                devName:             GSString32Ptr;
                characteristics:     Integer;
                totalBlocks:         Longint;
                slotNum:             Integer;
                unitNum:             Integer;
                version:             Integer;
                deviceID:            Integer;
                headLink:              Integer;
                forwardLink:           Integer;
                extendedDIBptr:        Longint;
            END;

     DIORecPtrGS              = ^DIORecGS;
     DIORecGS                 =
            RECORD
                pCount:                Integer;
                devNum:                Integer;
                buffer:                Ptr;
                requestCount:          Longint;
                startingBlock:         Longint;
                blockSize:             Integer;
                transferCount:         Longint;
            END;

     DirEntryRecPtrGS         = ^DirEntryRecGS;
     DirEntryRecGS            =
            RECORD
                pCount:                        Integer;
                refNum:                        Integer;
                flags:                         Integer;
                base:                          Integer;
                displacement:                  Integer;
                name:                          Ptr;
                entryNum:                      Integer;
                fileType:                      Integer;
                eof:                           Longint;
                blockCount:                    Longint;
                createDateTime:                TimeRec;
                modDateTime:                   TimeRec;
                access:                        Integer;
                auxType:                       Longint;
                fileSysID:                     Integer;
                optionList:                    ResultBuf255Ptr;
                resourceEOF:                   Longint;
                resourceBlocks:                Longint;
            END;

     ExpandPathRecPtrGS       = ^ExpandPathRecGS;
     ExpandPathRecGS          =
            RECORD
                pCount:                        Integer;
                inputPath:                     GSString255Ptr;
                outputPath:                    ResultBuf255Ptr;
                flags: Integer;
            END;
     FileInfoRecPtrGS         = ^FileInfoRecGS;
     FileInfoRecGS            =
            RECORD
                pCount:                        Integer;
                pathname:                      GSString255Ptr;
                access:                        Integer;
                fileType:                      Integer;
                auxType:                       Longint;
                storageType:                   Integer; {must be 0 for SetFileInfo}
                createDateTime:                TimeRec;
            modDateTime:                     TimeRec;
            optionList:                      Longint;
            eof:                             Longint;
            blocksUsed:                      Longint; {must be 0 for SetFileInfo}
            resourceEOF:                     Longint; {must be 0 for SetFileInfo}
            resourceBlocks:                  Longint; {must be 0 for SetFileInfo}
        END;

    FormatRecPtrGS              = ^FormatRecGS;
    FormatRecGS                 =
        RECORD
            pCount:                          Integer;
            devName:                         GSString32Ptr; {device name pointer}
            volName:                         GSString32Ptr; {volume name pointer}
            fileSysID:                       Integer; { file system ID }
            reqFileSysID:                    Integer; { in; }
        END;

    FSTInfoRecPtrGS             = ^FSTInfoRecGS;
    FSTInfoRecGS                =
        RECORD
            pCount:                          Integer;
            fstNum:                          Integer;
            fileSysID:                       Integer;
            fstName:                         ResultBuf255Ptr;
            version:                         Integer;
            attributes:                      Integer;
            blockSize:                       Integer;
            maxVolSize:                      Longint;
            maxFileSize:                     Longint;
        END;

    InterruptRecPtrGS           = ^InterruptRecGS;
    InterruptRecGS              =
        RECORD
            pCount:Integer;
            intNum:Integer;
            vrn:            Integer; { used only by BindInt }
            intCode:        Longint; { used only by BindInt }
        END;

    IORecPtrGS                  = ^IORecGS;
    IORecGS                     =
        RECORD
            pCount:                Integer;
            refNum:                Integer;
            dataBuffer:            Ptr;
            requestCount:          Longint;
            transferCount:         Longint;
            cachePriority:         Integer;
        END;

    LevelRecPtrGS               = ^LevelRecGS;
    LevelRecGS                  =
        RECORD
            pCount:                Integer;
            level:                 Integer;
        END;
```

```
NameRecPtrGS              = ^NameRecGS;
NameRecGS                 =
        RECORD
                pCount:              Integer;
                pathname:            GSString255Ptr; { full pathname or a
                                                  filename depending on call }
        END;

GetNameRecPtrGS           = ^GetNameRecGS;
GetNameRecGS              =
        RECORD
                pCount:              Integer;
                dataBuffer:          ResultBuf255Ptr; { full pathname or a
                                                  filename depending on call }
        END;

NewlineRecPtrGS           = ^NewlineRecGS;
NewlineRecGS              =
        RECORD
                pCount:              Integer;
                refNum:              Integer;
                enableMask:          Integer;
                numChars:            Integer;
                newlineTable:        Ptr;
        END;

OpenRecPtrGS              = ^OpenRecGS;
OpenRecGS                 =
        RECORD
                pCount:              Integer;
                refNum:              Integer;
                pathname:            GSString255Ptr;
                requestAccess:       Integer;
                resourceNumber:      Integer; {For extended files: dataFork/
                                                  resourceFork}
                access:              Integer; {Value of file's access attribute}
                fileType:            Integer; {Value of file's fileType attribute}
                auxType:             Longint;
                storageType:         Integer;
                createDateTime:      TimeRec;
                modDateTime:         TimeRec;
                optionList:          IntPtr;
                eof:                 Longint;
                blocksUsed:          Longint;
                resourceEOF:         Longint;
                resourceBlocks:      Longint;
        END;

OSShutdownRecPtrGS        = ^OSShutdownRecGS;
OSShutdownRecGS           =
        RECORD
                pCount:              Integer; { in }
                shutdownFlag:        Integer; { in }
        END;
```

```
PositionRecPtrGS          = ^PositionRecGS;
PositionRecGS             =
        RECORD
                pCount:       Integer;
                refNum:       Integer;
                position:     Longint;
        END;

EOFRecPtrGS               = ^EOFRecGS;
EOFRecGS                  =
        RECORD
                pCount:       Integer;
                refNum:       Integer;
                eof:          Longint;
        END;

PrefixRecPtrGS            = ^PrefixRecGS;
PrefixRecGS               =
        RECORD
                pCount:Integer;
                prefixNum:    Integer;
                CASE Integer OF
                        0:    (getPrefix: ResultBuf255Ptr; );
                        1:    (setPrefix: GSString255Ptr; );
        END;

QuitRecPtrGS              = ^QuitRecGS;
QuitRecGS                 =
        RECORD
                pCount:Integer;
                pathname:     GSString255Ptr; {pathname of next app to run}
                flags:        Integer;
        END;

RefnumRecPtrGS            = ^RefNumRecGS;
RefNumRecGS               =
        RECORD
                pCount:Integer;
                refNum:Integer;
        END;

SessionStatusRecPtrGS     = ^SessionStatusRecGS;
SessionStatusRecGS     .  =
        RECORD
                pCount:Integer; { in: min = 1 }
                status:Integer; { out: }
        END;

SetPositionRecPtrGS       = ^SetPositionRecGS;
SetPositionRecGS          =
        RECORD
                pCount:       Integer;
                refNum:       Integer;
                base:         Integer;
                displacement: Longint;
        END;

SysPrefsRecPtrGS          = ^SysPrefsRecGS;
```

```
    SysPrefsRecGS                    =
            RECORD
                    pCount:         Integer;
                    preferences:    Integer;
            END;

    VersionRecPtrGS                  = ^VersionRecGS;
    VersionRecGS                     =
            RECORD
                    pCount:         Integer;
                    version:        Integer;
            END;

    VolumeRecPtrGS                   = ^VolumeRecGS;
    VolumeRecGS                      =
            RECORD
                    pCount:         Integer;
                    devName:        GSString32Ptr;
                    volName:        ResultBuf255Ptr;
                    totalBlocks:    Longint;
                    freeBlocks:     Longint;
                    fileSysID:      Integer;
                    blockSize:      Integer;
            END;

PROCEDURE BeginSessionGS
                (VAR pblockPtr:     SessionStatusRecGS);
PROCEDURE BindIntGS
                (VAR pblockPtr:     InterruptRecGS);
PROCEDURE ChangePathGS
                (VAR pblockPtr:     ChangePathRecGS);
PROCEDURE ClearBackupBitGS
                (VAR pblockPtr:     NameRecGS);
PROCEDURE CloseGS
                (VAR pblockPtr:     RefNumRecGS);
PROCEDURE CreateGS
                (VAR pblockPtr:     CreateRecGS);
PROCEDURE DControlGS
                (VAR pblockPtr:     DAccessRecGS);
PROCEDURE DestroyGS
                (VAR pblockPtr:     NameRecGS);
PROCEDURE DInfoGS
                (VAR pblockPtr:     DInfoRecGS);
PROCEDURE DReadGS
                (VAR pblockPtr:     DIORecGS);
PROCEDURE DStatusGS
                (VAR pblockPtr:     DAccessRecGS);
PROCEDURE DWriteGS
                (VAR pblockPtr:     DIORecGS);
PROCEDURE EndSessionGS
                (VAR pblockPtr:     SessionStatusRecGS);
PROCEDURE EraseDiskGS
                (VAR pblockPtr:     FormatRecGS);
PROCEDURE ExpandPathGS
                (VAR pblockPtr:     ExpandPathRecGS);
PROCEDURE FlushGS
                (VAR pblockPtr:     RefNumRecGS);
PROCEDURE FormatGS
```

```
                (VAR pblockPtr:         FormatRecGS);
PROCEDURE GetBootVolGS
                (VAR pblockPtr:         NameRecGS);
PROCEDURE GetDevNumberGS
                (VAR pblockPtr:         DevNumRecGS);
PROCEDURE GetDirEntryGS
                (VAR pblockPtr:         DirEntryRecGS);
PROCEDURE GetEOFGS
                (VAR pblockPtr:         EOFRecGS);
PROCEDURE GetFileInfoGS
                (VAR pblockPtr:         FileInfoRecGS);
PROCEDURE GetFSTInfoGS
                (VAR pblockPtr:         FSTInfoRecGS);
PROCEDURE GetLevelGS
                (VAR pblockPtr:         LevelRecGS);
PROCEDURE GetMarkGS
                (VAR pblockPtr:         PositionRecGS);
PROCEDURE GetNameGS
                (VAR pblockPtr:         GetNameRecGS);
PROCEDURE GetPrefixGS
                (VAR pblockPtr:         PrefixRecGS);
PROCEDURE GetVersionGS
                (VAR pblockPtr:         VersionRecGS);
PROCEDURE GetSysPrefsGS
                (VAR pblockPtr:         SysPrefsRecGS);
PROCEDURE NewlineGS
                (VAR pblockPtr:         NewlineRecGS);
PROCEDURE NullGS
                (VAR pblockPtr:         IntPtr);
PROCEDURE OpenGS
                (VAR pblockPtr:         OpenRecGS);
PROCEDURE QuitGS
                (VAR pblockPtr:         QuitRecGS);
PROCEDURE ReadGS
                (VAR pblockPtr:         IORecGS);
PROCEDURE ResetCacheGS
                (VAR pblockPtr:         IntPtr);
PROCEDURE SessionStatusGS
                (VAR pblockPtr:         SessionStatusRecGS);
PROCEDURE SetEOFGS
                (VAR pblockPtr:         SetPositionRecGS);
PROCEDURE SetFileInfoGS
                (VAR pblockPtr:         FileInfoRecGS);
PROCEDURE SetLevelGS
                (VAR pblockPtr:         LevelRecGS);
PROCEDURE SetMarkGS
                (VAR pblockPtr:         SetPositionRecGS);
PROCEDURE SetPrefixGS
                (VAR pblockPtr:         PrefixRecGS);
PROCEDURE SetSysPrefsGS
                (VAR pblockPtr:         SysPrefsRecGS);
PROCEDURE UnbindIntGS
                (VAR pblockPtr:         InterruptRecGS);
PROCEDURE VolumeGS
                (VAR pblockPtr:         VolumeRecGS);
PROCEDURE WriteGS
                (VAR pblockPtr:         IORecGS);
```

```
PROCEDURE OSShutdownGS
              (VAR pblockPtr:       OSShutdownRecGS);
PROCEDURE FSTSpecific
              (VAR pBlockPtr:       Ptr);

IMPLEMENTATION
END.
```

IntMath

```
{********************************************
; File: IntMath.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT IntMath;

INTERFACE
USES Types;

CONST  imBadInptParam        = $0B01;      {error - bad input parameter }
       imIllegalChar         = $0B02;      {error - Illegal character in string}
       imOverflow            = $0B03;      {error - integer or long integer
                                            overflow }
       imStrOverflow         = $0B04;      {error - string overflow }
       minLongint            = $80000000;  {Limit - minimum negative signed
                                            long integer }
       minFrac               = $80000000;  {Limit - pinned value for negative
                                            Frac overflow }
       minFixed              = $80000000;  {Limit - pinned value for negative
                                            Fixed overflow }
       minInt                = $8000;      {Limit - Minimum negative signed
                                            integer }
       maxInt                = $7FFF;      {Limit - Maximum positive signed
                                            integer }
       maxUInt               = $FFFF;      {Limit - Maximum positive unsigned
                                            integer }
       maxLongint            = $7FFFFFFF;  {Limit - maximum positive signed
                                            Longint }
       maxFrac               = $7FFFFFFF;  {Limit - pinned value for positive
                                            Frac overflow }
       maxFixed              = $7FFFFFFF;  {Limit - pinned value for positive
                                            Fixed overflow }
       maxULong              = $FFFFFFFF;  {Limit - maximum unsigned Long }
       unsignedFlag          = $0000;      {SignedFlag - }
       signedFlag            = $0001;      {SignedFlag - }
```

```
TYPE    IntDivRecPtr                = ^IntDivRec;
        IntDivRec                   =
                RECORD
                        quotient:       Integer; { quotient from SDivide }
                        remainder:      Integer; { remainder from SDivide }
                END;

        LongDivRecPtr               = ^LongDivRec;
        LongDivRec                  =
                RECORD
                        quotient:       Longint; { Quotient from LongDiv }
                        remainder:      Longint; { remainder from LongDiv }
                END;

        DivRecPtr                   = ^DivRec; (* for backward compatability *)
        DivRec                      = LongDivRec;
        LongMulRecPtr               = ^LongMulRec;
        LongMulRec                  =
                RECORD
                        lsResult:       Longint; { low 2 words of product }
                        msResult:       Longint; { High 2 words of product }
                END;

        WordDivRecPtr               = ^WordDivRec;
        WordDivRec                  =
                RECORD
                        quotient:       Integer; { Quotient from UDivide }
                        remainder:      Integer; { remainder from UDivide }
                END;

PROCEDURE IMBootInit;
PROCEDURE IMStartUp;
PROCEDURE IMShutDown;
FUNCTION  IMVersion: Integer;
PROCEDURE IMReset;
FUNCTION  IMStatus: Boolean;
FUNCTION  Dec2Int
                (strPtr:                Ptr;
                 strLength:             Integer;
                 signedFlag:            Boolean): Integer;
FUNCTION  Dec2Long
                (strPtr:                Ptr;
                 strLength:             Integer;
                 signedFlag:            Boolean): Longint;
FUNCTION  Fix2Frac
                (fixedValue:            Fixed): Frac;
FUNCTION  Fix2Long
                (fixedValue:            Fixed): Longint;
PROCEDURE Fix2X
                (fixedValue:            Fixed;
                 VAR extendPtr:         Extended);
FUNCTION  FixATan2
                (input1:                Longint;
                 input2:                Longint): Fixed;
FUNCTION  FixDiv
                (dividend:              Longint;
                 divisor:               Longint): Fixed;
FUNCTION  FixMul
```

```
               (multiplicand:          Fixed;
                multiplier:            Fixed): Fixed;
FUNCTION  FixRatio
               (numerator:             Integer;
                denominator:           Integer): Fixed;
FUNCTION  FixRound
               (fixedValue:            Fixed): Integer;
FUNCTION  Frac2Fix
               (fracValue:             Frac): Fixed;
PROCEDURE Frac2X
               (fracValue:             Frac;
                VAR extendPtr:         Extended);
FUNCTION  FracCos
               (angle:                 Fixed): Frac;
FUNCTION  FracDiv
               (dividend:              Longint;
                divisor:               Longint): Frac;
FUNCTION  FracMul
               (multiplicand:          Frac;
                multiplier:            Frac): Frac;
FUNCTION  FracSin
               (angle:                 Fixed): Frac;
FUNCTION  FracSqrt
               (fracValue:             Frac): Frac;
FUNCTION  Hex2Int
               (strPtr:                Ptr;
                strLength:             Integer): Integer;
FUNCTION  Hex2Long
               (strPtr:                Ptr;
                strLength:             Integer): Longint;
FUNCTION  HexIt
               (intValue:              Integer): Longint;
FUNCTION  HiWord
               (longValue:             Longint): Integer;
PROCEDURE Int2Dec
               (wordValue:             Integer;
                strPtr:                Ptr;
                strLength:             Integer;
                signedFlag:            Boolean);
PROCEDURE Int2Hex
               (intValue:              Integer;
                strPtr:                Ptr;
                strLength:             Integer);
PROCEDURE Long2Dec
               (longValue:             Longint;
                strPtr:                Ptr;
                strLength:             Integer;
                signedFlag:            Boolean);
FUNCTION  Long2Fix
               (longValue:             Longint): Fixed;
PROCEDURE Long2Hex
               (longValue:             Longint;
                strPtr:                Ptr;
                strLength:             Integer);
FUNCTION  LongDivide
               (dividend:              Longint;
                divisor:               Longint): LongDivRec;
FUNCTION  LongMul
```

```
            (multiplicand:          Longint;
             multiplier:            Longint): LongMulRec;
FUNCTION  LoWord
            (longValue:             Longint): Integer;
FUNCTION  Multiply
            (multiplicand:          Integer;
             multiplier:            Integer): Longint;
FUNCTION  SDivide
            (dividend:              Integer;
             divisor:               Integer): IntDivRec;
FUNCTION  UDivide
            (dividend:              Integer;
             divisor:               Integer): WordDivRec;
FUNCTION  X2Fix
            (extendPtr:             ExtendedPtr): Longint;
FUNCTION  X2Frac
            (extendPtr:             ExtendedPtr): Longint;

IMPLEMENTATION
END.
```

LineEdit

```
{********************************************
; File: LineEdit.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT LineEdit;

INTERFACE
USES Types, QuickDraw, Events;

CONST  leDupStrtUpErr          = $1401;   {error - duplicate LEStartup call }
       leResetErr              = $1402;   {error - can't reset Line Edit }
       leNotActiveErr          = $1403;   {error - Line Edit not active }
       leScrapErr              = $1404;   {error - desk scrap too big to copy }
       leJustLeft              = $0000;   {Justification - }
       leJustCenter            = $0001;   {Justification - }
       leJustFill              = $0002;   {Justification - }
       leJustRight             = $FFFF;   {Justification - }

TYPE   LERecHndl               = ^LERecPtr;
       LERecPtr                = ^LERec;
       LERec                   =
              RECORD
                     leLineHandle:       Handle;
                     leLength:           Integer;
                     leMaxLength:        Integer;
                     leDestRect:         Rect;
```

```
                    leViewRect:          Rect;
                    lePort:              GrafPortPtr;
                    leLineHite:          Integer;
                    leBaseHite:          Integer;
                    leSelStart:          Integer;
                    leSelEnd:            Integer;
                    leActFlg:            Integer;
                    leCarAct:            Integer;
                    leCarOn:             Integer;
                    leCarTime:           Longint;
                    leHiliteHook:        VoidProcPtr;
                    leCaretHook:         VoidProcPtr;
                    leJust:              Integer;
                    lePWChar:            Integer;
               END;

PROCEDURE LEBootInit;
PROCEDURE LEStartUp
               (userID:        Integer;
                dPageAddr:     Integer);
PROCEDURE LEShutDown;
FUNCTION  LEVersion:     Integer;
PROCEDURE LEReset;
FUNCTION  LEStatus:            Boolean;
PROCEDURE LEActivate
               (leRecHandle:   LERecHndl);
PROCEDURE LEClick
               (eventPtr:      EventRecord;
                leRecHandle:   LERecHndl);
PROCEDURE LECopy
               (leRecHandle:   LERecHndl);
PROCEDURE LECut
               (leRecHandle:   LERecHndl);
PROCEDURE LEDeactivate
               (leRecHandle:   LERecHndl);
PROCEDURE LEDelete
               (leRecHandle:   LERecHndl);
PROCEDURE LEDispose
               (leRecHandle:   LERecHndl);
PROCEDURE LEFromScrap;
FUNCTION  LEGetScrapLen:       Integer;
FUNCTION  LEGetTextHand
               (leRecHandle:   LERecHndl): Handle;
FUNCTION  LEGetTextLen
               (leRecHandle:   LERecHndl): Integer;
PROCEDURE LEIdle
               (leRecHandle:   LERecHndl);
PROCEDURE LEInsert
               (textPtr:       Ptr;
                textLength:    Integer;
                leRecHandle:   LERecHndl);
PROCEDURE LEKey
               (theKey:        CHAR;
                modifiers:     Integer;
                leRecHandle:   LERecHndl);
```

```
FUNCTION   LENew
                (destRectPtr: Rect;
                 viewRectPtr: Rect;
                 maxTextLen:  Integer): LERecHndl;
PROCEDURE LEPaste
                (leRecHandle: LERecHndl);
FUNCTION  LEScrapHandle:     Handle;
PROCEDURE LESetCaret
                (caretProcPtr: VoidProcPtr;
                 leRecHandle: LERecHndl);
PROCEDURE LESetHilite
                (hiliteProcPtr: VoidProcPtr;
                 leRecHandle: LERecHndl);
PROCEDURE LESetJust
                (just:        Integer;
                 leRecHandle: LERecHndl);
PROCEDURE LESetScrapLen
                (newLength:   Integer);
PROCEDURE LESetSelect
                (selStart:    Integer;
                 selEnd:      Integer;
                 leRecHandle: LERecHndl);
PROCEDURE LESetText
                (textPtr:     Ptr;
                 textLength:  Integer;
                 leRecHandle: LERecHndl);
PROCEDURE LETextBox
                (textPtr:     Ptr;
                 textLength:  Integer;
                 rectPtr:     Rect;
                 just:   Integer);
PROCEDURE LETextBox2
                (textPtr:     Ptr;
                 textLength:  Integer;
                 rectPtr:     Rect;
                 just:   Integer);
PROCEDURE LEToScrap;
PROCEDURE LEUpdate
                (leRecHandle: LERecHndl);
FUNCTION  GetLEDefProc:      Ptr;

IMPLEMENTATION
END.
```

Lists

```
{********************************************
; File: Lists.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Lists;

INTERFACE
USES Types, QuickDraw, Events, Controls;

CONST  cString          = $0001;    {ListType bit mask - null terminated string type }
       LIST_STRG        = $0001;    {ListType bit mask - null terminated string type }
       selectOnlyOne    = $0002;    {ListType bit mask - only one selection allowed }
       LIST_SELECT      = $0002;    {ListType bit mask - single selection only }
       memDisabled      = $40;      {memFlag - Sets member flag to disabled }
       memSelected      = $80;      {memFlag - Sets member flag to selected }

TYPE   LColorTableHndl    = ^LColorTablePtr;
       LColorTablePtr     = ^LColorTable;
       LColorTable        =
            RECORD
                listFrameClr:   Integer;  { Frame color }
                listNorTextClr: Integer;  { Unhighlighted text color }
                listSelTextClr: Integer;  { Highlighted text color }
                listNorBackClr: Integer;  { Unhighlighted background color }
                listSelBackClr: Integer;  { Highlighted backgraound color }
            END;
       MemRecHndl         = ^MemRecPtr;
       MemRecPtr          = ^MemRec;
       MemRec             =
            PACKED RECORD
                memPtr:         Ptr;      { Pointer to string, or custom }
                memFlag:        Byte;     { Bit Flag }
            END;

       ListCtlRecHndl     = ^ListCtlRecPtr;
       ListCtlRecPtr      = ^ListCtlRec;
       ListCtlRec         =
            PACKED RECORD
                ctlNext:        CtlRecHndl;    { Handle of Next Control }
                ctlOwner:       GrafPortPtr;   { Window owner }
                ctlRect:        Rect;          { Enclosing Rect }
                ctlFlag:        Byte;          { Bit 7 visible, Bit 0 string type,
                                               Bit 1 multiple }
                ctlHilite:      Byte;          { (not used) }
                ctlValue:       Integer;       { First member in display }
                ctlProc:        LongProcPtr;   { Address of list definition procedure}
                ctlAction:      LongProcPtr;   { Address of list action procedure }
                ctlData:        Longint;       { Low = view size, High = total members }
                ctlRefCon:      Longint;        { Not used }
                ctlColor:       LColorTablePtr;{ Null for default colors }
                ctlMemDraw:     VoidProcPtr;    {Address of routine to draw members}
                ctlMemHeight:   Integer;        { Member's Height in Pixels }
                ctlMemSize:     Integer;        { Bytes in member record }
                ctlList:        MemRecPtr;      { Adress of first member record in array }
                ctlListBar:     CtlRecHndl;     { Handle of list contrlo's scroll bar
                                                control }
            END;

        ListRecHndl             = ^ListRecPtr;
        ListRecPtr              = ^ListRec;
        ListRec                 =
            RECORD
            listRect:       Rect;           { Enclosing Rectangle }
            listSize:       Integer;        { Number of List Members }
            listView:       Integer;        { Max Viewable members }
            listType:       Integer;        { Bit Flag }
            listStart:      Integer;        { First member in view }
            listCtl:        CtlRecHndl;     { List control's handle }
            listDraw:       VoidProcPtr;    { Address of Custom drawing routine}
            listMemHeight:  Integer;        { Height of list members }
            listMemSize:    Integer;        { Size of Member Records }
            listPointer:    MemRecPtr;      { Pointer to first element in MemRec array }
            listRefCon:     Longint;        { becomes Control's refCon }
            listScrollClr:  BarColorsPtr;   { Color table for list's scroll bar}
            END;

PROCEDURE ListBootInit;
PROCEDURE ListStartup;
PROCEDURE ListShutDown;
FUNCTION  ListVersion: Integer;
PROCEDURE ListReset;
FUNCTION  ListStatus: Boolean;
FUNCTION  CreateList
            (theWindowPtr:      WindowPtr;
             __listRecPtr:      ListRecPtr): ListCtlRecHndl;
PROCEDURE DrawMember
            (memberPtr:         MemRecPtr;
             __listRecPtr:      ListRecPtr);
FUNCTION  GetListDefProc:       LongProcPtr;
PROCEDURE NewList
            (memberPtr:         MemRecPtr;
             __listRecPtr: ListRecPtr);
FUNCTION  NextMember
            (memberPtr:         MemRecPtr;
             __listRecPtr:      ListRecPtr): MemRecPtr;
FUNCTION  ResetMember
            (__listRecPtr:      ListRecPtr): MemRecPtr;
PROCEDURE SelectMember
            (memberPtr:         MemRecPtr;
             __listRecPtr:      ListRecPtr);
PROCEDURE SortList
            (comparePtr:        VoidProcPtr;
             __listRecPtr:      ListRecPtr);
PROCEDURE DrawMember2
            (itemNumber:        Integer;
             ctlHandle:         CtlRecHndl);
```

```
FUNCTION  NextMember2
              (itemNumber:          Integer;
               ctlHandle:           CtlRecHndl): Integer;
FUNCTION  ResetMember2
              (ctlHandle:           CtlRecHndl): Integer;
PROCEDURE SelectMember2
              (itemNumber:          Integer;
               ctlHandle:           CtlRecHndl);
PROCEDURE SortList2
              (comparePtr:          Ptr;
               ctlHandle:           CtlRecHndl);
PROCEDURE NewList2
              (drawProcPtr:         ProcPtr;
               listStart:           Integer;
               listRef:             Ref;
               listRefDesc:         RefDescriptor;
               listSize:            Integer;
               ctlHandle:           CtlRecHndl);

IMPLEMENTATION
END.
```

Loader

```
{********************************************
; File: Loader.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Loader;

INTERFACE
USES Types;

CONST  idNotFound          = $1101; {error - segment/application/entry not found }
       idNotLoadFile       = $1104; {error - file is not a load file }
       idBusyErr           = $1105; {error - system loader is busy }
       idFilVersErr        = $1107; {error - file version error }
       idUserIDErr         = $1108; {error - user ID error }
       idSequenceErr       = $1109; {error - segnum out of sequence }
       idBadRecordErr      = $110A; {error - illegal load record found }
       idForeignSegErr     = $110B; {error - segment is foreign }

TYPE   InitialLoadOutputRecPtr  = ^InitialLoadOutputRec;
       InitialLoadOutputRec     =
           RECORD
               userID:Integer;
               startAddr:   Ptr;
               dPageAddr:   Integer;
```

```
                  buffSize:       Integer;
              END;

       RestartOutRecPtr            = ^RestartOutRec;
       RestartOutRec               =
              RECORD
                  userID:         Integer;
                  startAddr:      Ptr;
                  dPageAddr:      Integer;
                  buffSize:       Integer;
              END;

       LoadSegNameOutPtr           = ^LoadSegNameOut;
       LoadSegNameOut              =
              RECORD
                  segAddr:        Ptr;
                  fileNum:        Integer;
                  segNum:         Integer;
              END;

       UnloadSegOutRecPtr          = ^UnloadSegOutRec;
       UnloadSegOutRec             =
              RECORD
                  userID:         Integer;
                  fileNum:        Integer;
                  segNum:         Integer;
              END;

PROCEDURE LoaderInitialization;
PROCEDURE LoaderStartUp;
PROCEDURE LoaderShutDown;
FUNCTION  LoaderVersion:          Integer;
PROCEDURE LoaderReset;
FUNCTION  LoaderStatus:           Boolean;
PROCEDURE GetLoadSegInfo
              (userID:            Integer;
               loadFileNum:       Integer;
               loadSegNum:        Integer;
               bufferPtr:         Ptr);
FUNCTION  GetUserID
              (pathNamePtr: Ptr): Integer;
FUNCTION  GetUserID2
              (pathNamePtr: Ptr): Integer;
FUNCTION  InitialLoad
              (userID:            Integer;
               loadFileNamePtr:   Ptr;
               spMemFlag:         Boolean): InitialLoadOutputRec;
FUNCTION  InitialLoad2
              (userID:            Integer;
               loadFileNamePtr:   Ptr;
               spMemFlag:         Boolean;
               inputType:         Integer): InitialLoadOutputRec;
FUNCTION  LGetPathname
              (userID:            Integer;
               fileNumber:        Integer): Ptr;
FUNCTION  LGetPathname2
              (userID:            Integer;
               fileNumber:        Integer): Ptr;
```

```
FUNCTION  GetPathname
                (userID:                Integer;
                 fileNumber:            Integer): Ptr;
FUNCTION  GetPathname2
                (userID:                Integer;
                 fileNumber:            Integer): Ptr;
PROCEDURE RenamePathname
                (oldPathname:           Ptr;
                 newPathname:           Ptr);
FUNCTION  LoadSegName
                (userID:                Integer;
                 loadFileNamePtr:       Ptr;
                 loadSegNamePtr:        Ptr): LoadSegNameOut;
FUNCTION  loadSegNum
                (userID:                Integer;
                 loadFileNum:           Integer;
                 loadSegNum:            Integer): Ptr;
FUNCTION  Restart
                (userID:                Integer): RestartOutRec;
FUNCTION  UnloadSeg
                (segmentPtr:            Ptr): UnloadSegOutRec;
PROCEDURE UnloadSegNum
                (userID:                Integer;
                 loadFileNum:           Integer;
                 loadSegNum:            Integer);
FUNCTION  UserShutDown
                (userID:                Integer;
                 restartFlag:           Integer): Integer;

IMPLEMENTATION
END.
```

Locator

```
{*********************************************
; File: Locator.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Locator;

INTERFACE
USES Types;

CONST  toolNotFoundErr      = $0001; {error - }
       funcNotFoundErr      = $0002; {error - }
       toolVersionErr       = $0110; {error - }
       sysStrtMtErr         = $0100; {error - can't mount system startup volume }
       messNotFoundErr      = $0111; {error - }
       fileInfoType         = $0001; {MessageCenter - Message type parameter }
```

```
      addMessage            = $0001; {MessageCenter - action parameter }
      getMessage            = $0002; {MessageCenter - action parameter }
      deleteMessage         = $0003; {MessageCenter - action parameter }
      mvReturn              = $0001; {TLMountVolume - like ok for dialogs }
      mvEscape              = $0002; {TLMountVolume - like cancel for dialogs }
      sysTool               = $0000; {Tool Set Spec - }
      userTool              = $8000; {Tool Set Spec - }


TYPE  MessageRecHndl        = ^MessageRecPtr;
      MessageRecPtr         = ^MessageRec;
      MessageRec            =
            RECORD
                  messageNext:          MessageRecHndl;
                  messageType:          Integer;
                  messageData:          Integer;
                  fileNames:            PACKED ARRAY [1..1] OF Str255;
            END;

      ToolSpec              =
            RECORD
                  toolNumber:           Integer;
                  minVersion:           Integer;
            END;

      StartStopRecordPtr    = ^StartStopRecord;
      StartStopRecord       =
            RECORD
                  flags:                Integer;
                  videoMode:            Integer;
                  resFileID:            Integer;
                  dPageHandle:          Handle;
                  numTools:             Integer;
            END;

PROCEDURE TLBootInit;
PROCEDURE TLStartUp;
PROCEDURE TLShutDown;
FUNCTION TLVersion:                Integer;
PROCEDURE TLReset;
FUNCTION TLStatus:                 Boolean;
FUNCTION GetFuncPtr
            (userOrSystem:         Integer;
             funcTSNum:            Integer): Ptr;
FUNCTION GetTSPtr
            (userOrSystem:         Integer;
            tSNum:                 Integer): Ptr;
FUNCTION GetWAP
            (userOrSystem:         Integer;
             tSNum:                Integer): Ptr;
PROCEDURE LoadOneTool
            (toolNumber:           Integer;
             minVersion:           Integer);
PROCEDURE LoadTools
            (toolTablePtr:         Ptr);
PROCEDURE MessageCenter
            (action:               Integer;
             messageType:          Integer;
```

```
               messageHandle:      MessageRecHndl);
PROCEDURE RestoreTextState
               (stateHandle:       Handle);
FUNCTION SaveTextState:            Handle;
PROCEDURE SetDefaultTPT;
PROCEDURE SetTSPtr
               (userOrSystem:      Integer;
                tSNum:             Integer;
                fptablePtr:        FPTPtr);
PROCEDURE SetWAP
               (userOrSystem:      Integer;
                tSNum:             Integer;
                waptPtr:           Ptr);
FUNCTION TLMountVolume
               (whereX:            Integer;
                whereY:            Integer;
                line1:             Str255;
                line2:             Str255;
                but1:              Str255;
                but2:              Str255): Integer;
FUNCTION TLTextMountVolume
               (line1:             Str255;
                line2:             Str255;
                but1:              Str255;
                but2:              Str255): Integer;
PROCEDURE UnloadOneTool
               (toolNumber:        Integer);
FUNCTION StartUpTools
               (userID:            Integer;
                startStopRefDesc:  RefDescriptor;
                startStopRef:      Ref): Ref;
PROCEDURE ShutDownTools
               (startStopDesc:     RefDescriptor;
                startStopRef:      Ref);
FUNCTION MessageByName
               (createItFlag:      Boolean;
                recordPtr:         Ptr): Longint;

IMPLEMENTATION
END.
```

Memory

```
{********************************************
; File: Memory.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Memory;

INTERFACE
USES Types;

CONST  memErr            = $0201;   {error - unable to allocate block }
       emptyErr          = $0202;   {error - illegal operation, empty handle }
       notEmptyErr       = $0203;   {error - an empty handle was expected for this
                                    operation }
       lockErr           = $0204;   {error - illegal operation on a locked block }
       purgeErr          = $0205;   {error - attempt to purge an unpurgable block }
       handleErr         = $0206;   {error - an invalid handle was given }
       idErr             = $0207;   {error - an invalid owner ID was given }
       attrErr           = $0208;   {error - operation illegal on block with given
                                    attributes }
       attrNoPurge       = $0000;   {Handle Attribute Bits - Not purgeable }
       attrBank          = $0001;   {Handle Attribute Bits - fixed bank }
       attrAddr          = $0002;   {Handle Attribute Bits - fixed address }
       attrPage          = $0004;   {Handle Attribute Bits - page aligned }
       attrNoSpec        = $0008;   {Handle Attribute Bits - may not use special memory}
       attrNoCross       = $0010;   {Handle Attribute Bits - may not cross banks }
       attrPurge1        = $0100;   {Handle Attribute Bits - Purge level 1 }
       attrPurge2        = $0200;   {Handle Attribute Bits - Purge level 2 }
       attrPurge3        = $0300;   {Handle Attribute Bits - Purge level 3 }
       attrPurge         = $0300;   {Handle Attribute Bits - test or set both purge bits }
       attrHandle        = $1000;   {Handle Attribute Bits - block of master pointers }
       attrSystem        = $2000;   {Handle Attribute Bits - system handle }
       attrFixed         = $4000;   {Handle Attribute Bits - not movable }
       attrLocked        = $8000;   {Handle Attribute Bits - locked }


PROCEDURE MMBootInit;
FUNCTION  MMStartUp:         Integer;
PROCEDURE MMShutDown
               (userID:      Integer);
FUNCTION  MMVersion:         Integer;
PROCEDURE MMReset;
FUNCTION  MMStatus:          Boolean;
PROCEDURE BlockMove
               (srcPtr:      Ptr;
                dstPtr:      Ptr;
                count:       Longint);
PROCEDURE CheckHandle
               (theHandle:   Handle);
```

```
PROCEDURE CompactMem;
PROCEDURE DisposeAll
                (userID:        Integer);
PROCEDURE DisposeHandle
                (theHandle:     Handle);
FUNCTION  FindHandle
                (locationPtr: Ptr): Handle;
FUNCTION  FreeMem:              Longint;
FUNCTION  GetHandleSize
                (theHandle:     Handle): Longint;
PROCEDURE HandToHand
                (sourceHandle: Handle;
                 destHandle:    Handle;
                 count:         Longint);
PROCEDURE HandToPtr
                (sourceHandle: Handle;
                 destPtr:       Ptr;
                 count:         Longint);
PROCEDURE HLock
                (theHandle:     Handle);
PROCEDURE HLockAll
                (userID:        Integer);
PROCEDURE HUnlock
                (theHandle:     Handle);
PROCEDURE HUnlockAll
                (userID:        Integer);
FUNCTION  MaxBlock:             Longint;
FUNCTION  NewHandle
                (blockSize:     Longint;
                 userID:        Integer;
                 attributes:    Integer;
                 locationPtr:   Ptr): Handle;
PROCEDURE PtrToHand
                (sourcePtr:     Ptr;
                 destHandle:    Handle;
                 count:         Longint);
PROCEDURE PurgeAll
                (userID:        Integer);
PROCEDURE PurgeHandle
                (theHandle:     Handle);
FUNCTION  RealFreeMem:          Longint;
PROCEDURE ReallocHandle
                (blockSize:     Longint;
                 userID:        Integer;
                 attributes:    Integer;
                 locationPtr:   Ptr;
                 theHandle:     Handle);
PROCEDURE RestoreHandle
                (theHandle:     Handle);
PROCEDURE SetHandleSize
                (newSize:       Longint;
                 theHandle:     Handle);
PROCEDURE SetPurge
                (newPurgeLevel:Integer;
                 theHandle:     Handle);
PROCEDURE SetPurgeAll
                (newPurgeLevel:Integer;
                 userID:        Integer);
```

```
FUNCTION  TotalMem:          Longint;
PROCEDURE AddToOOMQueue
             (headerPtr:     Ptr);
PROCEDURE DeleteFromOOMQueue
             (headerPtr:     Ptr);

IMPLEMENTATION
END.
```

Menus

```
{*********************************************
; File: Menus.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
*********************************************}

UNIT Menus;

INTERFACE
USES Types, QuickDraw, Events, Controls, Windows;

CONST  mDrawMsg              = $0000;   {MenuDefProcCodes - }
       mChooseMsg            = $0001;   {MenuDefProcCodes - }
       mSizeMsg              = $0002;   {MenuDefProcCodes - }
       mDrawTitle            = $0003;   {MenuDefProcCodes - }
       mDrawMItem            = $0004;   {MenuDefProcCodes - }
       mGetMItemID           = $0005;   {MenuDefProcCodes - }
       mInvis                = $0004;   {MenuFlag - }
       mCustom               = $0010;   {MenuFlag - }
       mXor                  = $0020;   {MenuFlag - }
       mSelected             = $0040;   {MenuFlag - }
       mDisabled             = $0080;   {MenuFlag - }
       customMenu            = $0010;   {MenuFlagMasks - }
       xorMItemHilite        = $0020;   {MenuFlagMasks - }
       xorTitleHilite        = $0020;   {MenuFlagMasks - }
       underMItem            = $0040;   {MenuFlagMasks - }
       disableItem           = $0080;   {MenuFlagMasks - }
       disableMenu           = $0080;   {MenuFlagMasks - }
       enableItem            = $FF7F;   {MenuFlagMasks - }
       enableMenu            = $FF7F;   {MenuFlagMasks - }
       noUnderMItem          = $FFBF;   {MenuFlagMasks - }
       colorMItemHilite      = $FFDF;   {MenuFlagMasks - }
       colorTitleHilite      = $FFDF;   {MenuFlagMasks - }
       colorReplace          = $FFDF;   {MenuFlagMasks - }
       standardMenu          = $FFEF;   {MenuFlagMasks - }

TYPE   MenuBarRecHndl        = ^MenuBarRecPtr;
       MenuBarRecPtr         = ^MenuBarRec;
       MenuBarRec            = CtlRec;
       MenuRecHndl           = ^MenuRecPtr;
```

```
    MenuRecPtr              = ^MenuRec;
    MenuRec                 =
        PACKED RECORD
            menuID:         Integer;        { Menu's ID number }
            menuWidth:      Integer;        { Width of menu }
            menuHeight:     Integer;        { Height of menu }
            menuProc:       WordProcPtr;    { Menu's definition procedure }
            menuFlag:       Integer;        { Bit flags }
            firstItem:      Byte;
            numOfItems:     Byte;
            titleWidth:     Integer;        { Width of menu's title }
            titleName:      Ptr;
        END;


PROCEDURE MenuBootInit;
PROCEDURE MenuStartUp
            (userID:                Integer;
             dPageAddr:             Integer);
PROCEDURE MenuShutDown;
FUNCTION  MenuVersion:              Integer;
PROCEDURE MenuReset;
FUNCTION  MenuStatus:               Boolean;
PROCEDURE CalcMenuSize
            (newWidth:              Integer;
             newHeight:             Integer;
             menuNum:               Integer);
PROCEDURE CheckMItem
            (checkedFlag:           Boolean;
             itemNum:               Integer);
FUNCTION  CountMItems
            (menuNum:               Integer): Integer;
PROCEDURE DeleteMenu
            (menuNum:               Integer);
PROCEDURE DeleteMItem
            (itemNum:               Integer);
PROCEDURE DisableMItem
            (itemNum:               Integer);
PROCEDURE DisposeMenu
            (menuHandle:            MenuRecHndl);
PROCEDURE DrawMenuBar;
PROCEDURE EnableMItem
            (itemNum:               Integer);
FUNCTION  FixMenuBar:               Integer;
PROCEDURE FlashMenuBar;
FUNCTION  GetBarColors:             Longint;
FUNCTION  GetMenuBar:               MenuBarRecHndl;
FUNCTION  GetMenuFlag
            (menuNum:               Integer): Integer;
FUNCTION  GetMenuMgrPort:           GrafPortPtr;
FUNCTION  GetMenuTitle
            (menuNum:               Integer): Ptr;
FUNCTION  GetMHandle
            (menuNum:               Integer): MenuRecHndl;
FUNCTION  GetMItem
            (itemNum:               Integer): StringPtr;
FUNCTION  GetMItemFlag
            (itemNum:               Integer): Integer;
```

```
FUNCTION   GetMItemMark
                (itemNum:            Integer): Integer;
FUNCTION   GetMItemStyle
                (itemNum:            Integer): TextStyle;
FUNCTION   GetMTitleStart:           Integer;
FUNCTION   GetMTitleWidth
                (menuNum:            Integer): Integer;
FUNCTION   GetSysBar:                MenuBarRecHndl;
PROCEDURE HiliteMenu
                (hiliteFlag:         Boolean;
                 menuNum:            Integer);
PROCEDURE InitPalette;
PROCEDURE InsertMenu
                (addMenuHandle:      MenuRecHndl;
                 insertAfter:        Integer);
PROCEDURE InsertMItem
                (addItemPtr:         Ptr;
                 insertAfter:        Integer;
                 menuNum:            Integer);
FUNCTION   MenuGlobal
                (menuGlobalMask:     Integer): Integer;
PROCEDURE MenuKey
                (taskRecPtr:         WmTaskRec;
                 barHandle:          MenuBarRecHndl);
PROCEDURE MenuNewRes;
PROCEDURE MenuRefresh
                (redrawRoutinePtr:   VoidProcPtr);
PROCEDURE MenuSelect
                (taskRecPtr:         WmTaskRec;
                 barHandle:          MenuBarRecHndl);
FUNCTION   NewMenu
                (menuStringPtr:      Ptr): MenuRecHndl;
FUNCTION   NewMenuBar
                (theWindowPtr:       WindowPtr): MenuBarRecHndl;
PROCEDURE SetBarColors
                (newBarColor:        Integer;
                 newInvertColor:     Integer;
                 newOutColor:        Integer);
PROCEDURE SetMenuBar
                (barHandle:          MenuBarRecHndl);
PROCEDURE SetMenuFlag
                (newValue:           Integer;
                 menuNum:            Integer);
PROCEDURE SetMenuID
                (newMenuNum:         Integer;
                 curMenuNum:         Integer);
PROCEDURE SetMenuTitle
                (newStr:             Str255;
                 menuNum:            Integer);
PROCEDURE SetMItem
                (newItemLine:        Str255;
                 itemNum:            Integer);
PROCEDURE SetMItemBlink
                (count:              Integer);
PROCEDURE SetMItemFlag
                (newValue:           Integer;
                 itemNum:            Integer);
```

```
PROCEDURE SetMItemID
             (newItemNum:         Integer;
              curItemNum:         Integer);
PROCEDURE SetMItemMark
             (mark:               Integer;
              itemNum:            Integer);
PROCEDURE SetMItemName
             (str:                Str255;
              itemNum:            Integer);
PROCEDURE SetMItemStyle
             (theTextStyle:       TextStyle;
              itemNum:            Integer);
PROCEDURE SetMTitleStart
             (xStart:             Integer);
PROCEDURE SetMTitleWidth
             (newWidth:           Integer;
              menuNum:            Integer);
PROCEDURE SetSysBar
             (barHandle:          MenuBarRecHndl);
FUNCTION  PopUpMenuSelect
             (selection:          Integer;
              currentLeft:        Integer;
              currentTop:         Integer;
              flag:               Integer;
              menuHandle:         MenuRecHndl): Integer;
FUNCTION  GetPopUpDefProc:        Ptr;
PROCEDURE DrawPopUp
             (selection:          Integer;
              flag:               Integer;
              right:              Integer;
              bottom:             Integer;
              left:               Integer;
              top:                Integer;
              menuHandle:         MenuRecHndl);
FUNCTION  NewMenuBar2
             (refDesc:            RefDescriptor;
              menuBarTemplateRef: Ref;
              windowPortPtr:      GrafPortPtr): MenuBarRecHndl;
FUNCTION  NewMenu2
             (refDesc:            RefDescriptor;
              menuTemplateRef:    Ref): MenuRecHndl;
PROCEDURE InsertMItem2
             (refDesc:            RefDescriptor;
              menuTemplateRef:    Ref;
              insertAfter:        Integer;
              menuNum:            Integer);
PROCEDURE SetMenuTitle2
             (refDesc:            RefDescriptor;
              titleRef:           Ref;
              menuNum:            Integer);
PROCEDURE SetMItem2
             (refDesc:            RefDescriptor;
              menuItemTempRef:    Ref;
              menuItemID:         Integer);
PROCEDURE SetMItemName2
             (refDesc:            RefDescriptor;
              titleRef:           Ref;
              menuItemID:         Integer);
```

```
PROCEDURE HideMenuBar;
PROCEDURE ShowMenuBar;

IMPLEMENTATION
END.
```

MIDI

```
{********************************************
; File: MIDI.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT MIDI;

INTERFACE
USES Types;

CONST  miToolNum        = $0020;   {Midi - the tool number of the MIDI Tool Set }
       miDrvrFileType   = $00BB;   {Midi - filetype of MIDI device driver }
       miNSVer          = $0102;   {Midi - minimum version of Note Synthesizer
                                   required by MIDI Tool Set }
       miSTVer          = $0203;   {Midi - minimum version of Sound Tools needed by
                                   MIDI Tool Set }
       miDrvrAuxType    = $0300;   {Midi - aux type of MIDI device driver }
       miStartUpErr     = $2000;   {Midi - MIDI Tool Set is not started }
       miPacketErr      = $2001;   {Midi - incorrect length for a received MIDI command }
       miArrayErr       = $2002;   {Midi - a designated array had an insufficient or
                                   illegal size }
       miFullBufErr     = $2003;   {Midi - input buffer overflow }
       miToolsErr       = $2004;   {Midi - the required tools were not started up or
                                   had insufficient versions }
       miOutOffErr      = $2005;   {Midi - MIDI output must first be enabled }
       miNoBufErr       = $2007;   {Midi - no buffer is currently allocated }
       miDriverErr      = $2008;   {Midi - the designated file is not a legal MIDI
                                   device driver }
       miBadFreqErr     = $2009;   {Midi - the MIDI clock cannot attain the requested
                                   frequency }
       miClockErr       = $200A;   {Midi - the MIDI clock value wrapped to zero }
       miConflictErr    = $200B;   {Midi - conflicting processes for MIDI input }
       miNoDevErr       = $200C;   {Midi - no MIDI device driver loaded }
       miDevNotAvail    = $2080;   {Midi - the requested device is not available }
       miDevSlotBusy    = $2081;   {Midi - requested slot is already in use }
       miDevBusy        = $2082;   {Midi - the requested device is already in use }
       miDevOverrun     = $2083;   {Midi - device overrun by incoming MIDI data }
       miDevNoConnect   = $2084;   {Midi - no connection to MIDI }
       miDevReadErr     = $2085;   {Midi - framing error in received MIDI data }
       miDevVersion     = $2086;   {Midi - ROM version is incompatible with device
                                   driver }
       miDevIntHndlr    = $2087;   {Midi - conflicting interrupt handler is installed }
```

```
        miSetClock          = $0000;      {MidiClock - set time stamp clock }
        miStartClock        = $0001;      {MidiClock - start time stamp clock }
        miStopClock         = $0002;      {MidiClock - stop time stamp clock }
        miSetFreq           = $0003;      {MidiClock - set clock frequency }
        miRawMode           = $00000000;{MidiControl - raw mode for MIDI input & output }
        miSetRTVec          = $0000;      {MidiControl - set real-time message vector }
        miPacketMode        = $00000001;{MidiControl - packet mode for MIDI input and output }
        miSetErrVec         = $0001;      {MidiControl - set real-time error vector }
        miStandardMo        = $00000002;{MidiControl - standard mode for MIDI input and output }
        miSetInBuf          = $0002;      {MidiControl - set input buffer information }
        miSetOutBuf         = $0003;      {MidiControl - set output buffer information }
        miStartInput        = $0004;      {MidiControl - start MIDI input }
        miStartOutput       = $0005;      {MidiControl - start MIDI output }
        miStopInput         = $0006;      {MidiControl - stop MIDI input }
        miStopOutput        = $0007;      {MidiControl - stop MIDI output }
        miFlushInput        = $0008;      {MidiControl - discard contents of input buffer }
        miFlushOutput       = $0009;      {MidiControl - discard contents of output buffer }
        miFlushPacke        = $000A;      {MidiControl - discard next input packet }
        miWaitOutput        = $000B;      {MidiControl - wait for output buffer to empty }
        miSetInMode         = $000C;      {MidiControl - set input mode }
        miSetOutMode        = $000D;      {MidiControl - set output mode }
        miClrNotePad        = $000E;      {MidiControl - clear all notes marked on in the notepad }
        miSetDelay          = $000F;      {MidiControl - set minimum delay between output packets }
        miOutputStat        = $0010;      {MidiControl - enable/disable output of running - status}
        miIgnoreSysEx       = $0011;      {MidiControl - ignore system exclusive input }
        miSelectDrvr        = $0000;      {MidiDevice - display device driver selection dialog }
        miLoadDrvr          = $0001;      {MidiDevice - load and initialize device driver }
        miUnloadDrvr        = $0002;      {MidiDevice - shutdown MIDI device, unload driver}
        miNextPktLen        = $0;         {MidiInfo - return length of next packet }
        miInputChars        = $0001;      {MidiInfo - return number of characters in input buffer }
        miOutputChars       = $0002;      {MidiInfo - return number of characters in output buffer}
        miMaxInChars        = $0003;      {MidiInfo - return maximum number of characters in
                                          input buffer }
        miMaxOutChars       = $0004;      {MidiInfo - return maximum number of characters in
                                          output buffer }
        miRecordAddr        = $0005;      {MidiInfo - return current MidiRecordSeq address }
        miPlayAddr          = $0006;      {MidiInfo - return current MidiPlaySeq address }
        miClockValue        = $0007;      {MidiInfo - return current time stamp clock value}
        miClockFreq         = $0008;      {MidiInfo - return number of clock ticks per second }

TYPE    MiBufInfo     =
            RECORD
                bufSize:   Integer;    { size of buffer (0 for default) }
                address:   Ptr;        { address of buffer (0 for auto-allocation) }
            END;

        MiDriverInfo =
            RECORD
                slot:      Integer;    { device slot }
                external:  Integer;    { slot internal (=0) / external (=1) }
                pathname:  PACKED ARRAY[1..65] OF Byte; { device driver pathname }
            END;

PROCEDURE MidiBootInit;
PROCEDURE MidiStartUp
            (userID:          Integer;
             directPages:     Integer);
PROCEDURE MidiShutDown;
```

```
FUNCTION   MidiVersion:              Integer;
PROCEDURE  MidiReset;
FUNCTION   MidiStatus:               Boolean;
PROCEDURE  MidiClock
                (funcNum:            Integer;
                 arg:                Longint);
PROCEDURE  MidiControl
                (controlCode:        Integer);
PROCEDURE  MidiDevice
                (funcNum:            Integer;
                 driverInfo:         Ptr);
FUNCTION   MidiInfo
                (funcNum:            Integer): Longint;
PROCEDURE  MidiInputPoll;
FUNCTION   MidiReadPacket
                (arrayAddr:          Ptr;
                 arraySize:          Integer): Integer;
FUNCTION   MidiWritePacket
                (arrayAddr:          Ptr): Integer;


IMPLEMENTATION
END.
```

MiscTool

```
{********************************************
; File: MiscTool.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT MiscTool;

INTERFACE
USES Types;

CONST   badInputErr         = $0301; {error - bad input parameter }
        noDevParamErr       = $0302; {error - no device for input parameter }
        taskInstlErr        = $0303; {error - task already installed error }
        noSigTaskErr        = $0304; {error - no signature in task header }
        queueDmgdErr        = $0305; {error - queue has been damaged error }
        taskNtFdErr         = $0306; {error - task was not found error }
        firmTaskErr         = $0307; {error - firmware task was unsuccessful }
        hbQueueBadErr       = $0308; {error - heartbeat queue damaged }
        unCnctdDevErr       = $0309; {error - attempted to dispatch to unconnected device }
        idTagNtAvlErr       = $030B; {error - ID tag not available }
        pdosUnClmdIntErr    = $0001; {System Fail - ProDOS unclaimed interrupt error }
        divByZeroErr        = $0004; {System Fail - divide by zero error }
        pdosVCBErr          = $000A; {System Fail - ProDOS VCB unusable }
        pdosFCBErr          = $000B; {System Fail - ProDOS FCB unusable }
```

```
pdosBlk0Err          = $000C; {System Fail - ProDOS block zero allocated illegally }
pdosIntShdwErr       = $000D; {System Fail - ProDOS interrupt w/ shadowing off }
segLoader1Err        = $0015; {System Fail - segment loader error }
sPackage0Err         = $0017; {System Fail - can't load a package }
package1Err          = $0018; {System Fail - can't load a package }
package2Err          = $0019; {System Fail - can't load a package }
package3Err          = $001A; {System Fail - can't load a package }
package4Err          = $001B; {System Fail - can't load a package }
package5Err          = $001C; {System Fail - can't load a package }
package6Err          = $001D; {System Fail - can't load a package }
package7Err          = $001E; {System Fail - can't load a package }
package8Err          = $0020; {System Fail - can't load a package }
package9Err          = $0021; {System Fail - can't load a package }
package10Err         = $0022; {System Fail - can't load a package }
package11Err         = $0023; {System Fail - can't load a package }
package12Err         = $0024; {System Fail - can't load a package }
outOfMemErr          = $0025; {System Fail - out of memory error }
segLoader2Err        = $0026; {System Fail - segment loader error }
fMapTrshdErr         = $0027; {System Fail - file map trashed }
stkOvrFlwErr         = $0028; {System Fail - stack overflow error }
psInstDiskErr        = $0030; {System Fail - Please Insert Disk (file manager alert) }
memMgr1Err           = $0032; {System Fail - memory manager error }
memMgr2Err           = $0033; {System Fail - memory manager error }
memMgr3Err           = $0034; {System Fail - memory manager error }
memMgr4Err           = $0035; {System Fail - memory manager error }
memMgr5Err           = $0036; {System Fail - memory manager error }
memMgr6Err           = $0037; {System Fail - memory manager error }
memMgr7Err           = $0038; {System Fail - memory manager error }
memMgr8Err           = $0039; {System Fail - memory manager error }
memMgr9Err           = $003A; {System Fail - memory manager error }
memMgr10Err          = $003B; {System Fail - memory manager error }
memMgr11Err          = $003C; {System Fail - memory manager error }
memMgr12Err          = $003D; {System Fail - memory manager error }
memMgr13Err          = $003E; {System Fail - memory manager error }
memMgr14Err          = $003F; {System Fail - memory manager error }
memMgr15Err          = $0040; {System Fail - memory manager error }
memMgr16Err          = $0041; {System Fail - memory manager error }
memMgr17Err          = $0042; {System Fail - memory manager error }
memMgr18Err          = $0043; {System Fail - memory manager error }
memMgr19Err          = $0044; {System Fail - memory manager error }
memMgr20Err          = $0045; {System Fail - memory manager error }
memMgr21Err          = $0046; {System Fail - memory manager error }
memMgr22Err          = $0047; {System Fail - memory manager error }
memMgr23Err          = $0048; {System Fail - memory manager error }
memMgr24Err          = $0049; {System Fail - memory manager error }
memMgr25Err          = $004A; {System Fail - memory manager error }
memMgr26Err          = $004B; {System Fail - memory manager error }
memMgr27Err          = $004C; {System Fail - memory manager error }
memMgr28Err          = $004D; {System Fail - memory manager error }
memMgr29Err          = $004E; {System Fail - memory manager error }
memMgr30Err          = $004F; {System Fail - memory manager error }
memMgr31Err          = $0050; {System Fail - memory manager error }
memMgr32Err          = $0051; {System Fail - memory manager error }
memMgr33Err          = $0052; {System Fail - memory manager error }
memMgr34Err          = $0053; {System Fail - memory manager error }
stupVolMntErr        = $0100; {System Fail - can't mount system startup volume }
p1PrntModem          = $0000; {Battery Ram Parameter Ref Number - }
p1LineLnth           = $0001; {Battery Ram Parameter Ref Number - }
```

```
p1DelLine        = $0002; {Battery Ram Parameter Ref Number - }
p1AddLine        = $0003; {Battery Ram Parameter Ref Number - }
p1Echo           = $0004; {Battery Ram Parameter Ref Number - }
p1Buffer         = $0005; {Battery Ram Parameter Ref Number - }
p1Baud           = $0006; {Battery Ram Parameter Ref Number - }
p1DtStpBits      = $0007; {Battery Ram Parameter Ref Number - }
p1Parity         = $0008; {Battery Ram Parameter Ref Number - }
p1DCDHndShk      = $0009; {Battery Ram Parameter Ref Number - }
p1DSRHndShk      = $000A; {Battery Ram Parameter Ref Number - }
p1XnfHndShk      = $000B; {Battery Ram Parameter Ref Number - }
p2PrntModem      = $000C; {Battery Ram Parameter Ref Number - }
p2LineLnth       = $000D; {Battery Ram Parameter Ref Number - }
p2DelLine        = $000E; {Battery Ram Parameter Ref Number - }
p2AddLine        = $000F; {Battery Ram Parameter Ref Number - }
p2Echo           = $0010; {Battery Ram Parameter Ref Number - }
p2Buffer         = $0011; {Battery Ram Parameter Ref Number - }
p2Baud           = $0012; {Battery Ram Parameter Ref Number - }
p2DtStpBits      = $0013; {Battery Ram Parameter Ref Number - }
p2Parity         = $0014; {Battery Ram Parameter Ref Number - }
p2DCDHndShk      = $0015; {Battery Ram Parameter Ref Number - }
p2DSRHndShk      = $0016; {Battery Ram Parameter Ref Number - }
p2XnfHndShk      = $0017; {Battery Ram Parameter Ref Number - }
dspColMono       = $0018; {Battery Ram Parameter Ref Number - }
dsp40or80        = $0019; {Battery Ram Parameter Ref Number - }
dspTxtColor      = $001A; {Battery Ram Parameter Ref Number - }
dspBckColor      = $001B; {Battery Ram Parameter Ref Number - }
dspBrdColor      = $001C; {Battery Ram Parameter Ref Number - }
hrtz50or60       = $001D; {Battery Ram Parameter Ref Number - }
userVolume       = $001E; {Battery Ram Parameter Ref Number - }
bellVolume       = $001F; {Battery Ram Parameter Ref Number - }
sysSpeed         = $0020; {Battery Ram Parameter Ref Number - }
slt1intExt       = $0021; {Battery Ram Parameter Ref Number - }
slt2intExt       = $0022; {Battery Ram Parameter Ref Number - }
slt3intExt       = $0023; {Battery Ram Parameter Ref Number - }
slt4intExt       = $0024; {Battery Ram Parameter Ref Number - }
slt5intExt       = $0025; {Battery Ram Parameter Ref Number - }
slt6intExt       = $0026; {Battery Ram Parameter Ref Number - }
slt7intExt       = $0027; {Battery Ram Parameter Ref Number - }
startupSlt       = $0028; {Battery Ram Parameter Ref Number - }
txtDspLang       = $0029; {Battery Ram Parameter Ref Number - }
kybdLang         = $002A; {Battery Ram Parameter Ref Number - }
kyBdBuffer       = $002B; {Battery Ram Parameter Ref Number - }
kyBdRepSpd       = $002C; {Battery Ram Parameter Ref Number - }
kyBdRepDel       = $002D; {Battery Ram Parameter Ref Number - }
dblClkTime       = $002E; {Battery Ram Parameter Ref Number - }
flashRate        = $002F; {Battery Ram Parameter Ref Number - }
shftCpsLCas      = $0030; {Battery Ram Parameter Ref Number - }
fstSpDelKey      = $0031; {Battery Ram Parameter Ref Number - }
dualSpeed        = $0032; {Battery Ram Parameter Ref Number - }
hiMouseRes       = $0033; {Battery Ram Parameter Ref Number - }
dateFormat       = $0034; {Battery Ram Parameter Ref Number - }
clockFormat      = $0035; {Battery Ram Parameter Ref Number - }
rdMinRam         = $0036; {Battery Ram Parameter Ref Number - }
rdMaxRam         = $0037; {Battery Ram Parameter Ref Number - }
langCount        = $0038; {Battery Ram Parameter Ref Number - }
lang1            = $0039; {Battery Ram Parameter Ref Number - }
lang2            = $003A; {Battery Ram Parameter Ref Number - }
lang3            = $003B; {Battery Ram Parameter Ref Number - }
```

```
lang4                  = $003C; {Battery Ram Parameter Ref Number - }
lang5                  = $003D; {Battery Ram Parameter Ref Number - }
lang6                  = $003E; {Battery Ram Parameter Ref Number - }
lang7                  = $003F; {Battery Ram Parameter Ref Number - }
lang8                  = $0040; {Battery Ram Parameter Ref Number - }
layoutCount            = $0041; {Battery Ram Parameter Ref Number - }
layout1                = $0042; {Battery Ram Parameter Ref Number - }
layout2                = $0043; {Battery Ram Parameter Ref Number - }
layout3                = $0044; {Battery Ram Parameter Ref Number - }
layout4                = $0045; {Battery Ram Parameter Ref Number - }
layout5                = $0046; {Battery Ram Parameter Ref Number - }
layout6                = $0047; {Battery Ram Parameter Ref Number - }
layout7                = $0048; {Battery Ram Parameter Ref Number - }
layout8                = $0049; {Battery Ram Parameter Ref Number - }
layout9                = $004A; {Battery Ram Parameter Ref Number - }
layout10               = $004B; {Battery Ram Parameter Ref Number - }
layout11               = $004C; {Battery Ram Parameter Ref Number - }
layout12               = $004D; {Battery Ram Parameter Ref Number - }
layout13               = $004E; {Battery Ram Parameter Ref Number - }
layout14               = $004F; {Battery Ram Parameter Ref Number - }
layout15               = $0050; {Battery Ram Parameter Ref Number - }
layout16               = $0051; {Battery Ram Parameter Ref Number - }
aTalkNodeNo            = $0080; {Battery Ram Parameter Ref Number - }
irqIntFlag             = $0000; {GetAddr Param Ref No - }
irqDataReg             = $0001; {GetAddr Param Ref No - }
irqSerial1             = $0002; {GetAddr Param Ref No - }
irqSerial2             = $0003; {GetAddr Param Ref No - }
irqAplTlkHi            = $0004; {GetAddr Param Ref No - }
tickCnt                = $0005; {GetAddr Param Ref No - }
irqVolume              = $0006; {GetAddr Param Ref No - }
irqActive              = $0007; {GetAddr Param Ref No - }
irqSndData             = $0008; {GetAddr Param Ref No - }
brkVar                 = $0009; {GetAddr Param Ref No - }
evMgrData              = $000A; {GetAddr Param Ref No - }
mouseSlot              = $000B; {GetAddr Param Ref No - }
mouseClamps            = $000C; {GetAddr Param Ref No - }
absClamps              = $000D; {GetAddr Param Ref No - }
sccIntFlag             = $000E; {GetAddr Param Ref No - }
extVGCInt              = $01;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
scanLineInt            = $02;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
adbDataInt             = $04;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
ADTBDataInt            = $04;   {Hardware Interrupt Status - maintained for
                                 compatiblity with old interfaces }
oneSecInt              = $10;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
quartSecInt            = $20;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
vbInt                  = $40;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
kbdInt                 = $80;   {Hardware Interrupt Status - Returned by
                                 GetIRQEnable }
kybdEnable             = $0000; {Interrupt Ref Number - Parameter to IntSource }
kybdDisable            = $0001; {Interrupt Ref Number - Parameter to IntSource }
vblEnable              = $0002; {Interrupt Ref Number - Parameter to IntSource }
vblDisable             = $0003; {Interrupt Ref Number - Parameter to IntSource }
```

```
qSecEnable          = $0004; {Interrupt Ref Number - Parameter to IntSource }
qSecDisable         = $0005; {Interrupt Ref Number - Parameter to IntSource }
oSecEnable          = $0006; {Interrupt Ref Number - Parameter to IntSource }
oSecDisable         = $0007; {Interrupt Ref Number - Parameter to IntSource }
adbEnable           = $000A; {Interrupt Ref Number - Parameter to IntSource }
adbDisable          = $000B; {Interrupt Ref Number - Parameter to IntSource }
scLnEnable          = $000C; {Interrupt Ref Number - Parameter to IntSource }
scLnDisable         = $000D; {Interrupt Ref Number - Parameter to IntSource }
exVCGEnable         = $000E; {Interrupt Ref Number - Parameter to IntSource }
exVCGDisable        = $000F; {Interrupt Ref Number - Parameter to IntSource }
mouseOff            = $0000; {Mouse Mode Value - }
transparent         = $0001; {Mouse Mode Value - }
transParnt          = $0001; {Mouse Mode Value - (old name) }
moveIntrpt          = $0003; {Mouse Mode Value - }
bttnIntrpt          = $0005; {Mouse Mode Value - }
bttnOrMove          = $0007; {Mouse Mode Value - }
mouseOffVI          = $0008; {Mouse Mode Value - }
transParntVI        = $0009; {Mouse Mode Value - (old name) }
transparentVI       = $0009; {Mouse Mode Value - }
moveIntrptVI        = $000B; {Mouse Mode Value - }
bttnIntrptVI        = $000D; {Mouse Mode Value - }
bttnOrMoveVI        = $000F; {Mouse Mode Value - }
toolLoc1            = $0000; {Vector Ref Number - }
toolLoc2            = $0001; {Vector Ref Number - }
usrTLoc1            = $0002; {Vector Ref Number - }
usrTLoc2            = $0003; {Vector Ref Number - }
intrptMgr           = $0004; {Vector Ref Number - }
copMgr              = $0005; {Vector Ref Number - }
abortMgr            = $0006; {Vector Ref Number - }
_sysFailMgr         = $0007; {Vector Ref Number - }
aTalkIntHnd         = $0008; {Vector Ref Number - }
sccIntHnd           = $0009; {Vector Ref Number - }
scLnIntHnd          = $000A; {Vector Ref Number - }
sndIntHnd           = $000B; {Vector Ref Number - }
vblIntHnd           = $000C; {Vector Ref Number - }
mouseIntHnd         = $000D; {Vector Ref Number - }
qSecIntHnd          = $000E; {Vector Ref Number - }
kybdIntHnd          = $000F; {Vector Ref Number - }
adbRBIHnd           = $0010; {Vector Ref Number - }
adbSRQHnd           = $0011; {Vector Ref Number - }
deskAccHnd          = $0012; {Vector Ref Number - }
flshBufHnd          = $0013; {Vector Ref Number - }
kybdMicHnd          = $0014; {Vector Ref Number - }
oneSecHnd           = $0015; {Vector Ref Number - }
extVCGHnd           = $0016; {Vector Ref Number - }
otherIntHnd         = $0017; {Vector Ref Number - }
crsrUpdtHnd         = $0018; {Vector Ref Number - }
incBsyFlag          = $0019; {Vector Ref Number - }
decBsyFlag          = $001A; {Vector Ref Number - }
bellVector          = $001B; {Vector Ref Number - }
breakVector         = $001C; {Vector Ref Number - }
traceVector         = $001D; {Vector Ref Number - }
stepVector          = $001E; {Vector Ref Number - }
ctlYVector          = $0028; {Vector Ref Number - }
proDOSVctr          = $002A; {Vector Ref Number - }
osVector            = $002B; {Vector Ref Number - }
msgPtrVctr          = $002C; {Vector Ref Number - }
```

```
TYPE    ClampRecHndl         = ^ClampRecPtr;
        ClampRecPtr          = ^ClampRec;
        ClampRec             =
                RECORD
                    yMaxClamp: Integer;
                    yMinClamp: Integer;
                    xMaxClamp: Integer;
                    xMinClamp: Integer;
                END;

        FWRecHndl            = ^FWRecPtr;
        FWRecPtr             = ^FWRec;
        FWRec                =
                RECORD
                    yRegExit:  Integer;
                    xRegExit:  Integer;
                    aRegExit:  Integer;
                    status:    Integer;
                END;

        MouseRecHndl         = ^MouseRecPtr;
        MouseRecPtr          = ^MouseRec;
        MouseRec             =
                PACKED RECORD
                    mouseMode:   Byte;
                    mouseStatus: Byte;
                    yPos:        Integer;
                    xPos:        Integer;
                END;

        InterruptStateRecHndl     = ^InterruptStateRecPtr;
        InterruptStateRecPtr      = ^InterruptStateRec;
        InterruptStateRec         =
                PACKED RECORD
                    irq_A:       Integer;
                    irq_X:       Integer;
                    irq_Y:       Integer;
                    irq_S:       Integer;
                    irq_D:       Integer;
                    irq_P:       Byte;
                    irq_DB:      Byte;
                    irq_e:       Byte;
                    irq_K:       Byte;
                    irq_PC:      Integer;
                    irq_state:   Byte;
                    irq_shadow:  Integer;
                    irq_mslot:   Byte;
                END;

PROCEDURE MTBootInit;
PROCEDURE MTStartUp;
PROCEDURE MTShutDown;
FUNCTION MTVersion:          Integer;
PROCEDURE MTReset;
FUNCTION MTStatus:           Boolean;
PROCEDURE ClampMouse
            (xMinClamp:      Integer;
             xMaxClamp:      Integer;
```

```
                yMinClamp:    Integer;
                yMaxClamp:    Integer);
PROCEDURE ClearMouse;
PROCEDURE ClrHeartBeat;
PROCEDURE DeleteID
               (idTag:        Integer);
PROCEDURE DelHeartBeat
               (taskPtr:      Ptr);
FUNCTION FWEntry
               (aRegValue:    Integer;
                xRegValue:    Integer;
                yRegValue:    Integer;
                eModeEntryPt: Integer): FWRec;
FUNCTION GetAbsClamp:         ClampRec;
FUNCTION GetAddr
               (refNum:       Integer): Ptr;
FUNCTION GetIRQEnable:        Integer;
FUNCTION GetMouseClamp:       ClampRec;
FUNCTION GetNewID
               (idTag:        Integer): Integer;
FUNCTION GetTick:             Longint;
FUNCTION GetVector
               (vectorRefNum: Integer): Ptr;
PROCEDURE HomeMouse;
PROCEDURE InitMouse
               (mouseSlot:    Integer);
PROCEDURE IntSource
               (srcRefNum:    Integer);
FUNCTION Munger
               (destPtr:      Ptr;
                destLenPtr:   IntPtr;
                targPtr:      Ptr;
                targLen:      Integer;
                replPtr:      Ptr;
                replLen:      Integer;
                padPtr:       Ptr): Integer;
FUNCTION PackBytes
               (VAR srcBuffer:Ptr;
                VAR srcSize:  Integer;
                dstBuffer:    Ptr;
                dstSize:      Integer): Integer;
PROCEDURE PosMouse
               (xPos:         Integer;
                yPos:         Integer);
PROCEDURE ReadAsciiTime
               (bufferPtr:    Ptr);
FUNCTION ReadBParam
               (paramRefNum:  Integer): Integer;
PROCEDURE ReadBRam
               (bufferPtr:    Ptr);
FUNCTION ReadMouse:           MouseRec;
FUNCTION ReadTimeHex:         TimeRec;
FUNCTION ServeMouse:          Integer;
PROCEDURE SetAbsClamp
               (xMinClamp:    Integer;
                xMaxClamp:    Integer;
                yMinClamp:    Integer;
                yMaxClamp:    Integer);
```

```
PROCEDURE SetHeartBeat
               (taskPtr:      Ptr);
PROCEDURE SetMouse
               (mouseMode:    Integer);
PROCEDURE SetVector
               (vectorRefNum: Integer;
                vectorPtr:    Ptr);
PROCEDURE StatusID
               (idTag:        Integer);
PROCEDURE SysBeep;
PROCEDURE SysFailMgr
               (errorCode:    Integer;
                str:          Str255);
FUNCTION UnPackBytes
               (srcBuffer:    Ptr;
                srcSize:      Integer;
                VAR dstBuffer:Ptr;
                VAR dstSize:  Integer): Integer;
PROCEDURE WriteBParam
               (theData:      Integer;
                paramRefNum:  Integer);
PROCEDURE WriteBRam
               (bufferPtr:    Ptr);
PROCEDURE WriteTimeHex
               (month:        Byte;
                day:          Byte;
                curYear:      Byte;
                hour:         Byte;
                minute:       Byte;
                second:       Byte);
PROCEDURE AddToQueue
               (newEntryPtr:  Ptr;
                headerPtr:    Ptr);
PROCEDURE DeleteFromQueue
               (entryPtr:     Ptr;
                headerPtr:    Ptr);
PROCEDURE SetInterruptState
               (iStateRec:    InterruptStateRec;
                bytesDesired: Integer);
PROCEDURE GetInterruptState
               (VAR iStateRec:InterruptStateRec;
                bytesDesired: Integer);
FUNCTION GetIntStateRecSize: Integer;
FUNCTION ReadMouse2:          MouseRec;
FUNCTION GetCodeResConverter: ProcPtr;
FUNCTION GetRomResource:      Ptr;

IMPLEMENTATION
END.
```

NoteSeq

```
{********************************************
; File: NoteSeq.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT NoteSeq;

INTERFACE
USES Types;

CONST   pitchBend          = $0;          {Command - }
        tempo              = $00000001;   {Command - }
        turnNotesOff       = $00000002;   {Command - }
        jump               = $00000003;   {Command - }
        setVibratoDepth    = $00000004;   {Command - }
        programChange      = $00000005;   {Command - }
        setRegister        = $00000006;   {Command - }
        ifGo               = $00000007;   {Command - }
        incRegister        = $00000008;   {Command - }
        decRegister        = $00000009;   {Command - }
        midiNoteOff        = $0000000A;   {Command - }
        midiNoteOn         = $0000000B;   {Command - }
        midiPolyKey        = $0000000C;   {Command - }
        midiCtlChange      = $0000000D;   {Command - }
        midiProgChange     = $0000000E;   {Command - }
        midiChnlPress      = $0000000F;   {Command - }
        midiPitchBend      = $00000010;   {Command - }
        midiSelChnlMode    = $00000011;   {Command - }
        midiSysExclusive   = $00000012;   {Command - }
        midiSysCommon      = $00000013;   {Command - }
        midiSysRealTime    = $00000014;   {Command - }
        midiSetSysExl      = $00000015;   {Command - }
        commandMask        = $0000007F;   {Command - }
        volumeMask         = $0000007F;   {Command - }
        chord              = $00000080;   {Command - }
        vallMask           = $00007F00;   {Command - }
        toneMask           = $00007F00;   {Command - }
        noteMask           = $00008000;   {Command - }
        lByte              = $00FF0000;   {Command - meaning depends on midi command}
        durationMask       = $07FF0000;   {Command - }
        trackMask          = $78000000;   {Command - }
        delayMask          = $80000000;   {Command - }
        hByte              = $FF000000;   {Command - }
        noRoomMidiErr      = $1A00; {error - }
        noCommandErr       = $1A01; {error - can't understand current SeqItem }
        noRoomErr          = $1A02; {error - sequence is more than twelve levels deep }
        startedErr         = $1A03; {error - Note Sequencer is already started}
        noNoteErr          = $1A04; {error - can't find the note to be turned
                                     off by the current SeqItem }
        noStartErr          = $1A05; {error - Note Sequencer not started yet }
        instBndsErr         = $1A06; {error - Instrument number out of
                                       Instrument boundary range }
        nsWrongVer          = $1A07; {error - incompatible versions of
                                       NoteSequencer and oteSynthesizer }

TYPE    LocRecHndl          = ^LocRecPtr;
        LocRecPtr           = ^LocRec;
        LocRec              =
                RECORD
                        curPhraseItem: Integer;
                        curPattItem:   Integer;
                        curLevel:      Integer;
                END;


PROCEDURE SeqBootInit;
PROCEDURE SeqStartUp
                (dPageAddr:            Integer;
                 mode:                 Integer;
                 updateRate:           Integer;
                 increment:            Integer);
PROCEDURE SeqShutDown;
FUNCTION SeqVersion:                   Integer;
PROCEDURE SeqReset;
FUNCTION SeqStatus:                    Boolean;
PROCEDURE SeqAllNotesOff;
FUNCTION ClearIncr:                    Integer;
FUNCTION GetLoc:                       LocRec;
FUNCTION GetTimer:                     Integer;
PROCEDURE SetIncr
                (increment:            Integer);
PROCEDURE SetInstTable
                (instTable:            Handle);
PROCEDURE SetTrkInfo
                (priority:             Integer;
                 instIndex:            Integer;
                 trackNum:             Integer);
PROCEDURE StartInts;
PROCEDURE StartSeq
                (errHndlrRoutine:      VoidProcPtr;
                 compRoutine:          VoidProcPtr;
                 sequence:             Handle);
PROCEDURE StepSeq;
PROCEDURE StopInts;
PROCEDURE StopSeq
                (next: Integer);
PROCEDURE StartSeqRel
                (errHandlerPtr:        ProcPtr;
                 compRoutine:          ProcPtr;
                 sequence:             Handle);

IMPLEMENTATION
END.
```

NoteSyn

```
{*********************************************
; File: NoteSyn.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
*********************************************}

UNIT NoteSyn;

INTERFACE
USES Types;

CONST  nsAlreadyInit      = $1901;    {error - Note Syn already initialized }
       nsSndNotInit       = $1902;    {error - Sound Tools not initialized }
       nsNotAvail         = $1921;    {error - generator not available }
       nsBadGenNum        = $1922;    {error - bad generator number }
       nsNotInit          = $1923;    {error - Note Syn not initialized }
       nsGenAlreadyOn     = $1924;    {error - generator already on }
       soundWrongVer      = $1925;    {error - incompatible versions of Sound and NoteSyn}

TYPE   EnvelopeHndl       = ^EnvelopePtr;
       EnvelopePtr        = ^Envelope;
       Envelope           =
              PACKED RECORD
                  st1BkPt:            Byte;
                  st1Increment:       Integer;
                  st2BkPt:            Byte;
                  st2Increment:       Integer;
                  st3BkPt:            Byte;
                  st3Increment:       Integer;
                  st4BkPt:            Byte;
                  st4Increment:       Integer;
                  st5BkPt:            Byte;
                  st5Increment:       Integer;
                  st6BkPt:            Byte;
                  st6Increment:       Integer;
                  st7BkPt:            Byte;
                  st7Increment:       Integer;
                  st8BkPt:            Byte;
                  st8Increment:       Integer;
              END;

           WaveFormHndl             = ^WaveFormPtr;
           WaveFormPtr              = ^WaveForm;
           WaveForm                 =
              PACKED RECORD
                  wfTopKey:           Byte;
                  wfWaveAddress:      Byte;
                  wfWaveSize:         Byte;
                  wfDocMode:          Byte;
                  wfRelPitch:         Integer;
              END;
```

```
        InstrumentHndl              = ^InstrumentPtr;
        InstrumentPtr               = ^Instrument;
        Instrument                  =
           PACKED RECORD
              theEnvelope:           Envelope;
              releaseSegment:          Byte;
              priorityIncrement:     Byte;
              pitchBendRange:          Byte;
              vibratoDepth:            Byte;
              vibratoSpeed:            Byte;
              inSpare:               Byte;
              aWaveCount:              Byte;
              bWaveCount:              Byte;
              aWaveList:               ARRAY [1..1] OF WaveForm;
              bWaveList:               ARRAY [1..1] OF WaveForm;
           END;

PROCEDURE NSBootInit;
PROCEDURE NSStartUp
              (updateRate:          Integer;
               userUpdateRtnPtr:    Ptr);
PROCEDURE NSShutDown;
FUNCTION NSVersion:                 Integer;
PROCEDURE NSReset;
FUNCTION NSStatus:                  Boolean;
PROCEDURE AllNotesOff;
FUNCTION AllocGen
              (requestPriority:     Integer): Integer;
PROCEDURE DeallocGen
              (genNumber:           Integer);
PROCEDURE NoteOff
              (genNumber:           Integer;
               semitone:            Integer);
PROCEDURE NoteOn
              (genNumber:           Integer;
               semitone:            Integer;
               volume:              Integer;
               InstrumentPtr:       Ptr);
PROCEDURE NSSetUpdateRate
              (updateRate:          Integer);
FUNCTION NSSetUserUpdateRtn
              (updateRtn:           VoidProcPtr): VoidProcPtr;

IMPLEMENTATION
END.
```

Print

```
{********************************************
; File: Print.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Print;

INTERFACE
USES Types, QuickDraw, Events, Controls, Windows, Dialogs;

CONST  pntrConFailed      = $1308;    {error - connection to the printer failed }
       memFullErr         = $FF80;    {errors - }
       ioAbort            = $FFE5;    {errors - }
       prAbort            = $0080;    {errors - }
       missingDriver      = $1301;    {errors - specified driver not in system/drivers }
       portNotOn          = $1302;    {errors - specified port not selected in ctl panel }
       noPrintRecord      = $1303;    {errors - no print record was given }
       badLaserPrep       = $1304;    {errors - laser prep in laser writer incompatible }
       badLPFile          = $1305;    {errors - laser prep in system/drivers
                                      incompatible }
       papConnNotOpen     = $1306;    {errors - cannot connect to laser writer }
       papReadWriteErr    = $1307;    {errors - apple talk PAPRead or PAPWrite error }
       startUpAlreadyMade = $132;     {errors - low level startup already made }
       invalidCtlVal      = $1322;    {errors - invalid control value had been spec'd }
       reset              = $0001;    {LLDControl - Printer control value - reset printer }
       formFeed           = $0002;    {LLDControl - Printer control value - form feed }
       lineFeed           = $0003;    {LLDControl - Printer control value - line feed }
       bothDrivers        = $0;       {whichDriver - input to PMLoadDriver and PMUnloadDriver }
       printerDriver      = $0001;    {whichDriver - input to PMLoadDriver and PMUnloadDriver }
       portDriver         = $0002;    {whichDriver - input to PMLoadDriver and PMUnloadDriver }
       prPortrait         = $0000;    { - }
       prLandscape        = $0001;    { - }
       prImageWriter      = $0001;    { - }
       prImageWriterLQ    = $0002;    { - }
       prLaserWriter      = $0003;    { - }
       prEpson            = $0004;    { - }
       prBlackWhite       = $0001;    { - }
       prColor            = $0002;    { - }
       bDraftLoop         = $0000;    { - }
       bSpoolLoop         = $0080;    { - }

TYPE   PrPrinterSpecRec =
              RECORD
                     prPrinterType:        Integer;
                     prCharacteristics:    Integer;
              END;

       PrInfoRecHndl      = ^PrInfoRecPtr;
       PrInfoRecPtr       = ^PrInfoRec;
```

```
PrInfoRec       =
    RECORD
        iDev:       Integer; { reserved for internal use }
        iVRes:      Integer; { vertical resolution of printer }
        iHRes:      Integer; { horizontal resolution of printer }
        rPage:      Rect; { defining page rectangle }
    END;

PrJobRecPtr     = ^PrJobRec;
PrJobRec        =
    PACKED RECORD
        iFstPage:   Integer; { first page to print }
        iLstPage:   Integer; { last page to print }
        iCopies:    Integer; { number of copies }
        bJDocLoop:  Byte; { printing method }
        fFromUser:  Byte; { used internally }
        pIdleProc:  WordProcPtr; { background procedure }
        pFileName:  Ptr; { spool file name }
        iFileVol:   Integer; { spool file volume reference number }
        bFileVers:  Byte; { spool file version number }
        bJobX:      Byte; { used internally }
    END;

PrStyleRecHndl          = ^PrStyleRecPtr;
PrStyleRecPtr           = ^PrStyleRec;
PrStyleRec              =
    RECORD
        wDev:       Integer; { output quality information }
        internA:    ARRAY [1..3] OF Integer; { for internal use }
        feed:       Integer; { paper feed type }
        paperType:  Integer; { paper type }
        crWidth:    Integer; { carriage Width for image writer or
                               vertical sizing for lazer writer }
        reduction:  Integer; { % reduction, laser writer only }
        internB:    Integer; { for internal use }
    END;

PrRecHndl               = ^PrRecPtr;
PrRecPtr                = ^PrRec;
PrRec                   =
    RECORD
        prVersion:  Integer;        { print manager version }
        prInfo:     PrInfoRec;      { printer infomation subrecord }
        rPaper:     Rect;           { Defining paper rectangle }
        prStl:      PrStyleRec;     { style subrecord }
        prInfoPT:   PACKED ARRAY [1..14] OF Byte; {reserved for internal use}
        prXInfo:    PACKED ARRAY [1..24] OF Byte; {reserved for internal use}
        prJob:      PrJobRec;       { job subrecord }
        printX:     PACKED ARRAY [1..38] OF Byte; {reserved for future use}
        iReserved:  Integer;        { reserved for internal use }
    END;

PrStatusRecHndl         = ^PrStatusRecPtr;
PrStatusRecPtr          = ^PrStatusRec;
PrStatusRec             =
    RECORD
        iTotPages:  Integer;        { number of pages in spool file }
        iCurPage:   Integer;        { page being printed }
```

```
                iTotCopies:    Integer;       { number of copies requested }
                iCurCopy:      Integer;       { copy being printed }
                iTotBands:     Integer;       { reserved for internal use }
                iCurBand:      Integer;       { reserved for internal use }
                fPgDirty:      Integer;       { TRUE if started printing page }
                fImaging:      Integer;       { reserved for internal use }
                hPrint:        PrRecHndl;     { handle of print record }
                pPrPort:       GrafPortPtr;   { pointer to grafport being use for
                                              printing }
                hPic:          Longint;       { reserved for internal use }
            END;

PROCEDURE PMBootInit;
PROCEDURE PMStartUp
              (userID:                Integer;
               dPageAddr:             Integer);
PROCEDURE PMShutDown;
FUNCTION PMVersion:                   Integer;
PROCEDURE PMReset;
FUNCTION PMStatus:                    Boolean;
PROCEDURE LLDBitMap
              (bitMapPtr:             Ptr;
               rectPtr:               Rect;
               userID:                Integer);
PROCEDURE LLDControl
              (printerControlValue: Integer);
PROCEDURE LLDShutDown
              (userID:                Integer);
PROCEDURE LLDStartUp
              (dPageAddr:             Integer;
               userID:                Integer);
PROCEDURE LLDText
              (textPtr:               Ptr;
               textLength:            Integer;
               userID:                Integer);
PROCEDURE PMLoadDriver
              (whichDriver:           Integer);
PROCEDURE PMUnloadDriver
              (whichDriver:           Integer);
FUNCTION PrChoosePrinter:             Boolean;
FUNCTION PrChooser:                   Boolean;
PROCEDURE PrCloseDoc
              (printGrafPortPtr:      GrafPortPtr);
PROCEDURE PrClosePage
              (printGrafPortPtr:      GrafPortPtr);
PROCEDURE PrDefault
              (printRecordHandle:     PrRecHndl);
FUNCTION PrDriverVer:                 Integer;
FUNCTION PrError:                     Integer;
FUNCTION PrJobDialog
              (printRecordHandle:     PrRecHndl): Boolean;
FUNCTION PrOpenDoc
              (printRecordHandle:     PrRecHndl;
               printGrafPortPtr:      GrafPortPtr): GrafPortPtr;
PROCEDURE PrOpenPage
              (printGrafPortPtr:      GrafPortPtr;
               pageFramePtr:          rectPtr);
```

```
PROCEDURE PrPicFile
          (printRecordHandle:   PrRecHndl;
           printGrafPortPtr:    GrafPortPtr;
           statusRecPtr:        PrStatusRecPtr);
PROCEDURE PrPixelMap
          (srcLocPtr:           LocInfoPtr;
           srcRectPtr:          Rect;
           colorFlag:           Integer);
FUNCTION PrPortVer: Integer;
PROCEDURE PrSetError
          (errorNumber:         Integer);
FUNCTION PrStlDialog
          (printRecordHandle:   PrRecHndl): Boolean;
FUNCTION PrValidate
          (printRecordHandle:   PrRecHndl): Boolean;
PROCEDURE PrSetDocName
          (DocNamePtr:          StringPtr);
FUNCTION PrGetDocName:          StringPtr;
FUNCTION PrGetPgOrientation
          (prRecordHandle:      PrRecHndl): Integer;
FUNCTION PrGetPrinterSpecs:     PrPrinterSpecRec;
PROCEDURE PrGetZoneName
          (VAR ZoneNamePtr:     Str255);
PROCEDURE PrGetPrinterDvrName
          (VAR DvrNamePtr:      Str255);
PROCEDURE PrGetPortDvrName
          (VAR DvrNamePtr:      Str255);
PROCEDURE PrGetUserName
          (VAR UserNamePtr:     Str255);
PROCEDURE PrGetNetworkName
          (VAR NetworkNamePtr:  Str255);

IMPLEMENTATION
END.
```

QDAux

```
{********************************************
; File: QDAux.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT QDAux;

INTERFACE
        USES Types;

CONST   frameVerb                  = $00;    {PicInfo - PRIVATE - for reference only }
        picNop                     = $00;    {PicInfo - PRIVATE - for reference only }
        drawCharVerb               = $00;    {PicInfo - PRIVATE - for reference only }
        paintVerb              = $01;      {PicInfo - PRIVATE - for reference only }
        picClipRgn             = $01;      {PicInfo - PRIVATE - for reference only }
        drawTextVerb           = $01;      {PicInfo - PRIVATE - for reference only }
        eraseVerb              = $02;      {PicInfo - PRIVATE - for reference only }
        picBkPat               = $02;      {PicInfo - PRIVATE - for reference only }
        drawCStrVerb           = $02;      {PicInfo - PRIVATE - for reference only }
        invertVerb             = $03;      {PicInfo - PRIVATE - for reference only }
        picTxFont              = $03;      {PicInfo - PRIVATE - for reference only }
        fillVerb               = $04;      {PicInfo - PRIVATE - for reference only }
        picTxFace              = $04;      {PicInfo - PRIVATE - for reference only }
        picTxMode              = $05;      {PicInfo - PRIVATE - for reference only }
        picSpExtra             = $06;      {PicInfo - PRIVATE - for reference only }
        picPnSize              = $07;      {PicInfo - PRIVATE - for reference only }
        picPnMode              = $08;      {PicInfo - PRIVATE - for reference only }
        picPnPat               = $09;      {PicInfo - PRIVATE - for reference only }
        picThePat              = $0A;      {PicInfo - PRIVATE - for reference only }
        picOvSize              = $0B;      {PicInfo - PRIVATE - for reference only }
        picOrigin              = $0C;      {PicInfo - PRIVATE - for reference only }
        picTxSize              = $0D;      {PicInfo - PRIVATE - for reference only }
        picFGColor             = $0E;      {PicInfo - PRIVATE - for reference only }
        picBGColor             = $0F;      {PicInfo - PRIVATE - for reference only }
        picTxRatio             = $10;      {PicInfo - PRIVATE - for reference only }
        picVersion             = $11;      {PicInfo - PRIVATE - for reference only }
        lineNoun               = $20;      {PicInfo - PRIVATE - for reference only }
        picLine                = $20;      {PicInfo - PRIVATE - for reference only }
        picLineFrom            = $21;      {PicInfo - PRIVATE - for reference only }
        picShortL              = $22;      {PicInfo - PRIVATE - for reference only }
        picShortLFrom          = $23;      {PicInfo - PRIVATE - for reference only }
        picLongText            = $28;      {PicInfo - PRIVATE - for reference only }
        picDHText              = $29;      {PicInfo - PRIVATE - for reference only }
        picDVText              = $2A;      {PicInfo - PRIVATE - for reference only }
        picDVDHText            = $2B;      {PicInfo - PRIVATE - for reference only }
        rectNoun               = $30;      {PicInfo - PRIVATE - for reference only }
        rRectNoun              = $40;      {PicInfo - PRIVATE - for reference only }
        ovalNoun               = $50;      {PicInfo - PRIVATE - for reference only }
        arcNoun                = $60;      {PicInfo - PRIVATE - for reference only }
        polyNoun               = $70;      {PicInfo - PRIVATE - for reference only }
        rgnNoun                = $80;      {PicInfo - PRIVATE - for reference only }
        mapNoun                = $90;      {PicInfo - PRIVATE - for reference only }
        picBitsRect            = $90;      {PicInfo - PRIVATE - for reference only }
        picBitsRgn             = $91;      {PicInfo - PRIVATE - for reference only }
        picPBitsRect           = $98;      {PicInfo - PRIVATE - for reference only }
        picPBitsRgn            = $99;      {PicInfo - PRIVATE - for reference only }
        picShortComment        = $A0;      {PicInfo - PRIVATE - for reference only }
        picLongComment         = $A1;      {PicInfo - PRIVATE - for reference only }
        picEnd                 = $FF;      {PicInfo - PRIVATE - for reference only }
        resMode640PMask        = $00;      {SeedFill/CalcMask - }
        resMode640DMask        = $01;      {SeedFill/CalcMask - }
        resMode320Mask         = $02;      {SeedFill/CalcMask - }
        destModeCopyMask       = $0000;    {SeedFill/CalcMask - }
        destModeLeaveMask      = $1000;    {SeedFill/CalcMask - }
        destModeOnesMask       = $2000;    {SeedFill/CalcMask - }
        destModeZerosMask      = $3000;    {SeedFill/CalcMask - }
        destModeError          = $1212;    {Error - }

TYPE    QDIconRecordHndl       = ^QDIconRecordPtr;
        QDIconRecordPtr        = ^QDIconRecord;
        QDIconRecord           =
         RECORD
             iconType:      Integer;
             iconSize:      Integer;
             iconHeight:    Integer;
             iconWidth:     Integer;
             iconImage:     PACKED ARRAY [1..1] OF Byte;
             iconMask:      PACKED ARRAY [1..1] OF Byte;
         END;

     PicHndl                    = ^PicPtr;
     PicPtr                     = ^Picture;
     Picture                    =
         RECORD
             picSCB:        Integer;
             picFrame:      Rect; { Followed by picture opcodes }
         END;

PROCEDURE QDAuxBootInit;
PROCEDURE QDAuxStartUp;
PROCEDURE QDAuxShutDown;
FUNCTION  QDAuxVersion:            Integer;
PROCEDURE QDAuxReset;
FUNCTION  QDAuxStatus:             Boolean;
PROCEDURE CopyPixels
              (srcLocPtr:          LocInfo;
               destLocPtr:         LocInfo;
               srcRect:            Rect;
               destRect:           Rect;
               xferMode:           Integer;
               makeRgn:            RgnHandle);
PROCEDURE DrawIcon
              (iconPtr:            QDIconRecord;
               displayMode:        Integer;
               xPos:               Integer;
               yPos:               Integer);
PROCEDURE SpecialRect
              (rectPtr:            Rect;
               frameColor:         Integer;
               fillColor:          Integer);
PROCEDURE WaitCursor;
PROCEDURE SeedFill
              (srcLocInfoPtr:      LocInfo;
               srcRect:            Rect;
               dstLocInfoPtr:      LocInfo;
               dstRect:            Rect;
               seedH:              Integer;
               seedV:              Integer;
               resMode:            Integer;
               __patternPtr:       PatternPtr;
               leakTblPtr:         Ptr);
PROCEDURE CalcMask
              (srcLocInfoPtr:      LocInfo;
               srcRect:            Rect;
               dstLocInfoPtr:      LocInfo;
               dstRect:            Rect;
               resMode:            Integer;
               __patternPtr:       PatternPtr;
               leakTblPtr:         Ptr);
```

```
PROCEDURE PicComment
              (kind:                 Integer;
               dataSize:             Integer;
               dataHandle:           Handle);
PROCEDURE ClosePicture;
PROCEDURE DrawPicture
              (picHandle:            PicHndl;
               destRect:             Rect);
PROCEDURE KillPicture
              (picHandle:            PicHndl);
FUNCTION  OpenPicture
              (picFrame:             Rect): PicHndl;

IMPLEMENTATION
END.
```

QuickDraw

```
{********************************************
; File: QuickDraw.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Quickdraw;

INTERFACE
USES Types;

CONST alreadyInitialized        = $0401;   {error - Quickdraw already initialized }
      cannotReset               = $0402;   {error - never used }
      notInitialized            = $0403;   {error - Quickdraw not initialized }
      screenReserved            = $0410;   {error - screen reserved }
      badRect                   = $0411;   {error - bad rectangle }
      notEqualChunkiness        = $0420;   {error - Chunkiness is not equal }
      rgnAlreadyOpen            = $0430;   {error - region is already open }
      rgnNotOpen                = $0431;   {error - region is not open }
      rgnScanOverflow           = $0432;   {error - region scan overflow }
      rgnFull                   = $0433;   {error - region is full }
      polyAlreadyOpen           = $0440;   {error - poly is already open }
      polyNotOpen               = $0441;   {error - poly is not open }
      polyTooBig                = $0442;   {error - poly is too big }
      badTableNum               = $0450;   {error - bad table number }
      badColorNum               = $0451;   {error - bad color number }
      badScanLine               = $0452;   {error - bad scan line }
      notImplemented            = $04FF;   {error - not implemented }
      tsNumber                  = $04;     { - }
      _colorTable               = $0F;     {AnSCBByte - Mask for SCB color table }
      scbReserved               = $10;     {AnSCBByte - Mask for SCB reserved bit }
      scbFill                   = $20;     {AnSCBByte - Mask for SCB fill bit }
```

```
scbInterrupt            = $40;     {AnSCBByte - Mask for SCB interrupt bit }
scbColorMode            = $80;     {AnSCBByte - Mask for SCB color mode bit }
table320                = $32;     {ColorData - (val=size) }
table640                = $32;     {ColorData - (val=size) }
blueMask                = $000F;   {ColorValue - Mask for Blue nibble }
greenMask               = $00F0;   {ColorValue - Mask for green nibble }
redMask                 = $0F00;   {ColorValue - Mask for red nibble }
widMaxSize              = $0001;   {FontFlags - }
zeroSize                = $0002;   {FontFlags - }
maskSize                = $08;     {GrafPort - Mask Size (val=size) }
locSize                 = $10;     {GrafPort - Loc Size (val=size) }
patsize                 = $20;     {GrafPort - Pattern Size (val=size) }
pnStateSize             = $32;     {GrafPort - Pen State Size (Val=size) }
portSize                = $AA;     {GrafPort - Size of GrafPort }
black                   = $000;    {MasterColors - These work in 320 and 640 mode }
blue                    = $00F;    {MasterColors - These work in 320 and 640 mode }
darkGreen320            = $080;    {MasterColors - These work in 320 mode }
green320                = $0E0;    {MasterColors - These work in 320 mode }
green640                = $0F0;    {MasterColors - These work in 640 mode }
lightBlue320            = $4DF;    {MasterColors - These work in 320 mode }
purple320               = $72C;    {MasterColors - These work in 320 mode }
darkGray320             = $777;    {MasterColors - These work in 320 mode }
periwinkleBlue320       = $78F;    {MasterColors - These work in 320 mode }
brown320                = $841;    {MasterColors - These work in 320 mode }
lightGray320            = $0CCC;   {MasterColors - These work in 320 mode }
red320                  = $0D00;   {MasterColors - These work in 320 mode }
lilac320                = $0DAF;   {MasterColors - These work in 320 mode }
red640                  = $0F00;   {MasterColors - These work in 640 mode }
orange320               = $0F70;   {MasterColors - These work in 320 mode }
flesh320                = $0FA9;   {MasterColors - These work in 320 mode }
yellow                  = $0FF0;   {MasterColors - These work in 320 and 640 mode }
white                   = $0FFF;   {MasterColors - These work in 320 and 640 mode }
modeCopy                = $0000;   {PenModeDATA - }
modeOR                  = $0001;   {PenModeDATA - }
modeXOR                 = $0002;   {PenModeDATA - }
modeBIC                 = $0003;   {PenModeDATA - }
modeForeCopy            = $0004;   {PenModeDATA - }
modeForeOR              = $0005;   {PenModeDATA - }
modeForeXOR             = $0006;   {PenModeDATA - }
modeForeBIC             = $0007;   {PenModeDATA - }
modeNOT                 = $8000;   {PenModeDATA - }
notCopy                 = $8000;   {PenModeDATA - }
notOR                   = $8001;   {PenModeDATA - }
notXOR                  = $8002;   {PenModeDATA - }
notBIC                  = $8003;   {PenModeDATA - }
notForeCOPY             = $8004;   {PenModeDATA - }
notForeOR               = $8005;   {PenModeDATA - }
notForeXOR              = $8006;   {PenModeDATA - }
notForeBIC              = $8007;   {PenModeDATA - }
mode320                 = $0000;   {QDStartup - Argument to QDStartup }
mode640                 = $0080;   {QDStartup - Argument to QDStartup }
plainMask               = $0000;   {TextStyle - Mask for plain text bit }
boldMask                = $0001;   {TextStyle - Mask for bold bit }
italicMask              = $0002;   {TextStyle - Mask for italic bit }
underlineMask           = $0004;   {TextStyle - Mask for underline bit }
outlineMask             = $0008;   {TextStyle - Mask for outline bit }
shadowMask              = $0010;   {TextStyle - Mask for shadow bit }
```

```
TYPE    TextStyle       = Integer;
        ColorValue      = Integer;
        AnSCBByte       = Byte;
        PatternPtr      = ^Pattern;
        Pattern         = PACKED ARRAY [1..32] OF Byte;
        Mask            = PACKED ARRAY [1..8] OF Byte;
        CursorHndl      = ^CursorPtr;
        CursorPtr       = ^Cursor;
        Cursor          =
            RECORD
                cursorHeight:   Integer;     { size in bytes }
                cursorWidth:    Integer;     { enclosing rectangle }
                cursorData:     ARRAY [1..1, 1..1] OF Integer;
                cursorMask:     ARRAY [1..1, 1..1] OF Integer;
                cursorHotSpot: Point;
            END;

             RgnHandle = ^RgnPtr;
             RgnPtr    = ^Region;
             Region    =
            RECORD
                rgnSize:       Integer;    { size in bytes }
                rgnBBox:       Rect;          { enclosing rectangle }
            END;

        BufDimRecHndl       = ^BufDimRecPtr;
        BufDimRecPtr        = ^BufDimRec;
        BufDimRec           =
            RECORD
                maxWidth:                 Integer;
                textBufHeight:            Integer;
                textBufferWords:          Integer;
                fontWidth:                Integer;
            END;

        FontHndl            = ^FontPtr;
        FontPtr             = ^Font;
        Font                =
            RECORD
                offseToMF:      Integer; { fully defined front of the Font record. }
                family:         Integer;
                style:          TextStyle;
                size:           Integer;
                version:        Integer;
                fbrExtent:      Integer;
            END;

        FontGlobalsRecHndl          = ^FontGlobalsRecPtr;
        FontGlobalsRecPtr           = ^FontGlobalsRecord;
        FontGlobalsRecord           =
            RECORD
                fgFontID:      Integer; { currently 12 bytes long, but may be expanded }
                fgStyle:       TextStyle;
                fgSize:        Integer;
                fgVersion:     Integer;
                fgWidMax:      Integer;
                fgFBRExtent:   Integer;
            END;
```

```
FontIDHndl          = ^FontIDPtr;
FontIDPtr           = ^FontID;
FontID              =
    PACKED RECORD
        famNum:         Integer;
        fontStyle:      Byte;
        fontSize:       Byte;
    END;
FontInfoRecHndl     = ^FontInfoRecPtr;
FontInfoRecPtr      = ^FontInfoRecord;
FontInfoRecord      =
    RECORD
        ascent:         Integer;
        descent:        Integer;
        widMax:         Integer;
        leading:        Integer;
    END;

LocInfoHndl       = ^LocInfoPtr;
LocInfoPtr        = ^LocInfo;
LocInfo           =
    RECORD
        portSCB:          Integer;      { SCBByte in low byte }
        ptrToPixImage:    Ptr;          { ImageRef }
        width:            Integer;      { Width }
        boundsRect:       Rect;         { BoundsRect }
    END;
QDProcsHndl       = ^QDProcsPtr;
QDProcsPtr        = ^QDProcs;
QDProcs           =
    RECORD
        stdText:          VoidProcPtr;
        stdLine:          VoidProcPtr;
        stdRect:          VoidProcPtr;
        stdRRect:         VoidProcPtr;
        stdOval:          VoidProcPtr;
        stdArc:           VoidProcPtr;
        stdPoly:          VoidProcPtr;
        stdRgn:           VoidProcPtr;
        stdPixels:        VoidProcPtr;
        stdComment:       VoidProcPtr;
        stdTxMeas:        VoidProcPtr;
        stdTxBnds:        VoidProcPtr;
        stdGetPic:        VoidProcPtr;
        stdPutPic:        VoidProcPtr;
    END;

GrafPortHndl      = ^GrafPortPtr;
GrafPortPtr       = ^GrafPort;
GrafPort          =
    RECORD
        portInfo:         LocInfo;
        portRect:         Rect;         { PortRect }
        clipRgn:          RgnHandle;    { Clip Rgn. Pointer }
        visRgn:           RgnHandle;    { Vis. Rgn. Pointer }
        bkPat:            Pattern;      { BackGround Pattern }
        pnLoc:            Point;        { Pen Location }
        pnSize:          Point;       { Pen Size }
        pnMode:          Integer;     { Pen Mode }
        pnPat:           Pattern;     { Pen Pattern }
        pnMask:          Mask;        { Pen Mask }
        pnVis:           Integer;     { Pen Visable }
        fontHandle:      FontHndl;
        FontID:          FontID;      { Font ID }
        fontFlags:       Integer;     { FontFlags }
        txSize:          Integer;     { Text Size }
        txFace:          TextStyle;   { Text Face }
        txMode:          Integer;     { Text Mode }
        spExtra:         Fixed;       { Fixed Point Value }
        chExtra:         Fixed;       { Fixed Point Value }
        fgColor:         Integer;     { ForeGround Color }
        bgColor:         Integer;     { BackGround Color }
        picSave:         Handle;      { PicSave }
        rgnSave:         Handle;      { RgnSave }
        polySave:        Handle;      { PolySave }
        grafProcs:       QDProcsPtr;
        arcRot:          Integer;     { ArcRot }
        userField:       Longint;     { UserField }
        sysField:        Longint;     { SysField }
    END;

PaintParamHndl  = ^PaintParamPtr;
PaintParamPtr   = ^PaintParam;
PaintParam      =
    RECORD
        ptrToSourceLocInfo:    LocInfoPtr;
        ptrToDestLocInfo:      LocInfoPtr;
        ptrToSourceRect:       RectPtr;
        ptrToDestPoint:        PointPtr;
        mode:                  Integer;
        maskHandle:            Handle;       { clip region }
    END;

PenStateHndl      = ^PenStatePtr;
PenStatePtr       = ^PenState;
PenState          =
    RECORD
        psPnSize:      Point;
        psPnMode:      Integer;
        psPnPat:       Pattern;
        psPnMask:      Mask;
    END;

RomFontRecHndl      = ^RomFontRecPtr;
RomFontRecPtr       = ^RomFontRec;
RomFontRec          =
    RECORD
        rfFamNum:       Integer;
        rfFamStyle:     Integer;
        rfSize:         Integer;
        rfFontHandle:   FontHndl;
        rfNamePtr:      Ptr;
        rfFBRExtent:    Integer;
    END;
```

```
        ColorTableHndl      = ^ColorTablePtr;
        ColorTablePtr       = ^ColorTable;
        ColorTable          =
          RECORD
            entries:        ARRAY [1..16] OF Integer;
          END;

PROCEDURE QDBootInit;
PROCEDURE QDStartUp
               (dPageAddr:          Integer;
                masterSCB:          Integer;
                maxWidth:           Integer;
                userID:             Integer);
PROCEDURE QDShutDown;
FUNCTION QDVersion:                 Integer;
PROCEDURE QDReset;
FUNCTION QDStatus:                  Boolean;
PROCEDURE AddPt
               (VAR srcPtPtr:       Point;
                VAR destPtPtr:      Point);
PROCEDURE CharBounds
               (theChar:            CHAR;
                VAR resultPtr:      Rect);
FUNCTION CharWidth
               (theChar: CHAR):     Integer;
PROCEDURE ClearScreen
               (colorWord:          Integer);
PROCEDURE ClipRect
               (RectPtr:            Rect);
PROCEDURE ClosePoly;
PROCEDURE ClosePort
               (portPtr:            GrafPortPtr);
PROCEDURE CloseRgn
               (aRgnHandle:         RgnHandle);
PROCEDURE CopyRgn
               (srcRgnHandle:       RgnHandle;
                destRgnHandle:      RgnHandle);
PROCEDURE CStringBounds
               (cStringPtr:         Ptr;
                VAR resultRect:     Rect);
FUNCTION CStringWidth
               (cStringPtr:         Ptr): Integer;
PROCEDURE DiffRgn
               (rgn1Handle:         RgnHandle;
                rgn2Handle:         RgnHandle;
                diffRgnHandle:      RgnHandle);
PROCEDURE DisposeRgn
               (aRgnHandle:         RgnHandle);
PROCEDURE DrawChar
               (theChar:            CHAR);
PROCEDURE DrawCString
               (cStrPtr:            cStringPtr);
PROCEDURE DrawString
               (str:                Str255);
PROCEDURE DrawText
               (textPtr:            Ptr;
                textLength:         Integer);
FUNCTION NotEmptyRect
```

```
               (RectPtr:            Rect): Boolean;
FUNCTION EmptyRgn
               (aRgnHandle:         RgnHandle): Boolean;
FUNCTION EqualPt
               (VAR point1Ptr:      Point;
                VAR point2Ptr:      Point): Boolean;
FUNCTION EqualRect
               (rect1Ptr:           Rect;
                rect2Ptr:           Rect): Boolean;
FUNCTION EqualRgn
               (rgn1Handle:         RgnHandle;
                rgn2Handle:         RgnHandle): Boolean;
PROCEDURE EraseArc
               (RectPtr:            Rect;
                startAngle:         Integer;
                arcAngle:           Integer);
PROCEDURE EraseOval
               (RectPtr:            Rect);
PROCEDURE ErasePoly
               (polyHandle:         Handle);
PROCEDURE EraseRect
               (RectPtr:            Rect);
PROCEDURE EraseRgn
               (aRgnHandle:         RgnHandle);
PROCEDURE EraseRRect
               (RectPtr:            Rect;
                ovalWidth:          Integer;
                ovalHeight:         Integer);
PROCEDURE FillArc
               (RectPtr:            Rect;
                startAngle:         Integer;
                arcAngle:           Integer;
                PatternPtr:         Pattern);
PROCEDURE FillOval
               (RectPtr:            Rect;
                PatternPtr:         Pattern);
PROCEDURE FillPoly
               (polyHandle:         Handle;
                PatternPtr:         Pattern);
PROCEDURE FillRect
               (RectPtr:            Rect;
                PatternPtr:         Pattern);
PROCEDURE FillRgn
               (aRgnHandle:         RgnHandle;
                PatternPtr:         Pattern);
PROCEDURE FillRRect
               (RectPtr:            Rect;
                ovalWidth:          Integer;
                ovalHeight:         Integer;
                PatternPtr:         Pattern);
PROCEDURE ForceBufDims
               (maxWidth:           Integer;
                maxFontHeight:      Integer;
                maxFBRExtent:       Integer);
PROCEDURE FrameArc
               (RectPtr:            Rect;
                startAngle:         Integer;
                arcAngle:           Integer);
```

```
PROCEDURE FrameOval
               (RectPtr:           Rect);
PROCEDURE FramePoly
               (polyHandle:        Handle);
PROCEDURE FrameRect
               (RectPtr:           Rect);
PROCEDURE FrameRgn
               (aRgnHandle:         RgnHandle);
PROCEDURE FrameRRect
               (RectPtr:            Rect;
                ovalWidth:          Integer;
                ovalHeight:         Integer);
FUNCTION GetAddress
               (tableID:            Integer): Ptr;
FUNCTION GetArcRot:                 Integer;
FUNCTION GetBackColor:              Integer;
PROCEDURE GetBackPat
               (VAR PatternPtr:     Pattern);
FUNCTION GetCharExtra:              Fixed;
PROCEDURE GetClip
               (aRgnHandle:         RgnHandle);
FUNCTION GetClipHandle:             RgnHandle;
FUNCTION GetColorEntry
               (tableNumber:        Integer;
                entryNumber:        Integer): Integer;
PROCEDURE GetColorTable
               (tableNumber:       Integer;
                VAR destTablePtr:  ColorTable);
FUNCTION GetCursorAdr:             CursorPtr;
FUNCTION GetFGSize:                Integer;
FUNCTION GetFont:                  FontHndl;
FUNCTION GetFontFlags:             Integer;
PROCEDURE GetFontGlobals
               (VAR fgRecPtr:      FontGlobalsRecord);
FUNCTION GetFontID:                FontID;
PROCEDURE GetFontInfo
               (VAR FontInfoRecPtr:  FontInfoRecord);
FUNCTION GetFontLore
               (VAR recordPtr:     FontGlobalsRecord;
                recordSize:        Integer): Integer;
FUNCTION GetForeColor:             Integer;
FUNCTION GetGrafProcs:             QDProcsPtr;
FUNCTION GetMasterSCB:             Integer;
PROCEDURE GetPen
               (VAR PointPtr:      Point);
PROCEDURE GetPenMask
               (VAR maskPtr:       Mask);
FUNCTION GetPenMode:               Integer;
PROCEDURE GetPenPat
               (VAR PatternPtr:    Pattern);
PROCEDURE GetPenSize
               (VAR PointPtr:      Point);
PROCEDURE GetPenState
               (VAR __penStatePtr: PenState);
FUNCTION GetPicSave: Longint;
FUNCTION GetPixel
               (h:                 Integer;
                v:                 Integer): Integer;
```

```
FUNCTION GetPolySave:           Longint;
FUNCTION GetPort:               GrafPortPtr;
PROCEDURE GetPortLoc
               (VAR LocInfoPtr:  LocInfo);
PROCEDURE GetPortRect
               (VAR __rectPtr:   Rect);
FUNCTION GetRgnSave:            Longint;
PROCEDURE GetROMFont
               (VAR recordPtr:   RomFontRec);
FUNCTION GetSCB
               (scanLine:        Integer): Integer;
FUNCTION GetSpaceExtra:         Fixed;
FUNCTION GetStandardSCB:        Integer;
FUNCTION GetSysField:           Longint;
FUNCTION GetSysFont:            FontHndl;
FUNCTION GetTextFace:           TextStyle;
FUNCTION GetTextMode:           Integer;
FUNCTION GetTextSize:           Integer;
FUNCTION GetUserField:          Longint;
FUNCTION GetVisHandle:          RgnHandle;
PROCEDURE GetVisRgn
               (aRgnHandle:      RgnHandle);
PROCEDURE GlobalToLocal
               (VAR PointPtr:    Point);
PROCEDURE GrafOff;
PROCEDURE GrafOn;
PROCEDURE HideCursor;
PROCEDURE HidePen;
PROCEDURE InflateTextBuffer
               (newWidth:        Integer;
                newHeight:       Integer);
PROCEDURE InitColorTable
               (VAR tablePtr:    ColorTable);
PROCEDURE InitCursor;
PROCEDURE InitPort
               (portPtr:         GrafPortPtr);
PROCEDURE InsetRect
               (VAR __rectPtr:   Rect;
                deltaH:          Integer;
                deltaV:          Integer);
PROCEDURE InsetRgn
               (aRgnHandle:      RgnHandle;
                dH:              Integer;
                dV:              Integer);
PROCEDURE InvertArc
               (RectPtr:         Rect;
                startAngle:      Integer;
                arcAngle:        Integer);
PROCEDURE InvertOval
               (RectPtr:         Rect);
PROCEDURE InvertPoly
               (polyHandle:      Handle);
PROCEDURE InvertRect
               (RectPtr:         Rect);
PROCEDURE InvertRgn
               (aRgnHandle:      RgnHandle);
```

```
PROCEDURE InvertRRect
              (RectPtr:          Rect;
               ovalWidth:        Integer;
               ovalHeight:       Integer);
PROCEDURE KillPoly
              (polyHandle:       Handle);
PROCEDURE Line
              (dH:               Integer;
               dV:               Integer);
PROCEDURE LineTo
              (h:                Integer;
               v:                Integer);
PROCEDURE LocalToGlobal
              (VAR PointPtr:     Point);
PROCEDURE MapPoly
              (polyHandle:       Handle;
               srcRectPtr:       Rect;
               destRectPtr:      Rect);
PROCEDURE MapPt
              (VAR PointPtr:     Point;
               srcRectPtr:       Rect;
               destRectPtr:      Rect);
PROCEDURE MapRect
              (VAR RectPtr:      Rect;
               srcRectPtr:       Rect;
               destRectPtr:      Rect);
PROCEDURE MapRgn
              (aRgnHandle:       RgnHandle;
               srcRectPtr:       Rect;
               destdRectPtr:     Rect);
PROCEDURE Move
              (dH:               Integer;
               dV:               Integer);
PROCEDURE MovePortTo
              (h:                Integer;
               v:                Integer);
PROCEDURE MoveTo
              (h:                Integer;
               v:                Integer);
FUNCTION NewRgn: RgnHandle;
PROCEDURE ObscureCursor;
PROCEDURE OffsetPoly
              (polyHandle:       Handle;
               dH:               Integer;
               dV:               Integer);
PROCEDURE OffsetRect
              (VAR __rectPtr:    Rect;
               deltaH:           Integer;
               deltaV:           Integer);
PROCEDURE OffsetRgn
              (aRgnHandle:       RgnHandle;
               dH:               Integer;
               dV:               Integer);
FUNCTION OpenPoly: Handle;
PROCEDURE OpenPort
              (portPtr:          GrafPortPtr);
PROCEDURE OpenRgn;
PROCEDURE PaintArc
               (RectPtr:              Rect;
                startAngle:           Integer;
                arcAngle:             Integer);
PROCEDURE PaintOval
               (RectPtr:              Rect);
PROCEDURE PaintPixels
               (__paintParamPtr:      PaintParam);
PROCEDURE PaintPoly
               (polyHandle:           Handle);
PROCEDURE PaintRect
               (RectPtr:              Rect);
PROCEDURE PaintRgn
               (aRgnHandle:           RgnHandle);
PROCEDURE PaintRRect
               (RectPtr:              Rect;
               ovalWidth:             Integer;
               ovalHeight:            Integer);
PROCEDURE PenNormal;
PROCEDURE PPToPort
               (srcLocPtr:            LocInfoPtr;
                srcRectPtr:           Rect;
                destX:                Integer;
                destY:                Integer;
                transferMode:         Integer);
PROCEDURE Pt2Rect
               (VAR point1Ptr:        Point;
                VAR point2Ptr:        Point;
                VAR __rectPtr:        Rect);
FUNCTION PtInRect
               (VAR PointPtr:         Point;
                RectPtr:              Rect): Boolean;
FUNCTION PtInRgn
               (VAR PointPtr:         Point;
                aRgnHandle:           RgnHandle): Boolean;
FUNCTION Random: Integer;
FUNCTION RectInRgn
               (RectPtr:              Rect;
                aRgnHandle:           RgnHandle): Boolean;
PROCEDURE RectRgn
               (aRgnHandle:           RgnHandle;
                RectPtr:              Rect);
PROCEDURE RestoreBufDims
               (sizeInfoPtr:          BufDimRecPtr);
PROCEDURE SaveBufDims
               (VAR sizeInfoPtr:      BufDimRec);
PROCEDURE ScalePt
               (VAR PointPtr:         Point;
                srcRectPtr:           Rect;
                destRectPtr:          Rect);
PROCEDURE ScrollRect
               (RectPtr:              Rect;
                dH:                   Integer;
                dV:                   Integer;
                aRgnHandle:           RgnHandle);
```

```
FUNCTION SectRect
                (rect1Ptr:          Rect;
                 rect2Ptr:          Rect;
                 VAR intersectRectPtr:        Rect): Boolean;
PROCEDURE SectRgn
                (rgn1Handle:        RgnHandle;
                 rgn2Handle:        RgnHandle;
                 destRgnHandle:     RgnHandle);
PROCEDURE SetAllSCBs
                (newSCB:            Integer);
PROCEDURE SetArcRot;
PROCEDURE SetBackColor
                (backColor:         Integer);
PROCEDURE SetBackPat
                (PatternPtr:        Pattern);
PROCEDURE SetBufDims
                (maxWidth:          Integer;
                 maxFontHeight:     Integer;
                 maxFBRExtent:      Integer);
PROCEDURE SetCharExtra
                (charExtra:         Fixed);
PROCEDURE SetClip
                (aRgnHandle:        RgnHandle);
PROCEDURE SetClipHandle
                (aRgnHandle:        RgnHandle);
PROCEDURE SetColorEntry
                (tableNumber:       Integer;
                 entryNumber:       Integer;
                 newColor:          ColorValue);
PROCEDURE SetColorTable
                (tableNumber:       Integer; srcTablePtr: ColorTable);
PROCEDURE SetCursor
                (theCursorPtr:      Cursor);
PROCEDURE SetEmptyRgn
                (aRgnHandle:        RgnHandle);
PROCEDURE SetFont
                (newFontHandle:     FontHndl);
PROCEDURE SetFontFlags
                (fontFlags:         Integer);
PROCEDURE SetFontID
                (VAR __fontID:      FontID);
PROCEDURE SetForeColor
                (foreColor:         Integer);
PROCEDURE SetGrafProcs
                (grafProcsPtr:      QDProcsPtr);
PROCEDURE SetIntUse
                (useInt:            Integer);
PROCEDURE SetMasterSCB
                (masterSCB:         Integer);
PROCEDURE SetOrigin
                (h:                 Integer;
                 v:                 Integer);
PROCEDURE SetPenMask
                (maskPtr:           Mask);
PROCEDURE SetPenMode
                (penMode:           Integer);
PROCEDURE SetPenPat
                (PatternPtr:        Pattern);
```

```
PROCEDURE SetPenSize
               (penWidth:          Integer;
                penHeight:         Integer);
PROCEDURE SetPenState
               (VAR __penStatePtr: PenState);
PROCEDURE SetPicSave
               (picSaveValue:      Longint);
PROCEDURE SetPolySave
               (polySaveValue:     Longint);
PROCEDURE SetPort
               (portPtr:           GrafPortPtr);
PROCEDURE SetPortLoc
               (__locInfoPtr:      LocInfo);
PROCEDURE SetPortRect
               (RectPtr:           Rect);
PROCEDURE SetPortSize
               (portWidth:         Integer;
                portHeight:        Integer);
PROCEDURE SetPt
               (VAR srcPtPtr:      Point;
                h:                 Integer;
                v:                 Integer);
PROCEDURE SetRandSeed
               (randomSeed:         Longint);
PROCEDURE SetRect
               (VAR aRectPtr:      Rect;
                left:              Integer;
                top:               Integer;
                right:             Integer;
                bottom:            Integer);
PROCEDURE SetRectRgn
               (aRgnHandle:        RgnHandle;
                left:              Integer;
                top:               Integer;
                right:             Integer;
                bottom:            Integer);
PROCEDURE SetRgnSave
               (rgnSaveValue:      Longint);
PROCEDURE SetSCB
               (scanLine:          Integer;
                newSCB:            Integer);
PROCEDURE SetSolidBackPat
               (colorNum:          Integer);
PROCEDURE SetSolidPenPat
               (colorNum:          Integer);
PROCEDURE SetSpaceExtra
               (spaceExtra:        Fixed);
PROCEDURE SetStdProcs
               (stdProcRecPtr:     QDProcsPtr);
PROCEDURE SetSysField
               (sysFieldValue:     Longint);
PROCEDURE SetSysFont
               (fontHandle:        FontHndl);
PROCEDURE SetTextFace
               (textFace:          TextStyle);
PROCEDURE SetTextMode
               (textMode:          Integer);
```

```
PROCEDURE SetTextSize
              (textSize:          Integer);
PROCEDURE SetUserField
              (userFieldValue:    Longint);
PROCEDURE SetVisHandle
              (aRgnHandle:        RgnHandle);
PROCEDURE SetVisRgn
              (aRgnHandle:        RgnHandle);
PROCEDURE ShowCursor;
PROCEDURE ShowPen;
PROCEDURE SolidPattern
              (colorNum:          Integer;
               VAR PatternPtr:    Pattern);
PROCEDURE StringBounds
              (str:               Str255;
               VAR resultPtr:     Rect);
FUNCTION StringWidth
              (str: Str255):      Integer;
PROCEDURE SubPt
              (VAR srcPtPtr:      Point;
               VAR destPtPtr:     Point);
PROCEDURE TextBounds
              (textPtr:           Ptr;
               textLength:        Integer;
               VAR resultPtr:     Rect);
FUNCTION TextWidth
              (textPtr:           Ptr;
               textLength:        Integer): Integer;
PROCEDURE UnionRect
              (rect1Ptr:          Rect;
               rect2Ptr:          Rect;
               VAR unionRectPtr:  Rect);
PROCEDURE UnionRgn
              (rgn1Handle:        RgnHandle;
               rgn2Handle:        RgnHandle;
               unionRgnHandle:    RgnHandle);
PROCEDURE XorRgn
              (rgn1Handle:        RgnHandle;
               rgn2Handle:        RgnHandle;
               xorRgnHandle:      RgnHandle);

IMPLEMENTATION
END.
```

Resources

```
{********************************************
; File: Resources.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Resources;

INTERFACE
USES Types, Memory, GSOS;

CONST  resLogOut              = $0;     {ResourceConverter - }
       resLogIn               = $1;     {ResourceConverter - }
       resLogApp              = $0;     {ResourceConverter - }
       resLogSys              = $2;     {ResourceConverter - }
       resForkUsed            = $1E01;  {Error - Resource fork not empty }
       resBadFormat           = $1E02;  {Error - Format of resource fork unknown }
       resForkEmpty           = $1E03;  {Error - Resource fork is empty }
       resNoCurFile           = $1E04;  {Error - there are no current open resource
                                        files }
       resDupID               = $1E05;  {Error - ID is already used }
       resNotFound            = $1E06;  {Error - resource was not found }
       resFileNotFound        = $1E07;  {Error - resource file not found }
       resBadAppID            = $1E08;  {Error - User ID not found, please call
                                        ResourceStartup }
       resNoUniqueID          = $1E09;  {Error - a unique ID was not found }
       resBadAttr             = $1E0A;  {Error - reseved bits in attributes word
                                        are not zero }
       resHashGone            = $1E0B;  {Error - the hash count table is no longer
                                        valid }
       resIndexRange          = $1E0D;  {Error - index is out of range }
       resNoCurApp            = $1E0E;  {Error - no current application, please
                                        call ResourceStartup }
       resChanged             = $0020;  {Resources - }
       resPreLoad             = $0040;  {Resources - }
       resProtected           = $0080;  {Resources - }
       resAbsLoad             = $0400;  {Resources - }
       resConverter           = $0800;  {Resources - }
       resMemAttr             = $C3F1;  {Resources - }
       systemMap              = $0001;  {Resources - }
       mapChanged             = $0002;  {Resources - }
       romMap                 = $0004;  {Resources - }
       resNameOffset          = $10000; {Resources - type holding names }
       resNameVersion         = $0001;  {Resources - }
       rIcon                  = $8001;  {Resources - resource type holding names }
       rPicture               = $8002;  {Resources - resource type holding names }
       rControlList           = $8003;  {Resources - resource type holding names }
       rControlTemplate       = $8004;  {Resources - resource type holding names }
       rWindow                = $8005;  {Resources - resource type holding names }
       rString                = $8006;  {Resources - resource type holding names }
```

```
        rStringList                 = $8007;   {Resources - resource type holding names }
        rMenuBar                    = $8008;   {Resources - resource type holding names }
        rMenu                       = $8009;   {Resources - resource type holding names }
        rMenuItem                   = $800A;   {Resources - resource type holding names }
        rTextForLETextBox2          = $800B;   {Resources - resource type holding names }
        rCtlDefProc                 = $800C;   {Resources - resource type holding names }
        rCtlColorTbl                = $800D;   {Resources - resource type holding names }
        rWindParam1                 = $800E;   {Resources - resource type holding names }
        rWindParam2                 = $800F;   {Resources - resource type holding names }
        rWindColor                  = $8010;   {Resources - resource type holding names }
        rResName                    = $8014;   {Resources - resource type holding names }

TYPE
        ResID                       = Longint;
        ResType                     = Integer;
        ResAttr                     = Integer;
        ResHeaderRec                =
            RECORD
                rFileVersion:   Longint; { Format version of resource fork }
                rFileToMap:     Longint; { Offset from start to resource map record }
                rFileMapSize:   Longint; { Number of bytes map occupies in file }
                rFileMemo:      PACKED ARRAY [1..128] OF Byte;
                                         { Reserved space for application }
                rFileRecSize:   Longint; { Size of ResHeaderRec Record }
            END;

        FreeBlockRec    =
            RECORD
                blkOffset:      Longint;
                blkSize:        Longint;
            END;

        ResMapHandle    = ^ResMapPtr;
        ResMapPtr       = ^ResMap;
        ResMap          =
            RECORD
                mapNext:            ResMapHandle; { Handle to next resource map }
                mapFlag:            Integer; { Bit Flags }
                mapOffset:          Longint; { Map's file position }
                mapSize:            Longint; { Number of bytes map occupies in file }
                mapToIndex:         Integer;
                mapFileNum:         Integer;
                mapID:              Integer;
                mapIndexSize:       Longint;
                mapIndexUsed:       Longint;
                mapFreeListSize:    Integer;
                mapFreeListUsed:    Integer;
                mapFreeList:        ARRAY [1..1] OF FreeBlockRec; { n bytes (array of free
                                                                   block records) }
            END;

        ResRefRecPtr    = ^ResRefRec;
        ResRefRec       =
            RECORD
                ResType:        ResType;
                ResID:          ResID;
                resOffset:      Longint;
                ResAttr:        ResAttr;
```

```
         resSize:        Longint;
         resHandle:      Handle;
      END;

   ResourceSpec    =
      RECORD
         resourceType:   ResType;
         resourceID:     ResID;
      END;

   ResNameEntryPtr    = ^ResNameEntry;
   ResNameEntry       =
      RECORD
         namedResID:       ResID;
         resName:          Str255;
      END;

   ResNameRecordHandle       = ^ResNameRecordPtr;
   ResNameRecordPtr          = ^ResNameRecord;
   ResNameRecord             =
      RECORD
         version:                Integer;
         nameCount:              Longint;
         resNameEntries:         ARRAY [1..1] OF ResNameEntry;
      END;

PROCEDURE ResourceBootInit;
PROCEDURE ResourceStartup
            (userID:            Integer);
PROCEDURE ResourceShutdown;
FUNCTION  ResourceVersion:      Integer;
PROCEDURE ResourceReset;
FUNCTION  ResourceStatus:       Boolean;
PROCEDURE AddResource
            (resourceHandle:    Handle;
             resourceAttr:      Integer;
             resourceType:      Integer;
             resourceID:        Longint);
PROCEDURE CloseResourceFile(fileID: Integer);
FUNCTION  CountResources
            (resourceType:      Integer): Longint;
FUNCTION  CountTypes:           Integer;
PROCEDURE CreateResourceFile
            (auxType:           Longint;
             fileType:          Integer;
             fileAccess:        Integer;
             fileName:          GSString255Ptr);
PROCEDURE DetachResource
            (resourceType:      Integer;
             resourceID:        Longint);
FUNCTION  GetCurResourceApp:    Integer;
FUNCTION  GetCurResourceFile:   Integer;
FUNCTION  GetIndResource
            (resourceType:      ResType;
             resourceIndex:     Longint): ResID;
FUNCTION  GetIndType:           Integer;
FUNCTION  GetMapHandle
            (fileID:            Integer):      ResMapHandle;
```

```
FUNCTION  GetOpenFileRefNum
          (fileID:              Integer):      Integer;
FUNCTION  GetResourceAttr
          (resourceType:        ResType;
           resourceID:          ResID): ResAttr;
FUNCTION  GetResourceSize
          (resourceType:        Integer;
           currentID:           Longint): Longint;
FUNCTION  HomeResourceFile
          (resourceType:        Integer;
           resourceID:          Longint): Integer;
FUNCTION  LoadAbsResource
          (loadAddress:         Longint;
          maxSize               Longint;
          resourceType:         Integer;
          resourceID:           Longint): Handle;
FUNCTION  LoadResource
          (resourceType:        Integer;
           resourceID:          Longint): Handle;
PROCEDURE MarkResourceChange
          (__changeFlag:        Boolean;
           resourceType:        Integer;
           resourceID:          Longint);
PROCEDURE MatchResourceHandle
          (foundRec:            Longint;
           resourceHandle:      Handle);
FUNCTION  OpenResourceFile
          (openAccess:          Integer;
           resourceMapPtr:      ResMapPtr;
           fileName:            GSString255Ptr): Integer;
PROCEDURE ReleaseResource
          (purgeLevel:          Integer;
           resourceType:        Integer;
           resourceID:          Longint);
PROCEDURE RemoveResource
          (resourceType:        Integer;
           resourceID:          Longint);
PROCEDURE ResourceConverter
          (converterProc:       ProcPtr;
           resourceType:        ResType;
           logFlags:            Integer);
PROCEDURE SetCurResourceApp
          (userID:              Integer);
PROCEDURE SetCurResourceFile
          (fileID:              Integer);
PROCEDURE SetResourceAttr
          (resourceAttr:        Integer;
           resourceType:        Integer;
           currentID:           Longint);
FUNCTION  SetResourceFileDepth
          (searchDepth:         Integer): Integer;
PROCEDURE SetResourceID
          (newID:               ResID;
           resourceType:        ResType;
           currentID:           ResID);
FUNCTION  SetResourceLoad
          (readFlag:            Integer): Integer;
```

```
FUNCTION  UniqueResourceID
            (IDrange:          Integer;
             resourceType:     Integer): Longint;
PROCEDURE UpdateResourceFile
            (fileID:           Integer);
PROCEDURE WriteResource
            (resourceType:     Integer;
             resourceID:       Longint);

IMPLEMENTATION
END.
```

SANE

```
{*********************************************
; File: SANE.p
;
;
; Copyright Apple Computer, Inc.  1988
; All Rights Reserved
;
*********************************************}

UNIT SANE;

INTERFACE

CONST  DecStrLen                  = 255;
       SigDigLen                  = 28;   { 20 for 68K; 28 in 6502 SANE }

{----------------------------------------------------
 *  Exceptions.
 ----------------------------------------------------}

       Invalid                    = 1;
       Underflow                  = 2;
       Overflow                   = 4;
       DivByZero                  = 8;
       Inexact                    = 16;

{----------------------------------------------------
 *  IEEE default environment constant.
 ----------------------------------------------------}

       IEEEDefaultEnv             = 0;

{----------------------------------------------------
 *  Style constants for DecForm records.
 ----------------------------------------------------}

       FloatDecimal               = 0;
       FixedDecimal               = 1;


{----------------------------------------------------
 * Types for handling decimal representations.
 ----------------------------------------------------}
TYPE   DecStr                     = String[DecStrLen];

       CStrPtr                    = ^CHAR;

       Decimal        =
              RECORD
                     sgn:   integer; { 0 for positive, 1 for negative }
                     exp:   integer;
                     sig:   String[SigDigLen]
              END;

       DecForm        =
            RECORD
                    style: integer; { FloatDecimal, FixedDecimal }
                    digits:integer;
            END;
{-----------------------------------------------------
 * Ordering relations.
 -----------------------------------------------------}
        RelOp           = (GreaterThan, LessThan, EqualTo, Unordered);

{-----------------------------------------------------
 * Inquiry classes.
 -----------------------------------------------------}
        NumClass        = (SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum);

{-----------------------------------------------------
 *  Environmental control.
 -----------------------------------------------------}
        Exception       = integer;

        RoundDir        = (ToNearest, Upward, Downward, TowardZero);

        RoundPre        = (ExtPrecision, DblPrecision, RealPrecision);

        Environment     = integer;

{*=====================================================================*
 *  The functions and procedures of the SANE library        *
 *=====================================================================*}

{-----------------------------------------------------
 * Conversions between numeric binary types.
 -----------------------------------------------------}

FUNCTION  Num2integer(x: Extended): integer;
FUNCTION  Num2LongInt(x: Extended): LongInt;
FUNCTION  Num2Real(x: Extended):    real;
FUNCTION  Num2Double(x: Extended):  DOUBLE;
FUNCTION  Num2Extended(x: Extended): Extended;
FUNCTION  Num2Comp(x: Extended): comp;

{-----------------------------------------------------
 * Conversions between binary and decimal.
 -----------------------------------------------------}

PROCEDURE num2dec(f: DecForm; x: Extended; VAR d: Decimal); C;
{ d <-- x according to format f }

FUNCTION  dec2num(d: Decimal): Extended; C;
{ Dec2Num <-- d }

PROCEDURE Num2Str(f: DecForm; x: Extended; VAR s: DecStr);
{ s <-- x according to format f }

FUNCTION  Str2Num(s: DecStr): Extended;
{ Str2Num <-- s }
```

```
{--------------------------------------------------------
 * Conversions between decimal formats.
 --------------------------------------------------------}

PROCEDURE Str2Dec(s: DecStr; VAR Index: integer; VAR d: Decimal; VAR ValidPrefix: integer);
{ On input Index is starting index into s, on output Index is
       one greater than index of last character of longest numeric
       substring;
       d <-- Decimal rep of longest numeric substring;
       ValidPrefix <-- "s, beginning at Index, contains valid numeric
       String or valid prefix of some numeric String" }

PROCEDURE CStr2Dec(s: CStrPtr; VAR Index: integer; VAR d:Decimal; VAR ValidPrefix: integer);
{ Str2Dec for character buffers or C strings instead of Pascal
       strings: the first argument is the the address of a character
       buffer and ValidPrefix <-- "scanning ended with a null byte" }

PROCEDURE dec2str(f: DecForm; d: Decimal; VAR s: DecStr);
{ s <-- d according to format f }

{--------------------------------------------------------
 * Arithmetic, auxiliary, and elementary functions.
 --------------------------------------------------------}

FUNCTION  remainder(x, y: Extended; VAR quo: integer): Extended; C;
{ Remainder <-- x rem y; quo <-- low-order seven bits of integer
       quotient x/y so that -127 < quo < 127 }

FUNCTION  rint(x: Extended): Extended; C;
{ round to integral value }

FUNCTION  scalb(n: integer; x: Extended): Extended; C;
{ scale binary;  Scalb <-- x * 2^n }

FUNCTION  logb(x: Extended): Extended; C;
{ Logb <-- unbiased exponent of x }

FUNCTION  copysign(x, y: Extended): Extended;
{ CopySign <-- y with sign of x }

FUNCTION  NextReal(x, y: real): real;

FUNCTION  nextdouble(x, y: DOUBLE): DOUBLE; C;

FUNCTION nextExtended(x, y: Extended): Extended; C;
{ return next representable value from x toward y }

FUNCTION  log2(x: Extended): Extended; C;
{ base-2 log }

FUNCTION  Ln1(x: Extended): Extended;
{ ln(1+x) }

FUNCTION  exp2(x: Extended): Extended; C;
{ base-2 exponential }

FUNCTION  exp1(x: Extended): Extended; C;
{ exp(x) - 1 }
```

```
FUNCTION  XpwrI(x: Extended; i: integer): Extended;
{ XpwrI <-- x^i }

FUNCTION  XpwrY(x, y: Extended): Extended;
{ XpwrY <-- x^y }

FUNCTION  compound(r, n: Extended): Extended; C;
{ Compound <-- (1+r)^n }

FUNCTION  annuity(r, n: Extended): Extended; C;
{ Annuity <-- (1 - (1+r)^(-n)) / r }

FUNCTION  tan(x: Extended): Extended; C;
{ tangent }

FUNCTION randomx(VAR x: Extended): Extended; C;
{ returns next random number and updates argument;
       x integral, 1 <= x <= (2^31)-2 }

{-----------------------------------------------------
 * Inquiry routines.
 -----------------------------------------------------}

FUNCTION ClassReal(x: real): NumClass;

FUNCTION classdouble(x: DOUBLE): NumClass; C;

FUNCTION classcomp(x: comp): NumClass; C;

FUNCTION classExtended(x: Extended): NumClass; C;
{ return class of x }

FUNCTION SignNum(x: Extended): integer; C;
{ 0 if sign bit clear, 1 if sign bit set }

{-----------------------------------------------------
 * NaN function.
 -----------------------------------------------------}

FUNCTION nan(i: integer): Extended; C;
{ returns NaN with code i }

{-----------------------------------------------------
 * Environment access routines.
 -----------------------------------------------------}

PROCEDURE SetException(e: Exception; b: BOOLEAN);
{ set e flags if b is true, clear e flags otherwise; may cause halt }

FUNCTION TestException(e: Exception): BOOLEAN;
{ return true if any e flag is set, return false otherwise }

PROCEDURE SetHalt(e: Exception; b: BOOLEAN);
{ set e halt enables if b is true, clear e halt enables otherwise }

FUNCTION TestHalt(e: Exception): BOOLEAN; C;
{ return true if any e halt is enabled, return false otherwise }
```

```
PROCEDURE SetRound(r: RoundDir);
{ set rounding direction to r }

FUNCTION  GetRound: RoundDir;
{ return rounding direction }

PROCEDURE setprecision(p: RoundPre); C;
{ set rounding precision to p }

FUNCTION  getprecision: RoundPre; C;
{ return rounding precision }

PROCEDURE setenvironment(e: Environment); C;
{ set environment to e }

PROCEDURE getenvironment(VAR e: Environment); C;
{ e <-- environment }

PROCEDURE ProcEntry(VAR e: Environment);
{ e <-- environment;  environment <-- IEEE default env }

PROCEDURE ProcExit(e: Environment);
{ temp <-- exceptions; environment <-- e;
signal exceptions in temp }

FUNCTION  gethaltvVector: LongInt; C;
{ return halt vector }

PROCEDURE sethaltvector(v: LongInt); C;
{ halt vector <-- v }

{---------------------------------------------------
 * Comparison routine.
 ---------------------------------------------------}

FUNCTION  relation(x, y: Extended): RelOp; C;
 { return Relation such that "x Relation y" is true }



PROCEDURE SANEBootInit;
PROCEDURE SANEStartUp(dPageAddr:integer);
PROCEDURE SANEShutDown;
FUNCTION  SANEVersion: integer;
PROCEDURE SANEReset;
FUNCTION  SANEStatus: integer;

IMPLEMENTATION

END.
```

Scheduler

```
{********************************************
; File: Scheduler.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Scheduler;

INTERFACE

USES Types;

PROCEDURE SchBootInit;
PROCEDURE SchStartUp;
PROCEDURE SchShutDown;
FUNCTION  SchVersion:       Integer;
PROCEDURE SchReset;
FUNCTION  SchStatus:  Boolean;
FUNCTION  SchAddTask
               (taskPtr:    VoidProcPtr): Boolean;
PROCEDURE SchFlush;

IMPLEMENTATION
END.
```

Scrap

```
{********************************************
; File: Scrap.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Scrap;

INTERFACE
USES Types;

CONST  badScrapType    = $1610;             {error - No scrap of this type. }
       textScrap       = $0000;             {scrapType - }
       picScrap        = $0001;             {scrapType - }

PROCEDURE ScrapBootInit;
PROCEDURE ScrapStartUp;
PROCEDURE ScrapShutDown;
FUNCTION  ScrapVersion:            Integer;
PROCEDURE ScrapReset;
FUNCTION  ScrapStatus:             Boolean;
PROCEDURE GetScrap
          (destHandle:             Handle;
           scrapType:              Integer);
FUNCTION  GetScrapCount:           Integer;
FUNCTION  GetScrapHandle
          (scrapType:              Integer): Handle;
FUNCTION  GetScrapPath:            Ptr;
FUNCTION  GetScrapSize
          (scrapType:              Integer): Longint;
FUNCTION  GetScrapState:           Integer;
PROCEDURE LoadScrap;
PROCEDURE PutScrap
          (numBytes:               Longint;
           scrapType:              Integer;
           srcPtr:                 Ptr);
PROCEDURE SetScrapPath
          (path:                   Str255);
PROCEDURE UnloadScrap;
PROCEDURE ZeroScrap;

IMPLEMENTATION
END.
```

Sound

```
{**********************************************
; File: Sound.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
**********************************************}

UNIT Sound;

INTERFACE
USES Types;

CONST   noDOCFndErr           = $0810; {error - no DOC chip found }
        docAddrRngErr         = $0811; {error - DOC address range error }
        noSAppInitErr         = $0812; {error - no SAppInit call made }
        invalGenNumErr        = $0813; {error - invalid generator number }
        synthModeErr          = $0814; {error - synthesizer mode error }
        genBusyErr            = $0815; {error - generator busy error }
        mstrIRQNotAssgnErr    = $0817; {error - master IRQ not assigned }
        sndAlreadyStrtErr     = $0818; {error - sound tools already started }
        unclaimedSndIntErr    = $08FF; {error - sound tools already started }
        ffSynthMode           = $0001; {channelGenMode - Free form synthesizer mode }
```

```
      noteSynthMode       = $0002; {channelGenMode - Note synthesizer mode. }
      gen0off             = $0001; {genMask - param to FFStopSound }
      gen1off             = $0002; {genMask - param to FFStopSound }
      gen2off             = $0004; {genMask - param to FFStopSound }
      gen3off             = $0008; {genMask - param to FFStopSound }
      gen4off             = $0010; {genMask - param to FFStopSound }
      gen5off             = $0020; {genMask - param to FFStopSound }
      gen6off             = $0040; {genMask - param to FFStopSound }
      gen7off             = $0080; {genMask - param to FFStopSound }
      gen8off             = $0100; {genMask - param to FFStopSound }
      gen9off             = $0200; {genMask - param to FFStopSound }
      gen10off            = $0400; {genMask - param to FFStopSound }
      gen11off            = $0800; {genMask - param to FFStopSound }
      gen12off            = $1000; {genMask - param to FFStopSound }
      gen13off            = $2000; {genMask - param to FFStopSound }
      gen14off            = $4000; {genMask - param to FFStopSound }
      genAvail            = $0000; {genStatus - Generator available status }
      ffSynth             = $0100; {genStatus - Free Form Synthesizer status }
      noteSynth           = $0200; {genStatus - Note Synthesizer status }
      lastBlock           = $8000; {genStatus - Last block of wave }
      smReadRegister      = $00;  {Jump Table Offset - Read Register routine }
      smWriteRegister     = $04;  {Jump Table Offset - Write Register routine }
      smReadRam           = $08;  {Jump Table Offset - Read Ram routine }
      smWriteRam          = $0C;  {Jump Table Offset - Write Ram routine }
      smReadNext          = $10;  {Jump Table Offset - Read Next routine }
      smWriteNext         = $14;  {Jump Table Offset - Write Next routine }
      smOscTable          = $18;  {Jump Table Offset - Pointer to Oscillator table }
      smGenTable          = $1C;  {Jump Table Offset - Pointer to generator table }
      smGcbAddrTable      = $20;  {Jump Table Offset - Pointer to GCB address table}
      smDisableInc        = $24;  {Jump Table Offset - Disable Increment routine }

TYPE
      SoundPBHndl         = ^SoundPBPtr;
      SoundPBPtr          = ^SoundParamBlock;
      SoundParamBlock     =
         RECORD
            waveStart:    Ptr;        { starting address of wave }
            waveSize:     Integer;    { waveform size in pages }
            freqOffset:   Integer;    { ? formula to be provided }
            docBuffer:    Integer;    { DOC buffer start address, low byte = 0 }
            bufferSize:   Integer;    { DOC buffer start address, low byte = 0 }
            nextWavePtr:  SoundPBPtr; { Pointer to start of next wave's parameter
                                        block }
            volSetting:   Integer;    { DOC volume setting. High byte = 0 }
         END;

      DocRegParamBlkPtr   = ^DocRegParamBlk;
      DocRegParamBlk      =
         PACKED RECORD
            oscGenType:   Integer;
            freqLow1:     Byte;
            freqHigh1:    Byte;
            vol1:         Byte;
            tablePtr1:    Byte;
            control1:     Byte;
            tableSize1:   Byte;
            freqLow2:     Byte;
            freqHigh2:    Byte;
```

```
               vol2:            Byte;
               tablePtr2:       Byte;
               control2:        Byte;
               tableSize2:      Byte;
           END;

PROCEDURE SoundBootInit;
PROCEDURE SoundStartUp
               (dPageAddr:           Integer);
PROCEDURE SoundShutDown;
FUNCTION  SoundVersion:              Integer;
PROCEDURE SoundReset;
FUNCTION  SoundToolStatus:           Boolean;
FUNCTION  FFGeneratorStatus
               (genNumber:           Integer): Integer;
FUNCTION  FFSoundDoneStatus
               (genNumber:           Integer): Boolean;
FUNCTION  FFSoundStatus:             Integer;
PROCEDURE FFStartSound
               (genNumFFSynth:       Integer;
                pBlockPtr:           SoundPBPtr);
PROCEDURE FFStopSound
               (genMask:             Integer);
FUNCTION  GetSoundVolume
               (genNumber:           Integer): Integer;
FUNCTION  GetTableAddress:           Ptr;
PROCEDURE ReadRamBlock
               (destPtr:             Ptr;
                docStart:            Integer;
                byteCount:           Integer);
PROCEDURE SetSoundMIRQV
               (sMasterIRQ:          Longint);
PROCEDURE SetSoundVolume
               (volume:              Integer;
                genNumber:           Integer);
FUNCTION  SetUserSoundIRQV
               (userIRQVector:       Longint): Ptr;
PROCEDURE WriteRamBlock
               (srcPtr:              Ptr;
                docStart:            Integer;
                byteCount:           Integer);
PROCEDURE FFSetUpSound
               (channelGen:          Integer;
                paramBlockPtr:       SoundPBPtr);
PROCEDURE FFStartPlaying
               (genWord:             Integer);
PROCEDURE SetDOCReg
               (pBlockPtr:           DocRegParamBlk);
PROCEDURE ReadDOCReg
               (VAR pBlockPtr:       DocRegParamBlk);

IMPLEMENTATION
END.
```

StdFile

```
{********************************************
; File: StdFile.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT StdFile;

INTERFACE
USES Types;

CONST  noDisplay        = $0000;    {filterProc result - file not to be displayed }
       noSelect         = $0001;    {filterProc result - file displayed, but not
                                    selectable }
       displaySelect    = $0002;    {filterProc result - file displayed and
                                    selectable}
       sfMatchFileType  = $8000;    { - }
       sfMatchAuxType   = $4000;    { - }
       sfDisplayGrey    = $2000;    { - }


TYPE   SFReplyRecPtr    = ^SFReplyRec;
       SFReplyRec       =
          RECORD
             good:          Boolean;
             fileType:      Integer;
             auxFileType:   Integer;
             filename:      String[15];
             fullPathname:  String[128];
          END;

       SFReplyRec2Hndl     = ^SFReplyRec2Ptr;
       SFReplyRec2Ptr      = ^SFReplyRec2;
       SFReplyRec2         =
          RECORD
             good:          Boolean;
             fileType:      Integer;
             auxType:       Longint;
             nameDesc:      RefDescriptor;
             nameRef:       Ref;
             pathDesc:      RefDescriptor;
             pathRef:       Ref;
          END;

       multiReplyRecord    =
          RECORD
             good:            Integer;
             namesHandle:     Handle;
          END;
```

```
        SFTypeListHandle     = ^SFTypeListPtr;
        SFTypeListPtr        = ^SFTypeList;
        SFTypeList           =
            PACKED RECORD
                numEntries:              Byte;
                fileTypeEntries:         PACKED ARRAY [1..5] OF Byte;
            END;

        TypeSelector2    =
            RECORD
                flags:           Integer;
                fileType:        Integer;
                auxType:         Longint;
            END;

        SFTypeList2Ptr     = ^SFTypeList2;
        SFTypeList2        =
            RECORD
                numEntries:              Integer;
                fileTypeEntries:         ARRAY [1..5] OF TypeSelector2;
            END;

PROCEDURE SFBootInit;
PROCEDURE SFStartUp
                (userID:            Integer;
                 dPageAddr:         Integer);
PROCEDURE SFShutDown;
FUNCTION  SFVersion:                Integer;
PROCEDURE SFReset;
FUNCTION  SFStatus:                 Boolean;
PROCEDURE SFAllCaps
                (allCapsFlag:       Boolean);
PROCEDURE SFGetFile
                (whereX:            Integer;
                 whereY:            Integer;
                 prompt:            Str255;
                 filterProcPtr:     WordProcPtr;
                 typeListPtr:       SFTypeListPtr;
                 VAR replyPtr:      SFReplyRec);
PROCEDURE SFGetFile2
                (whereX:            Integer;
                 whereY:            Integer;
                 promptDesc:        RefDescriptor;
                 promptRef:         Ref;
                 filterProcPtr:     WordProcPtr;
                 typeListPtr:       SFTypeList2Ptr;
                 VAR replyPtr:      SFReplyRec2);
PROCEDURE SFMultiGet2
                (whereX:            Integer;
                 whereY:            Integer;
                 promptDesc:        RefDescriptor;
                 promptRef:         Ref;
                 filterProcPtr:     WordProcPtr;
                 typeListPtr:       SFTypeList2Ptr;
                 VAR replyPtr:      SFReplyRec2);
```

```
PROCEDURE SFPGetFile
                (whereX:         Integer;
                 whereY:         Integer;
                 prompt:         Str255;
                 filterProcPtr:  WordProcPtr;
                 typeListPtr:    SFTypeListPtr;
                 dialogTempPtr:  DialogTemplate;
                 dialogHookPtr:  VoidProcPtr;
                 VAR replyPtr:   SFReplyRecPtr);
PROCEDURE SFPGetFile2
                (whereX:         Integer;
                 whereY:         Integer;
                 itemDrawPtr:    ProcPtr;
                 promptDesc:     RefDescriptor;
                 promptRef:      Ref;
                 filterProcPtr:  WordProcPtr;
                 typeListPtr:    SFTypeList2Ptr;
                 dialogTempPtr:  DialogTemplate;
                 dialogHookPtr:  VoidProcPtr;
                 VAR replyPtr:   SFReplyRec2);
PROCEDURE SFPMultiGet2
                (whereX:         Integer;
                 whereY:         Integer;
                 itemDrawPtr:    ProcPtr;
                 promptDesc:     RefDescriptor;
                 promptRef:      Ref;
                 filterProcPtr:  WordProcPtr;
                 typeListPtr:    SFTypeList2Ptr;
                 dialogTempPtr:  DialogTemplate;
                 dialogHookPtr:  VoidProcPtr;
                 VAR replyPtr:   SFReplyRec2);
PROCEDURE SFPPutFile
                (whereX:         Integer;
                 whereY:         Integer;
                 prompt:         Str255;
                 origName:       Str255;
                 maxLen:         Integer;
                 dialogTempPtr:  DialogTemplate;
                 dialogHookPtr:  VoidProcPtr;
                 replyPtr:       SFReplyRecPtr);
PROCEDURE SFPPutFile2
                (whereX:         Integer;
                 whereY:         Integer;
                 itemDrawPtr:    ProcPtr;
                 promptDesc:     RefDescriptor;
                 promptRef:      Ref;
                 origNameDesc:   RefDescriptor;
                 origNameRef:    Ref;
                 dialogTempPtr:  DialogTemplate;
                 dialogHookPtr:  VoidProcPtr;
                 VAR replyPtr:   SFReplyRec2);
PROCEDURE SFPutFile
                (whereX:         Integer;
                 whereY:         Integer;
                 prompt:         Str255;
                 origName:       Str255;
                 maxLen:         Integer;
                 VAR replyPtr:   SFReplyRec);
```

```
PROCEDURE SFPutFile2
              (whereX:               Integer;
               whereY:               Integer;
               promptDesc:           RefDescriptor;
               promptRef:            Ref;
               origNameDesc:         RefDescriptor;
               origNameRef:          Ref;
               VAR replyPtr:         SFReplyRec2);
FUNCTION  SFShowInvisible
              (invisibleState:       Boolean) : Boolean ;
PROCEDURE SFReScan
              (filterProcPtr:        ProcPtr;
               typeListPtr:          SFTypeList2Ptr);
IMPLEMENTATION
END.
```

TextEdit

```
{*********************************************
; File: TextEdit.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
*********************************************}

UNIT TextEdit;

INTERFACE
USES Types, QuickDraw, Fonts, GSOS, Resources, Controls;

CONST   teAlreadyStarted          = $2201;        {error - }
        teNotStarted              = $2202;        {error - }
        teInvalidHandle           = $2203;        {error - }
        teInvalidVerb             = $2204;        {error - }
        teInvalidFlag             = $2205;        {error - }
        teInvalidPCount           = $2206;        {error - }
        teInvalidRect             = $2207;        {error - }
        teBufferOverflow          = $2208;        {error - }
        teInvalidLine             = $2209;        { - }
        teInvalidCall             = $220A;        { - }
        NullVerb                  = $0;           {TE - }
        PStringVerb               = $0001;        {TE - }
        CStringVerb               = $0002;        {TE - }
        C1InputVerb               = $0003;        {TE - }
        C1OutputVerb              = $0004;        {TE - }
        HandleVerb                = $0005;        {TE - }
        PointerVerb               = $0006;        {TE - }
        NewPStringVerb            = $0007;        {TE - }
        fEqualLineSpacing         = $8000;        {TE - }
        fShowInvisibles           = $4000;        {TE - }
        teInvalidDescriptor       = $2204;        { - }
        teInvalidParameter        = $220B;        { - }
```

```
teInvalidTextBox2        = $220C;          { - }
teEqualLineSpacing       = $8000;          { - }
teShowInvisibles         = $4000;          { - }
teJustLeft               = $0;             { - }
teJustRight              = $1;             { - }
teJustCenter             = $2;             { - }
teJustFull               = $3;             { - }
teNoTabs                 = $0;             { - }
teColumnTabs             = $1;             { - }
teAbsoluteTabs           = $2;             { - }
teLeftTab                = $0;             { - }
teCenterTab              = $1;             { - }
teRightTab               = $2;             { - }
teDecimalTab             = $3;             { - }
teInvis                  = $4000;          { - }
teCtlColorIsPtr          = $0000;          { - }
teCtlColorIsHandle       = $0004;          { - }
teCtlColorIsResource     = $0008;          { - }
teCtlStyleIsPtr          = $0000;          { - }
teCtlStyleIsHandle       = $0001;          { - }
teCtlStyleIsResource     = $0002;          { - }
teNotControl             = $80000000;      { - }
teSingleFormat           = $40000000;      { - }
teSingleStyle            = $20000000;      { - }
teNoWordWrap             = $10000000;      { - }
teNoScroll               = $08000000;      { - }
teReadOnly               = $04000000;      { - }
teSmartCutPaste          = $02000000;      { - }
teTabSwitch              = $01000000;      { - }
teDrawBounds             = $00800000;      { - }
teColorHilite            = $00400000;      { - }
teRefIsPtr               = $0000;          { - }
teRefIsHandle            = $0001;          { - }
teRefIsResource          = $0002;          { - }
teRefIsNewHandle         = $0003;          { - }
teDataIsPString          = $0000;          { - }
teDataIsCString          = $0001;          { - }
teDataIsC1Input          = $0002;          { - }
teDataIsC1Output         = $0003;          { - }
teDataIsTextBox2         = $0004;          { - }
teDataIsTextBlock        = $0005;          { - }
teTextIsPtr              = $0000;          { - }
teTextIsHandle           = $0008;          { - }
teTextIsResource         = $0010;          { - }
teTextIsNewHandle        = $0018;          { - }
tePartialLines           = $8000;          { - }
teDontDraw               = $4000;          { - }
teUseFont                = $0020;          { - }
teUseSize                = $0010;          { - }
teUseForeColor           = $0008;          { - }
teUseBackColor           = $0004;          { - }
teUseUserData            = $0002;          { - }
teUseAttributes          = $0001;          { - }
teReplaceFont            = $0040;          { - }
teReplaceSize            = $0020;          { - }
teReplaceForeColor       = $0010;          { - }
teReplaceBackColor       = $0008;          { - }
teReplaceUserField       = $0004;          { - }
```

```
        teReplaceAttributes         = $0002;            { - }
        teSwitchAttributes          = $0001;            { - }
        teEraseRect                 = $0001;            { - }
        teEraseBuffer               = $0002;            { - }
        teRectChanged               = $0003;            { - }


TYPE    TEColorTablePtr        = ^TEColorTable;
        TEColorTable           =
            RECORD
                contentColor:           Integer;
                outlineColor:           Integer;
                pageBoundaryColor:      Integer;
                hiliteForeColor:        Integer;
                hiliteBackColor:        Integer;
                vertColorDescriptor:    Integer;
                vertColorRef:           Longint;
                horzColorDescriptor:    Integer;
                horzColorRef:           Longint;
                growColorDescriptor:    Integer;
                growColorRef:           Longint;
            END;

        TEBlockEntry           =
            RECORD
                text:                   Handle;
                length:                 Handle;
                flags:                  Integer;
            END;

        TEBlocksHndl           = ^TEBlocksPtr;
        TEBlocksPtr            = ^TEBlocksRecord;
        TEBlocksRecord         =
            RECORD
                start:                  Longint;
                index:                  Integer;
                blocks:                 ARRAY [1..1] OF TEBlockEntry;
            END;

        TEHandle               = ^TERecordPtr;
        TERecordPtr            = ^TERecord;
        TERecord               =
            PACKED RECORD
                ctlNext:                CtlRecHndl;
                ctlOwner:               WindowPtr;
                ctlRect:                Rect;
                ctlFlag:                Byte;
                ctlHilite:              Byte;
                ctlValue:               Integer;
                ctlProc:                ProcPtr;
                ctlAction:              ProcPtr;
                ctlData:                Longint;
                ctlRefCon:              Longint;
                ctlColorRef:            TEColorTablePtr;
                textLength:             Longint;
                blockList:              TEBlocksHndl;
                filterProc:             ProcPtr;
                reserved1:              Longint;
        ctlID:                    Longint;
        ctlMoreFlags:             Integer;
        ctlVersion:               Integer;
        theChar:                  Integer;
        theModifiers:             Integer;
        extendFlag:               Integer;
        moveByWord:               Integer;
        inputPtr:                 Ptr;
        inputLength:              Longint;
        theRect:                  Rect;
    END;

TETabItem       =
    RECORD
        tabKind:        Integer;
        tabData:        Integer;
    END;

TERuler         =
    RECORD
        leftMargin:     Integer;
        leftIndent:     Integer;
        rightMargin:    Integer;
        just:           Integer;
        extraLS:        Integer;
        flags:          Integer;
        userData:       Integer;
        tabType:        Integer;
        tabs:           ARRAY [1..1] OF TETabItem;
        tabTerminator:  Integer;
    END;

TEStyle         =
    RECORD
        teFont:         FontID;
        foreColor:      Integer;
        backColor:      Integer;
        reserved:       Longint;
    END;

TEStyleGroupHndl     = ^TEStyleGroupPtr;
TEStyleGroupPtr      = ^TEStyleGroup;
TEStyleGroup         =
    RECORD
        count:      Integer;
        styles:     ARRAY [1..1] OF TEStyle;
    END;

TEStyleItem     =
    RECORD
        length:     Longint;
        offset:     Longint;
    END;

TEFormatHndl    = ^TEFormatPtr;
TEFormatPtr     = ^TEFormat;
TEFormat        =
    RECORD
```

```
        version:          Integer;
        rulerListLength:  Longint;
        theRulerList:     ARRAY [1..1] OF TERuler;
        styleListLength:  Longint;
        theStyleList:     ARRAY [1..1] OF TEStyle;
        numberOfStyles:   Longint;
        theStyles:        ARRAY [1..1] OF TEStyleItem;
    END;

TETextRef      =
    RECORD
        CASE Integer OF
            $0000:  (textIsPStringPtr:        StringPtr);
            $0001:  (textIsCStringPtr:        CStringPtr);
            $0002:  (textIsC1InputPtr:        GSString255Ptr);
            $0003:  (textIsC1OutputPtr:       ResultBuf255Ptr);
            $0004:  (textIsTB2Ptr:            Ptr);
            $0005:  (textIsRawPtr:            Ptr);

            $0008:  (textIsPStringHandle:     String255Hndl);
            $0009:  (textIsCStringHandle:     CStringHndl);
            $000A:  (textIsC1InputHandle:     GSString255Hndl);
            $000B:  (textIsC1OutputHandle:    ResultBuf255Hndl);
            $000C:  (textIsTB2Handle:         Handle);
            $000D:  (textIsRawHandle:         Handle);

            $0010:  (textIsPStringResource:   ResID);
            $0011:  (textIsCStringResource:   ResID);
            $0012:  (textIsC1InputResource:   ResID);
            $0013:  (textIsC1OutputResource:  ResID);
            $0014:  (textIsTB2Resource:       ResID);
            $0015:  (textIsRawResource:       ResID);

            $0018:  (textIsPStringNewH:       String255HndlPtr);
            $0019:  (textIsCStringNewH:       CStringHndlPtr);
            $001A:  (textIsC1InputNewH:       GSString255HndlPtr);
            $001B:  (textIsC1OutputNewH:      ResultBuf255HndlPtr);
            $001C:  (textIsTB2NewH:           HandlePtr);
            $001D:  (textIsRawNewH:           HandlePtr);
    END;

TEStyleRef     =
    RECORD
        CASE Integer OF
            $0000:  (styleIsPtr:              TEFormatPtr);
            $0001:  (styleIsHandle:           TEFormatHndl);
            $0002:  (styleIsResource:         ResID);
            $0003:  (styleIsNewH:             ^TEFormatHndl);
    END;

TEParamBlockHndl    = ^TEParamBlockPtr;
TEParamBlockPtr     = ^TEParamBlock;
TEParamBlock        =
    RECORD
        pCount:         Integer;
        controlID:      Longint;
        boundsRect:     Rect;
        procRef:        Longint;
```

```
            flags:            Integer;
            moreflags:        Integer;
            refCon:           Longint;
            textFlags:        Longint;
            indentRect:       Rect;
            vertBar:          CtlRecHndl;
            vertScroll:       Integer;
            horzBar:          CtlRecHndl;
            horzScroll:       Integer;
            styleRef :        TEStyleRef;
            textDescriptor:   Integer;
            textRef:          TETextRef;
            textLength:       Longint;
            maxChars:         Longint;
            maxLines:         Longint;
            maxHeight:        Integer;
            pageHeight:       Integer;
            headerHeight:     Integer;
            footerHeight:     Integer;
            pageBoundary:     Integer;
            colorRef:         Longint;
            drawMode:         Integer;
            filterProcPtr:    ProcPtr;
        END;

    TEInfoRec       =
        RECORD
            charCount:      Longint;
            lineCount:      Longint;
            formatMemory:   Longint;
            totalMemory:    Longint;
            styleCount:     Longint;
            rulerCount:     Longint;
        END;

    TEHooks         =
        RECORD
            charFilter:     ProcPtr;
            wordWrap:       ProcPtr;
            wordBreak:      ProcPtr;
            drawText:       ProcPtr;
            eraseText:      ProcPtr;
        END;

PROCEDURE TEBootInit;
PROCEDURE TEStartup
        (userId:                Integer;
         directPage:            Integer);
PROCEDURE TEShutdown;
FUNCTION  TEVersion:            Integer;
PROCEDURE TEReset;
FUNCTION  TEStatus:             Integer;
PROCEDURE TEActivate
        (teH:                   TEHandle);
PROCEDURE TEClear
        (teH:                   TEHandle);
```

```
PROCEDURE TEClick
        (VAR theEventPtr:           EventRecord;
         teH:                       TEHandle);
PROCEDURE TECut
        (teH:                       TEHandle);
PROCEDURE TECopy
        (teH:                       TEHandle);
PROCEDURE TEDeactivate
        (teH:                       TEHandle);
FUNCTION  TEGetDefProc:             ProcPtr;
PROCEDURE TEGetHooks
        (VAR hooks:                 TEHooks;
         count:                     Integer;
         teH:                       TEHandle);
PROCEDURE TEGetSelection
        (VAR selStart:              Longint;
         VAR selEnd:                Longint;
         teH:                       TEHandle);
FUNCTION  TEGetSelectionStyle
        (VAR commonStyle:           TEStyle;
         styleHandle:               TEStyleGroupHndl;
         teH:                       TEHandle): Integer;
FUNCTION  TEGetText
        (bufferDesc:                Integer;
         bufferRef:                 TETextRef;
         bufferLength:              Longint;
         styleDesc:                 Integer;
         styleRef:                  TEStyleRef;
         teH:                       TEHandle): Longint;
PROCEDURE TEGetTextInfo
        (VAR infoRec:               TEInfoRec;
         pCount:                    Integer;
         teH:                       TEHandle);
PROCEDURE TEIdle
        (teH:                       TEHandle);
PROCEDURE TEInsert
        (textDesc:                  Integer;
         textRef:                   TETextRef;
         textLength:                Longint;
         styleDesc:                 Integer;
         styleRef:                  TEStyleRef;
         teH:                       TEHandle);
PROCEDURE TEInsertPageBreak
        (teH:                       TEHandle);
PROCEDURE TEKey
        (theEventPtr:               EventRecord;
         teH:                       TEHandle);
PROCEDURE TEKill
        (teH:                       TEHandle);
FUNCTION  TENew
        (theParms:                  TEParamBlock): TEHandle;
FUNCTION  TEPaintText
        (thePort:                   GrafPortPtr;
         start:                     Longint;
         destRect:                  Rect;
         paintFlags:                Integer;
         teH:                       TEHandle): Longint;
```

```
PROCEDURE TEPaste
        (teH:                       TEHandle);
PROCEDURE TEReplace
        (textDesc:                  Integer;
         textRef:                   TETextRef;
         textLength:                Longint;
         styleDesc:                 Integer;
         styleRef:                  TEStyleRef;
         teH:                       TEHandle);
PROCEDURE TESetHooks
        (hooks:                     TEHooks;
         count:                     Integer;
         teH:                       TEHandle);
PROCEDURE TESetSelection
        (selStart:                  Longint;
         selEnd:                    Longint;
         teH:                       TEHandle);
PROCEDURE TESetText
        (textDesc:                  Integer;
         textRef:                   TETextRef;
         textLength:                Longint;
         styleDesc:                 Integer;
         styleRef:                  TEStyleRef;
         teH:                       TEHandle);
PROCEDURE TEStyleChange
        (flags:                     Integer;
         newStyle:                  TEStyle;
         teH:                       TEHandle);
PROCEDURE TEUpdate
        (teH:                       TEHandle);

IMPLEMENTATION
END.
```

TEXTTOOL

```
{********************************************
; File: TextTool.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT TextTool;

INTERFACE
USES Types;

CONST
badDevType          = $0C01; {error - not implemented }
badDevNum           = $0C02; {error - Illegal device number. }
badMode             = $0C03; {error - Bad mode: illegal operation. }
unDefHW             = $0C04; {error - Undefined hardware error }
lostDev             = $0C05; {error - Lost device: Device no longer on line }
lostFile            = $0C06; {error - File no longer in diskette directory }
badTitle            = $0C07; {error - Illegal Filename }
noRoom              = $0C08; {error - Insufficient space on specified diskette }
noDevice            = $0C09; {error - Volume not online }
noFile              = $0C0A; {error - File not in specified directory }
dupFile             = $0C0B; {error - Filename already exists }
notClosed           = $0C0C; {error - Attempt to open an open file }
notOpen             = $0C0D; {error - Attempt to close closed file }
badFormat           = $0C0E; {error - error reading real or integer }
ringBuffOFlo        = $0C0F; {error - Chars arriving too fast }
writeProtected      = $0C10; {error -  }
devErr              = $0C40; {error - Read or Write failed }
input               = $0000; {deviceNum -  }
output              = $0001; {deviceNum -  }
errorOutput         = $0002; {deviceNum -  }
basicType           = $0000; {deviceType -  }
pascalType          = $0001; {deviceType -  }
ramBased            = $0002; {deviceType -  }
noEcho              = $0000; {echoFlag -  }
echo                = $0001; {echoFlag -  }


TYPE
DeviceRecHndl          = ^DeviceRecPtr;
DeviceRecPtr           = ^DeviceRec;
DeviceRec              = RECORD
                             ptrOrSlot : Longint; { slot number or jump table ptr }
                             deviceType : Integer; { type of input device }
                         END;

TxtMaskRecHndl = ^TxtMaskRecPtr;
TxtMaskRecPtr          = ^TxtMaskRec;
```

```
TxtMaskRec            = RECORD
                             orMask : Integer;
                             andMask : Integer;
                        END;

PROCEDURE  TextBootInit;
PROCEDURE  TextStartUp;
PROCEDURE  TextShutDown;
FUNCTION   TextVersion  : Integer;
PROCEDURE  TextReset;
FUNCTION   TextStatus  : Boolean ;
PROCEDURE  CtlTextDev
               (deviceNum:    Integer;
                controlCode:  Integer);
PROCEDURE  ErrWriteBlock
               (textPtr:      Ptr;
               offset:        Integer;
               count:         Integer);
PROCEDURE  ErrWriteChar
               (theChar:      Integer);
PROCEDURE  ErrWriteCString
               (cStrPtr:      CStringPtr);
PROCEDURE  ErrWriteLine
               (str:          Str255);
PROCEDURE  ErrWriteString
               (str:          Str255);
FUNCTION   GetErrGlobals:     TxtMaskRec;
FUNCTION   GetErrorDevice:    DeviceRec;
FUNCTION   GetInGlobals:      TxtMaskRec;
FUNCTION   GetInputDevice:    DeviceRec;
FUNCTION   GetOutGlobals:     TxtMaskRec;
FUNCTION   GetOutputDevice:   DeviceRec;
PROCEDURE  InitTextDev
               (deviceNum:    Integer);
FUNCTION   ReadChar
               (echoFlag:     Integer): Integer;
FUNCTION   ReadLine
               (bufferPtr:    Ptr;
               maxCount:      Integer;
               eolChar:       Integer;
               echoFlag:      Integer): Integer;
PROCEDURE  SetErrGlobals
               (andMask:      Integer;
               orMask:        Integer);
PROCEDURE  SetErrorDevice
               (deviceType:   Integer;
               ptrOrSlot:     Longint);
PROCEDURE  SetInGlobals
               (andMask:      Integer;
               orMask:        Integer);
PROCEDURE  SetInputDevice
               (deviceType:   Integer;
               ptrOrSlot:     Longint);
PROCEDURE  SetOutGlobals
               (andMask:      Integer;
               orMask:        Integer);
```

```
PROCEDURE  SetOutputDevice
               (deviceType:    Integer;
               ptrOrSlot:      Longint);
PROCEDURE  StatusTextDev
               (deviceNum:     Integer;
               requestCode:    Integer);
PROCEDURE  TextReadBlock
               (bufferPtr:     Ptr;
               offset:         Integer;
               blockSize:      Integer;
               echoFlag:       Integer);
PROCEDURE  TextWriteBlock
               (textPtr:       Ptr;
               offset:         Integer;
               count:          Integer);
PROCEDURE  WriteChar
               (theChar:       Integer);
PROCEDURE  WriteCString
               (cStrPtr:       CStringPtr);
PROCEDURE  WriteLine
               (str:           Str255);
PROCEDURE  WriteString
               (str:           Str255);

IMPLEMENTATION
END.
```

TYPES

```
{********************************************
; File: Types.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
********************************************}

UNIT Types;
INTERFACE

CONST

{$IFC UNDEFINED noError } { -  }
noError = $0000;          {$SETC noError := 0}
                          {$ENDC}
refIsPointer          = $0000; {RefDescriptor -   }
refIsHandle           = $0001; {RefDescriptor -   }
refIsResource         = $0002; {RefDescriptor -   }
refIsNewHandle = $0003; {RefDescriptor -   }

TYPE   Byte                  = 0..255;
       Fixed                 = Longint;
       Frac                  = Longint;
```

```
ExtendedPtr            = ^Extended;

SignedByte             = -128..127;
PackedByte             = PACKED ARRAY [1..1] of SignedByte;
Ptr                    = ^PackedByte;
PointerPtr             = ^Ptr;
Handle                 = ^Ptr;

HandlePtr              = ^Handle;

CStringPtr             = Ptr;
CStringHndl            = ^CStringPtr;
CStringHndlPtr         = ^CStringHndl;
ProcPtr                = Ptr;
VoidProcPtr            = ProcPtr;
WordProcPtr            = ProcPtr;
LongProcPtr            = ProcPtr;

IntPtr                 = ^Integer;
FPTPtr                 = Ptr ;
String255              = STRING[255];
String255Ptr           = ^String255;
String255Hndl          = ^String255Ptr;
String255HndlPtr       = ^String255Hndl;
Str255                 = String255;
StringPtr              = String255Ptr;
StringHandle           = ^StringPtr;
String32               = STRING[32];
String32Ptr            = ^String32;
String32Handle         = ^String32Ptr;
Str32                  = String32;

PointPtr               = ^Point;
Point                  = RECORD
                              v : Integer;
                              h : Integer;
                              END;
RectHndl               = ^RectPtr;
RectPtr                = ^Rect;
Rect                   = RECORD
                              CASE INTEGER OF
                                                     1:
                                                            (top: Integer;
                                                            left: Integer;
                                                            bottom: Integer;
                                                            right: Integer);
                                                     2:
                                                            (topLeft: Point;
                                                            botRight: Point);
                                                     3: (
                                                            v1 : Integer;
                                                            h1 : Integer;
                                                            v2 : Integer;
                                                            h2 : Integer);
                                                     END;

TimeRecHndl            = ^TimeRecPtr;
TimeRecPtr             = ^TimeRec;
```

```
        TimeRec         = PACKED RECORD
                                                second:        Byte;
                                                minute:        Byte;
                                                hour:          Byte;
                                                year:          Byte;
                                                day:           Byte;
                                                month:         Byte;
                                                extra:         Byte;
                                                weekDay:       Byte;
                                                END;
        RefDescriptor = Integer;



        Ref = RECORD
                CASE Integer OF
                        refIsPointer:           ( refIsPointer : Ptr );
                        refIsHandle:            ( refIsHandle : Handle );
                        refIsResource:          ( refIsResource : Longint );
                        refIsNewHandle:         ( refIsNewHandle : Handle );
                END;
VAR

{$PUSH}
{$J+}
        _ownerid : Integer;
        _toolErr : Integer;
{$J-}
{$POP}
{these calls are only here temporarilty }

FUNCTION   BAND4(long1, long2: LongInt): LongInt;
FUNCTION   BOR4 (long1, long2: LongInt): LongInt;
FUNCTION   BXOR4(long1, long2: LongInt): LongInt;
FUNCTION   BNOT4(long1:        LongInt): LongInt;

FUNCTION   BSL4   (long1: LongInt; count: INTEGER): LongInt;
FUNCTION   BSR4   (long1: LongInt; count: INTEGER): LongInt;
FUNCTION   BRotL4(long1: LongInt; count: INTEGER): LongInt;
FUNCTION   BRotR4(long1: LongInt; count: INTEGER): LongInt;

FUNCTION   BTst4(     long1: LongInt; pos: INTEGER): BOOLEAN;
PROCEDURE BClr4(VAR long1: LongInt; pos: INTEGER);
PROCEDURE BSet4(VAR long1: LongInt; pos: INTEGER);

IMPLEMENTATION
END.
```

WINDOWS

```
{*********************************************
; File: Windows.p
;
;
; Copyright Apple Computer, Inc. 1986-89
; All Rights Reserved
;
*********************************************}

UNIT Windows;
INTERFACE
USES Types,QuickDraw,Events,Controls;
CONST

paramLenErr          = $0E01; {error - first word of parameter list is the wrong size }
allocateErr          = $0E02; {error - unable to allocate window record }
taskMaskErr          = $0E03; {error - bits 12-15 are not clear in WmTaskMask field of
                                  EventRecord }
wNoConstraint        = $0000; {Axis parameter - No constraint on movement. }
wHAxisOnly           = $0001; {Axis parameter - Horizontal axis only. }
wVAxisOnly           = $0002; {Axis parameter - Vertical axis only. }
FromDesk             = $00; {Desktop Command - Subtract region from desktop }
ToDesk               = $1;  {Desktop Command - Add region to desktop }
GetDesktop           = $2;  {Desktop Command - Get Handle of Desktop region }
SetDesktop           = $3;  {Desktop Command - Set Handle of Desktop region }
GetDeskPat           = $4;  {Desktop command - Address of  pattern or drawing routine }
SetDeskPat           = $5; {Desktop command - Change Address of  pattern or drawing
                              routine }
GetVisDesktop        = $6; {Desktop command - Get destop region less visible windows. }
BackGroundRgn        = $7; {Desktop command - For drawing directly on desktop. }
toBottom             = $FFFFFFFE; {SendBehind value - To send window to bottom. }
topMost              = $FFFFFFFF; {SendBehind value - To make window top. }
bottomMost           = $0000; {SendBehind value - To make window bottom. }
tmMenuKey      = $0001; {Task Mask -  }
tmUpdate       = $0002; {Task Mask -  }
tmFindW        = $0004; {Task Mask -  }
tmMenuSel      = $0008; {Task Mask -  }
tmOpenNDA      = $0010; {Task Mask -  }
tmSysClick     = $0020; {Task Mask -  }
tmDragW        = $0040; {Task Mask -  }
tmContent      = $0080; {Task Mask -  }
tmClose        = $0100; {Task Mask -  }
tmZoom         = $0200; {Task Mask -  }
tmGrow         = $0400; {Task Mask -  }
tmScroll       = $0800; {Task Mask -  }
tmSpecial      = $1000; {Task Mask -  }
tmCRedraw      = $2000; {Task Mask -  }
tmInactive     = $4000; {Task Mask -  }
tmInfo         = $8000; {Task Mask -  }
wNoHit         = $0000; {TaskMaster codes - retained for back compatibility.  }
inNull         = $0000; {TaskMaster codes - retained for back compatibility }
inKey          = $0003; {TaskMaster codes - retained for back compatibility }
inButtDwn      = $0001; {TaskMaster codes - retained for back compatibility }
```

```
inUpdate        = $0006; {TaskMaster codes - retained for back compatibility }
wInDesk         = $0010; {TaskMaster codes - On Desktop }
wInMenuBar      = $0011; {TaskMaster codes - On system menu bar }
wClickCalled    = $0012; {TaskMaster codes - system click called }
wInContent      = $0013; {TaskMaster codes - In content region }
wInDrag         = $0014; {TaskMaster codes - In drag region }
wInGrow         = $0015; {TaskMaster codes - In grow region, active window only }
wInGoAway       = $0016; {TaskMaster codes - In go-away region, active window only }
wInZoom         = $0017; {TaskMaster codes - In zoom region, active window only }
wInInfo         = $0018; {TaskMaster codes - In information bar }
wInSpecial      = $0019; {TaskMaster codes - Item ID selected was 250 - 255 }
wInDeskItem     = $001A; {TaskMaster codes - Item ID selected was 1 - 249 }
wInFrame        = $1B; {TaskMaster codes - in Frame, but not on anything else }
wInactMenu      = $1C; {TaskMaster codes - "selection" of inactive menu item }
wClosedNDA      = $001D; {TaskMaster codes - desk accessory closed }
wCalledSysEdit = $001E; {TaskMaster codes - inactive menu item selected }
wInSysWindow    = $8000; {TaskMaster codes - hi bit set for system windows }
wDraw           = $00; {VarCode - Draw window frame command. }
wHit            = $01; {VarCode - Hit test command. }
wCalcRgns       = $02; {VarCode - Compute regions command. }
wNew            = $03; {VarCode - Initialization command. }
wDispose        = $04; {VarCode - Dispose command. }
fHilited        = $0001; {WFrame - Window is highlighted. }
fZoomed         = $0002; {WFrame - Window is zoomed. }
fAllocated      = $0004; {WFrame - Window record was allocated. }
fCtlTie         = $0008; {WFrame - Window state tied to controls. }
fInfo           = $0010; {WFrame - Window has an information bar. }
fVis            = $0020; {WFrame - Window is visible. }
fQContent       = $0040; {WFrame -  }
fMove           = $0080; {WFrame - Window is movable. }
fZoom           = $0100; {WFrame - Window is zoomable. }
fFlex           = $0200; {WFrame -  }
fGrow           = $0400; {WFrame - Window has grow box. }
fBScroll        = $0800; {WFrame - Window has horizontal scroll bar. }
fRScroll        = $1000; {WFrame - Window has vertical scroll bar. }
fAlert          = $2000; {WFrame -  }
fClose          = $4000; {WFrame - Window has a close box. }
fTitle          = $8000; {WFrame - Window has a title bar. }
windSize        = $145; {WindRec - Size of WindRec. }
wmTaskRecSize  = $0046; {WmTaskRec - Size of WmTaskRec. }
wTrackZoom      = $001F; { -  }
wHitFrame       = $0020; { -  }
wInControl      = $0021; { -  }

TYPE
WmTaskRec       = EventRecord ;


WmTaskRecPtr    = EventRecordPtr ;
WindColorHndl  = ^WindColorPtr;
WindColorPtr    = ^WindColor;
WindColor       = RECORD
                    frameColor:   Integer;        { Color of window frame. }
                    titleColor:   Integer;        { Color of title and bar. }
                    tBarColor:    Integer;        { Color/pattern of title bar. }
                    growColor:    Integer;        { Color of grow box. }
                    infoColor:    Integer;        { Color of information bar. }
                    END;
```

```
WindRecPtr     = ^WindRec;
WindRec= RECORD
                    port:         GrafPort;   { Window's port }
                    wDefProc:     ProcPtr;
                    wRefCon:      Longint;
                    wContDraw:    ProcPtr;
                    wReserved:    Longint;    { Space for future expansion }
                    wStrucRgn:    RgnHandle;  { Region of frame plus content. }
                    wContRgn:     RgnHandle;  { Content region. }
                    wUpdateRgn:   RgnHandle;  { Update region. }
                    wControls:    CtlRecHndl; { Window's control list. }
                    wFrameCtrls:  CtlRecHndl; { Window frame's control list. }
                    wFrame: Integer;
                    END;

WindowChainPtr= ^WindowChain;
WindowChain    = RECORD
             wNext:        WindowChainPtr;
             theWindow:    WindRec;
             END;

ParamListHndl = ^ParamListPtr;
ParamListPtr = ^ParamList;
ParamList = RECORD
    paramLength:     Integer;       { Parameter to NewWindow. }
    wFrameBits:      Integer;       { Parameter to NewWindow. }
    wTitle:          Ptr;           { Parameter to NewWindow. }
    wRefCon:         Longint;       { Parameter to NewWindow. }
    wZoom:           Rect;          { Parameter to NewWindow. }
    wColor:          WindColorPtr;  { Parameter to NewWindow. }
    wYOrigin:        Integer;       { Parameter to NewWindow. }
    wXOrigin:        Integer;       { Parameter to NewWindow. }
    wDataH:          Integer;       { Parameter to NewWindow. }
    wDataW:          Integer;       { Parameter to NewWindow. }
    wMaxH:           Integer;       { Parameter to NewWindow. }
    wMaxW:           Integer;       { Parameter to NewWindow. }
    wScrollVer:      Integer;       { Parameter to NewWindow. }
    wScrollHor:      Integer;       { Parameter to NewWindow. }
    wPageVer:        Integer;       { Parameter to NewWindow. }
    wPageHor:        Integer;       { Parameter to NewWindow. }
    wInfoRefCon:     Longint;       { Parameter to NewWindow. }
    wInfoHeight:     Integer;       { height of information bar }
    wFrameDefProc:   LongProcPtr;   { Parameter to NewWindow. }
    wInfoDefProc:    VoidProcPtr;   { Parameter to NewWindow. }
    wContDefProc:    VoidProcPtr;   { Parameter to NewWindow. }
    wPosition:       Rect;          { Parameter to NewWindow. }
    wPlane:          WindowPtr;     { Parameter to NewWindow. }
    wStorage:        WindowChainPtr;        { Parameter to NewWindow. }
    END;

DeskMessageRecordPtr          = ^DeskMessageRecord;
DeskMessageRecord             =
        RECORD
                reserved      : Longint;
                messageType   : Integer;
                drawType      : Integer;
        END;
```

```
FUNCTION AlertWindow
        ( alertFlags:               Integer;
          subStrPtr:                Ptr;
          alertStrPtr:              Ptr) : Integer;
PROCEDURE DrawInfoBar
        ( theWindowPtr:             WindowPtr);
PROCEDURE EndFrameDrawing;
FUNCTION GetWindowMgrGlobals:       Longint;
PROCEDURE ResizeWindow
        ( hiddenFlag:               Integer;
          __rectPtr:                Rect;
          theWindowPtr:             WindowPtr);
PROCEDURE StartFrameDrawing
        ( windowPtr:                Longint);
PROCEDURE WindBootInit;
PROCEDURE WindStartUp
        ( userID:                   Integer);
PROCEDURE WindShutDown;
FUNCTION WindVersion:               Integer;
PROCEDURE WindReset;
FUNCTION WindStatus:                Boolean;
PROCEDURE BeginUpdate
        ( theWindowPtr:             WindowPtr);
PROCEDURE BringToFront
        ( theWindowPtr:             WindowPtr);
FUNCTION CheckUpdate
        ( theEventPtr:              EventRecordPtr): Boolean;
PROCEDURE CloseWindow
        ( theWindowPtr:             WindowPtr);
FUNCTION Desktop
        ( deskTopOP:                Integer;
        dtParam:                    Longint): Ptr;
PROCEDURE DragWindow
        ( grid:                     Integer;
          startX:                   Integer;
          startY:                   Integer;
          grace:                    Integer;
          boundsRectPtr:            RectPtr;
          theWindowPtr:             WindowPtr);
PROCEDURE EndInfoDrawing;
PROCEDURE EndUpdate
        ( theWindowPtr:             WindowPtr);
FUNCTION FindWindow
        (VAR theWindowPtr:          WindowPtr;
          pointX:                   Integer;
          pointY:                   Integer): Integer;
FUNCTION FrontWindow:               WindowPtr;
FUNCTION GetContentDraw
        ( theWindowPtr:             WindowPtr): VoidProcPtr;
FUNCTION GetContentOrigin
        ( theWindowPtr:             WindowPtr): Point;
FUNCTION GetContentRgn
        ( theWindowPtr:             WindowPtr): RgnHandle;
FUNCTION GetDataSize
        ( theWindowPtr:             WindowPtr): Longint;
FUNCTION GetDefProc
        ( theWindowPtr:             WindowPtr): LongProcPtr;
```

```
FUNCTION GetFirstWindow:             WindowPtr;
PROCEDURE GetFrameColor
        (VAR colorPtr:               WindColorPtr;
         theWindowPtr:               WindowPtr)  ;
FUNCTION GetInfoDraw
        ( theWindowPtr:              WindowPtr): VoidProcPtr;
FUNCTION GetInfoRefCon
        ( theWindowPtr:              WindowPtr): Longint;
FUNCTION GetMaxGrow
        ( theWindowPtr:              WindowPtr): Longint;
FUNCTION GetNextWindow
        ( theWindowPtr:              WindowPtr):   WindowPtr;
FUNCTION GetPage
        ( theWindowPtr:              WindowPtr): Longint;
PROCEDURE GetRectInfo
        (VAR infoRectPtr:            Rect;
        theWindowPtr:                WindowPtr);
FUNCTION GetScroll
        ( theWindowPtr:              WindowPtr): Longint;
FUNCTION GetStructRgn
        ( theWindowPtr:              WindowPtr): RgnHandle;
FUNCTION GetSysWFlag
        ( theWindowPtr:              WindowPtr): Boolean;
FUNCTION GetUpdateRgn
        ( theWindowPtr:              WindowPtr): RgnHandle;
FUNCTION GetWControls
        ( theWindowPtr:              WindowPtr): CtlRecHndl;
FUNCTION GetWFrame
        ( theWindowPtr:              WindowPtr): Integer;
FUNCTION GetWKind
        ( theWindowPtr:              WindowPtr): Integer;
FUNCTION GetWMgrPort:                WindowPtr;
FUNCTION GetWRefCon
        ( theWindowPtr:              WindowPtr): Longint;
FUNCTION GetWTitle
        ( theWindowPtr:              WindowPtr): Ptr;
FUNCTION GetZoomRect
        ( theWindowPtr:              WindowPtr): RectPtr;
```

```
FUNCTION GrowWindow
        (minWidth:          Integer;
         minHeight:         Integer;
         startX:            Integer;
         startY:            Integer;
         theWindowPtr:      WindowPtr): Longint;
PROCEDURE HideWindow
        (theWindowPtr:      WindowPtr);
PROCEDURE HiliteWindow
        (fHiliteFlag:       Boolean;
         theWindowPtr:      WindowPtr);
PROCEDURE InvalRect
        ( badRectPtr:       Rect);
PROCEDURE InvalRgn
        ( badRgnHandle:     RgnHandle);
PROCEDURE MoveWindow
        (newX:              Integer;
         newY:              Integer;
         theWindowPtr:      WindowPtr);
FUNCTION NewWindow
        (theParamListPtr:   ParamList): WindowPtr;
FUNCTION PinRect
        (theXPt:            Integer;
         theYPt:            Integer;
         theRectPtr:        Rect): Point;
PROCEDURE RefreshDesktop
        ( redrawRect:       RectPtr);
PROCEDURE SelectWindow
        ( theWindowPtr:     WindowPtr);
PROCEDURE SendBehind
        (behindWindowPtr:   WindowPtr;
         theWindowPtr:      WindowPtr);
PROCEDURE SetContentDraw
        (contentDrawPtr:    VoidProcPtr;
         theWindowPtr:      WindowPtr);
PROCEDURE SetContentOrigin
        (xOrigin:           Integer;
         yOrigin:           Integer;
         theWindowPtr:      WindowPtr);
PROCEDURE SetContentOrigin2
        (scrollFlag:        Integer;
         xOrigin:           Integer;
         yOrigin:           Integer;
         theWindowPtr:      WindowPtr);
PROCEDURE SetDataSize
        (dataWidth:         Integer;
         dataHeight:        Integer;
         theWindowPtr:      WindowPtr);
PROCEDURE SetDefProc
        (wDefProcPtr:       LongProcPtr;
         theWindowPtr:      WindowPtr);
PROCEDURE SetFrameColor
        (newColorPtr:       WindColorPtr;
         theWindowPtr:      WindowPtr);
PROCEDURE SetInfoDraw
        (infoRecCon:        VoidProcPtr;
         theWindowPtr:      WindowPtr);
```

```
PROCEDURE SetInfoRefCon
        (infoRefCon:          Longint;
        theWindowPtr:         WindowPtr);
PROCEDURE SetMaxGrow
        (maxWidth:            Integer;
         maxHeight:           Integer;
         theWindowPtr:        WindowPtr);
PROCEDURE SetOriginMask
        (originMask:          Integer;
         theWindowPtr:        WindowPtr);
PROCEDURE SetPage
        (hPage:               Integer;
         vPage:               Integer;
         theWindowPtr:        WindowPtr);
PROCEDURE SetScroll
        (hScroll:             Integer;
         vScroll:             Integer;
         theWindowPtr:        WindowPtr);
PROCEDURE SetSysWindow
        (theWindowPtr:        WindowPtr);
PROCEDURE SetWFrame
        (wFrame:              Integer;
         theWindowPtr:        WindowPtr);
FUNCTION SetWindowIcons
        (newFontHandle:       FontHndl): FontHndl;
PROCEDURE SetWRefCon
        (wRefCon:             Longint;
         theWindowPtr:        WindowPtr);
PROCEDURE SetWTitle
        (title:               Str255;
         theWindowPtr:        WindowPtr);
PROCEDURE SetZoomRect
        (wZoomSizePtr:        Rect;
         theWindowPtr:        WindowPtr);
PROCEDURE ShowHide
        (showFlag:            Boolean;
         theWindowPtr:        WindowPtr);
PROCEDURE ShowWindow
        (theWindowPtr:        WindowPtr);
PROCEDURE SizeWindow
        (newWidth:            Integer;
         newHeight:           Integer;
         theWindowPtr:        WindowPtr);
PROCEDURE StartDrawing
        (theWindowPtr:        WindowPtr);
PROCEDURE StartInfoDrawing
        (VAR infoRectPtr:     Rect;
         theWindowPtr:        WindowPtr);
FUNCTION TaskMaster
        (taskMask:            Integer;
         taskRecPtr:          WmTaskRec): Integer;
FUNCTION TrackGoAway
        (startX:              Integer;
         startY:              Integer;
         theWindowPtr:        WindowPtr): Boolean;
```

```
FUNCTION TrackZoom
        (startX:              Integer;
         startY:              Integer;
         theWindowPtr: WindowPtr): Boolean;
PROCEDURE ValidRect
        (goodRectPtr:         Rect);
PROCEDURE ValidRgn
        (goodRgnHandle:       RgnHandle);
FUNCTION WindDragRect
        (actionProcPtr:       VoidProcPtr;
        dragPatternPtr:       Pattern;
        startX:               Integer;
        startY:               Integer;
        dragRectPtr:          Rect;
        limitRectPtr:         Rect;
        slopRectPtr:          Rect;
        dragFlag:             Integer): Longint;
PROCEDURE WindNewRes;
FUNCTION WindowGlobal
        (WindowGlobalMask:    Integer): Integer;
PROCEDURE ZoomWindow
        (theWindowPtr: WindowPtr);
FUNCTION TaskMasterDA
        (eventMask:           Integer;
         taskRecPtr:          Ptr): Integer;
FUNCTION CompileText
        (subType:             Integer;
         subStringsPtr:       Ptr;
         srcStringPtr:        Ptr;
         srcSize:             Integer): Handle;
FUNCTION NewWindow2
        (titlePtr:            StringPtr;
         refCon:              Longint;
         contentDrawPtr:      ProcPtr;
         defProcPtr:          ProcPtr;
         paramTableDesc:      RefDescriptor;
         paramTableRef:       Ref;
         resourceType:        Integer): WindowPtr;
FUNCTION ErrorWindow
        (subType:             Integer;
         subStringPtr:        Ptr;
         errNum:              Integer): Integer;

IMPLEMENTATION
END.
```

Appendix D

Inside Complete Pascal

Complete Pascal Memory Model

The environment in which an Apple IIGS application runs may be divided into 4 basic components: the *Application Code*, the *Application Globals*, the *Runtime Stack*, and the *Application Heap*. All of these components of an application coexist in the Apple IIGS's RAM memory. Memory in the Apple IIGS is partitioned into 64K byte *banks* which are managed by the Apple IIGS Memory Manager. A standard Apple IIGS comes with 4 banks of 64K byte RAM memory numbered $00, $01, $E0, and $E1. RAM expansion cards can be added to the Apple IIGS beginning at bank $02 and may extend to bank $7F, other bank numbers are reserved or not available.

For a thorough introduction to the architecture to the Apple IIGS see Apple Computer's *Technical Introduction to the Apple IIGS.*

The Application Code

An Apple IIGS application may consist of one or more *code segments.* Small programs are usually made up of only a single code segment, but larger programs are divided into several code segments because the Apple IIGS limits the size of an individual code segment to 64K bytes. The reason for the size restriction is that a code segment must not cross the boundries of a *bank* of memory.

Complete Pascal generates a separate *code module* for each Pascal procedure and function declared in a program. Each of these code modules is associated with a *load segment name* which is used to organize separate code segments together by the linker. The default segment name is *main.* When an application has grown large enough to require more than one code segment, the *{$CSeg segname }* directive is used to change the segment name assigned to subsequent code modules. The compiler can be restored to use the default code segment name by specifying *{$CSeg main }.*

The Application Globals

Complete Pascal allocates storage for global variables in *data segments.* By default, the data segments are given the *load segment name* "~global". The Linker uses the load segment names associated with each data segment to group them together into *load segments.* Programs are usually made up of only a single data segment, but programs which require a large amount of global storage are divided into several data segments because the Apple IIGS limits the size of an individual data segment to 64K bytes. The reason for the size restriction is that a data segment must not cross the boundries of a *bank* of memory.

When an application requires a large amount of global storage, it should use the *{$DSeg segname }* directive to instruct the compiler to allocate subsequent global variable declarations into a new data segment. The compiler can be restored to the default data segment by specifying *{$DSeg ~global }*.

During the execution of program initialization code generated by Complete Pascal, the 65816 *Data Bank Register* is set to point to the bank of memory which contains the global variables declared in the *~global* data segment. Because of this, references to global variables in the *~global* data segment can use the *Absolute Addressing Mode.* Global variables in all other data segments are addressed using the less efficient *Absolute Long Addressing Mode.*

The Runtime Stack

The runtime stack is a special block of memory that the application uses to maintain the return addresses of procedures and functions, and to store parameters and local variables. During the execution of application initialization code, a block of memory is allocated in *bank $00* of the Apple IIGS. The block is allocated in bank $00 because this is the only bank of memory in which the 65816 *Stack Register* is able to operate.

By default, Complete Pascal allocates a stack containing 8096 bytes (8K) for an application. If an application requires additional or less stack space, then you should specify the *{$StackSize numbytes }* directive in order to change the amount of space allocated for the stack.

The *{$StackSize numbytes }* directive must appear before the reserved word PROGRAM for it to have any affect. For example, the following code fragment would cause a 10K stack to be allocated.

```
{$StackSize 10240 }
PROGRAM MyApp;
...
```

Desk Accessories do not have initialization code which allocates and initializes a runtime stack since they run within the environment of other applications. Thus, when writing an application be sure to leave a reasonable amount of stack space for use by desk accessories, and of course when writing a desk accessory, be sure to use as little stack space as possible (Remember the default, and typical, runtime stack is only 8K bytes).

❖ **Warning:** Neither the Apple IIGS nor Complete Pascal has any way to detect the amount of stack space actually used by an application. If insufficient space has been reserved for the runtime stack, then execution of the application will destroy the contents of memory.

The Application Heap

The *application heap* is the memory still available after the application's code, global data, and runtime stack have been allocated. This memory is available to an application via the Memory Manager routines defined in the Memory.p.o unit provided with Complete Pascal. Memory may also be allocated and deallocated in the application heap using Complete Pascal's *New* and *Dispose* procedures.

Nearly every application will allocate at least one memory block in bank $00 for initializing the Apple IIGS tools used in the application. Many of the Apple IIGS toolsets require one or more *pages* allocated in bank $00 to function. These pages are sometimes called *zero pages.*

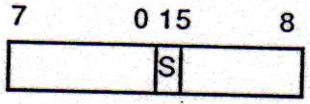
Data Representation

This section shows how each of the Pascal data types is represented in memory. Note that the 65816 stores bytes of word size data in memory backwards from its representation in the accumlator. That is, the least significant bits are in low memory while the most significant bits are in high memory. Consider the following Pascal declaration:

```
     type trick = packed record
                    case integer of
                       0: (int: integer);
                       1: (highbyte: 0..255;
                            lowbyte:  0..255);
                 end;
```

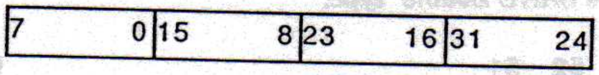
This record type does not give the results one might expect. On the 65816, referencing *highbyte* would actually return the low order byte of the integer *int,* and not the high order byte. The following paragraphs should clarify the storage layout of the Pascal data types.

Integer A signed two's complement integer in the range -32,768 to 32,767 requiring 2 bytes of storage. Bit 15 is the sign bit.

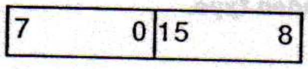


(Note: S denotes the sign bit)

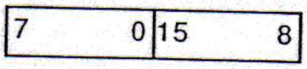
LongInt A signed two's complement integer in the range -2,147,483,648 to 2,147,483,647 requiring 4 bytes of storage. Bit 31 is the sign bit.



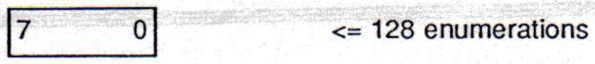
Boolean An enumerated type of *(False,True)* requiring one byte of storage, where the boolean value is in bit 0. One byte of storage is used inside of a packed array or record.

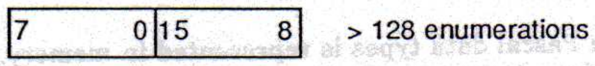


Char An enumerated type of the ASCII character set having 256 values. The character value requires two bytes of storage where the value is in the lower order byte (bits 7-0). One byte of storage is used inside of a packed array or record.

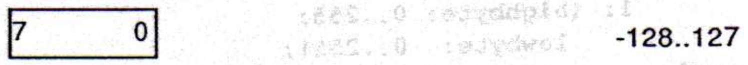


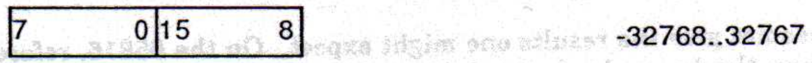
Enumeration An unsigned byte or word size integer. If the enumeration type consists of 128 or fewer enumeration constants, then the a value of the type occupies a single byte of storage if used inside of a packed array or record., otherwise, it occupies a word of storage.

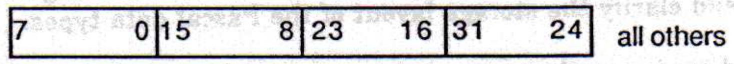




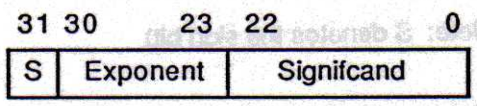
Subrange A signed byte, word, or longword. If the range is within -128..127 a word is used in unpacked structures or simple variables, but inside of packed arrays and records a byte is used to represent the subrange. If the range is within -32768..32767 a word is used, otherwise a longword is used to represent the subrange.



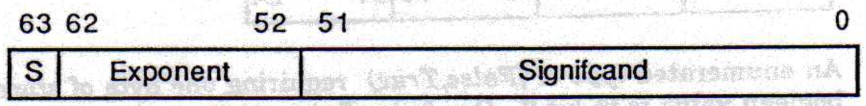




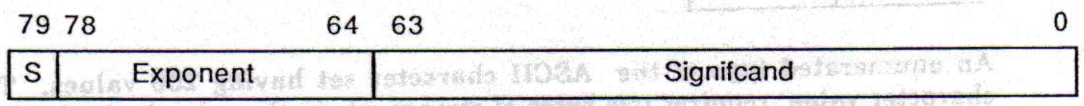
Single A 32-bit real number represented in IEEE standard single precision format implemented as the SANE Single type.



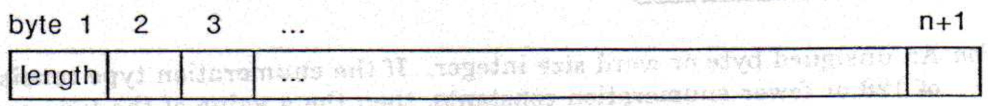
Double A 64-bit real number represented in IEEE standard double precision format implemented as the SANE Double type.



Real / Extended An 80-bit real number represented in IEEE standard extended format. Both are implemented as the SANE Extended type.



String[n] An n+1 byte size *Pascal String* consisting of a byte containing the current string length (not counting the length byte itself) followed by bytes containing ASCII characters.



Pointers

A 24-bit physical memory address occupying 4 bytes of storage. Only 3 bytes are needed to store the 24-bit address value so bits 31-24 are always zero. The *nil* pointer is represented as the 32-bit value zero.

7 0	15 8	23 16	31 24

Sets

A sequence of bytes up to a maximum of 32 bytes or 256 bits representing the powerset of the base-type. The number of bytes used is the minimum number required to represent the powerset. An ordinal value of the base-type is represented by a single bit whose bit number is the ordinal value. If an ordinal value is a member of a set then its bit is set to 1, otherwise, it is set to 0. If the ordinal values of the base-type are in the range 0..15, then two bytes are used to represent the set. If the ordinal values of the base-type are in the range 0..31, then four bytes are used to represent the set, etc.

Files

A 22 byte data structure used internally by the Complete Pascal runtime routines. In addition to the file variable itself, an open file associated with an external disk file has a *access buffer* allocated in the application heap for use by GS/OS; additionally, a textfile has a 256 byte line buffer.

Arrays / Records

Components of unpacked arrays and records are allocated contiguously as defined above. Arrays are stored in row order. That is, the last index varies fastest. Record components are allocated as they textually appear in their declaration.

The implementation of Complete Pascal performs data packing to byte boundries only, no bit packing is available. A data type is represented as a byte in a packed structure if and only if the number of bits required to store all values of the type is less than or exactly eight bits. For example, the standard type Char or Boolean require less than or equal to 8 bits to represent all of their values, therefore in a packed structure Char and Boolean are allocated a byte of storage, but otherwise require a word of storage.

Calling Conventions

This section outlines the Complete Pascal compiler conventions for calling procedures and functions, parameter passing, and function result values.

Calling a Subprogram

Complete Pascal uses a stack-based parameter passing convention for calling subprograms. Before calling a procedure or function, the parameters are pushed onto the stack in the same order as their declaration. If a function is being called, the storage for the function result is allocated on the stack before pushing any parameters. After the call has completed, control is returned to the calling environment with parameters (if any) removed from the stack, but the function result value (if a function) is left on top of the stack. The calling environment must remove the function result value after it has used it.

Following is skeleton code for a procedure call.

```
        lda     pppp          ; Push first parameter
        pha
        ...
        lda     pppp          ; Push last parameter
        pha
        jsl     >Aproc              ; Call the procedure
                              ; Parameters removed from stack
```

The following is skeleton code for a function call.

```
        pha                   ; Reserve space for result value
        lda     pppp          ; Push first parameter
        pha
        ...
        lda     pppp          ; Push last parameter
        pha
        jsl     >Afunc              ; Call the function
                              ; Parameters removed from stack

        pla                   ; Remove function result from stack
                              ;  and place in accumulator
```

Subprograms are always called in full 65816 native mode (ie. 16-bit accumulator and index registers). Therefore, if the processor is not in full native mode before the call, it is forced to full native mode before the call is made. For example, assume that the accumulator is in 8-bit mode and the index registers are in 16-bit mode, then the following code is generated.

```
                              ; Accumulator currently in 8-bit mode
        rep     #$20          ; Change accumlator to 16-bit mode
        LONGA ON
        jsl     >ASubprog
```

If the subprogram being called is declared at a level other than the global level (not in the program or unit block), then a *static link* is pushed on the stack after all the parameters have been pushed. The static link serves as a mechanism to address local variables in nested stackframes. Because of this static link, the address of a nested subrogram should never be passed to an Apple IIGS Toolbox routine as a definition procedure since this is not the calling convention the Toolbox expects.

Variable Parameters

Actual variable parameters (var parameters) are always passed by reference to the formal parameter, that is, as a pointer that points to the storage occupied by the actual parameter. The pointer is passed as a 32-bit (4-byte) value. The high order word is pushed first followed by the low order word.

Consider the following example for a passing the global variable GlobVar as a variable parameter using the absolue addressing mode.

```
pea    GlobVar|-16  ; Push high order word first
pea    GlobVar      ; Push low order word second
```

Value Parameters

Actual value parameters are passed with their value on the stack or by reference depending on the size of the value. If the size of the value parameter occupies 4 bytes or less, then its value is passed on the stack and is the formal paramter. If the size of the value parameter occupies more than 4 bytes, then a 32-bit pointer to the value is passed on the stack. The called procedure or function then copies the value into local storage for the formal parameter so that changing the value of the formal parameter does not affect the value of the actual parameter.

Static Parameters

Actual static parameters are passed on the stack exactly like actual value parameters. The difference between static and value parameters is that if the size of the actual parameter is greater than 4 bytes, the called procedure or function DOES NOT copy the value into local storage for the formal parameter. Thus, it is illegal to give the formal static parameter a new value since it would change the value of the actual parameter.

Static parameters have been introduced in Complete Pascal to conserve the amount of space used by the runtime stack for storing formal parameters as well as improve the execution speed of programs since no unnecessary copying is performed.

❖ **Note:** Complete Pascal does not check that a new value is never assigned into a static parameter. It is the responsibility of the programmer to ensure that static parameters are used correctly.

Function Results

Storage for function results is reseved on the stack by the calling subprogram before any parameters are pushed onto the stack. If the function result is of type *Integer, LongInt, Char, Boolean,* or any subrange, enumerated or pointer type, or the real type *Single*, 2 or 4 bytes of storage are allocated. If the result type requires only 1 byte of storage, 2 bytes are allocated and the low memory address byte contains the value.

If the result type is of type *Double, Comp, Extended,* or any array, string, or record type, then the calling subprogram allocates temporary space within its stackframe for the result value, and pushes a 4 byte pointer to the temporary storage. The calling subprogram removes the pointer from the stack when the function returns, and the temporary storage is deallocated when no references to the value exist.

Entry/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 as follows:

```
        phd                 ; Save previous frame pointer
        tsc
        sec
        sbc     #xx
        tcd                 ; Establish new frame pointer
        clc
        adc     #yy
        tcs                 ; Allocate local storage
```

First, the direct page register from the previous activation is saved. The direct page register is used as a *frame pointer* for an activation. The saved frame pointer is called the *dynamic link*, and is required to restore the state of the previous activation.

After saving the previous frame pointer, *xx* bytes are subtracted from the current stack pointer to establish the frame pointer for this activation. *xx* is computed so that the first *word* of storage in the stack activation (ie. the function result or first parameter) is at direct page offset 254. Choosing this offset allows all parameters and as many local variables as possible to be addressed using the very efficient direct page addressing mode.

Once the frame pointer for the activation is established, *yy* bytes are added to that value to allocate the storage needed for local variables, value parameters copied local, and compiler temporaries.

Note that no registers are saved, and it is assumed that the processor is in full native mode.

The standard exit code is as follows:

```
        tdc
        clc
        adc     #xx
        tcs                 ; Deallocate local storage
        pld                 ; Restore previous frame pointer

        lda     2,S
        sta     mm,S
        lda     1,S         ; Move the return address down over
        sta     mm-1,S      ;     the parameters
        tsc
        clc
        adc     #mm-2       ; Deallocate the parameters
        tcs
        rtl
```

Local storage is first removed by adding the value *xx* to the frame pointer. Then the frame pointer from the previous activation is restored with the pld instruction. Then the parameters are "removed" from the stack by moving the return address down overtop of the first parameter(s) and then positioning the stack pointer to the new location of the return address. And finally, the RTL is executed to return to the calling subprogram (typically in the Apple IIGS ROM).

Note that if a procedure or function does not have any parameters then its exit code is the following:

```
tdc
clc
adc     #xx
tcs                     ; Deallocate local storage
pld                     ; Restore previous frame pointer
rtl
```

Appendix E

Complete Pascal versus TML Pascal

Complete Pascal implements many new features and makes several changes to the original TML Pascal product for the Apple IIGS. In this appendix we review the significant differences between these two products in order that users of the original version of TML Pascal can more quickly adapt to Complete Pascal.

The following sections are organized based on the chapters in this manual. Each section addresses the signifanct changes in Complete Pascal as they are mentioned in this book.

Chapter 1 – Getting Started

Perhaps the most imporant and significant change in Complete Pascal is that it is specifically designed for System Software version 5.0 and GS/OS. Complete Pascal, and programs created with Complete Pascal, cannot run on any version of ProDOS/16 or versions of GS/OS earlier than version 5.0.

This restriction is primarily because of the heavy use of the Resouce Manager by Complete Pascal which, of course, is not available in versions of the Apple IIGS System Software earlier than 5.0.

Chapter 2 – Using the Desktop Environment

The Complete Pascal editor n~w allows for an unlimited number windows open on the desktop. In addition, windows can use any font and font size for the text and each window may have a different tab setting. In addition, the editor now supports the Undo command.

The editing environment also supports a second window type for editing resource files.

Chapter 3 – Creating Programs

The file naming conventions have changed for Complete Pascal. Pascal source code files should end with the suffix ".p" instead of ".pas". This allows for two additional characters in the filename of source code files. The filename conventions for compiled unit symbol files has changed as well. Instead of the ".usym" suffix, the compiler appends a ".o" to the source code file name resulting in a suffix of ".p.o". These new conventions must be followed when using Units in order that the compiler can locate compiled symbol files on the disk.

The Resources... menu item in the Compile menu has been added to specify which resources should be compiled into a final application.

Chapter 4 – Creating Resources

Resources are a completely new feature to Complete Pascal. Programs written with TML Pascal should be converted where possible to take advantage of the new resources cababilities of System Software version 5.0.

Chapter 7 – Textbook Graphics Applications

TML Pascal implemented the "Plain Vanilla" application type. This has been replaced by the "Textbook Graphics" application. Previously, a program could place the parameters Input and Output in the program header to invoke the Plain Vanilla environment. For example:

```
    Program Test(Input,Output);
```

Plain vanilla turned on the grahics screen in 640 mode and created a large window with the title "TML Pascal". This allowed for easy programming of simple graphics.

Complete Pascal has changed this feature by adding the predefined procedure titled "Graphics". The procedure has a parameter allowing the program to specify either 320 or 640 mode graphics, and instead of creating a window on the screen, the program can now use the entire graphics screen.

Chapter 8 – Desktop Applications

Desktop applications should now use resources to create menus, windows, dialogs, etc. Resources are created using the TML Resouce Editor. Resources are then added to the compiled program by selecting a resource file with the Resources... menu item in the Compile menu.

Chapter 9 – New Desk Accessories (NDAs)

The source code for a New Desk Accessory should now be written as a Unit rather than a program. Desk accessories do not have a main program, thus it is not necessary to use the program structure.

Chapter 10 – Classic Desk Accessories (CDAs)

The source code for a Classic Desk Accessory should now be written as a Unit rather than a program. Desk accessories do not have a main program, thus it is not necessary to use the program structure.

Chapter 11 – Tokens

Complete Pascal now defines three real number constants to facilitate the writing of numerical applications. The compiler defines Inf (Infinity), Nan (Not A Number) and PI.

Chapter 19 – Input and Output

Complete Pascal now implements the Open procedure for opening a file for random read/write access.

Complete Pascal now supports GS/OS. Thus, filenames may use any legal ProDOS/16 or GS/OS pathname reference. In addition, any legal predefined or generated GS/OS device name may also be used for I/O. In particular, the device name used to access the printer is now ".PRINTER" and not "PRINTER:" Note that the device name ".PRINTER" is case sensitive and must be spelled with upper case letters.

Chapter 20 – Standard Procedures and Functions

The procedure Graphics is new to Complete Pascal.

The following standard procedures and functions have been renamed with Complete Pascal in order to conform with naming standards adopted by Apple Computer.

TML Pascal name	Complete Pascal name	
BitAnd	BAND	
BitOr	BOR	
BitXor	BXOR	
BitNot	BNOT	
BitSL	BSL	
BitSR	BSR	
BitRotl	BROTL	
BitRotr	BROTR	
HiWord	HiWrd	(IntMath.p has HiWord)
LoWord	LoWrd	(IntMath.p has LoWord)

Finally, the predefined global variable which returns the error code from Toolbox procedure and function calls has been renamed "_ToolErr". The name "ToolErrorNum" is still supported, but the new name should be used when new code is written to conform to the new naming standards.

Appendix B – Compiler Directives

The following three compiler directives have been renamed.

$DeskAcc	has been renamed	$NDA
$P	has been renamed	$U
$XrefVar	has been renamed	$J

The following compiler directives have been added since TML Pascal v1.0. However they were available in TML Pascal v1.50

$CDA
$DefProc

Appendix C – Toolbox Interfaces

The Apple IIGS Toolbox interfaces have gone through substantial changes. Several new interfaces have been added to support System Software version 5.0 and all previous interfaces have been changed to conform to the new standards adopted by Apple Computer.

Complete Pascal has chosen to use the new standard interfaces created by Apple Computer in order that Complete Pascal programmers may take advantage of the documentation, technical notes, examples and many other technical resources created by Apple Computer which will use Apple's new interfaces. TML Pascal programs will require some degree of change to use the new interfaces, but will enjoy the benifit of standardization.

Appendix D – Inside TML Pascal

The Boolean type, enumeration types and small integer subranges are now represented using a word (2 bytes) of memory rather than just one byte of memory. However, when these types appear in a *packed* record or array they are represented as using one byte of memory.