SOFTECH MICROSYSTEMS

Software
Reference Library

The MacAdvantage™: UCSD Pascal®

Thank you for your purchase of **The MacAdvantage: UCSD Pascal**. You have selected a precision engineered product that will allow you to write sophisticated applications on your Macintosh. But before you start, please read the following:

- **REGISTRATION CARD.** Please complete and return the enclosed registration card immediately. Not only will it serve as your "key" to our Customer Support Department, but it will also allow us to keep you informed of new releases and other information that may interest you.

- **UPGRADES.** We are continually making our software better by adding new features and by correcting problems. Customers who return their registration card will be eligible to upgrade their software for a nominal charge to cover our costs. You will be notified by mail when new releases are available.

- *INSIDE MACINTOSH.* Although we have documented our software in detail, we highly recommend that you obtain a copy of *Inside Macintosh* for more information on developing Macintosh applications. *Inside Macintosh* is available through:

  Apple Computer, Inc.
  467 Saratoga Avenue; Suite 621
  San Jose, CA 95129

- **APPLICATIONS.** If you plan to distribute applications that you write using this product, you will be pleased to know that we offer several economical licensing plans. Please contact our Customer Sales Department at (619) 451–1230 for more information. Additionally, we may be interested in publishing your application through our distribution channels. If you are interested in having SofTech Microsystems market your application, please contact our Applications Product Marketing Manager.

SofTech Microsystems, Inc.

The MacAdvantage: UCSD Pascal

SofTech Microsystems, Inc.
San Diego, California

1–182–MA

Printed in the United States of America.

Disclaimer

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

PREFACE

Congratulations on your purchase of UCSD Pascal for your Apple Macintosh computer!

The discussions in this manual assume that you are already familiar with your Macintosh. If you are not, we suggest that you first read the introductory chapters of *Macintosh*, your owner's guide.

The product you have purchased contains a UCSD Pascal compiler and a group of program development tools. The tools include a program and text file editor, a resource compiler, a symbolic debugger, a librarian utility, a runtime option configuration utility, and a set of interface units to the Macintosh ROM.

With these tools, you can build sophisticated application programs directly on a Macintosh with 128K or 512K of memory. The Macintosh interface units give you access to virtually all of the Macintosh ROM routines. Thus, you can write programs that make use of overlapping windows, a menu bar and desk accessories. We have included an example program that shows you how to access some of these features.

The Pascal language supported by the compiler is an extended version of UCSD Pascal designed for access to the Macintosh ROM. The new language features include:

- Support for 32–bit integers (type integer2).
- A new setlength intrinsic that makes it easier to set the length of a string.

- New bit manipulation intrinsics: band, bor, bxor, shiftleft, and shiftright.

- An enhanced sizeof intrinsic that allows you to specify the units that sizeof counts in.

- Pointer intrinsics that help you to make use of 32–bit absolute addresses used by the Macintosh ROM: adr, pointer, offset, ptrinc, absadr, reladr, absmove, derefhnd, and locate.

- A new type of **external** procedure that generates an in–line call to a Macintosh ROM routine.

UCSD Pascal programs are supported by a sophisticated runtime package that eliminates many of the worries associated with writing large programs. The runtime package

- provides simplified I/O through the Pascal I/O intrinsics.

- supports dynamic memory management through the Pascal intrinsics new and dispose.

- handles dynamic segment overlays automatically.

TABLE OF CONTENTS

TABLE OF CONTENTS

1
GETTING STARTED

This chapter gets you started writing UCSD Pascal programs for your Macintosh. The chapter is organized into the following sections:

HARDWARE REQUIREMENTS discusses the hardware components that are required or recommended for effective use of this product.

DISK CONTENTS details the composition of the disks you received with this product.

BACKING UP DISKS tells you how to make back up copies of your master disks.

RUNNING A PROGRAM guides you through the steps of creating and running a simple UCSD Pascal program.

ORGANIZATION OF THE MANUAL introduces you to the organization of the remainder of this user manual.

HARDWARE REQUIREMENTS

The product you have purchased is designed to work on the Macintosh with 128K or 512K bytes of memory and on Lisa under MacWorks. Programs may operate slightly differently in different hardware environments, based on memory size. In particular, when running on a machine with more memory, programs will tend to run faster and be able to handle more data.

Although this product will run on a one drive Macintosh, if you plan on developing programs that use the Macintosh interface we strongly suggest that you use two disk drives.

One option of the Debugger allows you to interact with it using an external terminal attached to the Printer port on the back of your Macintosh. An external terminal is not necessary for using the Debugger, but if you have one, it can make debugging easier, particularly when writing programs which put up windows on the screen.

DISK CONTENTS

You received two disks when you purchased this product. One of the disks, labeled **UCSD Pascal 1**, is a bootable Macintosh disk. The other disk, labeled **UCSD Pascal 2**, is not bootable.

The following files are located on **UCSD Pascal 1**:

- **Set Options.** Set Options is a utility program that allows you to set the runtime options of a code file.
- **Mac Library.** Mac Library is a collection of interface units that are used by programs that access the Macintosh ROM routines.
- **Compiler.** Compiler is the UCSD Pascal compiler.
- **Editor.** Editor is a program and text file editor.
- **Executive.** Executive provides menu style access to your program development tools.
- **Pascal Runtime.** Pascal Runtime is the runtime support package for Pascal programs.
- **p–Machine.** p–Machine is the virtual machine emulator that supports running the p–code generated by the UCSD Pascal compiler.

- **Empty Program.** Empty Program contains the standard program resources.

Three of the files, Pascal Runtime, p–Machine and Empty Program, are located within a folder called Pascal Folder.

The following files are located on **UCSD Pascal 2:**

- **RMaker.** RMaker is a resource compiler program that allows you to add your own resource definitions to a program.
- **Librarian.** Librarian is a utility program that allows you to combine UCSD Pascal units into a single library file.
- **Debug Runtime.** Debug Runtime is a version of Pascal Runtime that contains the Debugger and performance monitor.
- **Errorhandl.CODE.** Errorhandl.CODE is a utility unit that provides various program control functions to the user.
- **Mac Interface.** Mac Interface is a library of code files that contain the interface to the Macintosh ROM routines.
- **Grow.** Grow is the source to an example UCSD Pascal program that accesses the Macintosh ROM to handle a menu bar, windows and desk accessories.
- **Grow.R.** Grow.R is the resource definition file for the Grow program.

Two of the files, Grow and Grow.R, are located within a folder called Example Folder.

BACKING UP DISKS

You should immediately make a backup copy of the disks that you received with this product. This will insure that you don't accidentally loose any information contained on the disks.

Macintosh, your user's guide, describes in detail how you make backup copies of disks on the Macintosh. Here is a summary of the steps:

1. Insert the disk you want to copy.

2. Insert the disk you want to copy to.

3. Drag the icon of the disk you want to copy to the icon of the other disk.

If you have a one drive Macintosh it is faster for you to use the Disk Copy program to make backup copies of your disks.

Once you have made the backup copies, put the copies in a safe place.

WARNING: You cannot arbitrarily move UCSD Pascal programs to different volumes and expect them to run. The names of the two runtime support files are embedded in each code file. If you move a code file to a different volume, you may need to update the runtime support file names with the Set Options utility. See the GENERAL OPERATIONS chapter for details.

RUNNING A PROGRAM

This section guides you through the steps of compiling and running a simple Pascal program. Even if you don't know the Pascal language, you should be able to follow the steps outlined here.

Boot up your Macintosh with the **UCSD Pascal 1** disk. All of the operations described below will be done on this disk.

Editing the Program

First you must create a text file to compile. You create a text file by using the Editor. Start the Editor by double–clicking its icon.

You can probably figure out by yourself how to run the Editor, based on your knowlege of MacWrite. If you are not familiar with MacWrite, or if you have trouble using the Editor, refer to the EDITOR chapter.

Enter the program listed below, or a program of your own design:

```
program first;
begin
  writeln('hi there');
  readln;
end.
```

Now exit the editor, saving what you have typed in a file called FIRST.

Compiling the Program

The compiler translates the program you have edited into an executable code file. You start the compiler by double–clicking its icon. The compiler will ask you four questions:

1. **Compile what text?** Type FIRST, then press <Return>.

2. **To what code file?** Press <Return>. The output will be put in FIRST.CODE, by default.

3. **Use what resource file?** Press <Return>. The compiler will use the standard resources from the Empty Program file.

4. **File for listing?** Press <Return>. This disables listing generation.

If all goes well, the compiler will write something like the following to your screen:

```
<    0>..
TEST
<    2>..
TEST        .
     5 lines compiled in 0:00:20, 15 lines per minute
```

If the compiler finds a problem in your program, it will generate a syntax error message. At this point, you should press <Enter> to exit the compiler, then fix the problem using the Editor. Check that you typed in the example program exactly like it appears above, then recompile the program.

Running the Program

If the compiler has run to completion, you will find a file called FIRST.CODE on your disk. This file contains the Pascal code generated from your text file by the compiler. Running the program is easy—just double–click its icon. If you used the example program listed above, you should see the words

```
hi there
```

printed to the screen when you run the program. Press <Return> to terminate the program.

This section showed you how to run a very simple Pascal program. For more information about compiling and running programs, refer to the GENERAL OPERATIONS chapter.

ORGANIZATION OF THE MANUAL

This section describes the content of each chapter in the manual and gives some hints on how to use the manual.

Chapter 2, GENERAL OPERATIONS, discusses how to compile and run a program and how to develop an application.

Chapter 3, EDITOR, covers how to run the program and text file editor.

Chapter 4, PASCAL LANGUAGE, is a supplement to *The UCSD Pascal Handbook*. It describes the new language features found in this version of UCSD Pascal.

Chapter 5, MACINTOSH INTERFACE, discusses how to use the Macintosh interface units to call the Macintosh ROM.

Chapter 6, RMAKER, describes the resource compiler program, which allows you to add resources to a code file.

Chapter 7, LIBRARIAN, describes the Librarian utility, which allows you to combine Pascal units into a single library file.

Chapter 8, DEBUGGER, describes the operation of the Pascal debugger, which allows you to set break points, single step p–code, and examine and patch memory.

Chapter 9, MEMORY MANAGEMENT, describes the memory management of this implementation of UCSD Pascal. This chapter will be useful if you need to understand Pascal's memory management in order to write an application program.

Chapter 10, P–MACHINE ARCHITECTURE, describes the p–code instruction set that is supported by the underlying p–Machine. You will need to refer to this chapter if you use the Debugger.

Appendix A, MACINTOSH INTERFACE, contains listings of the Macintosh interface units, and index of interface identifiers and a table of unit dependencies.

Appendix B, ERROR MESSAGES, lists the error messages that may be generated by programs and the runtime support package.

Appendix C, P–CODE TABLES, contains numerical and alphabetical p–code tables. An index is also provided which you can use to locate the description for a p–code within the P–MACHINE ARCHITECTURE chapter.

If you are not a Pascal programmer, we suggest that you read the tutorial section (Part II) of *The UCSD Pascal Handbook* first. This will give you a quick introduction the the Pascal language. You can use some of the example programs to practice editing and compiling programs on the Macintosh. **WARNING:** a few of the programs are not appropriate for this version of UCSD Pascal.

If you are already a Pascal programmer, start by reading the first two sections of the GENERAL OPERATIONS chapter. This will give you the details of compiling and running Pascal programs. Next, you should read the EDITOR and PASCAL LANGUAGE chapters. *The UCSD Pascal Handbook* will be useful if you are not familiar with the UCSD dialect of Pascal. You may want to read the DEBUGGER chapter to learn how to use the Debugger. Some further secions of the GENERAL OPERATIONS chapter may be useful.

If you want to write programs that call the Macintosh ROM routines to do graphics or to display windows and menu bars, you must first acquire a copy of the *Inside Macintosh* manual. As of this printing, *Inside Macintosh* is only available in draft form from Apple. *Inside Macintosh* gives you the definitions of the Macintosh ROM routines. You must use *Inside Macintosh* in conjunction with the MACINTOSH INTERFACE chapter of this manual. Also, you will need to be very familiar with the UCSD Pascal extensions described in the PASCAL LANGUAGE chapter.

Finally, if you want to build sophisticated applications on the Macintosh you will need to read the RMAKER chapter and the later sections of the GENERAL OPERATIONS chapter.

2
GENERAL OPERATIONS

This chapter contains information and instructions on using **The MacAdvantage: UCSD Pascal.** It explains how to use this product to create UCSD Pascal programs for your Macintosh. Your programs can take full advantage of the power of UCSD Pascal and the Macintosh ROM to provide meaningful solutions to the kind of applications the Macintosh was designed to solve.

This chapter consists of five sections as follows:

CREATING PROGRAMS instructs you on use of the compiler.

RUNNING PROGRAMS contains the information you need to take full advantage of the UCSD Pascal runtime environment.

USING EXECUTIVE explains the operation of the Executive utility which you can use to make your program development process easier and faster.

ACCESSING FILES describes how your programs can interact with Macintosh files and serial devices.

BUILDING AN APPLICATION outlines the steps you need to go through in order to construct a sophisticated Macintosh application.

CREATING PROGRAMS

This section discusses how to run the UCSD Pascal compiler to create programs for your Macintosh. For information on the UCSD Pascal language, refer to *The UCSD Pascal Handbook* and the PASCAL LANGUAGE chapter.

Using the Compiler

The Compiler takes a text file as input and generates a code file as output. The code file generated consists of two parts: the data fork and the resource fork. The data fork contains p–code, which is executed by a p–Machine emulator. The resource fork contains information about the runtime environment required by your program. More information on the resource fork of an application can be found later in this chapter and in the chapter RMAKER.

The Compiler will accept for input any standard Macintosh text file. This file will usually be generated by the Editor supplied with this compiler, but it could be generated by MacWrite or by another Pascal program. If you use MacWrite files as input to the compiler, you must specify that the output file from MacWrite be stored in "text only" mode.

You start the Compiler by opening its icon:

Figure 2–1. Compiler Icon.

Responding To Startup Questions

The Compiler begins by asking four questions to obtain file names. Either the Macintosh or Pascal I/O conventions for file names may be used. These conventions are defined in File

Naming Conventions later in this chapter. The Compiler accepts only 40 characters of input to each question, so be sure to enter no more than 40 characters. Entering more than 40 characters will cause a string overflow runtime error to occur.

The first question asked is:

```
Compile what text?
```

The possible responses to this question are:

- Entering the name of the text file you wish to compile. The Compiler uses the name exactly as you specify it, including leading, embedded, and trailing blanks. It does not append any kind of suffix to the name you specify in order to locate the file.

- Pressing <Return> or <Enter><Return> to terminate the compilation without generating an output code file.

The second question asked is:

```
To what code file?
```

You should respond to this question in one of the following ways:

- Entering the name of the code file you wish the compiler to create. The Compiler will add a .CODE suffix to the name you specify. (The suffix is added by the Compiler only as a safeguard to prevent the accidental destruction of text files. It is not necessary to maintain the suffix for execution of the resulting code file.)

- Pressing <Return>. This causes the Compiler to generate its output to a code file with the same name as the input file with a .CODE suffix added. If you choose to use this default, be sure that you did not specify an input text file name longer than 35 characters.

- Pressing <Enter><Return> to immediately terminate the compilation.

The third question asked is:

```
Use what resource file?
```

This question is asking you to specify a source for the resources that the Compiler should copy to the output code file. You should respond to this question in one of the following ways:

- Entering the name of a file that contains the resources you want copied.
- Pressing <Return>. This directs the Compiler to attempt to copy the resources from the file Empty Program. (The file Empty Program must be on the same disk as the Compiler.) As supplied to you, Empty Program contains the standard set of resources required by a UCSD Pascal program.
- Pressing <Enter><Return> to terminate the compilation.

The resource file name you specify should either be Empty Program or another file known to have valid UCSD Pascal resources. Such files can be created either with the Compiler or RMaker. The Compiler will use the resources of the file you specify, regardless of whether they are valid UCSD Pascal resources. Should the file you specify not have valid UCSD Pascal resources, the resultant object code file will be unuseable. For more information on creating resource files, see the RMAKER chapter.

If the text file you are compiling is not a **program** (i.e. you are compiling one or more units), the standard set of resources in Empty Program will always be sufficient.

You cannot specify the same name for your resource file source as you specified for the code file. This means that if you want to preserve a unique set of resources for your program to be used each time it is compiled, these resources will have to be stored in

a file which has a different name than that which you are giving to your program.

The fourth question asked by the Compiler is:

```
File for listing?
```

You should respond to this question in one of the following ways:

- Pressing <Return> if you do not want the Compiler to generate a listing.
- Entering the name of the file or Macintosh serial device to which you want the compiled listing written. Unless the first character of the file name is a period, the suffix .LIST will be added.
- Pressing <Enter><Return> to terminate the compilation.

After the Compiler is finished, you may examine or print the listing file using the Editor. Note, however, that listing files consume large amounts of disk space. Should the disk containing the listing file become full during compilation, the Compiler will abort and both the code and listing files will be lost. A common listing output file is .BOUT, which directs the listing to the printer. Other permissable listing output files include the other serial devices: .AOUT, .CONSOLE, and .DBGTERM; the characteristics of these files are discussed in Serial Devices later in this chapter.

NOTE: When using the Apple Imagewriter in normal text mode, some print lines generated by the Compiler will be longer than 8.5". The extra characters past the end of the page margin will be over—printed on top of the beginning of the line. To avoid this, you can change the character pitch selected when the printer is powered on to ultracondensed. Page 40 of the *Imagewriter User's Manual* describes how to do this.

The $L compiler option can also be used to specify a name for the listing file.

Evaluating Compiler Progress

While the Compiler is running, it displays a report of its progress on the screen:

```
<   0>.................
INITIALI
<  19>.............................
<  50>...........
AROUTINE
<  61>.....................................
< 100>.................
MYPROG
< 119>.............................
< 150>.............

INITIALI .
MYPROG   ..

  165 lines compiled in 0:00:25, 396 lines per minute
```

During the first pass, the Compiler displays the name of each routine (INITIALI, AROUTINE and MYPROG in this example). The numbers enclosed by angle brackets, < >, are current line numbers. Each dot represents one source line compiled.

During the second pass, the names of the segments are displayed (INITIALI and MYPROG in the example). Here, each dot represents the compilation of one procedure or function.

You can suppress this output by using the $Q compiler option in your input textfile.

Syntax Errors

If the Compiler detects an error while compiling a program, it generates a syntax error. When this happens, the text where the error occurred is displayed along with an error number and message. Here is an example:

```
<   0>......
```

```
  q,r,s: string;
  w,x,y: reel <---
Syntax Error 104:Undeclared identifier
Line 7
Type <space> to continue, <Enter> to terminate
```

For each syntax error, a message like the one above is displayed. The Compiler gives you the option of pressing either <Space> to continue the compilation or <Enter> to terminate the compilation. Error numbers greater than 400 are always considered fatal and the Compiler will abort regardless of your input.

The Compiler issues three additional fatal error messages. Their occurrance is rare, as they usually mean that some kind of internal error condition has been detected. All three messages wait for you to respond to them by pressing any key on the keyboard. The actual response is immaterial; it is just an acknowledgement that you have seen the message. This is done because the screen contents will be erased by the Finder after the Compiler terminates.

```
Compilation aborted due to I/O error XXX
Press any key to exit.
```

The Compiler was unable to perform an I/O operation on one of the files it has open. XXX is the ioresult code passed back from the Macintosh Operating System. A list of these result codes appears in Appendix B. You should check that your Macintosh and its peripherals are all working correctly and then retry the compilation.

```
Error writing file, not enough room.
Press any key to exit.
```

The Compiler was unable to write a block of information to disk because the disk it was trying to write to was full. This error usually occurs when you are trying to write a listing file to disk. It can also happen when trying to write out the .CODE file you are creating. Make sure that the disk you are trying to write to is not full. Alternatively, if you are making a listing file, try

sending it to the file .BOUT. Then retry the compilation.

```
Compilation aborted due to back-end error XXX
Press any key to exit.
```

The Compiler has detected an abnormal condition within the files it creates while compiling your program. XXX is an internal code signifying the error. Retry the compilation. Please contact your technical support representative if the error appears again.

Compiled Listings

The Compiler optionally produces a compiled listing of the program. This listing contains source text, along with information about the compilation. Compiled listings are useful when you're using the Debugger.

You can produce a compiled listing in two ways. You can give a file name to the compiler's listing file question, or you can use the $L compiler option.

Here is the entire compiled listing for a small program:

```
UCSD Pascal Compiler [1R0.0]      10/ 8/84

    1   2     1:d     1  program Fact;
    2   2     1:d     1    var
    3   2     1:d     1      i: integer;
    4   2     1:d     2      prod: real;
    5   2     1:0     0    begin
    6   2     1:1     0      writeln('n  factorial of n');
    7   2     1:1    18      prod:= 1.0;
    8   2     1:1    23      for i:= 1 to 20 do
    9   2     1:2    41        begin
   10   2     1:3    41          prod:= prod * i;
   11   2     1:3    50          writeln(i,' ',prod);
   12   2     1:2    89        end;
   13   2      :0     0    end.

End of Compilation.
```

The numbers that precede each source line are:

- The first column is the line number. Line numbers start with 1 and are incremented for each line encountered by the Compiler during compilation. Lines found in files which are included by the $I compiler directive and the **uses** statement are also counted.

- The second column is the Pascal segment number. This entire example is segment 2.

- In the third column is the procedure number followed by a colon and the statement nesting level. All of the example is procedure 1. Procedure numbers are important in determining program locations either in the Debugger or when a runtime error occurs. The statement nesting level is an indication of how deeply the text is nested within Pascal structured statements. The statement nesting level field of data lines contains the letter "d".

- The fourth column contains the word offset of data or the byte offset of code. Data word offsets are relative to either the start of a segment for global data, or to the beginning of a procedure's activation record for a procedure's local data. Data offsets are useful for finding data using the Debugger. Code offsets are useful for setting break points with the Debugger.

RUNNING PROGRAMS

To run a program, either one that you have compiled or one someone else compiled, you just double—click its icon. Executing a program in this manner causes the disk it is on to become the default disk.

Once you have become familiar with creating and running programs as described here, you should also explore the faster method offered by the Executive utility which is discussed in USING EXECUTIVE.

Pressing the interrupt button on the programmer's switch will cause the currently executing UCSD Pascal program to be interrupted. The button to the rear of the programmer's switch is the interrupt button. The button to the front of the programmer's switch is the reset button. Pressing the reset button will cause the Macintosh to be restarted. Obviously, if you are in the middle of a hard to reproduce situation, you don't want to accidently press the wrong button.

When a program is interrupted, a standard Runtime Error dialog box will appear on the screen as described later on in the Runtime Errors section. Refer to that section for instructions on the options available when a **Program interrupted by user** runtime error occurs.

NOTE: The Runtime Support Library disables the interrupt button while it is starting up a program. Also, if you have one of Apple Computer's MacsBug debuggers installed, pressing the interrupt button while running a UCSD Pascal program will cause you to enter MacsBug. If you are running UCSD Pascal programs under the MacWorks software on a Lisa, there is no way to interrupt a program and receive the standard Runtime Error dialog box.

You may need to do some more steps before a program you just compiled is ready to run. You may need to run the Set Options utility to change the default runtime environment for your program; you may also need to run RMaker to install some additional resources.

The p–code produced by the UCSD Pascal compiler resides within the data fork of the output file. The resource fork of the file usually contains a standard set of resources that are used to start up (bootstrap) the p–code file. The standard resources are explained in detail in the section Standard Resources.

The important thing you need to know about the standard resources is that some of them define the runtime environment a program starts up in. The effects of the settings of these standard resources are explained in the next three sections.

Required Files

In order to run a program that was compiled with the UCSD Pascal compiler, two Pascal Runtime Files must be available to the program. One of the files is named Pascal Runtime and the other is named p–Machine. The p–Machine file is the p–Machine emulator program, which allows p–code to run on the Macintosh. The Pascal Runtime file is a group of Pascal and assembly language routines that support running UCSD Pascal on the Macintosh. The Pascal Runtime file is also called the Runtime Support Library. Usually, these files are found in the folder called Pascal Folder. As it is supplied to you, the Pascal Folder is located on the **UCSD Pascal 1** disk.

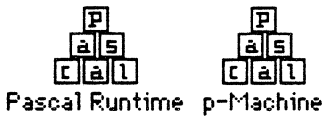


Figure 2–2. Pascal Runtime and p–Machine Icons.

The resource fork of every UCSD Pascal program contains references which define the location and names of these files. These references consist of file names which adhere to the Macintosh file naming conventions. These conventions are described in File Naming Conventions later in this chapter. Should the Pascal Runtime Files be on the default disk, you can omit the volume name. Note that the two files do not need to be on the same disk. The p–Machine file is read only when your program is started and not used thereafter. This means that you can keep it on a separate disk which can be removed from the Macintosh after the program starts.

Two versions of the Runtime Support Library were shipped to you. The first, named Pascal Runtime, is on the **UCSD Pascal 1** disk. It contains the necessary runtime support for executing UCSD Pascal applications. The other, named Debug Runtime, is on the **UCSD Pascal 2** disk. It provides the same runtime services as Pascal Runtime and in addition, provides the Debugger and the Performance Monitor tools. The usage of these additional tools is described in the DEBUGGER chapter.

Empty Program, in its original form, as supplied to you on the **UCSD Pascal 1** disk, specifies no volume name in the references to the Pascal Runtime Files. Hence, the resources in Empty Program assume that the required files are located on the default disk. Furthermore, the names of these files are assumed to be Pascal Runtime and p–Machine.

Any program you compile that uses Empty Program as its source for resources will inherit these references to the required Pascal Runtime Files. Should this configuration (i.e. the names of the files or their locations) not suit your requirements, you can use Set Options to change the file names and locations in each program you compile. Alternatively, Set Options can be used to change the names and locations specified in Empty Program.

If one or the other of these Pascal Runtime Files is not available to your program, an error message describing the problem will be displayed when you attempt to start the program.

Startup Options

The settings of five Startup options are contained within a program's standard resources that specify the the runtime environment in which your program executes. These option settings are obtained by the Compiler from the resource file you specify when a program is compiled. Each option is described below, along with the default setting specified in Empty Program.

- **Create Default Window.** The default value of this option is enabled. If this option is enabled, a standard program window is opened by the bootstrap. The title of the window is the value of the version number string, if it is nonempty. Otherwise, the title is the file name of the program. (The version number string is another type of resource. See Standard Resources for instructions on defining a non–empty version number string.) If this option is disabled, no default window is opened. To see how this affects which ROM initialization routines are done by the bootstrap, see Initialization in the MACINTOSH INTERFACE chapter.

- **Create .DBGTERM Device.** The default value of this option is disabled. If this option is enabled, the .DBGTERM device is available to the program. See Special Devices for details about the .DBGTERM device.

- **Startup in Debugger.** The default value of this option is disabled. If this option is enabled, the bootstrap calls the Debugger before the program starts. See the DEBUGGER chapter for instructions on using the Debugger. The Debugger interacts either by using the Macintosh screen and keyboard through the .DBGTERM device, or by using an external terminal, based on the setting of the Debug to Modem Port option. The Startup in Debugger option must be enabled if you intend to use the Debugger at all.

- **Enable Performance Monitor.** The default value of this option is disabled. If this option is enabled, the Performance Monitor is enabled for the duration of your program. See the DEBUGGER chapter for information on using the Performance Monitor. The Performance Monitor writes information about the faults that occur during the execution of a UCSD Pascal program. This information is written either to the .DBGTERM device or to an external terminal based on the setting of the Debug to Modem Port option. See the MEMORY MANAGEMENT chapter for an explanation of the various kinds of faults.

- **Debug to Modem Port.** The default value of this option is disabled. This option has no meaning unless either the Startup in Debugger option or the Enable Performance Monitor option is enabled. If this option is enabled, the Debugger interacts using an external terminal connected to the modem port of the Macintosh. The modem port is the one with the telephone icon, and corresponds to the channel used for the serial devices .AIN and .AOUT.

In addition to using Set Options to change the option values assigned to your program by the Compiler, you may also override them by specifying the type which define them when using RMaker. The implementation of the Runtime Options as resources is discussed in Standard Resources.

As with the required files, the default settings are obtained by the Compiler from Empty Program. Should modifying your program with Set Options after compilation become cumbersome, you can use Set Options on Empty Program to change the default settings. This way, every time you compile, the compiler will automatically give you the file locations and option settings that you prefer.

Library Files

All of the units that a program references with the **uses** statement must be available to the program when it is executed. This can be accomplished three ways:

- Each unit can be moved into the same code file as the program. The Librarian utility, described in the LIBRARIAN chapter, does this.

- The units may be combined into a single library code file using the Librarian. If this is done, you can then use Set Options to add the name of your library code file to your program's Library Files list. The file Mac Library on **UCSD Pascal 1** is an example of such a library of units that can be referenced by your program.

- You can use Set Options to add the names of all your code files containing individual units to your program's Library Files list. This works provided that you don't have more than five code files that you want your program to reference in this manner.

As outlined above, a program's Library Files list is usually specified using Set Options. Set Options allows you to specify up to five code files. It is also possible to augment a program's Library Files list by adding the appropriate resources using RMaker, but using Set Options is easier and less error prone.

The Library Files list is used by the Runtime Support Library when it needs to locate a referenced **unit** that it cannot find inside your program's code file. When it searches for a unit, the Runtime Support Library examines the code files in the order in

which you have listed them. If a library code file listed in your program's Library Files list cannot be found, the Runtime Support Library simply ignores that entry in the list and continues its search.

The limit of five code files in the Library Files list imposed by Set Options is a practical limit rather than an absolute limit. The Macintosh Operating System limits the number of files that a program can have open simultaneously, and every code file that must be opened and examined by the Runtime Support Library increases the time required to start a program.

Using Set Options

Set Options is the utility program that you use to modify a UCSD Pascal program's Runtime Options. A program's Runtime Options specify the location of the Pascal Runtime Files, the Library Files list, and the settings of the Startup options.

Set Options is executed just like any other application: just double—click its icon.

Set Options initially presents you with a standard Macintosh file selection box. See Figure 2—3. Select the file you wish to modify by clicking its name in the selection box and then select the Open button. You can cause the files residing on the disk in the other drive to be shown by selecting the Drive button. Selecting the Eject button causes the disk in the indicated drive to be ejected; this allows you to insert another disk if you wish. To terminate Set Options, select the Cancel button.

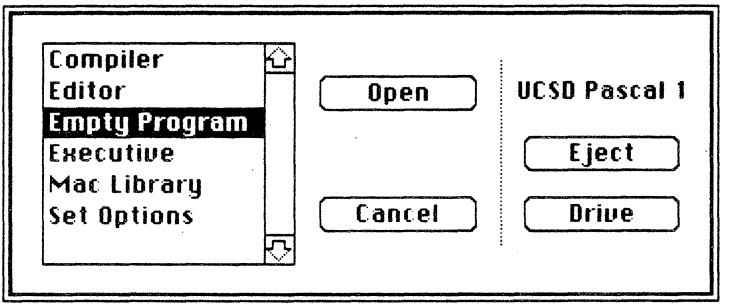


Figure 2–3. Set Options File Selection.

Once you have selected a file, a Macintosh dialog box is displayed that presents you with the settings of the current Runtime Options. See Figure 2–4.

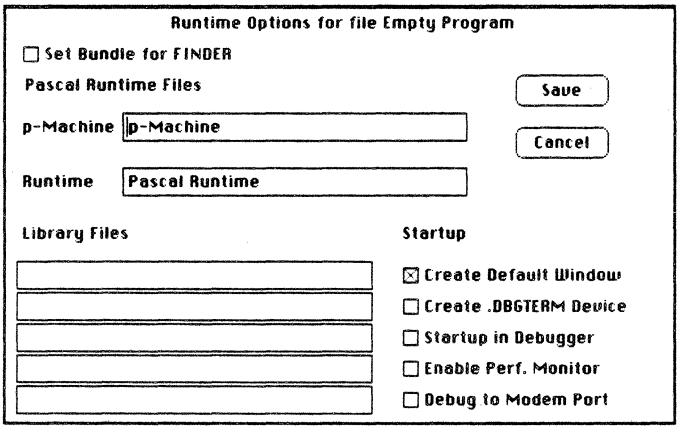


Figure 2–4. Runtime Options.

Four option groups are available:

- **Pascal Runtime Files.** These entries are the program's specification of the names and locations of the required files. Use the Mouse to move the cursor into the box for the name you wish to change. Normal Macintosh editing rules and file

naming conventions apply.

- **Startup.** These check boxes are used to specify the settings of the Startup options. An empty box indicates that the option is disabled; an X through the box indicates that it is enabled. To change the setting, move the cursor into the box and click the mouse button.

- **Library Files.** These entries make up the program's Library Files list. As explained previously, the Library Files list can identify the name and location of up to five library files. Enter or change these boxes using the methods described for changing the Pascal Runtime Files entries.

- **Set Bundle for Finder.** This check box is used to specify the setting of the program's Finder "bundle bit." The usage of the bundle bit is explained in the Application Interface to the Finder section later in this chapter. You should not change this option unless you understand why you are doing so. Indiscriminate setting of the bundle bit can cause the Desktop to become "polluted" with conflicting icon and other resource definitions. This is a condition which is often evidenced by the Finder using the wrong icons to decorate files.

To save the changes you have made, click the Save button, and Set Options will update the program with the Runtime Options shown on the screen and return you to the file selection box it presented to you earlier. Clicking the the Cancel button causes Set Options to return to the file selection box without updating the program, effectively discarding any changes you have made.

Once you have been returned to the file selection box, you can select another program and change its options, or click the Cancel button to exit Set Options.

NOTE: Set Options will not allow you to change the Runtime Options in the Set Options code file being used. To change the settings of the Runtime Options within Set Options, first use the Finder's Duplicate command to create a copy of Set Options. Then run the copy, and modify the original copy of Set Options.

Finally, exit back to the Finder and drag the copy of Set Options to the trash.

Program Startup Errors

If the Runtime Support Library has trouble starting your program, you will get a program startup error displayed within a dialog box on your screen. Actually, there are two catagories of program startup errors. The first catagory contains those startup errors which are detected and reported by the initial "bootstrap" program which is located in your program's standard resources. The second catagory contains the startup errors that can be generated by the Runtime Support Library during its construction of your program's execution environment.

The startup errors generated by the bootstrap are:

- **Could not open p—Machine file.** This error occurs if the file p—Machine could not be opened. The runtime environment description for the program's p—Machine file is wrong. Execute Set Options to correct the reference and then try the program again.

- **Could not allocate memory for p—Machine.** This error occurs if the bootstrap cannot allocate memory to read in the p—Machine file. This error should not occur; if it does, contact your technical support representative.

- **Error reading p—Machine file.** This error occurs if the bootstrap has trouble reading the p—Machine file. It is likely that your p—Machine file is damaged. Replace it, and try again.

- **Could not locate MSTR resource.** This error occurs if your program is missing the standard MSTR resource. You must use RMaker in such a way that all the standard program resources are in the resource fork of an application in addition to any new resources you define.

- **Could not open program data fork.** This error occurs if the bootstrap has trouble opening the p–code portion of your application program. Make sure you have not done any operation in building the application that might delete the p–code generated by the Compiler.

- **Could not open Runtime Support Library file.** This error occurs if the bootstrap could not open the Pascal Runtime file. Make sure that a Runtime Support Library file is installed where the program's runtime environment description says it should be. The two runtime libraries are Pascal Runtime and Debug Runtime.

- **Could not allocate stack/heap.** This error occurs if the bootstrap could not allocate a 64K byte area of memory for the Pascal Data Area. This error also should not occur and indicates a serious hardware or software failure.

The program startup errors generated by the Runtime Support Library are:

- **Error reading segment dictionary.** This error indicates that an I/O error occurred reading the segment dictionary within the program code file or a code file listed in the Library File list.

- **Error reading library.** This error indicates that an I/O error occurred reading a library code file.

- **Required unit not found ( ).** The **unit** whose name appears in the error message enclosed in parentheses is referenced by your program, but it cannot be found in the program code file or in any of the library code files listed in the Library File list.

- **Duplicate unit ( ).** This error indicates that there is more than one instance of the indicated **unit** in the program, or the unit's name is the same as one of the Runtime Support Library's units.

- **Too many library code files referenced.** This error indicates that the units used by your program are distributed into too many separate library code files. Use the Librarian utility to combine library code files.

- **Too many system units referenced.** This error should not occur. If it does, contact your technical support representative.

- **No program in code file to execute.** This error indicates that you have attempted to run a library code file that doesn't contain a **program**.

- **Program or unit must be linked first.** This error indicates that your **program** or one of the units that you are using needs to have one or more assembly language routines linked into it before it can be used. If this error occurs, it may be due to an improperly constructed Macintosh Interface **unit**, so you should contact your technical support representative.

- **Obsolete code segment ( ).** The indicated code segment was either not created properly or it was created by an incompatible version of the UCSD Pascal compiler.

- **Insufficient memory to construct environment.** There isn't enough memory for the Runtime Support Library to construct your program's environment. The best work around for this error is to combine separate library code files together into a fewer library code files. Another possible remedy is to eliminate any unneccesary entries in your program's Library File list.

- **Program environment too complicated: run QUICKSTART first.** This error indicates that the number of units used by your program and the complexity of their relationships is greater than can be handled directly by the Runtime Support Library. The QUICKSTART remedy suggested by the error message refers to a preprocessor program that you can use to prepare your program for execution. A version of this preprocessor utility is not currently available for the Macintosh environment. If you get this startup error, try using the Librarian utility to package

all of the units required by your program together with the program's code segments. If this doesn't eliminate the error, you may have to resort to merging the services provided by several small units into a single unit.

- **Error reading program code file.** This error indicates an I/O error reading your program code file.

- **Error reading library code file.** This error indicates an I/O error reading one of your library files.

- **Insufficient memory to allocate data segment.** Your program or one of the units it references has a large amount of global variables, and the Runtime Support Library is unable to allocate the storage for them in the Pascal Data Area. The most likely cause of the trouble is a declaration of one or more large array variables.

- **Insufficient memory to load fixed position segment.** A code segment containing one or more nonrelocatable assembly language routines cannot be loaded into the Pascal heap due to a lack of space in the Pascal Data Area.

- **Unknown environment construction error.** This error indicates an internal error in the Runtime Support Library's environment construction process. If you get this error, contact your technical support representative.

Runtime Errors

When the p–Machine emulator and Runtime Support Library detect certain errors, the Runtime Support Library will generate an execution error. If the Debugger is enabled and currently in its active state, then the Debugger is entered, and an error message is printed. Otherwise, the system displays the execution error message within a dialog box on the screen, and the user is given a choice of how to procede. Here is a sample execution error dialog box:

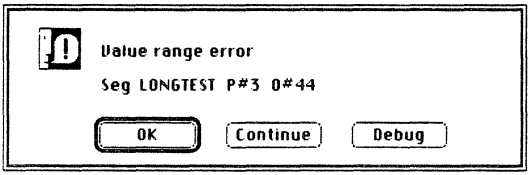


Figure 2–5. Execution Error.

The first line is the error message. The second line gives the p–code coordinates of where the error happened. In this example, LONGTEST is the segment. The procedure number is 3 and the offset within the procedure is 444. The coordinates can be checked against a compiled listing of the program to determine where in the program the error occurred.

Depending on the error, there is either one or three continuation buttons. If it is a fatal error, only the OK button is shown.

- **OK button.** Clicking this button will cause the program to terminate.

- **Continue button.** Clicking this button will cause the program to continue execution. Only some execution errors may be continued from, so you cannot depend on continuing from arbitrary errors.

- **Debug button.** Clicking this button will cause the Debugger to be invoked if it is enabled in the Runtime Options. The Debugger is enabled when the Start in Debugger runtime option is true. If the Debugger is not enabled, this button does nothing.

Here is a short explanation of each of the execution error messages:

- **Fatal runtime support error.** This error indicates a corrupted Runtime Support Library file.

- **Value range error.** This error occurs if (1) an array subscript is out of range, or (2) an assignment to a subrange variable is out of range. You can disable detection of this error by using the $R compiler option.

- **No proc in segment table.** This error should not occur on the Macintosh.

- **Exit from uncalled proc.** This error occurs when exit(A) is executed and A is not in the dynamic call chain.

- **Stack overflow.** This error occurs when there is no room in memory to expand the runtime stack by the desired amount.

- **Integer overflow.** This error occurs if (1) an integer2 operation overflows, or (2) a conversion to integer or integer2 is too large to fit in the destination type.

- **Division by zero.** This error occurs whenever a divide or mod operation is performed with a zero denominator.

- **Invalid memory reference.** This error indicates an attempt to access memory through a bad pointer or handle value.

- **Program interrupted by user.** This error occurs if the user presses the interrupt button on the programmer's switch and the Debugger is not enabled.

- **Runtime support I/O error.** This message indicates an I/O error was detected either during startup of the Runtime Support Library, or later attempting to read in a program segment. This is a fatal error.

- **I/O Error.** This error occurs if an I/O operation detects an error. You can disable I/O checking by using the $I compiler option.

- **Unimplemented instruction.** This error occurs if the p–Machine attempts to execute an invalid p–code. If you get this execution error, then something has gone drastically wrong in your program.

- **Floating point error.** This error occurs when a floating point operation overflows the size of floating point numbers.

- **String overflow.** This error occurs if (1) the source string is too large in string assignment, or (2) conversion of a number to a string overflows the size of the string.

- **Programmed halt.** This error occurs upon execution of the halt intrinsic.

- **Illegal heap operation.** This error indicates improperly paired mark and release operations, or an illegal dispose operation.

- **Break point.** This error occurs if a BPT p–code is executed and the debugger is not enabled. The BPT p–code is used by the debugger to implement break points.

- **Incompatible real number size.** This error cannot occur on the Macintosh unless you use the $R2 compiler directive, which is something you should not do.

- **Set too large.** This error occurs if an attempt is made to create a set larger than the maximum allowed size of a set. A set is allowed to have 4080 members.

- **Segment too large.** This error occurs if an attempt is made to load a segment that is over 32K bytes in size.

- **Heap expansion error.** This error occurs if there is no room for the heap to expand. This is most likely to occur due to the presence of a nonrelocatable Macintosh heap block immediately above the Pascal heap in memory. The Compiler will likely terminate with this message if you try to compile a program having too many symbols.

- **Insufficient memory to load code segment.** This error occurs if there is no more room in memory to load a required code segment. Again, the presence of locked or nonrelocatable Macintosh heap blocks can interfere with the acquisition of memory for code segments.

Refer to the P–MACHINE ARCHITECTURE chapter for additional information on execution errors.

USING EXECUTIVE

The Executive utility provides you with menu access to all of the programs that comprise **The MacAdvantage: UCSD Pascal**. That is, you can run the Editor, Compiler, RMaker, Set Options, and Librarian programs by selecting the appropriate entry in a pull–down menu. Other options on the Executive menu allows you to run any other program, or return to the Macintosh Finder program.

The advantage of using the Executive to start programs instead of the Finder is that a transition from one program to another is considerably faster. Moving between programs using the Executive is faster, because the time consuming activities related to the saving and recreation of the desktop display (done by the Finder) are avoided.

For example, the time it takes to go from the Compiler to the Editor should be reduced by approximately 50% if you use the Executive instead of the Finder to accomplish the transition. Of course, you may notice more or less time reduction depending on the number of disks you have inserted, the number of files on those disks, and the complexity of your current desktop arrangement.

When a program started by the Executive terminates, the Executive is restarted. This means that once you have started the Executive, you effectively remain inside it until you use its Quit option to reactivate the Finder.

The Executive isn't intended to be a complete substitute for the Finder. You will still need to use the Finder for a variety of tasks. Most notably, these tasks include: transferring files between disks, copying disks, maintaining the organization of the folders on your desktop, and running the Desk Accessories.

Note that it is possible to have your Macintosh start up in the Executive utility instead of the Finder if you wish. (See how to use the Finder "Set Startup" command in *Macintosh*, your user guide.)

The operation of the Executive utility is described in the following sections.

Starting The Executive

As it is supplied to you, the Executive is located on the **UCSD Pascal 1** disk. To start the Executive, double click its icon. Since the Executive is not a UCSD Pascal program, it can be run off of any disk without configuring it with Set Options.

The Executive Menu Bar

The Executive utility's menu bar consists of the following menus:

Set. The Set menu allows you to set the locations of the programs that comprise **The MacAdvantage: UCSD Pascal.**

Edit. The Edit menu will start the Editor program.

Compile. The Compile menu will start either the Compiler or RMaker (the resource compiler).

Utilities. The Utilities menu will start either Librarian or Set Options.

Run. The run menu puts up a standard file selection box. To run a program, select the program file name and select the Open button. (Or simply double–click the file name.)

Quit. The Quit menu allows you to exit back to the Macintosh FINDER.

The Editor, Compiler, RMaker, Librarian, and Set Options can also be started by entering a command key sequence from the keyboard. This is done by holding down the command key and typing the appropriate letter. The command key sequences supported by the Executive are shown in its pull down menus.

Setting Program Locations

The Executive is preconfigured to know about the locations of the Compiler, RMaker, Set Options, Editor and Librarian programs as they are shipped on the **UCSD Pascal 1** and **UCSD Pascal 2** disks. If you wish to execute these programs from other volumes (such as a hard disk) you must use the Set menu to change the location of these programs.

In the Set menu, there is one menu item for each program. Select the item that corresponds to the program whose location you wish to change. After you select the item, a dialog box will appear that contains the current location setting for the program. Type in the new location of the program, or click the Cancel button to retain the previous location setting. When specifying the location of a program, you must specify both a volume name and a file name using the standard Macintosh file name conventions. After typing in the new location, select the Save button to make the change permanent.

NOTE: If you are moving the Compiler, Librarian or Set Options programs to another volume don't forget to move the files in the Pascal Folder. You will also need to run the Set Options utility on these programs to change the location of the p–Machine and Pascal Runtime files.

If you receive the error message

```
Program XXX is not on-line
```

when attempting to start a program using Executive, check that the location for the program is set correctly.

ACCESSING FILES

Your UCSD Pascal program can access Macintosh files two ways. First, it can use the UCSD Pascal intrinsics described in *The UCSD Pascal Handbook*. Second, *Inside Macintosh* describes the interfaces to the Macintosh Operating System File Manager. By using the UCSD Pascal interfaces to these ROM routines, your program can have full access to all the file handling capability of your Macintosh.

Programs which use UCSD Pascal intrinsics to access files generally need to be aware of disk volume names or disk drive assignments. Their user interface has to be tailored accordingly. Two examples of programs like this are the Compiler and Librarian. Programs which use the Macintosh Standard File Package and File Management units generally don't need to worry about these details. Examples of this type of program are Editor, Set Options, and RMaker.

Regardless of which method you choose to use, this section provides you with information to help interface your program to Macintosh files.

File Naming Conventions

File names can be specified using the conventions of the Macintosh Operating System. These file naming conventions are as follows. A file name consists of up to 255 characters. Any character except a colon (:) may be used in a file name. In particular, spaces are allowed in a file name. File names are not case–sensitive for the purpose of comparison. However, the original type case of the name is retained in the directory when a file is created. Here are some example file names:

```
MYFILE
A rather long file name.
My File
```

The first and the third file names in the example are distinct names, because of the presence of a space in one of them. Remember that all spaces are considered part of the file name——even trailing spaces.

NOTE: According to *Inside Macintosh* there is a practical limit of about 40 characters for a file name.

A file name may be preceeded by an optional volume name, separated from the file name by a colon. A volume name may be up to 27 characters long, and may consist of any characters except a colon (:). Volume names follow the same case convention as file names. Here are some examples of file names preceeded by volume names:

```
My Disk:My File
Mac Boot:PBOOT
```

Any file name that is opened using the Pascal reset or rewrite calls may use some additional conventions supported only by the Pascal Runtime Package. These conventions are called Pascal I/O file naming conventions.

A volume may be refered to by the drive number of the disk drive it is mounted in. A drive number is represented by a number sign (#) followed by a positive integer representing the drive number. #1 refers to the internal drive. #2 refers to the external drive. Higher numbers refer to other drives that your Macintosh knows about. Which numbers correspond to which drives is system—specific.

WARNING: Drive numbers are used to open the named file on any disk in the specified drive. Should you be using multiple disks in the specified drive, the use of drive numbers is dangerous. A file will not be found or will be created on the wrong disk if the disk in the disk drive changes before the file is opened.

Here are some file names preceeded by drive numbers:

```
#1:My File
#2:RMaker
```

You may also specify a device by name. The syntax of a device name is the same as for a volume name, except that a device name may not be followed by a colon or a file name. All device names begin with a period (.) character, by convention. Here is a list of the standard Macintosh device names:

- **.AIN** is used to receive input from the modem port.
- **.AOUT** is used to send output to the modem port.
- **.BIN** is used to receive input from the printer port.
- **.BOUT** is used to send output to the printer port.

The Runtime Support Library also supports some nonstandard serial devices:

- **.CONSOLE** refers to a terminal–like device that uses the keyboard and the current QuickDraw grafport on the Macintosh screen.
- **.SYSTERM** refers to a device that is identical to characters are not echoed to the screen on input.
- **.DBGTERM** refers to a terminal–like device that uses the keyboard and the bottom eight lines of the Macintosh screen.

For more information on these devices, see Serial Devices.

File Types

Disk files that are created by the Runtime Support Library are one of three types. Each type has its own icon that distinguishes the file type. It is possible to associate your own icons to these file types. See BUILDING AN APPLICATION.

Using the facilities in the Error_Handling **unit**, your program can exercise additional control over the file types and creator identifiers for the files it creates. See Execution Environment Control later in this chapter for more information.

The file types and standard icons are as follows:

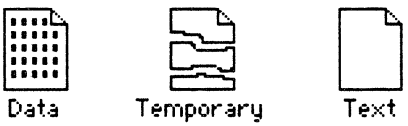


Figure 2–6. File Icons.

- **text file.** A text file results when a program creates a file of type text or **file of** char.
- **data file.** A data file results when any other type of file is created.
- **temporary file.** A temporary file results when a file has not been properly closed. A temporary file may not be opened using the Runtime Support Library. Currently, there is no utility program that will change a temporary file into a permanent file.

Pascal I/O Operation

This section is a collection of notes on how the Pascal I/O operations work under the Macintosh. Most operations will produce the result you would expect, but there are some restrictions imposed by the Macintosh Operating System and by this implementation of Pascal that you should be aware of.

- When you read certain types of text data from one of the special serial devices (.CONSOLE, .SYSTERM or .DBGTERM) you may use the <Backspace> key to correct typing errors. The data types that allow this are strings and the numeric data types. This handling of the <Backspace> key is independent of the general–purpose backspace character handling that is done by the special serial device driver, and works even if you are not using a fixed–pitch font.

- The standard file input defaults to the .CONSOLE device, as does the standard file output. This means that read, readln, write, and writeln intrinsics that do not specify a file name will cause output to go to the current window and input to come from the keyboard. If the program doesn't have a current window, output is written to the QuickDraw grafport which defines the screen. A program which has the Create Default Window runtime option enabled gets a current window which satisfies the requirements for these intrinsics.

- Tabs are expanded, as you would expect, on the special serial devices. That is, writing a tab character to one of the special serial devices causes the QuickDraw character drawing pen to be positioned at the start of next column to the right of its current position. Each column has a width of eight space characters. (This means that font and character size used determines the actual width of the columns on the screen.) Note that this same style of tab expansion may not occur when you write text on an Imagewriter printer unless you have set the tab stops on the printer.

- A disk file that is opened with reset or rewrite is opened with both read and write permission. Therefore, disk files may not be opened more than once simultaneously within an application. An important consequence of this is that none of the files that the system opens automatically may be opened by a program.

- Devices, on the other hand, are opened with whatever permissions are available. Thus, one device (like .BOUT) may be opened more than once simultaneously within an application. There are however some anomalies regarding the standard Macintosh devices .AIN and .BIN which you need to be aware of. You must open the corresponding output device first, before you open one of these input devices, otherwise the system will crash. For example, if you want to open a file to .AIN, first open it to .AOUT, close it, then re—open it to .AIN.

- Disk files are stored as normal Macintosh files, and share all the properties of Macintosh files. For instance, disk files may be located in multiple extents on a disk. Thus, a file may expand until the disk is completely full without the user worrying about the placement of files on the disk.

- The Runtime Support Library returns an I/O result code for each I/O operation. The codes that are returned correspond to the I/O result codes used by the Macintosh Operating System. Where possible, I/O result codes manufactured by the Runtime Support Library will be one of the codes known to the Macintosh Operating System. However, a few new codes have been defined that are unique to the Runtime Support Library.

- When the Runtime Support Library is reading a character from one of the special serial devices, this condition is made known to the user through the display of a block cursor at the current pen position on the screen.

Limits On Open Files

A UCSD Pascal program can have a maximum of eight files open at one time on the Macintosh. The Macintosh Operating System imposes a limit of 12 open files, but the Pascal Runtime Library keeps four files open while a normal program is running. These are the files that the system keeps open:

1. Pascal Runtime (data fork)
2. Pascal Runtime (resource fork)
3. Application (data fork)
4. Application (resource fork)

In addition, each library code file that your program uses will be open at runtime. Therefore, if you plan to have many files open at once in your application, you will need to restrict your use of library code files.

Special Keyboard Sequences

The Macintosh Operating System takes special actions on certain keyboard inputs. These actions take the form of special key sequences that the Macintosh Operating System recognizes. Your application can disable these actions by using GetOSEvent to retrieve keyboard input rather than GetNextEvent or Pascal I/O. These key sequences are as follows:

- **<command–shift–1>** ejects the disk in the internal drive.
- **<command–shift–2>** ejects the disk in the external drive.
- **<command–shift–3>** writes a copy of the current window to a disk file that is suitable for input to MacPaint. If <Caps Lock> is also down, then it writes the whole screen contents. The file is written to the default disk.

- **<command–shift–4>** writes a copy of the current window to the printer. If <Caps Lock> is down, then it writes the contents of the entire screen image (print screen).

Serial Devices

This section describes the special serial devices that are supported by the Runtime Support Library. The Macintosh Operating System does not treat the screen and keyboard as files at all, so these are really just virtual devices to give the screen and keyboard a file interface.

The Runtime Support Library uses QuickDraw to draw characters on these virtual devices. Therefore, QuickDraw terminology (eg. font, pen location) is used to describe the output characteristics of these devices.

- **.CONSOLE** refers to a terminal–like device that uses the Macintosh screen and keyboard. A write to .CONSOLE writes characters to the current window in the currently selected font. If the Create Default Window runtime option is enabled, this default window is the current window when a program starts. The default font is Geneva–12. A read from .CONSOLE reads characters from the keyboard. These characters are echoed on the screen in the current window.

- **.SYSTERM** refers to a device that is identical to .CONSOLE, except that characters read from the keyboard are not echoed on the screen.

- **.DBGTERM** refers to a terminal–like device that uses the keyboard and the bottom eight lines of the Macintosh screen. Characters written to .DBGTERM will appear in Monaco–9 font. The .DBGTERM device does not write to the screen within a window. Instead, it destructively modifies the bits at that position of the screen. Because the characters may be superimposed over other information on the screen, .DBGTERM draws its characters with some surrounding white space. This device is used by the Debugger when the External Terminal Debugging runtime option is disabled. It is also useful for programs that want to display their own

debugging information without interfering with the current window. Like .SYSTERM, .DBGTERM does not echo characters on input. When the .DBGTERM device is available, the size of the default window created for the .CONSOLE and .SYSTERM devices is made smaller so that .DBGTERM output is not intermingled with .CONSOLE output. Of course, if you are not using the default window option, it depends on the current grafport as to whether or not .CONSOLE output will ever conflict with .DBGTERM output.

The three serial devices mentioned have a number of characteristics in common. All of them display a block cursor when the program reads from the device. This block cursor is an indication to the user that keyboard input is expected.

The following special characters are handled by the special serial devices:

- **carriage return.** Writing a carriage return character (0D hex) causes the current pen position to be moved to the beginning of the next line. The vertical distance the pen is moved is based on the height of the current font. The pen is moved horizontally to coordinate 0. If the new pen location is below the bottom of the grafport, the grafport is scrolled by one line to accomodate the new line of characters.

- **line feed.** Writing a line feed character (0A hex) performs no action. Line feed is ignored on output.

- **tab.** Writing a tab character (09 hex) aligns the pen location at the next tab stop. The tab stops have a width of eight spaces in the current font, and are spaced evenly across the grafport starting at horizontal coordinate 0. If the pen is currently at a tab stop, writing a tab advances the pen to the next tab stop.

- **backspace.** Writing a backspace character (08 hex) erases a character the width of a space (in the current font) immediately before the current pen location, and moves the pen location to the left by the width of a space. Backspace is most useful if you are using a fixed–pitch font like

Monaco–9.

- **bell.** Writing a bell character (07 hex) causes an audible beep to be generated. The volume of the beep can be controlled via the Control Panel desktop accessory.

Disk Swapping

All of the disks having icons on the desktop prior to the start of a program are accessible to the program. This means that files may be opened or created on these disks, even if the disk is no longer in the disk drive. Additionally, any disks inserted in a disk drive while a program is executing are also accessible to the program, provided that the disk is inserted prior to its being accessed to open or create a file. Once a file has been opened, the Macintosh Operating System will request that the disk it is on be inserted into a disk drive whenever the file is referenced. This capability increases the amount of disk storage available to Macintosh programs, but at a severe cost in access speed.

If your application is going to depend on using multiple disks per drive, you should be aware of several factors:

- Swapping disks places additional burdens on the amount of stack slop required by your program. This is explained further in How to Set Stack Slop in the MACINTOSH INTERFACE chapter.

- Programs which use UCSD Pascal intrinsics to access files can use either Macintosh or Pascal I/O file naming conventions in the reset or rewrite statements. However, use of explicit drive numbers in file names can be dangerous because there is no assurance that the correct disk will be in the disk drive when the file is actually opened.

- Programs which use the high–level Macintosh File Manager unit can use Macintosh file naming conventions to open their files.

When they are requesting the names of the files that they are to operate on, the Compiler and the Librarian accept file names which contain explicit drive numbers. However, care should be taken when using the explicit drive number notation with these programs when using multiple disks in a drive.

BUILDING AN APPLICATION

This section discusses more advanced topics regarding putting together an application using UCSD Pascal on the Macintosh.

Putting it All Together

This section describes the use of segments, units, and libriaries. It presents some useful strategies for designing a large program.

Units and segments are used to divide large programs into independent modules. On the Macintosh, the main bottlenecks in developing large programs are:

- A large number of declarations that consume space while a program is compiling.
- Large pieces of code that use up memory space while the program is executing.

The use of units solves the first problem by: (1) allowing separate compilation; and (2) minimizing the number of identifiers needed to communicate between separate tasks. The use of segments alleviates the second problem by allowing the code for a large program to be partitioned into manageable chunks in such a way that only portions of the program need to be in main memory at any given time, and any unused portions reside on disk.

You can write a program with runtime memory management and separate compilations already planned, or you can write as a whole and then break it into segments and units. The latter approach is feasible when you're unsure about having to use

segments or quite sure that they will be used only rarely. The former approach is preferred and is easier to accomplish.

The following steps outline a typical procedure for constructing a relatively large application program:

1. Design the program (user and machine interfaces).
2. Determine needed additions to the library of units, both general and applied tools.
3. Write and debug units and add them to libraries.
4. Code and debug the program.
5. Tune the program for better performance.

During the design of a program, try to use existing procedures to decrease coding time and increase reliability. You can accomplish this strategy by using units.

To determine segmentation, consider the expected execution sequence and try to group routines inside segments so that the segment routines are called as infrequently as possible.

While designing the program, consider the logical (functional) grouping of procedures into units. Besides making the compilation of a large program possible, this can help the program's conceptual design and make testing easier.

Units may contain segment routines within them. You should be aware that a unit occupies a segment of its own; except, possibly, for any segment routines it may contain. The unit's segment, like other code segments, remains disk resident except when its routines are being called.

You can put into the interface section the headings for procedures and functions that are needed by other units. Then you can hide the implementation section from the users of the unit.

Steps 2 and 3 of the typical construction procedure are aimed at capturing some of the new routines in a form that allows them to be used in future programs. At this point, you should review, and perhaps modify, the design to identify those routines that may be useful in the future. In addition, useful routines might be made more general and put into libraries.

Write and test the Library routines before moving on to writing the rest of the program. This adds more generally useful procedures to the library.

The **interface** part of a unit should be completed before the **implementation** part, especially if several programmers are working on the same project.

Tuning a program usually involves performance tuning. Since segments offer greater memory space at reduced speed, performance is improved by turning routines into segment routines or turning segment routines back into normal routines. Either route is feasible. Pay attention to the rules for declaring segments.

Segmenting a Program

An entire program need not to be in main memory at runtime. Most programs can be described in terms of a working set of code that is required over a given time period. For most (if not all) of a program's execution time, the working set is a subset of the entire program, sometimes a very small subset. Portions of a program that are not part of the working set can reside on disk, thus freeing main memory for other uses.

When your program executes, it is read into main memory. When the code has finished running, or the space it occupies is needed for some action having higher priority, the space it occupies may be overwritten with new code. Code is swapped into main memory a segment at a time.

In its simplest form, a code segment includes a main program and all of its routines. A routine may occupy a segment of its own; this is accomplished by declaring it to be a **segment** routine. Segment routines may be swapped independently of the main program; declaring a routine to be a **segment** is useful in managing main memory.

Routines that are not part of a program's main working set are prime candidates for occupying their own segment. Such routines include initialization and wrap–up procedures and routines that are used only once or only rarely while a program is executing. Reading a procedure in from disk before it is executed takes time. Therefore, the way that you divide up a program is important.

The UCSD Pascal Handbook describes the syntax for creating separate segments in a program.

Separate Compilation

Separate compilation is a technique in which individual parts of a program are compiled separately and subsequently executed as a coordinated whole.

Many programs are too large to compile within the memory confines of the Macintosh. Such programs might comfortably run though, especially if they are segmented properly. Compiling small pieces of a program separately can overcome this memory problem.

Separate compilation also allows small portions of a program to be changed without necessarily affecting the rest of the code. This saves time and is less error prone. Libraries of routines may be built up and used in developing other programs. This capability is important if a large program is being developed and is invaluable if the project involves several programmers.

In UCSD Pascal, separate compilation is achieved by the **unit** construct—a unit being a group of routines and data structures. The contents of a unit usually relate to some common

application, such as screen control or data file handling. A program or another unit may use the routines and data structures of a unit by simply naming it in a **uses** declaration. In addition to being a separately compiled module, a unit is also a code segment; it can be swapped, as a whole, into and out of memory.

A unit consists of two main parts: the **interface** section, where constant, type, variable, procedure, process, and function declarations, which are "public" (available to any client module) are found; and the **implementation** section, where private declarations are found.

The UCSD Pascal Handbook describes the syntax for creating and using units.

Libraries

This section describes where you may place the code files that contain units so those units are available at compile time or runtime. At compile time, only the **interface** section of a called unit is needed. At runtime, only the **implementation** section is needed. (It is allowed, however, to have both the implementation and interface sections available at both runtime and compile time.) If you wish, a unit can be compiled with the complete interface section, but with empty routines defined in the implementation section. This allows clients which require the interface section to be compiled before the unit has been fully implemented. Also, for runtime purposes, the interface section can be stripped out of a unit's code file using the Librarian. This leaves only the implementation section and saves disk space at runtime.

A program or a unit which uses another unit is called a client of that unit. An anology can be made with someone who offers a service (the unit) and someone else who is a client of that service (the using program or unit). At runtime, the Runtime Support Library searches for a unit in the following places:

- The Runtime Support Library
- The client code file
- The files listed in the client's Library Files list.

The Runtime Support Library units reside in either Pascal Runtime or Debug Runtime. DO NOT place units that you write there.

To place a unit directly into a program's code file, use the Librarian. After the unit's code and the program's code are unified, the unit will be available when the program is executed. Refer to the LIBRARIAN chapter for more information on placing units into a client's code file.

A library can be a code file which is a collection of compiled units (usually stitched together with the Librarian) or it can contain just a simple unit within a code file created by the Compiler when you compile that unit. The Library Files section in this chapter describes how to modify the client's runtime environment description to reference libraries.

At compile time, as opposed to runtime, the code for a unit resides in a code file specified in the text you are compiling. *The UCSD Pascal Handbook* describes how clients can use the interface section of units at compile time.

Standard Resources

This section describes the RMaker input used to create the generic resources for Empty Program. This is the file used by the Compiler on the Macintosh to install resources into the program code files that it creates. Various parts of the Runtime Support Library expect to access these resources using the resource type identifiers and numbers defined here. You should be careful when defining resources for your program that you do not accidently redefine the resources described here.

The first input specifies the RMaker output file name. Following that is the file type and signature:

```
UCSD Pascal 1:Empty Program
APPLPROG
```

The next resource entry is the applications's signature and version number string. The generic application signature is PROG; Generic version data is the empty string. (Used as the title for the default screen I/O window.) If you want the title of the default screen I/O window to be other than the name of the program's code file, change the third line of the following example from the empty string to whatever string of characters you want to use. See the RMAKER chapter for instructions on how to append new resources onto an existing resource fork.

```
TYPE PROG = STR
  ,0 (32)
```

The required Pascal Runtime Files location names are next. First is the file name of file containing the Runtime Support Library. Next is the file name of file containing the p–Machine.

```
TYPE SYSF = STR
  ,0 (32)
Pascal Runtime
  ,1 (32)
p-Machine
```

Next is the number of Macintosh Memory Manager master pointer blocks to preallocate before the Pascal Heap Block is allocated as a nonrelocatable heap block. (Master pointer blocks are nonrelocatable, and must never be allowed to reside above the runtime support's heap block. If any nonrelocatable blocks are allocated above the Pascal Heap Block, it may not be possible for the Runtime Support Library to extend the Pascal Heap Block, even when sufficient free memory space is available. See the MEMORY MANAGEMENT chapter for more details on the Pascal Heap Block.)

Each allocated master pointer block has room for 64 master pointers. The Macintosh Finder starts any application with a single master pointer block (i.e. 64 master pointers).

```
TYPE MSTR = GNRL
  ,0 (32)
 .H
0001
```

The Startup option settings are defined next. Options are specified by individual characters in the string resource. A plus (+) enables an option, a minus (−) disables it. The position of the character in the string determines which option is set. The following table lists the Startup options and their default settings in Empty Program:

Option	Position	Default
Create Default Window	1	+
Create .DBGTERM Device	2	-
Startup in Debugger	3	-
Enable Performance Monitor	4	-
Debug to Modem Port	5	-

The following resource specifies the default settings:

```
TYPE OPTN = STR
,0 (32)
+----
```

The following strings define the text used in the bootstrap's error messages:

```
TYPE PRME = STR
  ,0 (32)
Could not open p-machine file
  ,1 (32)
Could not allocate memory for p-machine
  ,2 (32)
Error reading p-machine file
  ,3 (32)
Could not locate MSTR resource
  ,4 (32)
Could not open program data fork
  ,5 (32)
Could not open Runtime Support Library file
  ,6 (32)
Could not allocate stack/heap
```

The following resource definitions are used for the bootstrap's ALERT Dialog boxes:

```
TYPE DITL
   ,256 (32)
2

BtnItem Enabled
90 267 110 337
OK

StatText Disabled
10 60 70 350
^0

TYPE ALRT
   ,256 (32)
60 81 180 431
256
5555
```

One additional resource type is needed to complete the definition of Empty Program. It causes the assembly language bootstrap program to be included in the resource fork. This is the native Macintosh application which begins executing when you open the icon of a UCSD Pascal program. This bootstrap reads in the p–Machine. The p–Machine builds a runtime environment and reads in the Runtime Support Library. The Runtime Support Library stitches the pieces of your program together and begins executing it. The actual resource definition is not included here because it does not follow the conventions and syntax of the Macintosh RMaker.

Execution Environment Control

The Error_Handling unit may be used by a UCSD Pascal program to control its execution environment, or perform certain special functions. This unit may be found in the file Errorhandl.CODE on the **UCSD Pascal 2** disk. The entry points to the Error_Handling unit allow a program to:

1. Override the standard handling of runtime errors performed by the Runtime Support Library by installing a custom error handling routine. Such an error handler routine can attempt some corrective action for certain errors, or simply report runtime errors in a different manner.

2. Force entry into the Debugger.

3. Cancel a process.

4. Establish a procedure as the "interaction procedure" which is activated by the Debugger's "I" command.

5. Turn the Performance Monitor output ON and OFF.

6. Adjust the "stack slop" for the main task.

7. Establish a specific Macintosh file type identifier and signature for an open **file** variable.

The following is the interface to the Error_Handling unit:

```
unit error_handling;
interface

  type  eh_results = ( cant_handle, re_initialize, continue );
        eh_info = record
                    {used internally by operating system}
                    a:tinteger; b:tinteger; c:integer;
                    d:tinteger; e:tinteger; f:tinteger;
                  end;

        eh_file_ptr = tinteger;  {Actually a pointer to a
                                  file variable.}
        eh_res_type = packed array[1..4] of char;
                                 {A Macintosh Resource Type
                                  Identifier.}

  {User error handling facilities.}
  procedure set_err_handler(
              var info:eh_info;
              function err_handler(err, ior: integer):
  procedure clr_err_handler(var info: eh_info);
  procedure err_to_message(err: integer; var message: string);
  procedure ior_to_message(ior: integer; var message: string);
  procedure debugger;

  {Process control.}
  procedure cancel(taskid: processid);

  {Performance monitor control.}
  procedure set_pm_interaction(procedure pm_interactive);
  procedure pm_start_stop(start: boolean);


  {Stack space checking control.}
  procedure set_stack_slop(slop: integer);
  function get_stack_slop: integer;

  {File type and signature control.}
  procedure set_file_type(f: eh_file_ptr; ftype: eh_res_type);
  procedure set_file_signature(f: eh_file_ptr;
                               signature: eh_res_type);
```

The following paragraphs discuss each of the entry points to the Error_Handling unit:

1. The routine SetErrHandler establishes its parameter ERR_HANDLER as an error–handling function. After such an error–handling function is established, the UCSD Pascal Runtime Support Library will call it whenever a non–fatal runtime error occurs. The runtime error number and the current ioresult values are passed to an error–handling function in its ERR and IOR parameters.

   An error–handling function returns one of these possible results:

   ReInitialize. Causes immediate termination of the program.

   Continue. Asks the UCSD Pascal Runtime Support Library to attempt to continue execution.

   CantHandle. Indicates that the particular runtime error cannot be handled by this error–handling function, and that it should be reported to any previously established error–handling function (if any). If none of the established error–handling functions can handle the error, the standard UCSD Pascal Runtime Support Library error–handling mechanism is used to report the error.

   The INFO parameter passed to SetErrHandler is an information record which is used internally by the UCSD Pascal Runtime Support Library. Each distinct error–handling function you establish must have a separate information record. To cancel the establishment of an error–handling function, you should call ClrErrHandler passing the appropriate information record.

   The following is a simple example of how you might create your own error–handling function and use it in a program:

```
PROGRAM no_interruptions;

USES {$U UCSD Pascal 2:Errorhandl.CODE}
     Error_Handling;

VAR info: eh_info;
```

```
  FUNCTION my_error_routine(
             errnum, iorslt: integer): eh_results;
  BEGIN
    IF errnum = 8 {User Break} THEN
      my_error_routine := continue
    ELSE
      my_error_routine := cant_handle;
  END; {my_error_routine}

BEGIN
 {Assume program is entering some critical
  operation that shouldn't be interrupted.}
 SetErrHandler(info, my_error_routine);

 {Do the critical operation}

 {Restore User Break facility.}
 ClrErrHandler(info);

 {Resume normal processing.}
END. {no_interruptions}
```

In the example, an error—handling function is used to prevent the user from interrupting the program during a certain critical section of the program. All runtime errors except User Break will be handled in the usual fashion by the UCSD Pascal Runtime Support Library.

You can establish an error—handling function anywhere in your program. However, be sure that you call ClrErrHandler prior to leaving the context in which your function is declared.

Error—handling functions may be nested, and the most recently established function is called first. A unique information record variable must be used each time SetErrHandler is called.

WARNING: The exit intrinsic cannot be used to exit a function that is established as an Error—handling function.

2. ErrToMessage and IorToMessage are routines that you can call to obtain a textual message describing a particular runtime error or ioresult value. Both routines return the text of the message in the string variable you pass as the MESSAGE parameter. The possible messages returned by these routines are listed in Appendix B.

3. To enter the UCSD Pascal Debugger from an error—handling function (or from anywhere else in a UCSD Pascal program), you can call the routine Debugger. This facility is intended for use only during the development and checkout of a program. If you call the Debugger without having the appropriate runtime options set (those which are required in order to use the Debugger), your program will fail unpredictably.

4. The Cancel entry point cancels a process that was previously started via the start intrinsic. You pass the processid value returned by start to designate the process to be cancelled. Cancel cancels the process immediately, interrupting whatever was happening, and releases the space used for its stack. The canceled process is effectively forced to do an "exit(**process**)" statement, since the routine activations on the process's stack are "unwound" and any exit code for those routines is executed.

5. SetPmInteraction is used to establish a procedure within your program as the Debugger's "interaction procedure". The interaction procedure is called when the Debugger "I" command is typed from the Macintosh keyboard. (In order to use the interaction procedure mechanism, the Performance Monitor must be activated by setting the appropriate options with Set Options.) One typical kind of interaction procedure is one which produces a formatted display of the contents of some variables or a complicated data structure. Using the interaction procedure facility, you can make the debugging of a large and complex program much easier, since you are effectively customizing the Debugger to suit the needs of your program .

6. PmStartStop is used to control the built—in Performance Monitor. The Boolean value you pass as the parameter START indicates whether the Performance Monitor output should be enabled or disabled. If the Performance Monitor is not active when your program starts its execution, PmStartStop does nothing.

7. The routines SetStackSlop and GetStackSlop are used to control the stack slop for the currently executing task. SetStackSlop sets the stack slop to the number of words that you specify. GetStackSlop returns the current stack slop setting. SetStackSlop will not allow the slop setting to be less than the minimum setting of 1024 (2Kb). For further details concerning the usage of these routines, see the MACINTOSH INTERFACE chapter.

8. SetFileType and SetFileSignature are used to specify the permanent file type or signature for a Macintosh file being created by your program using the standard Pascal file I/O. The first parameter to these routines is a Pascal pointer value that points to a **file** variable that you have opened using the standard Pascal procedure rewrite. (Use the adr intrinsic to obtain this pointer value.) The second parameter is the four character file type or signature. When you close the file variable with the LOCK option, the created file's type and signature are set as specified. If your program creates a Macintosh file without calling SetFileType, the file type is set according to the type of the file. If you don't call SetFileSignature, the signature of your program is used when you close the file.

Application Interface to the Finder

The default interface between applications and the Macintosh Finder program simply allows programs to be started by the Finder. If you want your application to be started when an associated document is clicked or you wish to have special program and document icons displayed on the desktop then you must go through a little extra work.

Associating Programs With Documents

In order for the Finder to associate a document with an application two conditions must be met:

1. The application program must be "bundled" into the Desktop.

2. The document must have the same "creator" as the application.

For more details on these topics see the section entitled "FILE INFORMATION USED BY THE FINDER" in the FILE MANAGER chapter of *Inside Macintosh.*

To bundle a UCSD Pascal program into the desktop you run the Set Options program and set the "Set Bundle for FINDER" checkbox. The Set Options utility was described earlier in this chapter.

Since UCSD Pascal programs normally have a creator of PROG any document with the same creator that is double clicked from the Finder will start your program (assuming no other programs with the same creator have been bundled into the desktop). Note that files created by the UCSD Pascal Runtime package do not have a creator of PROG. You will have to use the File Manager interface unit to create documents with the correct creator or use the appropriate Error_Handler entry point.

In order to override the default application creator you use the RMaker program to set a new creator. For example:

```
Example.Rsrc                ;; Output File Name
APPLEXMP                    ;; Type is APPL , Creator is EXMP

INCLUDE UCSD Pascal 1:Empty Program
                            ;; Resources required by all
                            ;; UCSD Pascal Programs
```

Running RMaker using the above example will produce a resource file of type APPL with a creator of type 'EXMP'. Using this as the resource input file to the UCSD Pascal compiler will produce a UCSD Pascal program with the same creator and type. In order to make this creator type known to the Finder you need to run Set Options on the program and set the bundle bit.

NOTE: Apple Computer would like to maintain a unique set of creator identifiers. If you wish to bundle your application into the desktop then you should call Apple Technical Support to get a unique creator identifier.

Associating Icons With Files

In addition to being able to let the Finder know an application's creator, you can also bundle in other information into the desktop. This is achieved by defining a resource of type 'BNDL'.

For example suppose you wanted to define two new file icons, one for your application and another for the data files that your application will create and use. You could create a resource file for your program as follows:

```
Example.Rsrc                ;; Resource Output File
APPLEXMP

INCLUDE UCSD Pascal 1:Empty Program

TYPE EXMP = STR             ;; Version String
  ,0 (32)
Version 1 of Example Program

TYPE ICN# = GNRL            ;; The program Icon
  ,2000 (32)                ;; Defined later
  .H
0000 0000 0000 0000
................
0000 0000 0000 0000

TYPE ICN# = GNRL            ;; The Data File Icon
  ,2001 (32)                ;; Defined later
  .H
0000 0000 0000 0000
................
0000 0000 0000 0000

TYPE FREF                   ;; File References
   ,2000                    ;;  for application file
APPL 0                      ;; Type followed by Local
                            ;;  Icon ID
   ,2001                    ;;  for data file
DATA 1

TYPE BNDL                   ;; Bundle Resource
   ,2000
EXMP 0                      ;; Signature and Version ID
ICN#                        ;; ICONs
0 2000 1 2001               ;; Local IDs to Resource IDs
FREF                        ;; File References
0 2000 1 2001               ;; Local IDs to Resource IDs
```

In the above example we have defined a Version String, two icon lists, and two file references. A file is associated with a particular icon list using the FREF resource. This resource defines a file type and a local icon identifier. The mapping from resource identifiers to local identifiers is accomplished in the BNDL resource.

After you have created your program using the UCSD Pascal compiler you still need to run the Set Options program and set the bundle bit. After Set Options is run you will normally see your program icon switch from the standard application icon to the icon you defined as the program icon.

Defining Icons Using RMaker

An icon is defined as two 32 by 32 bit images. The first image is the icon in its dormant (unclicked) state. The second image is an icon mask which is used by the Finder to produce the image representing the icon in its active (clicked) state. The mask should be a filled in outline of the first icon.

The UCSD Pascal compiler icon is defined using the following icon list:

```
TYPE ICN# = GNRL
  ,2000 (32)                          ;; Resource ID
  .H
0001 0000 0002 8000                   ;; ---------------
0004 4000 0008 2000
0010 1000 0020 4800
0040 0400 0081 0200
0100 0100 0204 0080                   ;; The first set of
04E0 0040 0820 1020                   ;; of 16 rows define
1220 0010 2100 4008                   ;; the ICON.
488E 3F04 8802 4082
4E32 8041 2029 3022
1089 C814 088E 7F0F
04E2 3007 0201 0007
0100 8007 0080 6007
0040 1FE7 0020 021F
0010 0407 0008 0800
0004 1000 0002 2000
0001 4000 0000 8000                   ;; -------------
0001 0000 0003 8000
0007 C000 000F E000
001F F000 003F F800
007F FFC0 00FF FE00                   ;; The next 16 rows
01FF FF00 03FF FF80                   ;; define the ICON
07FF FFC0 0FFF FFE0                   ;; mask.
1FFF FFF0 3FFF FFF8
7FFF FFFC FFFF FFFE
```

```
7FFF FFFF 3FFF FFFE
1FFF FFFC 0FFF FFFF
07FF FFFF 03FF FFFF
01FF FFFF 00FF FFFF
007F FFFF 003F FE1F
001F FC07 000F F800
0007 F000 0003 E000
0001 C000 0000 8000                ;; -----------
```

3
EDITOR

The Editor is used to create and modify text files. These files can be used for many purposes including input to the Pascal Compiler and creating textual data for Pascal program consumption.

If the file you are editing is too big to fit on the screen, a portion of the file is displayed. This "window" into the file can be moved to display any part of the file you want. An example of the Editor display is shown in Figure 3—1.

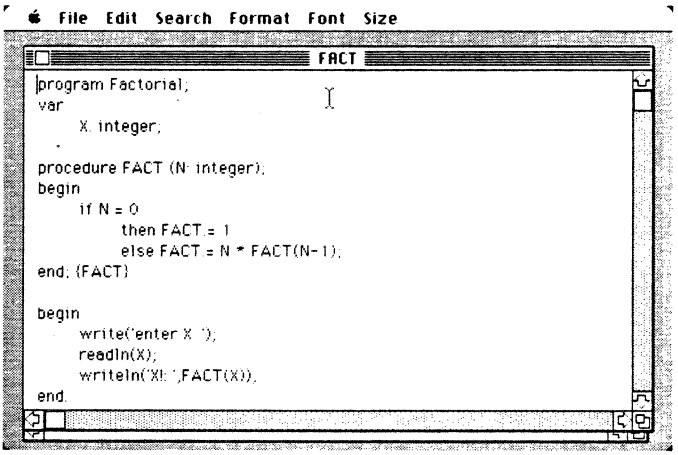


Figure 3—1. The Editor Display.

The basic editing operations are inserting characters, cutting a portion of the text, and pasting text into a new location. Text that is cut goes into a special window called the Clipboard. Text in the Clipboard can be pasted into any place in the file or into

another file. The Clipboard also allows you to transfer data between applications.

All editing action takes place at the insertion point. The insertion point is marked by a blinking vertical line where the next character will be placed. Any characters typed or pasted from the Clipboard are inserted at this point. This is true even if the insertion point is not currently displayed in the window. The window is automatically scrolled to show the insertion point.

The mouse is used to scroll the text in the window, move the insertion point, select text to be cut or copied, point to menus, and select items on menus.

The Editor is disk based. This means that the size of a file you can edit is limited only by the available space on the disk. However, as a file grows larger it takes longer to do simple editing operations on it. When a file becomes very large, you should split it into multiple pieces.

USING THE EDITOR

Start the Editor by double—clicking the Editor icon. For more information on starting applications refer to *Macintosh*, your owner's guide.

You direct the Editor to work on a file by using the New or Open... command in the File menu. Selecting a command from a menu is discussed below in The Menus.

The file that you are working on is called the "active document." Although you can have several documents open and accessible at any one time, you can edit only the active document. The active document appears in the "active window," which is indicated by a darkened title bar and scroll bars, and is always on top of all the other windows.

To leave the Editor, select Quit from the File menu, and you will return to the Finder.

Entering and Deleting Text

Text is entered into the active window at the insertion point by typing characters. Text is deleted at the insertion point by typing the <Backspace> key. Large deletions are done by selecting the text with the mouse and then typing <Backspace>. You change text by selecting the text to change and then typing the replacement text.

Editing Operations

The basic editing operations are cut, copy, and paste. To cut or copy text, you must first select the text to be cut or copied, then select either Cut or Copy from the Edit menu. Select text by moving the mouse while holding down the button. See SELECTING TEXT for complete information on selecting text. Text that is selected and then cut is removed from the active document and placed in a special window called the Clipboard. Text that is copied is placed on the Clipboard and also left in place in the active document.

The contents of the Clipboard can be pasted at any point in the active document by placing the insertion point where you want the text inserted and choosing Paste from the Edit menu.

The Menus

Operations are provided in six menus:

- The File menu is used to access files, print text, and exit the Editor.
- The Edit menu is used to edit text.

- The Search menu provides commands to find and change strings in the active document.
- The Format menu handles setting the tab stops and enabling auto indent mode.
- The Font menu allows you to select the font of the current document for display and printing.
- The Size menu allows you to set the size of the current font.

Each of these menus is described in more detail in the sections that follow.

Creating, Opening and Closing Files

Files are created, opened and put away using the functions of the File menu. The New command creates a new file. The Open... command opens an existing file. The Close command puts away the active document.

The Open... function uses the Open Box to help you select the file to open. This dialog box is shown in Figure 3–2.

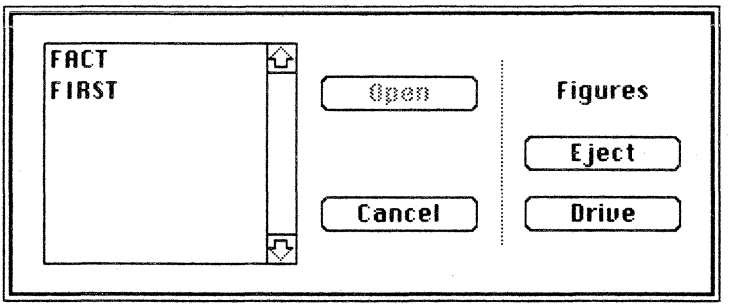


Figure 3–2. The Open Box.

To open a file, first scroll the file list by clicking the mouse in the scroll arrows until the file you want to open is in the list. Next, select the file by using the mouse to click its file name. Finally,

click the Open button to open the file. An alternative method of opening a file is to double—click its file name.

The file list displays only the files in the current drive that have a file type of TEXT. The name of the disk in the current drive is displayed above the Eject button. The other buttons are as follows: Cancel aborts the operation, Drive switches to the other drive, and Eject ejects the disk from the current drive.

Various File menu functions cause the active document to be saved. If the Editor needs you to supply a file name it uses the Save Box, shown in Figure 3—3.

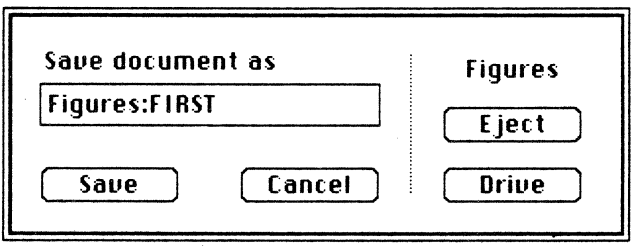


Figure 3—3. The Save Box.

To save a file, first type its file name. Next, use the Eject and Drive buttons to make the disk it is to be saved on the current drive. (The current drive name is shown above the Eject button.) Finally, click the Save button. The Cancel button is to abort the save operation and return to the Editor.

The field where you type the file name is a standard Macintosh editable text field. This means that you can use the mouse to edit the file name until it is correct. See *Macintosh* for more information on editing text fields.

Editing Multiple Files

Up to four documents can be open at one time, but only one document is the active document. To read in a document when you already have an active document, choose Open... from the File menu. It asks you for the document name. The new document is read into a window on the screen and becomes the active document. To make another document that is already open the active document, use the mouse to move the pointer into a portion of that document and click the mouse button. If you have several documents open, you might have to move some out of the way.

This capability of working with more than one document at a time can be used to copy text from one document to another. This process is described in detail in EDIT FUNCTIONS.

SELECTING TEXT

The basic editing functions are cut, copy and paste. Before you can cut or copy text, you must select the text to be cut or copied. Before you can paste, you place the insertion point by using the mouse to move the pointer on the screen.

Within an active document, the pointer will have one of three shapes:

- Text pointer in a document.
- Arrow pointer for menus and scroll bars.
- Wrist watch when an operation will take some time.

Use the mouse to move the pointer on the screen. The shape of the pointer changes when you move into and out of the document window.

Within the window, the text pointer is used to move the insertion point and to select text.

In selecting text, you can select characters or words. You can also select any number of characters or words. Selected text is displayed in reverse video.

Moving the Insertion Point

The insertion point is indicated by a blinking vertical line where the next character will be inserted. All insertion, whether from typing or pasting, takes place at this point in the file, even if it is not visible in the window.

To move the insertion point, move the mouse, directing the pointer to where you want it to be and click. Note that the insertion point moves when you select text. The insertion point is never placed beyond the end of a document.

Selecting Characters

To select characters, move the text pointer to the beginning of the characters you want to select, press and hold the mouse button while moving to the last character you want to select. You may select in either a forward or backward direction through the file.

An alternate method of selecting characters that is especially useful when selecting a large block of text is also available. Using this method, you move the pointer to the beginning of the text you want to select and click the mouse button. Then you move the pointer to the end of the text you want selected and shift—click. Shift—click means to hold down the shift key on the keyboard and click the mouse button. You can use the scrolling controls to display the end of the text you want selected if it is too big to fit in the window.

Selecting Words

To select a word, move the pointer into the word and click the mouse button twice. To select multiple words, click the mouse button twice and hold. Move the pointer to the last word you want selected and release. If you double—click, and hold down the mouse button while you move the insertion point to the left or right, the selection expands or contracts by words.

Adjusting the Amount of Text Selected

To change the amount of text selected, move the pointer to the position that you want the selection to extend to and shift—click. This can be used to either expand or contract the selection.

SCROLLING AND MOVING THE DISPLAY

When a document is longer than will fit into the display window, only part of the document is displayed at one time. You can change what part is displayed by "scrolling" through the display either horizontally or vertically. The vertical bars on the right and bottom sides of the active window are the scroll bars. An example of a text window showing the scroll bars is in Figure 3—1.

The display window can be changed in size and moved on the screen. This enables you to have multiple documents displayed on the screen. These operations are done using the title bar. and size control box (See Moving the Window, below.)

Scrolling the Display

There are three ways of moving the display window through the document. In the first method you use the elevators. The elevators are the white rectangles in each scroll bar. Its position in the grey portion of the scroll bar (the "elevator shaft") indicates the relative position of the currently displayed text window in the document. If it is near the middle, the text

displayed in the window is near the middle of the document, and so on. To change the position of the text window, you can move the pointer into the elevator, click and hold the mouse button down while you move the elevator to another position in the elevator shaft. When you release the button, the window will display the new position in the file.

The second way of moving the window uses the scroll arrows, which are just to either side of the elevator shafts. If you move the arrow pointer to the bottom scroll arrow and click, the display window will move one line toward the end of the document. If you hold the button down, the window will continue to move a line at a time until you release it. The other three arrows work in a similar way.

The third way of moving the window uses the gray regions to either side of the elevators. Clicking the mouse in one of the gray regions causes the Editor to scroll one window—full of information. You can use this feature to page through a file.

Moving the Window

You can move the window on the screen and change its size. This lets you display multiple documents on the screen. You can make any visible window the active window by moving the pointer into it and clicking.

To move the position of a window on the screen, move the pointer to the title bar (but not in the close box!), press the mouse button and hold it while you move the window. When you release the button, the window is redisplayed at the new location.

To change the size or shape of the active window, move the pointer to the size control box, press the button, and move the pointer until the window is the right size and shape. Release the button and the resized window is displayed. The size control box is the box in the lower right hand corner of the window. Only the active window can be resized.

THE FILE MENU

The File menu provides functions for reading in and writing out documents, updating documents, printing documents, and exiting the Editor. The File menu is shown in Figure 3—4. Each function is explained below.

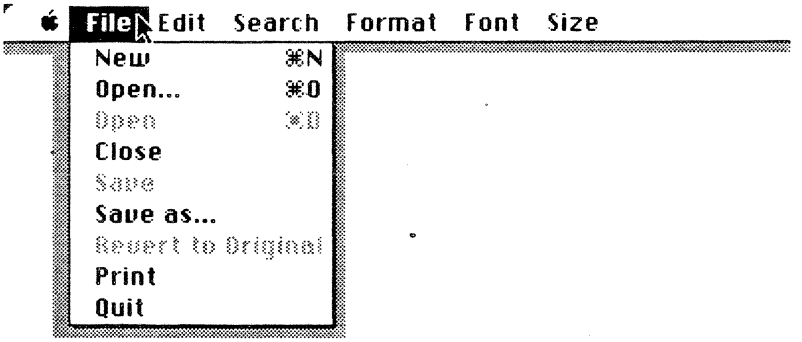


Figure 3—4. The File Menu.

- **New.** The New command creates a new document with the name Untitled and makes it the active document. You can also execute the New command by typing N while holding down the Command key.

- **Open...** This tells the Editor to get a new document. It prompts you for the document name using the Open Box, then reads it in and makes it the active document. You can also execute the Open... command by typing O while holding down the Command key. Another method of opening a new document is to type K while holding down the Command key, and then type in the name of the document you want to open followed by <Return>. This option does not appear in the menu.

- **Open.** This opens a file whose name corresponds to the contents of the currently selected text in the active window. This is used primarily to open an include file based on its name in the current document. You can also execute the Open command by typing D while holding down the

Command key.

- **Close.** This puts away the active document discarding any changes that have been made. You are asked to confirm whether the changes are to be discarded. If the document does not have a name, you are asked to supply one using the Save Box.

- **Save.** This writes out the active document, but does not close it.

- **Save as...** This writes out a copy of the active document to another document name. You are prompted for the name of the document to write to with the Save Box.

- **Revert to Original.** This returns the document to the way it was before you started editing it, or when you last saved it. This is done by reading the document from the disk.

- **Print.** The Print command prints the active document using the current font and font size. Executing the Print command causes the standard Print dialog box to be displayed in which you select various print options. If the Print dialog box fails to appear, you probably do not have an Imagewriter file on the same disk as the Editor. Refer to *MacWrite* for more information on the standard Print dialog box.

- **Quit.** This first asks you if you want to put away any modified documents. If you answer yes, they are written out to disk. Then it exits the Editor.

THE EDIT MENU

The Edit menu provides editing functions and tab setting. It is shown in Figure 3–5.

The three basic edit functions are cut, paste and copy. These make use of the special window called the Clipboard. The Clipboard can hold only one piece of text. Text is put into the Clipboard by selecting it in the active document, and either cutting it or copying it. Text is copied from the Clipboard and

inserted at the insertion point with the paste operation.

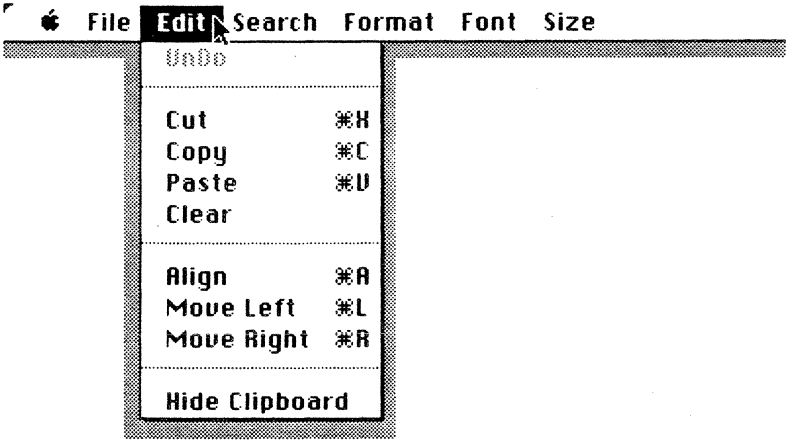


Figure 3–5. The Edit Menu.

For example, to move text from one place in a document to another:

1. Select the text to be moved.

2. Choose cut from the Edit menu. The text is removed from the active document and placed on the Clipboard.

3. Place the insertion point where you want the text to be.

4. Choose Paste from the Edit menu. The text on the Clipboard is inserted at the insertion point.

The Edit menu also enables you to adjust selected text left or right by inserting or deleting spaces. Here are the Edit functions:

- **Undo.** This command puts the document back the way it was before the previous operation, if possible. If there is no change to undo, the function is called Can't Undo.

- **Cut.** Cut places a copy of the currently selected text onto the Clipboard and removes the text from the active document. You can also Cut by pressing the X key while holding down the Command key.

- **Copy.** Copy places a copy of the currently selected text onto the Clipboard, but does not remove it from the active document. You can also copy by presing the C key while holding down the Command key.

- **Paste.** Paste inserts a copy of the text on the Clipboard at the insertion point in the active document. If a section of text is selected, Paste replaces it. You can also Paste by pressing the V key while holding down the Command key.

- **Clear.** Clear removes the currently selected text from the active document. The text is not placed in the Clipboard.

- **Align.** The Align command lines up the left edges of the selected lines. The align command is most often used to undo indentation in Pascal programs. You can also Align by pressing the A key while holding down the Command key.

- **Move Left.** Move Left moves selected text left by deleting a single space from the left of each line. It does not delete any characters other than spaces. It is most often used to adjust the left margin of a block of text. You can shift left by pressing the L key while holding down the Command key.

- **Move Right.** Move Right is similar to Move Left, except that it moves the selected text to the right by inserting spaces at the beginning of each line. This can also be done by pressing the R key while holding down the Command key.

- **Show Clipboard.** This enables the display of the Clipboard window and selects it. If the Clipboard is already displayed, this command is called Hide Clipboard.

THE SEARCH MENU

The Search menu gives you the ability to search for a text string in the active document. The basic operation is Find, which locates the next occurrence of the string and selects it. Change allows you to find a string and replace occurrences of it with a different string. Both of these operations search from the current insertion point to the end of the document. If you want to search from the beginning of a document, you must move the insertion point to the beginning of the document. The Search menu is shown in Figure 3–6.

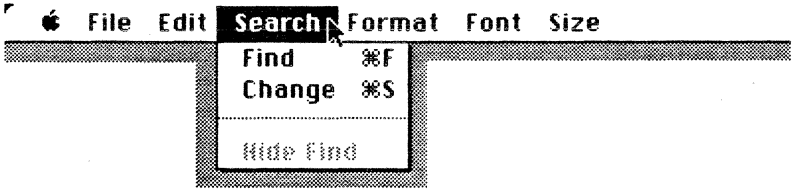


Figure 3–6. The Search Menu.

All searches start at the insertion point, and go to the end of the document. The search functions are as follows:

- **Find.** Find enables the Find Window, and displays it on the screen. The Find command can also be executed by pressing the F key while holding down the Command key.

- **Change.** Change enables the Change Window and displays it on the screen. The Change command can also be executed by pressing the S key while holding down the Command key.

- **Hide Find.** If the Find Window is enabled, the Hide Find command will close the Find Window. If the Change Window is enabled, this command is called Hide Change.

The Find Function

The Find function is performed using the Find Window, shown in Figure 3–7. To find an occurrence of a string, first, you edit the string to be found by using the standard Macintosh editing functions. Next, select Whole Word search or Partial Word search by clicking the appropriate box with the mouse. In Whole Word search, the string will only match complete words separated by spaces or other punctuation. In Partial Word search, the string may match any part of a word. Finally, you click the Find Next button.

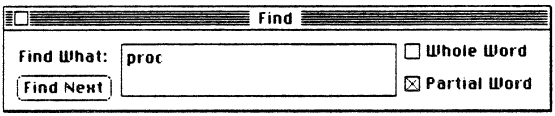


Figure 3–7. The Find Window.

If there is an occurrence of the string, it is selected. If no occurrence can be found, the Editor gives a warning message. Succeeding occurrences of the string can be found by just clicking the Find Next button.

To put away the Find Window, click in the close box within the title bar of the Find Window.

The Change Function

The Change function is performed using the Change Window, shown in Figure 3–8. To change all occurrences of a string for another, first edit the Find What and Change To strings in the Change Window. This is done using the standard Macintosh editing functions. Next, select Whole Word search or Partial Word search. Whole Word search only allows the string to match words separated by spaces. Partial Word search allows the string to match any string of characters. Finally, you click the Change All button.

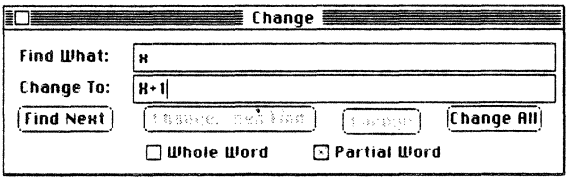


Figure 3–8. The Change Window

The other Change options are as follows: Find Next finds and selects the next occurrence of the Find What string; Change, Then Find changes the current selection, then finds the next one; and Change changes the current selection.

To put away the Change Window, click in the close box within the title bar of the Change Window.

THE FORMAT MENU

The functions in the Format menu allow you to set the spacing of the tab stops, configure auto indenting mode, display nonprinting characters, and set the printing page format. The Format menu is shown in Figure 3–9.

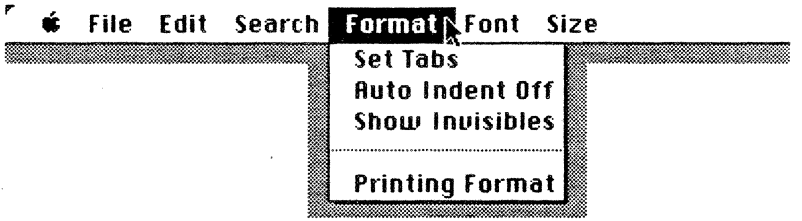


Figure 3–9. The Format Menu.

- **Set Tabs.** Set Tabs enables you to set the spacing of the tab stops. You may only select a spacing between 1 and 20. Note that the compiler listing pass assumes 8 spaces per tab stop. If you create Pascal source text with different tab settings,

your listing won't precisely match you source text.

- **Auto Indent Off.** This toggles the auto indent mode on and off. In auto indent mode, carriage returns puts the insertion point in line with the indenting of the previous line. This option is especially useful for indenting Pascal programs. If auto indenting is already off, this function is called Auto Indent On.

- **Show Invisibles.** Show Invisibles will display the non–printing characters (i.e. blanks, carriage returns, and tabs) in the currently active window. If the non–printing characters are currently being displayed, this command is called Hide Invisibles.

- **Printing Format.** The Printing Format command brings up the standard Page Setup dialog box. Refer to *MacWrite* for more information.

THE FONT MENU

The Font menu enables you to change the display font. The Font menu is shown in Figure 3–10. A check appears in front of the font in which the active document is currently displayed. You can change the font by selecting another font from the menu.

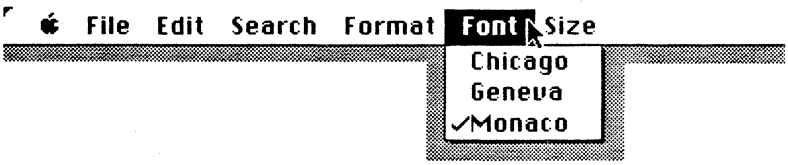


Figure 3–10. The Font Menu.

The font selected affects how many characters can be displayed on a line, and whether or not the display is proportionally spaced. Different fonts can be active in different windows at the same time. Which fonts can be selected depends on the fonts available on the system disk that you booted with.

NOTE: The **UCSD Pascal 1** disk has a System file that contains only the Chicago–12, Geneva–12, and Monaco–9 fonts installed on it. If you wish to use other fonts from the Editor, you must replace the System file, or use the Font Mover program to augment the font set of the System file.

THE SIZE MENU

The Size menu enables you to choose the size of the current font. The Size menu is shown in Figure 3–11. A check appears in front of the font in which the active document is currently displayed. You can change the font size by selecting another size from the menu.

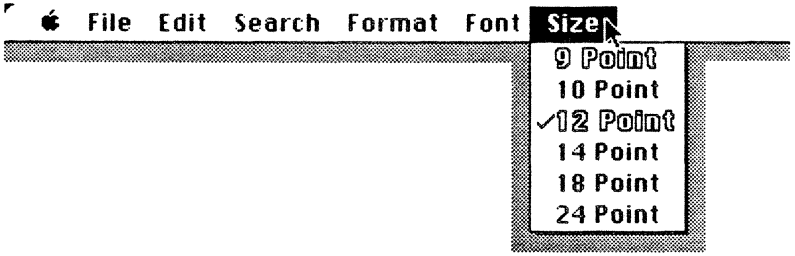


Figure 3–11. The Size Menu.

For each font, only certain sizes are available. These sizes are shown within the size menu in hollow letters. The font will look best if one of these sizes is selected. Otherwise, the Macintosh must do "scaling" which can detract from the appearance of the characters and slow down the speed of drawing characters.

4
PASCAL LANGUAGE

OVERVIEW

This chapter is a supplement to *The UCSD Pascal Handbook* which describes the version of the UCSD Pascal language supported by **The MacAdvantage: UCSD Pascal**.

The UCSD Pascal Handbook contains a thorough description of the basic UCSD Pascal language as it is implemented under Version IV of the **p–System**. **The MacAdvantage: UCSD Pascal** is an extended version of this UCSD Pascal language. In the creation **The MacAdvantage: UCSD Pascal**, some major new language features were introduced, and a few **p–System** specific features were removed.

In addition to the language features added for interfacing to the Macintosh, this supplement describes all of the enhancements to UCSD Pascal that have been introduced since the publication of *The UCSD Pascal Handbook*. There is also a section that identifies material in *The UCSD Pascal Handbook* that is not applicable to **The MacAdvantage: UCSD Pascal** environment. The last two sections contain revised descriptions of the compiler options and the conditional compilation facility.

Throughout the remainder of this chapter, the name UCSD Pascal refers to **The MacAdvantage: UCSD Pascal** version of the language.

Language Enhancements

The language features not described in *The UCSD Pascal Handbook* include:

1. The rules regarding the ordering of **label**, **const**, **type**, **var**, **procedure**, and **function** declarations within a declaration section have been relaxed. Identifiers must still be appropriately declared before they are used, but the usage of include files no longer influences the ordering that the compiler will accept. This gives you considerable freedom in the logical arrangement of large declaration sections. However, the compiler does require that it be able to resolve any accumulated forward references within pointer type declarations upon encountering a **procedure** or **function** declaration.

2. A new form of **uses** declaration called the "selective" **uses** declaration has been added to the language. This form of **uses** declaration is useful for economizing on symbol table space and resolving name conflicts between units.

3. Procedural and functional parameters are supported. This is a Standard Pascal construct for passing procedures and functions as parameters which was not implemented in earlier versions of UCSD Pascal.

4. Conformant arrays are supported. Conformant arrays are **array** parameters in which the **array** bounds are not fixed. The implementation follows the definition in the ISO Pascal standard.

5. A variant of the conformant array parameter construct called an "**interface** conformant array" is also supported. This construct is primarily useful in system programming for writing procedures which operate on parameters of arbitrary types.

6. The sizeof and pmachine intrinsics have been enhanced to make the writing of portable and efficient programs easier. These enhancements make it possible to (1) obtain the size of a variable or type in whatever units you wish, (2) store

pointer values in a size independent manner, and (3) easily generate the set of two byte p–code opcodes used by **The MacAdvantage: UCSD Pascal.**

7. Long integer arguments may be passed to the standard functions pred, succ, ord, and abs.

8. Due to a need for a clean interface to the Macintosh Operating System, a 32–bit integer data type, integer2, is supported. Unlike the long integers in UCSD Pascal, this data type may be used in all of the contexts where the integer data type may be used. (Long integers are still available, and have the same characteristics as before.)

9. Pointer manipulation intrinsics have been added to support manipulation of 32–bit absolute addresses. These intrinsics are: absadr, reladr, derefhnd, absmove, locate. Additional pointer manipulation intrinsics were added which can be used to manufacture or manipulate pointers in a size and implementation independent manner. These intrinsics are: adr, pointer, offset.

10. Bit manipulation intrinsics have also been added. These include band, bor, bxor, bnot, shiftleft, shiftright. These new intrinsics make efficient data manipulation operations easier to write.

11. An intrinsic called setlength has been added for setting the length of a string variable in an implementation independent fashion.

12. A new type of **external procedure**, called an "in–line **procedure**," is supported. A call to an in–line **procedure** becomes a direct call to a Macintosh Operating System routine.

Language Changes

The following are the language changes from the UCSD Pascal language under the **p–System:**

1. Two unadvertised constructs involving pointers are no longer allowed: (1) The standard **function** ord does not accept pointer arguments, and (2) pointers may only be compared for equality (=) or inequality (<>).

2. The unit I/O intrinsics are not supported. These are: unitread, unitwrite, unitstatus, unitbusy, unitwait, unitclear.

3. The gotoxy intrinsic is not supported due to the ambiguity of such an operation when a proportionally spaced character font is used for the .CONSOLE device.

USING THE HANDBOOK

This section is intended to bring to your attention certain material in *The UCSD Pascal Handbook* which either does not apply to you, or needs to be interpreted differently because you will not be writing UCSD Pascal programs under the **p–System.**

Using the Macintosh version of UCSD Pascal isn't radically different from what is described in the handbook. Most of the differences involve small details which will become clearer after you have absorbed the material in this chapter and the GENERAL OPERATIONS chapter.

In the handbook, there are a number of places where you are referred to manuals that are not included with the version of UCSD Pascal that you have purchased. The following table may give you some clues as to which chapter of this user manual to read in order to look up some of the topics referred to in *The UCSD Pascal Handbook.* The short explanations given here are intended to help you quickly sort out the differences between the descriptions in the handbook and the way things work with your Macintosh version.

p. 16: Library handling	See the LIBRARIAN chapter.
p. 17: Runtime Errors	See the GENERAL OPERATIONS chapter.
p. 19: Textfile maintenance	See the EDITOR chapter.
p. 27: Predeclared identifiers	List is not complete and includes identifiers that are no longer predeclared.
p. 27: pmachine intrinsic	The pmachine intrinsic is described in this chapter.
p. 59: trunc(L)	Produces overflow error if long integer L is outside of the range −maxint2−1 .. maxint2.
p. 86: Character−devices	The names of the character−devices are slightly different. Redirection of I/O on these devices is not supported. See the GENERAL OPERATIONS chapter.
p. 87: Keyboard End Of File	This feature is not available.
p. 95: Space Allocation	See the P−MACHINE ARCHITECTURE chapter.
p. 97: Real numbers	Only 64−bit real numbers are supported.
p. 101: sizeof intrinsic	The warning about the sizeof intrinsic is no longer accurate. See the revised description of the sizeof intrinsic in this chapter.
p. 103: declaration ordering	Include files no longer influence the ordering of declarations that the compiler will accept. See the discussion of this topic in the OVERVIEW section

	of this chapter.
p. 115: Library files	There is no file called *SYSTEM.LIBRARY. See the GENERAL OPERATIONS and LIBRARIAN chapters.
p. 133: Debugger	See the DEBUGGER chapter.
p. 135: input and output	The standard files input and output are permanently opened to the .CONSOLE device. See the GENERAL OPERATIONS chapter.
p. 140: File naming conventions	The Macintosh file naming conventions are similiar, but slightly different. See the GENERAL OPERATIONS chapter.
p. 146: keyboard	The file keyboard is opened to the .SYSTERM device. See the GENERAL OPERATIONS chapter.
p. 146: Device I/O	Material in this section is not applicable to the Macintosh environment. Low level device I/O can be done using the Macintosh interface unit PBIOMGR instead. See the MACINTOSH INTERFACE chapter.
p. 151: ioresult values	The ioresult intrinsic returns values different from those listed. In fact, ioresult can return negative values. See the ERROR MESSAGES Appendix.
p. 152: Screen I/O	There is no screen control **unit**. The gotoxy intrinsic is not supported.
p. 153: Memory Management	See the MEMORY MANAGEMENT chapter.
p. 163: Interrupts	No p–Machine events are supported. Thus the attach intrinsic cannot be used.

p. 167: Quiet compile option	Default setting is always "–". There is no file SYSTEM.MISCINFO.
p. 167: Realsize compile option	Only 64 bit real numbers are supported.
p. 170: Copyright notices	Up to 77 characters of copyright notice can be placed into the segment dictionary. The structure of the segment dictionary is described in the P–MACHINE ARCHITECTURE chapter.
p. 171: U(ser restart command	This feature is not available.
p. 172: External routines	The compiler will allow the form of **external** routine declaration shown here; but you need the appropriate assembler and linker to write external routines in assembly language. See IN–LINE PROCEDURES AND FUNCTIONS.
p. 280: BOOT_COPY program	This example program uses the unsupported unit I/O intrinsics, therefore it will not compile.
p. 307: ord(odd)	Technique discussed here still works with type integer; but will not work with type integer2. Use the bit manipulation intrinsics instead. See Bit Manipulation Intrinsics and Integer2 Routines.

INTEGER2 DATA TYPE

UCSD Pascal supports a 32–bit integer data type called integer2, which represents integral values in the range –2147483648 to 2147483647. The integer2 data type can be used in all contexts where it is legal to use integer. The integer2 data type is an extension to Standard Pascal.

Except for their differing sizes, the only difference in operation between integer2 and integer is the way that overflow is handled. Operations on the integer data type do not report integer overflow——the result of an overflow "wraps" back into the integer range, producing strange arithmetic results. Operations on integer2 report an execution error if the result of an expression is out of range.

Since the type integer2 can be used anywhere it is legal to use type integer, it is possible to:

- Index arrays with integer2 values.
- Use integer2 variables as **for** statement control variables.
- Use integer2 constants as **case** label constants in record type declarations and **case** statements.
- Use integer2 typed expressions as selectors in case statements.
- Define functions that return integer2 results.

Generally, you should use the integer2 type only when a particular Macintosh interface requires that you use it, or when the program you are writing requires the extended range of values offered by the integer2 type. This is because integer2 variables occupy twice the amount of memory as integer variables, and integer2 operations are somewhat slower than integer operations.

Integer2 Format

An integer2 constant value is represented by a sequence of digits, preceded by an optional '+' or '−'. If no sign is present, the constant is positive.

Each of the following is an integer2 constant:

```
0
7777777
-4582354
-1
```

Integer2 constant values can be specifed in the range −maxint2 .. maxint2. The identifier maxint2 is a UCSD Pascal predeclared constant identifier that has the value 2147483647. The constant identifier maxint2 is an extension to Standard Pascal. As with the integer data type, there is a negative integer2 value (−2147483648) that does not have a corresponding positive value.

An integer constant takes its type from the context in which it appears. Thus, 0 may represent an integer constant in one context and an integer2 constant in another, depending on what the compiler judges to be the required type.

Integer constants outside the range of values −maxint2 .. maxint2 are considered to be long integer constants.

Type Compatibility

As with the standard type integer, additional 32−bit integer data types may be declared via subrange type declarations. Any integer subrange type which includes integer values outside the range −maxint .. maxint is considered a subrange of the integer2 type. If either bound of such a subrange type lies outside of the range −maxint2 .. maxint2, the compiler reports a syntax error, since long integer subrange type declarations are not allowed.

The following example contains subranges of the integer and integer2 types:

```
0..maxint        {an integer subrange}
-55555..4        {an integer2 subrange}
5..maxint2       {an integer2 subrange}
```

The integer2 data types are assignment compatible with the integer data types, and vice versa. However, there can be a difference in meaning between a use of integer2 and integer, because the overflow conditions of the two types differ.

The type compatibility rules between the integer2 data types and the long integer data types are identical to the compatiblity rules between type integer and long integers. Briefly, these rules are as follows:

- In an expression, any integer or integer2 operand is compatible with a long integer operand. The conversion from integer or integer2 to long integer is done automatically.
- Long integers may be assigned the values of expressions of either integer or integer2 types. The conversion to long integer is done automatically.
- A variable of type integer or integer2 cannot be assigned the value of an expression of a long integer type. First, the long integer must be converted to an integer2 using the standard **function** trunc.

Integer2 Comparisons

All the comparisons legal for integer are also legal for the integer2 data type:

```
=    ... means ...   equal to
<>                   not equal to
>                    greater than
>=                   greater than or equal to
<                    less than
<=                   less than or equal to
```

Integer2 Operations

All the operations legal for integers are also legal for the integer2 data type.

These are the legal operations on a single integer:

```
+   ... means ... unary plus
-                 unary minus
```

A unary operator may not be strung together with a binary operator. The following example shows this:

```
5*-4            {illegal}
5*(-4)          {legal}
```

These are the legal operations on two integer2 operands:

```
+               plus
-               minus
*               times
div             integer divide
mod             remainder of integer divide
```

If the second operand of **div** is zero, a runtime error occurs. If the seond operand of **mod** is less than or equal to zero, a runtime error occurs. The integer2 **div** and **mod** operations are defined to perform the same functions as the integer **div** and **mod** operations.

The multiplicative operators *, **div**, and **mod** take precedence over the additive operators + and −. To override operator precedence, subexpressions may be grouped together with parentheses.

Integer2 Routines

The following routines take an integer2 parameter and return an integer2 result.

abs(I2) returns the absolute value of I2, which is an integer2.

sqr(I2) returns the square of I2, which is an integer2.

succ(I2) returns the I2+1, where I2 is an integer2.

pred(I2) returns the I2−1, where I2 is an integer2.

The standard functions (odd, chr, and ord) accept integer2 arguments. The functions ord, sqr, and abs will return type integer2 values when passed integer2 arguments.

NOTE: The standard **function** odd, when supplied with an integer2 argument produces exactly the Boolean values true and false. That is, ord(odd(E)), where E is an expression of type integer2, will always return zero (0) or one (1). This is not the case when the argument to odd is an expression of type integer, since odd only serves to change the type of the expression to Boolean and does not change the value in any way. What this implies is that you should NOT use odd as a type conversion function. Use the bit manipulation intrinsics instead of tricks which rely on the implementation of odd. For example, the obscure statement

```
Y := ord(odd(X) and odd(Z))
```

should be written as:

```
Y := band(X, Z)
```

The standard procedures read, and readln can be used to read values into integer2 variables. Similiarly, write and writeln can be used to write integer2 values to text files.

The standard functions trunc and round return type integer2.

Integer2 Conversions

In arithmetic expressions involving a mixture of data types, operands are automatically converted so the two operands of any one operation are of the same type.

The result type of an operation is established from the type of the operands. If both operands are of the same type, the type of the expression is the same as the type of the operands. If on the other hand the operands are of different types, the type of an expression is the type of whichever operand has the highest type–precedence.

The term "type–precedence" refers to a conceptual ordering of the various arithmetic data types. The type–precedence of a given type may be thought of as a measurement of the number of different types whose values can be converted to that type. Type real has the highest type–precedence, followed by integer2, and integer, in that order.

NOTE: Long integers are not compatible with type real. In an expression with a mixture of long integer operands and integer or integer2 operands, the type–precedence ordering is as follows: long integer, integer2, integer.

Two type conversion intrinsics called extend and reduce are defined which provide the programmer with facilities for controlling the type of an integer expression:

extend(X) causes the integer expression X to be converted to integer2 type. If the expression X is of the integer2 type, extend(X) is a null operation. It is natural to use extend in situations where both operands are of type integer, but the result

of an operation is expected to be outside the range of type integer. For example, the assignment statement

```
Grand_total := Last_year + This_year
```

could be written as

```
Grand_total := extend (Last_year) + This_year
```

in order to force the calculation to be performed using 32–bit integer arithmetic. In the situation depicted in the above example, GRAND_TOTAL would be a variable of type integer2, and the sum of the two integer values LAST_YEAR and THIS_YEAR is potentially larger than maxint.

reduce(X) causes the integer expression X to be reduced to type integer. If the value of X is outside the range of values −maxint−1 .. maxint an Integer Overflow execution error is reported. If X is already an expression of integer type, reduce(X) is a null operation.

PASCAL INTRINSICS

Setlength Intrinsic

A new string intrinsic called setlength is available in UCSD Pascal. Its definition is as follows:

setlength(DESTINATION, SIZE) is a **procedure.** It sets the current length of the string variable DESTINATION to the value of the integer expression SIZE.

For example,

```
setlength(S,length(S)+1);
S[length(S)] := '*';
```

appends an asterisk to S.

An advantage of using setlength as opposed to making an assignment to the "length character" is that range checking does not have to be disabled around the statement that sets the length of the string.

Bit Manipulation Intrinsics

UCSD Pascal contains a set of bit manipulation intrinsics to aid in disecting integer and integer2 values into fields. These are: band, bor, bxor, bnot, shiftleft, shiftright. Each of these intrinsics is a **function**. The four logical operations (band, bor, bxor, and bnot) provide a clean alternative to the old style "ord(odd(X) **and** odd(Y))" constructions.

band(P,Q) where P and Q are integer or integer2 expressions returns the bit—wise *and* of P and Q as an integer or integer2. If both P and Q are integer the result is an integer. Otherwise, the result is an integer2.

```
X:= band(X,255)      {masks X to its lower byte}
X:= band(X,-2)       {forces X to be even}
```

bor(P,Q) where P and Q are integer or integer2 expressions returns the bit—wise *or* of P and Q as an integer or integer2. If both P and Q are integers the result is an integer. Otherwise, the result is an integer2.

```
X:= bor(X,256)       {turns on bit 8 of X}
X:= bor(X,1)         {forces X to be odd}
```

bxor(P,Q) where P and Q are integer or integer2 expressions returns the bit—wise *exclusive—or* of P and Q as an integer or integer2. If both P and Q are integers the result is an integer. Otherwise, the result is an integer2.

```
X:= bxor(X,-1)       {inverts the bits of X}
X:= bxor(X,1)        {changes the parity of X}
```

bnot(P) where P is an integer or integer2 expression returns the bit—wise *ones—complement* of P as an integer or integer2. The type of the result is the same as the type of P.

```
X:= bnot(X)          {inverts the bits of X}
```

shiftleft(P,N) where P and N are integer or integer2 expressions returns the value of P shifted left by N bits. The bits that are shifted out of P are lost. The bits that are shifted into P are zero bits. If P is an integer, the result is an integer. Otherwise, the result is an integer2.

```
X:= shiftleft(X,1)   {doubles the value of X}
X:= shiftleft(band(X,255),8)
                     {shift the low byte of X}
```

shiftright(P,N) where P and N are integer or integer2 expressions returns the value of P shifted right by N bits. The bits that are shifted out of P are lost. The bits that are shifted into P are zero bits. If P is an integer, the result is an integer. Otherwise, the result is an integer2.

```
X:= shiftright(X,1) {halves the value of X}
X:= band(shiftright(X,8),255)
                    {returns the second byte of X}
```

Here is an example routine that uses the bit manipulation intrinsics to multiply (the hard way) two positive integers.

```
function TIMES(X,Y: integer): integer;
var
  RESULT,I: integer;
begin
  RESULT:= 0;
  for I:= 0 to 15 do
    begin
      if band(X,1) = 1
        then RESULT:= RESULT+Y;
      X:= shiftright(X,1);
      Y:= shiftleft(Y,1);
    end;
  TIMES:= RESULT;
end; {TIMES}
```

Pointer Intrinsics

UCSD Pascal contains a set of pointer manipulation intrinsics. These intrinsics were added to the language for the following reasons.

- They eliminate much of the need for the pmachine intrinsic and the ord(POINTER) construct to do pointer manipulation. This makes system–level pointer manipulation much cleaner and in some instances more efficient.

- They make pointer manipulation code independent of the representation or size of pointers. This paves the way for a larger pointer size in UCSD Pascal.

- They make it possible to manipulate data outside the Pascal Data Area. This is necessary in order to communicate with the Macintosh Operating System.

WARNING: The use of these intrinsics should be restricted to use in systems and application programs that must do unusual pointer manipulation or must call the Macintosh Operating System. Many of these routines do little or no type checking, so their use could be error–prone.

Besides the 16–bit representation of pointers used by UCSD Pascal for pointer variables, there are two other representations of pointers in UCSD Pascal. First, there is the "offset" representation of a pointer. An offset is a 16–bit signed integer that maps to a unique UCSD Pascal memory location. The representation of pointers as offsets is undefined. However, offsets have the following properties:

- Higher pointer addresses are represented by higher offset values. Thus offsets may be compared to determine the ordering of their respective pointers.

- A one word difference in pointer values is represented by an offset value change of 1. Thus offsets may be subtracted to determine the distance between two pointers in words.

The routines pointer and offset map between pointers and offsets.

The second alternative pointer representation is the "absolute pointer". An absolute pointer is represented by a positive integer2 value which is a 68000 32–bit address. The absolute pointer is provided in order to pass data to and from the Macintosh Operating System. The routines absadr and reladr map between pointers and absolute pointers.

Here are the pointer intrinsics:

offset(P) is a **function** which returns the word memory offset of the pointer P. The parameter P can be any expression of a pointer type. The result of offset(nil) is undefined.

pointer(O) is a pointer valued **function** which returns the pointer indicated by the offset O. The type of the result is the same type as the universal pointer constant nil.

adr(V) is a pointer valued **function** which returns a pointer to the variable reference V. (V may not be a component of a **packed array** or a field of a **packed record**. The variable reference V may be a reference to a subcomponent of a variable, as long as that subcomponent is word aligned and occupies at least one word of storage.) The type of the result is the same type as the universal pointer constant nil.

ptrinc(P,N) is a pointer valued **function** which returns the word pointer value obtained by adding the positive word offset N to the word pointer value P. The parameter P is an expression of any pointer type. The value of the parameter P may not be the same as the value of the pointer constant nil. The parameter N is an expression whose type is compatible with type integer or integer2. If the value of the parameter N is negative, the result of this **function** is undefined. The type of the result is the same type as the universal pointer constant nil.

NOTE: ptrinc is designed to be an efficient mechanism for stepping a pointer in short increments thru an allocated variable. If it is necessary to "back up" a pointer (i.e. add a negative offset) this can be done using offset and pointer.

absadr(P) is a **function** which returns the absolute address of the word pointed to by pointer expression P. The result is undefined if P is the pointer constant nil.

reladr(A) is a pointer valued **function** which returns a pointer to the word at the absolute address A. The result is undefined if the absolute address A is odd, or is not in the range of addresses that can be represented by a pointer. The type of the result is the same type as the universal pointer constant nil.

derefhnd(A) is an integer2 valued **function** which returns the absolute address of the word pointed to by the Macintosh handle A. (A handle is an abolute pointer to another absolute pointer called a "master pointer." The **function** derefhnd returns the low order three bytes of the master pointer.)

locate(V) is an integer2 valued **function** which returns the absolute address of the variable reference V. (V may not be a component of a **packed array** or a field of a **packed record.** The variable reference V may be a reference to a subcomponent of a variable, as long as that subcomponent is word aligned and occupies at least one word of storage.) The construction locate(V) is equivalent to absadr(adr(V)).

An alternative form of locate, locate(PROC, N), returns the absolute address of a PME entry–point which will cause activation of the routine specified by the **procedure** or **function** identifier PROC. The parameter N is an integer expression which specifies the number of the PME entry–point to be associated with the routine PROC. N must be in the range 1 to 9. (PME entry–point 0 is reserved for use by the Runtime Support Library.) The association of the entry–point with PROC remains in effect until a subsequent locate operation uses the same entry–point. The entry–point may only be called during the execution of an assembly language routine or during

an in—line procedure call.

absmove(SRC,DEST,NBYTES) is a procedure that moves NBYTES of data from SRC to DEST. SRC and DEST are absolute addresses. NBYTES is an integer2 expression. The action performed by absmove is equivalent the action of moveleft intrinsic, except that it can move data that is outside the Pascal Data Area. absmove is often used to move data into the Pascal Data Area so that it can be manipulated in a Pascal variable.

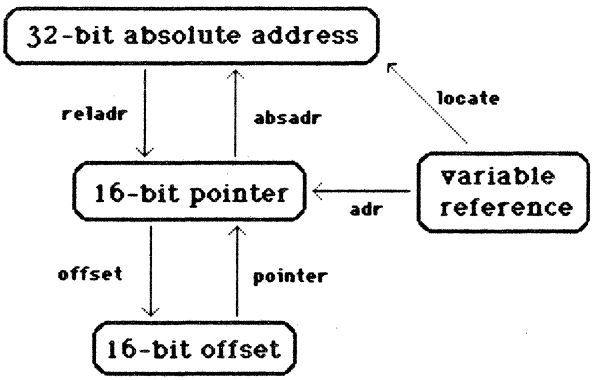


Figure 4—1. Pointer Intrinsics.

For more information on use of these pointer intrinsics, see the examples in the MACINTOSH INTERFACE chapter.

Pmachine Intrinsic

This section describes the pmachine intrinsic. The pmachine intrinsic allows you to generate in—line p—code. Its primary use is for performing tasks which the compiler does not ordinarily allow. In—line p—code can be useful in very low—level system programming. To use pmachine, you must understand the p—code operators described in the P—MACHINE ARCHITECTURE chapter.

The use of pmachine is discouraged for the following reasons:

1. In some cases, the p—codes you specify are altered by the compiler at compile time, producing unpredicatable results.

2. Software written with pmachine is often less maintainable than other software.

3. Software written using pmachine may be incompatible with future UCSD Pascal environments.

WARNING: Absolutely no protection is provided by this intrinsic or the system; use it with EXTREME CAUTION.

The following example shows the form of a call to pmachine:

```
"pmachine" "(" pmachine-item { "," pmachine-item } ")"
```

The parameters to pmachine are a list of one or more p—Machine—items. A p—Machine—item describes a portion of p—code, and causes one or more bytes to be generated.

The following list describes the four varieties of p—Machine—item:

1. p—code syllable: The simplest item is a scalar constant. This item produces a single p—code. If the constant is less than 255, the constant is the p—code. If the constant is greater than or equal to 255, a two byte p—code is generated consisting of a byte containing the value 255 followed by a byte containing the value (constant—255).

2. Expression value: If the item is an expression enclosed in parentheses, then a p—code sequence is generated which will compute the value of the expression and leave it on the stack.

3. Address reference: If the first token of the item is a caret (^), then the item is the specification of a variable, and p—code is generated which leaves the address of that variable on the stack. (The generated address is a pointer value, not an absolute address.)

4. Indirect store of pointer value: If the item consists of the Pascal assignment symbol, :=, the compiler is directed to generate code which accomplishes the storing of a p—Machine pointer value on the top of the stack into the pointer variable pointed to by a second pointer value on the stack (see the explanation below).

Given the following declarations:

```
const
  STO = 196;

type
  REC = record
          FIRST,SECOND: integer;
        end;
  RECP = ^REC;

var
  VECTOR: array[0..9] of RECP;
  I: integer;


pmachine(^VECTOR[5]^.FIRST, (I*I), STO);
```

would cause the square of I to be stored in the first field of the sixth element of the array VECTOR.

The fourth type of p—Machine—item is a syntactic mechanism for directing the compiler to generate the correct p—code sequence for an indirect store of a p—Machine pointer value regardless of pointer size.

The following pmachine construct illustrates the old way of storing a pointer value:

```
pmachine(^VECTOR[0], ^MYREC, STO);
```

The pmachine construct in the example pushes the address of VECTOR[0] onto the stack; pushes the address of the variable MYREC onto the stack; and finally uses the p–Machine STO instruction to store the pointer into VECTOR[0].

The following example shows how this same operation could be coded in a manner independent of the size of p–Machine pointer values:

```
pmachine(^VECTOR[0], ^MYREC, :=);
```

The appearance of the Pascal assignment symbol, :=, as a p–Machine–item causes the Pascal compiler to generate the p–Machine STO instruction.

NOTE: Always use the assignment symbol syntax to store pointer values into variables. This keeps your software independent of the size of p–Machine pointers. Also, DO NOT use the assignment symbol syntax to store anything other than pointer values; otherwise, your software may be invalidated by future UCSD Pascal implementations in which pointers are larger in size than a single p–Machine word.

Sizeof Intrinsic

The sizeof intrinsic has been enhanced in two ways. First, you may supply an optional units field, which allows you to select the units in which sizeof is to return the size. Second, you may supply optional tag fields, which allow sizeof to calculate the size of a particular variant of a record.

The syntax for sizeof is as follows:

```
"sizeof" "(" (type-identifier | variable)
             [ "," units { "," case-constant} ] ")"
```

The optional UNITS parameter is an integer constant that specifies the units in which the size of the type or variable is to be returned. If UNITS is omitted, a default value of 8 (the number of bits in a byte) is assumed. The units specification is a number of bits. If the size of the specified type or variable is N bits, the value returned is obtained by the following formula:

```
(N + UNITS - 1) div UNITS
```

The UNITS parameter may be followed by a list of tag field values that select a specific variant of a record type. The syntax and rules used for the specification of a variant are the same as for the standard **procedure** new.

The sizeof intrinsic may be used to determine the size of the actual parameter which corresponds to a formal conformant **array** parameter. When sizeof is used for this purpose, the compiler generates code to calculate the size of the actual parameter at runtime. In all other situations, the result is a constant value calculated by the compiler at compile time.

The following examples illustrate the various forms of the syntax for sizeof:

```
sizeof(V)          Returns size of variable V in bytes.
sizeof(T)          Returns size of type T in bytes.
sizeof(V, 1)       Returns size of V in bits.
sizeof(V, 8)       Returns size of V in bytes.
sizeof(V,16)       Returns size of V in words.
sizeof(R, 8, T1, T2, ...)
                   Returns size of the specified
                   variant of the record type or
                   variable R in bytes.
sizeof(P^.A[X], 16)
                   Returns size of the variable
                   P^.A[X] in words.
```

The following example also illustrates the use of tag fields in sizeof:

```
type
  R = record
        F1: integer;
        case HAS_MORE_THINGS: boolean of
          false: ();
```

```
            true:  (F2, F3: integer);
        end;
var
  I: integer;
  V: R;
  P: ^R;
begin
  new(P, false);
  V.F1 := 0;
  V.HAS_MORE_THINGS := false;
  moveleft(V, P^, sizeof(R, 8, false));
end.
```

The moveleft call moves only the F1 field of R, because the tag field of sizeof selected the false variant of R, which contains no additional fields.

IN–LINE PROCEDURES AND FUNCTIONS

UCSD Pascal has an alternative form of an **external procedure** or **function** declaration that allows you to gain immediate access to the Macintosh Toolbox routines. This form of **external** routine declaration is called an "in–line" routine. The syntax for declaring an in–line routine requires that you follow the reserved word **external** in your declaration by an integer constant enclosed in parentheses. The integer constant specifies the Macintosh "trap" instruction for the routine you wish to call. This syntax is illustrated by the following example:

```
procedure GetMouse( mouseLoc: PointPtr );
  external(-22158); {A972}
```

The example shows the declaration of the interface to the Macintosh Event Manager GETMOUSE routine, which is accessed by executing the trap instruction A972 (hexadecimal). The integer constant –22158 is the decimal value equivalent to A972. (UCSD Pascal does not allow you to specify constants in hexadecimal.)

When an in–line routine is called, the compiler generates code to pass the indicated parameters on the stack and then generates a special p–code which causes your program to execute the Macintosh machine instruction you have specified.

When you declare in—line routines, you do have to make sure that the number and types of the parameters are correct for the Macintosh routine you intend to call. It is also crucial that the **function** result type (if any) and the trap instruction number be correct. See the MACINTOSH INTERFACE chapter for detailed information about interfacing to the Macintosh Toolbox routines.

Unlike ordinary **external** routines, in—line routines may be declared within the **interface** section of a **unit**. Thus units can be used to organize collections of Macintosh Toolbox interfaces into manageable packages. This is precisely what was done to create the Macintosh Interface listed in Appendix A.

SELECTIVE USES DECLARATIONS

A selective **uses** declaration is a special form of the UCSD Pascal **uses** declaration that allows a client of a **unit** to select only those declarations that it needs from the **interface** section of the **unit**. (A client is a **program** or **unit** that **uses** another unit.) Selective **uses** declarations have two primary purposes. First, the client can better document which parts of a **unit** it is using by only selecting the pertinent declarations. Second, by only selecting declarations that are needed, symbol table space is conserved, so larger programs may be compiled.

Declarations from the **unit** are selected by listing the appropriate identifiers in parentheses after the **unit** name. The following defines the complete syntax of a **uses** declaration.

```
uses-declaration = "uses" unit-identifier
                   [ "(" identifier-list ")" ] ";" .

unit-identifier = identifier .

identifier-list = { identifier } .
```

A selective **uses** declaration consists of a simple **uses** declaration followed by a list of one or more identifiers enclosed in parentheses. Each of the identifiers in the list must be defined in the **interface** section of the **unit** being used. If a selected declaration is not present in the **interface** section, a syntax error

results.

Here is an example of a selective **uses** declaration:

```
uses MYUNIT (A_CONST, VAR1, VAR2, MY_ROUTINE);
```

A selective **uses** declaration specifies that only those declarations whose identifiers are listed are imported from the **unit**. The compiler first compiles the **interface** text for the **unit**, then discards the portions of the symbol table that describe declarations which were not selected. Thus, only the symbol table entries for the selected declarations are retained in the symbol table. The net result is a considerable savings in symbol table space when a client only requires a few declarations from a **unit** whose **interface** section is large. This makes it possible to compile larger programs than would otherwise be possible without selective **uses**.

While the primary advantage of the selective **uses** declaration is that the compiler's symbol table need not contain unnecessary declarations, there are other advantages as well.

First, a selective **uses** declaration can be a valuable documentation aid. The selective **uses** makes it easy to identify the specific declarations that a client needs from the **unit**.

Second, a selective **uses** declaration can remedy situations where there is a name conflict between units. This is done by not selecting one of the colliding declarations.

For example, suppose your **program** has a **procedure** called NEXT_LINE, and you decide to use a **unit** that also declares NEXT_LINE in its **interface** section. If you try to compile without a selective **uses**, you will get the syntax error "101:Identifier declared twice". You can avoid this situation by using a selective **uses** declaration to select only the identifiers you need, thereby avoiding the conflict with NEW_LINE.

WARNING: Despite the advantages of selective **uses** declarations, there are two anomalies which you should be aware of:

1. You must still have enough memory to compile the **interface** sections of the units that you use. Only after the interface for the **unit** is fully compiled does the compiler eliminate the declarations which are not selected.

2. Because selective **uses** declarations can be used to correct conflicts due to multiple declarations of the same identifier, a client which contains selective **uses** declarations may not compile successfully if the selective **uses** declarations are changed to simple **uses** declarations.

Here are the rules for inclusion of identifiers in a selective **uses** clause:

- If a selected declaration is not present in the **interface** section of the unit, a syntax error is issued by the compiler.

- Many identifiers do not need to be named explicitly in the selective **uses** list if they are referred to directly or indirectly within a selected identifier. For instance, field identifiers of a **record** are automatically included. An exception is that the names of type identifiers are never included.

The following is an example of selective **uses**.

```
unit TOOLS;

interface

type
  ALPHA = packed array[0..7] of char;
  SYM_TYPE = (BAD_SYMBOL, IDENTIFIER, OPERATOR);

  SYM_REC_P = ^SYM_REC;
  SYM_REC = record
              NAME: ALPHA;
              LLINK,RLINK: SYM_REC_P;
            END;

function CLASSIFY (NAME: ALPHA): SYM_TYPE;
{ Classifies a symbol as BAD_SYMBOL, IDENTIFIER
  or OPERATOR. }

procedure ENTER (NAME: ALPHA; var P: SYM_REC_P);
{ Creates a symbol table record with symbol NAME
```

```
  and installs it in the symbol table. }
implementation
  ...
end.
```

TOOLS is a **unit** with two procedures that manipulate a symbol table. Some clients of the **unit** call the **procedure** CLASSIFY while others call ENTER. If a client does not call ENTER, then the identifiers SYM_REC_P, SYM_REC, NAME, LLINK and RLINK are not needed. Likewise, if a client does not call CLASSIFY then the identifiers SYM_TYPE, BAD_SYMBOL, IDENTIFIER and OPERATOR are not needed.

The use of selective **uses** is demonstrated by two programs that are clients of **unit** TOOLS. Here is the first client of TOOLS:

```
program EXAMPLE_A;
uses {$U TOOLS.CODE} TOOLS (CLASSIFY,ALPHA);
var
  S: ALPHA;
begin
  S:= 'NewSym**';
  if CLASSIFY(S) = BAD_SYMBOL
    then writeln('the symbol is bad');
end.
```

EXAMPLE_A selects declarations for CLASSIFY and ALPHA from TOOLS. The following identifiers are imported from TOOLS: CLASSIFY, ALPHA, BAD_SYMBOL, IDENTIFIER, OPERATOR. The first two were specified explicitly in the selective **uses** declaration. The last three were included automatically because they are the constants of the scalar type SYM_TYPE, which is the function result type of CLASSIFY. Note that SYM_TYPE was not included, because indirectly referenced type names are never included. That is why EXAMPLE_A needed to specify type ALPHA explicitly in the selective **uses** declaration.

As expected, identifiers ENTER, SYM_REC_P, SYM_REC, NAME, LLINK and RLINK were not included, since none of them were even indirectly referenced.

EXAMPLE_B is a second client of TOOLS:

```
program EXAMPLE_B;

uses {$U TOOLS.CODE} TOOLS (ENTER,SYM_REC_P);

var
  REC_P: SYM_REC_P;

begin
  ENTER('NewSym**',REC_P);
  if REC_P^.NAME <> 'NewSym**'
    then writeln('symbol not entered');
end.
```

EXAMPLE_B specifies identifiers ENTER and SYM_REC_P in the selective **uses** declaration. The following identifiers are imported from TOOLS: ENTER, SYM_REC_P, NAME, LLINK, RLINK. As in EXAMPLE_A, the first two identifiers were named explicitly in the selective **uses** declaration. The last three identifiers were included automatically because they are fields of SYM_REC, which is indirectly referenced by both ENTER and SYM_REC_P. No other identifiers are imported from TOOLS.

CONFORMANT ARRAYS

This section describes conformant **array** parameters. Conformant arrays are **array** parameters in which the **array** bounds are not known until the procedure is called. Different size arrays of the same index type and base type may be passed on each call. The size of the **array** is determined by the upper and lower bound parameters, which are automatically passed to the routine.

Since the rules for using conformant arrays are a bit complicated, we will start with a small example. Here is an example conformant **array** parameter:

```
procedure A(var X: array[LO..HI: integer] of integer);
```

The occurrence of "**array** [...] **of** ..." signifies that x is a conformant **array** parameter. This syntax should be familiar from **array** declarations. However, instead of constant **array** bounds this **array** definition contains bounds parameter declarations (HI and LO in the example).

In the example, X is a conformant **array** parameter that may take any **array** of integers indexed by integers as a parameter. When **procedure** A is called, the bounds parameters LO and HI are set to the constant bounds of the actual parameter. Here is an example of two calls to A:

```
var
  B: array[0..9] of integer;
  C: array[-4..20] of integer;
  ...
begin
  A[B];    { LO is 0, HI is 9 }
  A[C];    { LO is -4, HI is 20 }
end;
```

Conformant arrays make it possible to write procedures that perform the same function on an assortment of **array** sizes. Consider the following example:

```
program CONFORMANT_ARRAYS;
  var
    X: array[1..10] of integer;
    Y: array[-100..100] of integer;
    X1,Y1: integer;

function SUM(A: array[LO..HI: integer] of integer): integer;
var
  I,RESULT: integer;
begin
  RESULT := 0;
  for I := LO to HI do
    RESULT := RESULT + A[I];
  SUM := RESULT;
end; {SUM}

begin
  { Assume the arrays contain some values. }
  X1 := SUM(X);
  Y1 := SUM(Y);
end. {CONFORMANT_ARRAYS}
```

SUM is a general purpose **function** to calculate the sum of an integer **array**. Because the parameter is a conformant **array** parameter, SUM is able to calculate the SUM of any integer

array it is passed. The first time SUM is called, LO will be 1 and HI will be 10. On the second call LO will be −100 and HI will be 100. The bounds parameters may be used in the procedure just as if they were normal integer parameters, except that you cannot assign anything to them or pass them as **var** parameters. The actual array parameter may be either a value or **var** parameter as desired.

The syntax of a conformant **array** parameter definition is as follows:

```
conformant-array-schema =
  packed-conformant-array-schema |
  unpacked-conformant-array-schema .

packed-conformant-array-schema =
  "packed" "array" "[" index-type-specification "]"
    "of" type-identifier .

unpacked-conformant-array-schema =
  "array" "[" index-type-specification
              { ";" index-type-specification } "]"
    "of" ( type-identifier |
           conformant-array-schema ) .

index-type-specification =
  identifier ".." identifier ":"
    ordinal-type-identifier .
```

A conformant **array** may be multidimensional. A multidimensional conformant **array** is specified by separating multiple index type specifications by semicolons, or by declaring an **array** of an **array**. The symbol ";" is a short–hand notation for "] **array of** [". Here is an example of a multidimensional conformant **array** parameter:

```
procedure A(var X: array[LO1..HI1: integer;
                         LO2..HI2: char] of integer);
```

Note that only the last index component of a conformant **array** may be specified as **packed**. Thus, a two dimensional conformant **array** with a **packed** second component must be specified:

```
procedure A(var X:
       array[LO1..HI1: integer] of
         packed array[LO2..HI2: char] of integer);
```

The short cut version using the semicolon notation may not be used in this case.

Any **array** that "conforms" to the conformant **array** parameter definition may be passed to the conformant **array** parameter. An **array** conforms if:

1. It has the same base type as the conformant **array**.

2. It has the same number of dimensions as the conformant **array**.

3. The type of each index is compatible with the index components in the conformant **array**.

4. The range of values of each index is within the range of the corresponding index type in the conformant **array**.

5. The **array**'s packing matches the packing of the conformant **array**.

A conformant **array** may be passed to another conformant **array** parameter as long as the parameter is declared as a **var** parameter. This restriction is due to the fact that the size of the value conformant **array** parameter must be known at compile time in order to allocate temporary storage for a copy of the actual parameter.

If more than one formal **array** parameter is named in an identifier list sharing the same conformant **array** definition, the actual parameters passed to those formal parameters must have the same bounds. Standard Pascal requires that the actual parameters be declared with the same type identifier. UCSD Pascal is not so strict. Consider the following example. Some of the calls are illegal:

```
program MORECONFORMANTARRAYS;
var
  A,B: array[1..10] of integer;
  C: array[0..99] of integer;
  D: array[1..10] of integer;

procedure SWAP(var X,Y:
                array[LO..HI: integer] of integer);
```

```
var
  I,TEMP: integer;
begin
  for I := LO to HI do
    begin
      TEMP := X[I];
      X[I] := Y[I];
      Y[I] := TEMP;
    end;
end; {SWAP}

begin
  {Assume the arrays have some values.}
  SWAP(A, B); {Legal.}
  SWAP(A, C); {Illegal--a and c have different bounds.}
  SWAP(A, D); {OK in UCSD Pascal,
               illegal in Standard Pascal.}
end. {MORECONFORMANTARRAYS}
```

Interface Conformant Arrays

UCSD Pascal supports a variant of the conformant **array** parameter called an **interface** conformant **array** that is even more flexible than the conformant **array** in the type of parameters it will accept. The **interface** conformant **array** is used primarily in system programming, where the need to write procedures that operate on arbitrary types is common.

WARNING: Because **interface** conformant arrays skirt all the type checking inherent in Pascal, they should be used only when necessary and with care.

An **interface** conformant **array** is declared just like a conformant **array**, except that the reserved word **interface** appears in front of the declaration. The following are some restrictions on **interface** conformant arrays.

- An **interface** conformant **array** must be a **var** parameter.

- An **interface** conformant **array** must be one dimensional.

Here is an example of an **interface** conformant **array** parameter declaration:

```
procedure A(var X:
        interface array[LO..HI: integer] of integer);
```

Here are some calls to the **procedure**:

```
var
 P: set of char;
 Q: packed array[0..100] of (red,green,blue);
  ...
begin
  A(P);
  A(Q);
end;
```

As this example shows, an **interface** conformant **array** will accept absolutely any type of variable as an actual parameter. Within **procedure** A, both P and Q are looked at as if they were each an **array** of integers.

The bounds parameters in an **interface** conformant **array** behave somewhat differently than in a conformant **array**. First, the low bound parameter is always set to zero. Second, the high bound parameter is set to the lowest value such that the **interface** conformant **array** will access all of the actual parameter. How large the high bound is set depends on the storage size of the actual parameter and the base type of the **interface** conformant **array**.

The following example shows how **interface** conformant arrays might be used in order to calculate a check sum of various pieces of data:

```
program SHOWINTERFACECONFORMANTARRAYS;
type
  BYTE = 0..255;
var
  S: string;
  BLOCK: packed array[0..511] of 0..255;
  A: array[1..10] of integer;
```

```
  I: integer;

function CHECKSUM
  (var X: interface packed array[L..H: integer] of
   BYTE ): integer;
var
  I,SUM: integer;
begin
  SUM := 0;
  for I := L to H do
    SUM := SUM + X[I];
  CHECKSUM := SUM;
end; {CHECKSUM}

begin
  {Assume the variables have some useful values.}
  writeln(CHECKSUM(S));       { L = 0, H = 80 }
  writeln(CHECKSUM(BLOCK)); { L = 0, H = 511 }
  writeln(CHECKSUM(A));       { L = 0, H = 19 }
  writeln(CHECKSUM(I));       { L = 0, H = 1 }
end. {SHOWINTERFACECONFORMANTARRAYS}
```

COMPILER OPTIONS

You may direct some of the compiler's actions by the use of compiler options embedded in the source code. Compiler options are a set of commands that may appear within "pseudo comments," and like any other Pascal comment, they are surrounded by either of the following pairs of delimiters:

```
parentheses/asterisks   (*      *)
braces                  {       }
```

The only difference is that a dollar sign ($) immediately follows the left–hand delimiter, for example:

```
{$I+}
(*$U MOLD.CODE*)
{$I+,S-,L+}
(*$R^*)
```

There are two kinds of compiler options: "switch" options and "string" options. A switch option is a letter followed by a plus (+), minus (–), or a caret (^). A string option is a letter followed by a string. (In the examples shown above, the second one is a string option; the others are switch options.) A pseudo comment may contain any number of switch options (separated by commas), and zero or one string options.

NOTE: If a string option is present in a pseudo comment, it must be the last option. The string is delimited by the option letter and the end of the comment. Also, if the pseudo comment uses the parenthesis/asterisk delimiters, (* and *), the string in the string option must not contain an asterisk.

Some options may appear anywhere within the source text. Others must appear at the beginning of the file (before the reserved word **program** or **unit**).

Switch options are either "toggles" or "stack" options. If a switch option is a toggle, a plus (+) turns it ON, and a minus (−) turns it OFF. The options 'I,' 'L,' and 'R' are stack options, as are the conditional compilation flags (see below).

With each stack option, the current state, either plus (+) or minus (−), is saved on the top of the stack, which can be up to 15 states deep). The stack may be "popped" by a caret (^) thus enabling the previous state of that option again. If the stack is "pushed" deeper than 15 states, the bottom state of the stack is lost. If the stack is popped when it is empty, the value is always minus (−).

```
{$I-}  ... current value is '-' — no I/O checking
       ...
{$I+}  ... current value is '+'
       ...
{$I^}  ... current value is '-' again
       ...
{$I^}  ... current value is '+', (this was the default)
{$I^}  ... current value is '-', (the stack is now empty)
```

The individual compiler options are described below in alphabetical order. If you do not use any compiler options, their default values will be in effect. Here are the default values for the compiler options:

```
{$Q-,R+,I+,L-,U+,P+,D-,N-}
```

These remain in effect unless you override them. The settings of the U and N options should not be changed.

Conditional compilation is also controlled by compile time options as described below.

$B — Begin Conditional Compilation

$B is a string option. It starts compilations of a section of conditionally compiled source code. See the section on conditional compilation, below.

$C — Copyright Field

$C is a string option. It places the string directly into the copyright field of the code file's segment dictionary. The purpose of this is to have a copyright notice embedded in the code file.

$D — Conditional Compilation Flag

There are two $D compiler options. This one is a string option. It is used to declare or alter the value of a conditional compilation flag. See the section on conditional compilation, below.

$D — Symbolic Debugging

The second $D compiler option is a switch option. $D+ turns on symbolic debugging information. $D− turns off symbolic debugging information. The default is $D−.

$E — End Conditional Compilation

$E is a string option. It ends a section of conditionally compiled

source code.

$I – I/O Check Option

There are two options named by $I. The first is a stack switch option (IOCHECK).

$I+, which is the default, instructs the compiler to generate code after each I/O statement in a program. This code verifies, at runtime, that the I/O operation was successful. If the operation was not successful, the program terminates with a runtime error.

$I– instructs the compiler not to generate any I/O checking code. In the case of an unsuccessful I/O operation, the program continues.

When you use the $I– option, your programs should specifically test ioresult when there is the chance of an I/O failure. If $I– is used and you don't test ioresult, the effects of an I/O error are unpredictable.

$I – INCLUDE File

This is a string option. The string (delimited by the letter 'I' and the end of the comment) is interpreted as the name of a file. If that file can be found, it is included in the source file and compiled.

```
{$I PROG2}
```

The example shown above "includes" the file PROG2 into the compilation unit's source code.

If the attempt to open the include file fails, or if an I/O error occurs while reading the include file, the compiler reports a fatal syntax error.

Include files may be nested up to a maximum of three files deep.

NOTE: Any leading spaces in a file name are discarded by the compiler. On the Macintosh, trailing spaces are significant in file names. Thus it is important that the end of comment delimiter be immediately adjacent to the last character in the file name. Furthermore, if a file name begins with a plus (+) or minus (−), a space must be inserted between the letter 'I' and the string. For example:

```
(*$I +PROG2*)
```

$L − Compiled Listing

$L is a stack option. You may use $L option either as a toggle switch option or as a string option. When used as a toggle, it turns the listing ON or OFF at that point in the source text. When used as a string option, it indicates the name of the listing file.

When used as a toggle, $L+ turns the listing ON and $L− turns it OFF. Using these options, you can list only parts of a compilation if you wish. The default for the toggle is $L− if you have not named a listing file using the compiler prompt or by using $L with a string option. The default value is $L+ if you have named a listing file in either of these ways. No matter which way you name the listing file, you can switch the listing ON or OFF by using $L+ or $L−.

If you do not specifically name a listing file and $L+ is in effect, the compiler writes to the file *SYSTEM.LST.TEXT.

$N – Native Code Generation

This is a switch option. $N+ outputs compiler information which allows native code generation to take place. $N– doesn't output this information. The default is $N–. Until such time as a Native Code Generator is available for this version of UCSD Pascal, you should not use $N+.

$P – Page and Pagination

The compiler can place page breaks in the compiled listing. It does this so that listings sent to the printer break across page boundaries. A form feed character (ASCII FF) is output every 66 lines if $P+ is in effect (this is the default). If you don't want this, use $P–.

You can cause a page break at any point in a compiled listing by using the $P option without a plus or minus sign.

$Q – Quiet

This is used to suppress the compiler's standard output to the console. $Q+ causes the compiler to suppress this output and $Q– causes it to resume outputting status information. If you have specified $Q+ and are obtaining a listing, the compiler does not pause when syntax errors are reported.

$R – Range Checking

$R is a stack switch option. The default value, $R+, causes the compiler to output code after every indexed access (for example, to Pascal arrays) to check that it is within the correct range. This is called range checking. The value $R– turns range checking off.

Programs compiled with the $R– are slightly smaller and faster since they require less code. However, if an invalid index occurs or a invalid assignment is made, the program isn't terminated

with a runtime error. Until a program has been completely tested, it is suggested that you compile with the R+ option left on.

$R2 and $R4 – Real Size

$R2 causes the code file's floating point arithmetic operations to be performed with two word (32–bit) precision. $R4 causes four word (64–bit) precision. The default and only supported real size for the Macintosh version of UCSD Pascal is four word reals. Therefore, you cannot use the $R2 directive, and never need to use the $R4 directive. If you do use the $R4 directive, it must occur before the first non–comment symbol in the compilation unit.

$T – Title

$T is a string option. The string becomes the new title of pages in the listing file.

$U – Use Library

Two options are indicated by $U. One is a string option (Use Library). The other, described below, is a toggle switch option (User Program).

With the Use Library option, the string is interpreted as a file name. This file should contain the unit(s) that your program is about to use. If the file is found, the compiler attempts to locate the unit(s) that it needs for the subsequent **uses** declarations. If a particular unit isn't found there the compiler issues a syntax error.

If a client (**program** or **unit**) contains **uses** declarations but no $U option, the compiler looks for the used units in the units (if any) that were compiled previously in the same compilation source file as the client.

The following is an example of a valid USES clause using the $U option:

```
USES UNIT1,UNIT2, { Found in current library }
  {$U A.CODE}
     UNIT3,       { Found in A.CODE }
  {$U B.LIBRARY}
     UNIT4,UNIT5; { Found in B.LIBRARY }
```

NOTE: Any leading spaces in a file name are discarded by the compiler. On the Macintosh, trailing spaces are significant in file names. Thus it is important that the end of comment delimiter be immediately adjacent to the last character in the file name.

$U – User Program

The $U– directive is used to specify that you are compiling a Runtime Support Library unit. This is how the Runtime Support Library units are compiled using the set of reserved unit names. $U– also sets $R– and $I–. You should not use $U–, and you never need to specify $U+. If you do specify $U+, it must appear before the heading (that is, before the reserved word **program** or **unit**).

CONDITIONAL COMPILATION

You may conditionally compile portions of the source text. At the beginning of a **program**'s text you can set a compile time flag which determines whether or not the conditionally compiled text will be compiled.

In order to designate a section of text as conditionally compilable, you must delimit it by the options $B (for begin) and $E (for end). Both of these options must name the flag which determines whether the code between them is compiled. The flag itself is declared by a $D option at the beginning of the source. $D options may be used at other locations in the source to change the value of an existing flag.

Here is an example:

```
{$D DEBUG}  {declares DEBUG and sets it TRUE}
program SIMPLE;
begin
   ...
   {$B DEBUG}  {if DEBUG is TRUE,
                this section is compiled}
   writeln('There is a bug.');
   {$E DEBUG}  {this ends the section}
   ...
   ...
   {$B DEBUG-}  {if DEBUG is FALSE,
                 this section is compiled}
   writeln('Nothing has failed.');
   {$E DEBUG}
   ...
end {SIMPLE}.
```

Each flag in a **program** must appear in a $D option before the source heading. The name of the flag follows the rules for Pascal identifiers. If the flag's name is followed by a minus (−), that flag is set false. The flag may be followed by a plus (+), which sets it true. If no sign is present, the flag is true. The flag's name may also be followed by a caret (^) as shown below.

The state of a flag may be changed by a $D option which appears after the source heading, but the flag must have first been declared before the heading.

The $B and $E options delimit a section of code to be conditionally compiled. The $B option may follow the flag's name with a minus (−), which causes the delimited code to be compiled if the flag is false. In the absence of a minus (−), the code is compiled if the flag is true. The flag's name may also be followed by a plus (+) or a caret (^); these are ignored. In a $E option, the flag's name may be followed by a plus (+), minus (−), or a caret (^); these symbols are ignored.

The state of each flag is saved in a stack, just as the state of a stack switch option is saved. Thus, using a $D option with a caret (^) yields the previous value of the flag. Each flag's stack may be as many as 15 values deep. If a 16th value is pushed, the bottom of the stack is lost. If an empty stack is popped with a caret (^), the value returned is always false.

If a section of code isn't compiled, any pseudo comments it may contain are ignored as well.

```
{$D DEBUG-}  {declares DEBUG and sets it FALSE}
program SIMPLE;
begin
   {$D DEBUG+}  {changes DEBUG to TRUE}
   ...
   {$B DEBUG}  {if DEBUG is TRUE, this section is
                compiled}
   writeln('There is a bug.');
   {$E DEBUG}  {this ends the section}
   ...
   ...
   {$D DEBUG^}  {restores previous value of DEBUG}
                 {... in this case, FALSE}
   {$B DEBUG-}  {if DEBUG is FALSE,
                 this section is compiled}
   writeln('Nothing has failed.');
   {$E DEBUG}
   ...
end {SIMPLE}.
```

5
MACINTOSH INTERFACE

This chapter describes the UCSD Pascal interface to the Macintosh Operating System and Toolbox. Because it is so large and complex, the Toolbox is not described in full here. You are encouraged to reference the Macintosh technical guide, *Inside Macintosh*, for a complete description of the Toolbox. The intent of this chapter is to describe the differences between the UCSD Pascal interface to the Toolbox and the Lisa Pascal interface described in *Inside Macintosh.*

Throughout this chapter "Toolbox" will refer to both the Macintosh Operating System and the Macintosh Toolbox. As far as the interface units are concerned, there is little difference between Toolbox routines and Operating System routines.

The Toolbox is a very complex piece of software. No one can be expected to learn how to use it in one reading, or even a few readings. The best thing to do is to learn the Toolbox in pieces, writing small programs as you go.

The most important part of this chapter (as well as the most complicated) is the section on DATA CONVENTIONS. You should probably skim this section on your first reading, then refer to it as necessary while writing programs that use the Toolbox interface.

Overall, the UCSD Pascal Toolbox interface is quite consistent with *Inside Macintosh.* However, for various reasons there are some restrictions and omissions in the UCSD Pascal interface. These are described in DIFFERENCES FROM *INSIDE MACINTOSH.*

The UCSD Pascal Toolbox interface is also quite consistent with the organization of *Inside Macintosh.* In general, each manager described in *Inside Macintosh* corresponds to a **unit** bearing the same name. There are some differences in the organization, however.

- There is a set of four "core" units that provide type declarations that are shared by the other units. In *Inside Macintosh* these declarations are included in the interface units themselves. Separating out some declarations saves having to use a whole unit where only some of its declarations are needed.

- The file manager and device manager routines have been redistributed as follows. High level file and device I/O have been combined in a unit called FileMgr. Low level file and device I/O have been combined in a unit called PBIOMgr (Parameter Block I/O Manager).

- The routines CountAppFiles, GetAppFiles, and ClrAppFiles have been moved from the Segment Loader to the OsUtility unit. There is no Segment Loader unit.

The rest of this chapter is arranged as follows:

HOW TO USE THE INTERFACE UNITS discusses making the interface units available to a program.

DIFFERENCES FROM *INSIDE MACINTOSH* discusses how use of the Toolbox routines from UCSD Pascal differs from *Inside Macintosh.*

DATA CONVENTIONS discusses issues regarding how Toolbox data is represented in UCSD Pascal. In particular, this affects how parameters are passed to the Toolbox routines.

SPECIFIC TECHNIQUES contains a set of example programming techniques that are helpful when using the Toolbox interface.

EXAMPLE APPLICATION contains a complete small application that uses the interface units.

HOW TO USE THE INTERFACE UNITS

This section discusses how to use the Toobox interface units from a UCSD Pascal program. There are two issues to consider.

1. How to make the **interface** sections of the units available at compile time.

2. How to make the code of the units available at runtime.

The use of units in general is discussed in *The UCSD Pascal Handbook.* This section focuses on the special considerations for use of the Toolbox interface units.

Appendix A contains listings of the **interface** sections of the interface units.

Compile Time Considerations

The interface units are contained in the file Mac Interface on the disk **UCSD Pascal 2**. The Librarian utility can be used to examine this file.

You make an interface unit available to your application through use of the **uses** statement. Often it is convenient to use the selective **uses** feature. Suppose you need to use the EraseRect and DrawChar routines from QuickDraw. Here is how you make them available.

```
program APPLICATION;

uses
  {$U UCSD Pascal 2:Mac Interface}
    MacCore,
    QDTypes,
    QuickDraw(EraseRect,DrawChar);
```

In the above example, the $U compiler option is used to open the library file Mac Interface on the volume **UCSD Pascal 2**. The volume prefix would not be needed if the library file were on the same volume as the UCSD Pascal compiler (the default volume). If you will not be swapping disks when compiling, you may also use #1: (which specifies the internal drive) or #2: (which specifies the external drive) to specify volume locations.

Nearly all of the interface units make use of other interface units. If one unit **uses** another unit within its **interface** section, you must include references to both units in your **uses** statement. The order of the units in the **uses** statement is important. In the example above, QuickDraw needs definitions from MacCore and QDTypes. Thus, they are both included in the **uses** statement before QuickDraw. QDTypes needs definitions from the MacCore unit, so MacCore is included before QDTypes. The selective **uses** declaration is discussed further in the PASCAL LANGUAGE chapter.

Appendix A contains a table of dependencies among the interface units. This table should help you to figure out which units are needed by other units. The column called 'Compile Time Dependencies' contains codes that indicate the units that are required by each unit.

The **interface** sections of the Toolbox interface units are very large. One of the problems with developing programs on a Macintosh with 128K bytes of memory is the lack of symbol table space while compiling. This can critically limit the size of a program that can be compiled unless steps are taken to conserve symbol table space.

Here are the things you can do to conserve symbol table space.

- Use selective **uses** to prevent unused definitions from being kept in the symbol table.

- Use the largest units with selective **uses** first, so that there is more symbol table space available while they are being compiled.

- Divide your program into units to minimize the number of interface units needed by each unit.

Here is an example of the first two points. Suppose you are using the Control Manager and QuickDraw. These require the use of MacCore, QDTypes and TBTypes. However, QuickDraw does not need any definitions from TBTypes. Therefore, you should arrange the units this way.

```
uses
  {$U #2:Mac Interface}
    MacCore,
    QDTypes,
    QuickDraw(...),
    TBTypes,
    CntrlMgr(...);
```

QuickDraw is much larger than the Control Manager, so it goes first. MacCore and QDTypes are used by QuickDraw so they must preceed QuickDraw. The Control Manager needs TBTypes in addition to MacCore and QDTypes.

It is possible to do even better than this. By looking at the **uses** declarations of QuickDraw and the Control Manager (in Appendix A), it is possible to make selective **uses** with the auxiliary units. QuickDraw needs all of MacCore and QDTypes, so nothing can be gained there. The Control Manager needs (GrafPort, GrafPtr, Point, VHSelect, FPoint, Rect, RectPtr) from QDTypes, and (EvtRecPtr, EventRecord, windowptr, windowhandle) from TBTypes. Therefore, the uses declaration could be made as follows.

```
uses
  {$U #2:Mac Interface}
    MacCore,
    QDTypes,
    QuickDraw(...),
    TBTypes (EvtRecPtr,EventRecord,windowptr,
```

```
                windowhandle),
    CntrlMgr(...);
```

You would add to the above **uses** statement any additional symbols your program requires from the QuickDraw and CntrlMgr units. This declaration makes optimum use of symbol table space.

If you have used these methods, and you still have trouble with running out of room while compiling, there is one other space–saving method that will help.

- Use in–line Toolbox routines right in your application without including a unit. This method is explained in detail in the section SPECIFIC TECHNIQUES.

Runtime Considerations

At runtime you must make the interface units available to your program. This is done by using the Library Files list facility in the Set Options utility or by using the Librarian utility to combine the units with your program. The Set Options utility is described in the chapter GENERAL OPERATIONS. The Librarian utility is described in the chapter LIBRARIAN.

Some of the interface units do not contain any code, and thus do not need to be included at runtime. The table in Appendix A indicates which units have code by a 'C' in the column called Code. The interface units that contain code are bound together in a library called Mac Library on the disk **UCSD Pascal 1.**

While you are developing and testing your program, we suggest that you use the Set Options utility to make Mac Library available to your program. This has the advantage that you can run the program immediately after compiling. When you complete the final version of the program, you should probably use the Librarian utility to include the interface units from Mac Library directly in your program. This makes the program self–contained, and reduces startup time.

DATA CONVENTIONS

UCSD Pascal is a different dialect and implementation of Pascal than Lisa Pascal, so there are differences in the interface units, accordingly. Most of these differences stem from the differences in the implementation of the Pascal language. Some of these implementation differences are related to different representations for data types, while others are a consequence of the different storage allocation algorithms used in the two implementations. Also, parameter passing methods differ between the two implementations.

An attempt has been made to provide Toolbox interface units whose interface is as close as possible to what is described in *Inside Macintosh*. In particular, it is nearly always the case that an interface routine takes the same number of parameters in the same order as in *Inside Macintosh*.

This section describes the data representation scheme used in the interface units. For information on the actual parameters of a particular routine in an interface unit, you must look at the description of the routine in *Inside Macintosh* and the declaration of the routine in Appendix A.

Passing Parameters to the ToolBox

Most of the ToolBox procedures in the Macintosh ROM were designed to work with the Lisa Pascal data and parameter passing conventions. In order to accommodate that interface, UCSD Pascal was extended to produce **The MacAdvantage: UCSD Pascal**. The extensions that are important to the Macintosh interface units are:

- A new type, integer2, was added to support 32–bit integers and addresses.

- The intrinsics locate and absadr were added to allow conversion from 16–bit UCSD Pascal addresses to 32–bit Lisa Pascal addresses.

- The intrinsic derefhnd was added to enable programs to dereference Macintosh Memory Manager handles.

- The intrinsic absmove was added to allow programs to move data to and from the Pascal Data Area.

- The new external procedure syntax **external**(...) was added to allow the UCSD Pascal compiler to generate in–line ToolBox calls in much the same way as the Lisa Pascal compiler.

In order to call the ToolBox procedures it is important that you understand all of these features. They are all documented in the PASCAL LANGUAGE chapter. Most of these features are used in the example program, GROW, located at the end of this chapter. They are also discussed with respect to their use in calling the ToolBox procedures later in this chapter.

The primary difference between UCSD Pascal and Lisa Pascal is that UCSD Pascal uses 16–bit addresses while Lisa Pascal uses 32–bit addresses. This affects the way in which you pass parameters to most of the ToolBox procedures. For example, a **var** parameter must be passed as a 32–bit pointer value parameter. Any Lisa Pascal value parameter that is larger than 32 bits must also be passed as a 32–bit pointer to the parameter.

Where necessary, the interface units make use of what are called "substitution types" instead of types whose declaration exactly matches those of *Inside Macintosh*. For example, the following types are declared in the MacCore unit (which contains most of the basic substitution type declarations):

```
type
  MacPtr = integer2 ;
  StringPtr = MacPtr ;
```

MacPtr represents a 32–bit pointer, while StringPtr represents a 32–bit pointer to a string variable. The StringPtr type is substituted in many of the interface unit procedures for the Str255 type that appears in *Inside Macintosh*. When you see StringPtr in a procedure declaration it means that you should be passing a 32–bit pointer to a string variable. Note that MacPtr and StringPtr types are the same type as integer2. Since the UCSD Pascal compiler will allow any integer2 value to be passed you must be careful to pass the correct value.

The following sections discuss all of the data representation and parameter passing differences between Lisa Pascal and **The MacAdvantage: UCSD Pascal**. After you read these sections, study the GROW program source. By looking at GROW you should begin to see how the ToolBox routines are called from a UCSD Pascal program.

UCSD Pascal Pointers vs Lisa Pascal Pointers

Lisa Pascal pointers are 32–bit absolute addresses, while UCSD Pascal pointers on the Macintosh are 16–bit offsets from the 68000 A6 register. This difference in pointer format between UCSD Pascal pointers and Toolbox pointers must be thoroughly understood in order to make use of the Toolbox interface.

An absolute address is represented in the Toolbox interfaces by the substitution type integer2. Two intrinsics are provided in UCSD Pascal to convert between pointers and absolute addresses: absadr converts a pointer into an absolute address; reladr converts an absolute address into a pointer.

NOTE: The pointer constant nil does not convert to the Macintosh value of nil. The constant AbsNil, declared in the MacCore unit, corresponds to a Lisa Pascal nil pointer. Also, there is no pointer value that corresponds to a odd absolute address.

The intrinsic adr takes a variable reference as a parameter and returns a pointer to that variable. The intrinsic locate takes a variable reference as a parameter and returns the absolute

address of that variable. The variable reference may be a reference to a sub—component of a variable, as long as that sub—component is word—aligned and occupies at least one word of storage. Locate(x) is equivalent to absadr(adr(x)).

Here are some examples of using absadr, reladr, locate,and adr.

```
var
  X: integer;
  P: ^integer;
  A,B: MacPtr;              {actually an integer2}
begin
  P:= adr(X);             {points p at the variable x}
  A:= absadr(P);          {sets a to the absolute address of x}
  B:= absadr(adr(X));     {sets b to the same thing}
  B:= locate(X);          {a shorter version of the last line}
  P:= reladr(A);          {points p at the variable x}
end;
```

Two more intrinsics round out the set of intrinsics that deal with pointer manipulation. The intrinsic derefhnd (dereference handle) returns the absolute address of the location the handle references. A handle is a Macintosh pointer—to—a—pointer used to reference relocatable blocks on the Macintosh heap.

NOTE: Derefhnd returns only the lower three bytes of the address. The upper byte, which contains Memory Manager attribute bits, is set to zero. For more information on Memory Manager attribute bits, see the Memory Manager chapter of *Inside Macintosh.*

Finally, the routine absmove is a block move intrinsic that acts like moveleft with absolute source and destination pointers. This intrinsic is useful for moving Macintosh—created data into a UCSD Pascal variable.

An example of the use of derefhnd and absmove is given below. This example allocates a 256 byte relocatable block by using the Memory Manager procedure NewHandle. It dereferences the handle returned in order to get the 32—bit absolute address of the block. Absmove is then used to move the string S into the block.

```
var
  sHandle :      Handle ;
```

```
  s :             String ;
  p :             MacPtr ;
begin
  s := 'Move this string to a relocatable block' ;
  sHandle := NewHandle (256) ;
  p := DeRefHnd (sHandle) ;
  Abs_Move ( Locate (s), p, Sizeof (s)) ;
end ;
```

LongInt

The Lisa Pascal type LongInt is used throughout the Toolbox as a parameter type and function result type. The UCSD Pascal equivalent to LongInt is integer2. In the MacCore unit there is a type declaration for LongInt:

```
type
  LongInt = integer2;
```

Pointer Types

All pointers within the Toolbox are represented in the interface units by the substitution type integer2 (interpreted as an absolute address). Because all Toolbox pointer types are integer2, there is effectively no type checking done when pointers are passed as parameters to a Toolbox routine. You should be very careful when passing pointer values to the Toolbox.

OpenPort in QuickDraw takes a pointer as a parameter. The following code fragment shows how a locally declared GrafPort could be passed to OpenPort:

```
var
  GP: GrafPort;

begin
  OpenPort(locate(GP));
end;
```

Call–by–reference Parameters

Call–by–reference parameters are parameters that are passed indirectly by passing a pointer to the item. One example of call–by–reference in Pascal is **var** parameters. Another example (one which depends on the implementation) is passing value (non–**var**) structures (e.g. arrays and records). In Lisa Pascal, value structures that are over 32 bits in size are always passed by reference.

Since call–by–reference parameters in UCSD Pascal are passed as 16–bit pointers on the stack, they cannot be used in calls to the Toolbox. Therefore, all call–by–reference parameters to the Toobox are passed as value absolute addresses.

For example, the Lisa Pascal definition

```
procedure GetFontInfo(var info: FontInfo);
```

is transformed into the UCSD Pascal definition

```
type
  FontInPtr = integer2;
procedure GetFontInfo(info: FontInPtr);
```

This calling mechanism is used for all **var** parameters and all value structure parameters over 32 bits in size. Here is an example call to GetFontInfo (declared in QuickDraw):

```
var
  FI: FontInfo;
begin
  GetFontInfo(locate(FI));
end;
```

Var pointer parameters are an especially confusing case. Here is an example:

```
var
```

```
  P: GrafPtr;
  GP: GrafPort;

begin
  GetPort(locate(P));
  absmove(P,locate(GP),sizeof(GP));
end;
```

This example loads the contents of the current GrafPort record into the local copy GP. If you understand this example, you should have no problems with call–by–reference parameters in the Toolbox interface.

Boolean

The Lisa Pascal representation of type Boolean differs somewhat from the UCSD Pascal representation, as follows:

- The UCSD Pascal Boolean is represented in a full 16–bit word. Only bit 0 of the word is significant. Zero (0) represents false. One (1) represents true.

- A Lisa Pascal Boolean value is represented in an 8–bit byte. As a parameter it is passed in the upper byte (bits 8 to 15) of a 16–bit word. All of these 8 bits are significant. Zero (0) represents false. Any nonzero value represents true. As a field in a record, a Boolean value is automatically packed into a byte.

Because of these differences, type Boolean is represented by the substitution types MacBool and SmallBool. MacBool is for Boolean parameters and SmallBool is for Boolean fields in a record. Unfortunately, MacBool and SmallBool are not compatible types. It is necessary to use the conversion routines when converting between them and UCSD Pascal Booleans.

Four conversion functions are available in the MacCore unit to map between MacBool or SmallBool and UCSD Pascal Boolean values:

```
ToMacBool(UB)    {converts UCSD format --> MacBool     }
FrMacBool(LB)    {converts MacBool     --> UCSD format}
ToSmall(UB)      {converts UCSD format --> SmallBool   }
```

```
FrSmallI(SB)      {converts SmallBool  --> UCSD format}
```

GetPixel in Quickdraw returns a Boolean value. Here is a call to GetPixel:

```
if FrMacBool(GetPixel(100,100))
  then ...
```

WARNING: When converting from SmallBool to MacBool it is necessary to go through the intermediate type Boolean; there are no provisions for converting directly between MacBool and SmallBool.

For example, suppose you want to pass the contrlVis field of a ControlRecord (a SmallBool) into the Visible parameter (a MacBool) of the Control Manager procedure NewControl. It is done as follows:

```
CH:= NewControl(...,ToMacBool(FrSmallI(CR.contrlVis)),...);
```

Packed Data

Lisa Pascal packs data differently from UCSD Pascal. The following differences have an effect on the Toolbox interface:

- Type Boolean within a **record** is automatically packed into a byte in Lisa Pascal. UCSD Pascal does not automatically pack any type.
- Lisa Pascal packs the fields of a record in a different order from UCSD Pascal.

Because of these differences, **packed** data is represented somewhat differently in the UCSD Pascal interfaces to the Toolbox.

First, records containing Booleans that will be automatically packed by Lisa Pascal are declared **packed**. Second, the order of declaration of fields in a **packed record** may be changed.

For example, the data type WindowRecord in the unit TBTypes contains four SmallBool fields. They are represented thus:

```
type
  WindowRecord =  packed record
                    port:        GrafPort;
                    windowKind:  integer;
                    hilited:     SmallBool;
                    visible:     SmallBool;
                    spareFlag:   SmallBool;
                    goAwayFlag:  SmallBool;
                    ...
                  end;
```

The **record** has been **packed** and the four SmallBool fields are declared in a different order from the Lisa Pascal interface.

Procedure Pointers

Procedure pointers are used to implement a procedure data type (including procedural parameters) in the Toolbox. Procedure pointers are usually used to pass some sort of "action procedure" to a Toolbox routine. For example, TrackControl in the Control Manager takes a parameter called actionProc. Periodically during a call to TrackControl, the Toolbox may call the user procedure actionProc. This procedure is passed to TrackControl as a procedure pointer, which is represented by the absolute address of its entry point.

The procedure pointer concept is supported in UCSD Pascal by an alternative form of the intrinsic locate. In this form, locate takes two parameters: a procedure or function identifier and an entry point number. It returns the absolute address of the entry point.

There are nine entry point numbers available for use by application programs. They are numbered one (1) through nine (9).

Here is an example of how to use locate.

```
procedure MYPROC;
begin
  ...
end;

begin
  i:= TrackControl(CH,P,locate(MYPROC,1));
  ...
end;
```

CH and P are other parameters to TrackControl (which is declared in the Control Manager unit) that can be ignored for the purpose of this discussion. Locate installs MYPROC in entry point 1, and passes the address of entry point 1 to TrackControl. When TrackControl wants to call the actionProc, it calls entry point 1, which causes MYPROC to be invoked.

Some action procedures are called immediately by the routine they are passed to. Others are called at a later time, or are not passed directly as parameters, but instead are installed in a data structure. There is a convention for selection of entry point numbers that will help eliminate some errors when using procedure pointers.

The convention is as follows.

- Entry point 0 is reserved for UCSD Pascal's grow zone procedure. You may not use entry point 0 in your application.

- Entry point 1 should be used for action procedures that have very limited scope. The parameter to TrackControl is an example. There, actionProc will only be called while TrackControl is executing. When TrackControl returns control to the user program, actionProc will no longer be called.

- The entry points greater than 1 should be used by action procedures of larger scope——those that will be called long after they are installed. The user is responsible for making sure that there is no conflict of entry point numbers within an application. Otherwise, serious errors will result.

Here is an example of using entry points greater than one. The grafProcs field of a GrafPort contains an array of low–level procedures that replace the default procedures in QuickDraw. You can customize QuickDraw by installing your own version of these procedures.

```
var
  GP: GrafPort;
  QDP: QDProcs;

begin
  SetStdProcs(locate(QDP));
  QDP.rectProc:= locate(MYRECT,2);
  QDP.rRectProc:= locate(MYRRECT,3);
  GP.grafProcs:= locate(QDP);
end;
```

In the example, entry points 2 and 3 must not be reused until the original rectangle and rounded rectangle primitives have been restored.

Enumerated Types

Enumerated types are affected by the order in which Lisa Pascal packs byte sized quantities. Lisa Pascal expects the small enumerated types to be passed in the upper half of a word. UCSD Pascal expects it in the lower half. Therefore, enumerated type parameters are represented by the substitution type integer, and the values of the enumerated type are represented by integer constants. DateForm in the Package Manager and GrafVerb in QuickDraw are two examples of enumerated types that have been replaced with constants.

Packed Array of Bit

Packed arrays of bits also suffer from byte—order problems. Lisa Pascal arranges the array indices in a word as follows:

```
 7  6  5  4  3  2  1  0 15 14 13 12 11 10  9  8
```

UCSD Pascal arranges the indices in a word as follows:

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

The best way to handle this rearrangement is to write an index mapping function from the Lisa Pascal index to the UCSD Pascal index. Here is an example mapping function for type KeyMap (declared in the Event Manager unit), which is a **packed array**[1..128] **of** Boolean.

```
function MKI(i: integer): integer; {Map Key Index}
begin
  if (i-1) mod 16 < 8
    then MapKeyIndex:= i+8
    else MapKeyIndex:= i-8;
end;
```

This function works by "switching" the upper and lower halves of each index range within a word. Suppose you want to set bits 32 and 55 in a KeyMap:

```
var
  KM: KeyMap;
begin
  KM[MKI(32)]:= true;
  KM[MKI(55)]:= true;
end;
```

Other bit arrays will require different mapping functions.

OSType and Point

OSType and Point are two Toolbox data structures that require special care when passed as value parameters. These two records fall into the category of structures that are 32 bits in size. When they are passed as value parameters, they are passed directly on the stack, instead of by reference. OSType is declared in the MacCore unit and Point is declared in the QDTypes unit.

Both these data types are represented by the substitution type integer2. The UCSD Pascal declarations of OSType and Point are case variant records that have a parameter field that is an integer2. This field must be passed as the parameter.

EqualPt in QuickDraw takes two value point parameters.

```
var
  P,Q: Point;

begin
  if EqualPt(P.Param,Q.Param)
    then ...
end;
```

CountResources in the Resource Manager takes a value parameter of type OSType.

```
var
  theType: OSType;
  x: integer;
begin
  theType.c:= 'STR ';
  x:= CountResources(theType.p);
end;
```

NOTE: If a Point or an OSType is passed as a **var** parameter, you must not pass it by the method shown above. Instead, it should be passed in the same way that other **var** parameters are passed.

DIFFERENCES FROM *INSIDE MACINTOSH*

The last section explained the differences between the UCSD Pascal Toolbox interface and the Lisa Pascal interface with regard to data representation. This section deals with the differences from *Inside Macintosh* with regard to which Toolbox routines may be called.

The differences explained here stem from three causes. First, UCSD Pascal uses memory in a slightly different way than Lisa Pascal does. Second, the UCSD Pascal implementation performs many of the necessary initialization steps described in *Inside Macintosh.* Finally, the implementation of procedure pointers (ProcPtrs) imposes some restrictions.

Memory Restrictions

This section explains briefly how UCSD Pascal uses Macintosh memory, and how this affects application programs. For a more detailed description of memory usage see the chapter MEMORY MANAGEMENT.

The important points about UCSD Pascal memory usage are as follows:

- UCSD Pascal uses the Macintosh stack for its stack.
- The UCSD Pascal heap is implemented as a nonrelocatable Macintosh block within the Application Heap Zone. This block expands and contracts according to heap usage. All data allocated with new or varnew is allocated here.

- The boundary between the end of the Application Heap and the stack (ApplLimit) moves to accomodate the growth of the stack.

The rule to remember when making Memory Manager calls from UCSD Pascal is:

- DON'T allocate a nonrelocatable block immediately above the UCSD Pacal heap if you plan to make use of the Pascal heap. The nonrelocatable block you allocate will most likely be positioned immediately above the heap by the Macintosh Memory Manager. This will prevent expansion of the Pascal heap. When you need to create a nonrelocatable memory area, you should use the UCSD Pascal intrinsics new or varnew. You can then convert the 16—bit pointer returned by these intrinsics into a 32—bit address by using the function absadr.

For reference, here is a list of the ways that a nonrelocatable block can be created.

- A call to NewPtr creates a nonrelocatable block.
- A call to HLock makes a relocatable block nonrelocatable.
- A call to NewHandle can cause a new block of master pointers to be allocated. These are put in a nonrelocatable block. The UCSD Pascal runtime software preallocates a block of 64 master pointers. In order to increase this number you need to define a new resource file for your program. The example RMaker input below will allocate 2 master pointer blocks for a total of 128 master pointers. The GNRL type MSTR defines the number of master pointer blocks that should be prellocated.

```
MY.RSRC                    ;; Output file name
APPLPROG                   ;; Type = APPL, Creator = PROG

INCLUDE UCSD Pascal 1:Empty Program
                           ;; Required resources

TYPE MSTR = GNRL
  ,0 (32)
.H
0002                       ;; Allocates 2 master pointer blocks
```

Here is a list of which routines from the memory manager must be used differently from what is described in *Inside Macintosh.*

SetGrowZone. You must not install your own grow zone function for the Application Heap Zone. The Pascal runtime system already has one. You may, however, use your own grow zone function in a heap zone of your own creation.

InitApplZone. This routine is not supported, because calling it will corrupt the UCSD Pascal code and data structures that are kept in the Application Heap Zone.

SetApplBase. This routine is not supported, because it will interfere with Pascal's use of the Application Heap Zone.

SetApplLimit. This routine is not supported, because the UCSD Pascal runtime support software automatically adjusts the Macintosh's ApplLimit variable for you. Calling this routine will interfere with Pascal's use of the Application Heap Zone.

There are two general strategies of memory use that an application can employ. An application could make use of the Pascal heap. If so, the program must be especially careful about use of the Macintosh memory management routines. Alternatively, an application could avoid use of the Pascal heap altogether. In this case, the program may use the Macintosh memory management routines with a little less care than if the Pascal heap were being used.

There are some special considerations regarding dereferencing a handle under UCSD Pascal. In particular, there are more ways that the Memory Manager can be called "behind your back" when UCSD Pascal code is running. Here is a list of ways that the memory manager may be called.

- Calling a procedure (especially one with local data) can cause a stack fault, which will result in some memory management functions being performed. A stack fault can also occur when using long integers and sets in UCSD Pascal.

- Calling an external procedure or a system intrinsic can cause a segment fault, which causes a code segment to be read into memory. This action will result in some memory management functions being performed.

- Allocating data on the Pascal heap with new or mark can cause a heap fault, which can result in memory management functions being performed.

NOTE: Calling a Macintosh ROM routine that is declared as an in—line procedure or is an external procedure implemented in assembly language will never cause a stack fault. Thus, it is safe to pass a dereferenced handle to most ROM routines.

Initialization

This section describes some initialization routines described in *Inside Macintosh* that do not need to be called from a UCSD Pascal program. Some of these routines are not available at all.

InitGraf. InitGraf is not available in the UCSD Pascal interface to QuickDraw. The operations performed by InitGraf are done automatically.

FlushEvents. FlushEvents(everyEvent,0) is done by the UCSD Pascal runtime support initialization code. There is no need to call FlushEvents in the initialization of your program.

InitDialogs. InitDialogs is done by the UCSD Pascal runtime support initialization code. You may call InitDialogs yourself if you want to install a restart procedure in the system.

InitFonts. InitFonts is done by the UCSD Pascal runtime support initialization code. There is no need for your application to call InitFonts.

InitWindows. InitWindows is done by the UCSD Pascal runtime support initialization code. You should not call InitWindows yourself, since it allocates a nonrelocatable block on the Application Heap Zone.

TEInit. TEInit is done by the UCSD Pascal runtime support initialization code. You must not call TEInit yourself.

The following calls are made for your program when the "Create Default Window" option described by Runtime Parameters in GENERAL OPERATIONS is enabled. In that case, you do not need to call them.

SetPort. If the "Create Default Window" option (which can be enabled or disabled by using the utility Set Options) is disabled, you must call SetPort yourself before using any QuickDraw routines.

NewWindow. If the "Create Default Window" option is disabled, you must open a window yourself before you do any writing to the screen.

InitCursor. If the "Create Default Window" option is turned off you will need to call InitCursor from your application in order to reset the cursor to be an arrow.

HideCursor. In order to make the cursor visible, call ShowCursor.

Procedure Parameter Restrictions

Due to the implementation of procedure parameters to the Macintosh Toolbox, there are some restrictions on their use beyond what is described in *Inside Macintosh.* These restrictions are as follows.

- You may not supply an I/O completion routine to an asynchronous I/O call. Instead, you must poll the parameter block to determine I/O completion.

- You may not implement a vertical retrace procedure.

These restrictions are due to the fact that the implementation of ProcPtrs will not handle asynchronous calls to an action procedure.

SPECIFIC TECHNIQUES

This section presents some techniques that will be of use in writing applications that use the interface units. Some complicated topics from earlier sections of this chapter were postponed until this section, because a more thorough discussion could be accomplished here.

Data Outside the Pointer Range

As discussed above, UCSD Pascal pointers have limited scope. In particular, they are only able to address memory within the 64K region that encompasses the Pascal Data Area. When it is necessary to access some data outside the Pascal Data Area, there are two ways it may be done.

1. Copy the data into a Pascal variable. After it is copied into the Pascal Data Area, it may be examined directly. If it is to be modified, then the modified copy must be installed by copying the data back into the original.

2. Access the data in place. Here, modification may be done directly, although without the help of record field names. With this method, you must know much more about how data is represented in the interface units.

The routine absmove is used to move data from one location to another within Macintosh memory.

Suppose you want to update the grafProcs field of the current GrafPort. Using method 1, it would be done as follows:

```
var
  GPP: GrafPtr;     { pointer to a graf port }
  GP: GrafPort;     { will contain copy of grafport record }
  QDP: QDProcs;     { the graf procedures record }
begin
  GetPort(locate(GPP));
  absmove(GPP,locate(GP),sizeof(GP));
  GP.grafProcs:= locate(QDP);
  absmove(locate(GP),GPP,sizeof(GP));
end;
```

Using method 2, it would be done as follows:

```
var
  GPP: GrafPtr;
  QDPP: MacPtr;
  QDP: QDProcs;
begin
  GetPort(locate(GPP));
  QDPP:= locate(QDP);
  absmove(locate(QDPP),GPP+sizeof(GrafPort)-4,sizeof(QDPP));
end;
```

Accessing a Macintosh Operating System Global

Globals may be accessed by manufacturing a pointer to them. For instance, the global ScrVRes is at location 102H. This word may be accessed as follows:

```
var
  CopyOfScrVRes: integer;
begin
  absmove(258 {102H},locate(CopyOfScrVRes),sizeof(integer));
end;
```

How to Dereference a Handle Safely

In UCSD Pascal, a handle is dereferenced into a pointer by using the intrinsic derefhnd. However, you must be somewhat careful when dereferencing a handle in UCSD Pascal, because there are some additional places where memory management routines will be called that may invalidate the dereferenced handle. Memory management routines are called "behind your back" when Pascal handles one of its internal faults (stack fault, heap fault or segment fault).

The following actions may cause a fault to occur:

- Calling a procedure may cause a stack or segment fault.
- Calling a Toolbox interface procedure that is not declared using the **external**(...) syntax may cause a stack or segment fault. Procedures declared with the **external**(...) syntax will never cause segment or stack faults. They may, however, cause relocatable blocks to move.
- Allocating data on the Pascal heap with new or varnew may cause a heap fault.

If you must dereference a handle across one of the dangerous calls mentioned above (or across one of the dangerous calls mentioned in *Inside Macintosh*), you must work on a copy of the data or use the Memory Manager procedure HLock to position lock the data.

How to Set Stack Slop

UCSD Pascal operates its stack in an unusual way, by Macintosh standards. In particular, UCSD Pascal moves the boundary between the stack and the application heap. Most Macintosh applications leave this boundary fixed.

In order to detect when the boundary needs to be moved, the runtime system knows about a "stack slop" value that represents the minimum distance between the top—of—stack and the top of the application heap. This stack slop has a minimum size of 2K (2048) bytes.

Most of the time, 2Kb of slop is plenty of extra stack space for calling Macintosh ROM routines. (ROM routines steal stack space without telling UCSD Pascal or your program.) However, there are some ROM routines that place an extra burden on stack space.

If you are going to be calling one of these routines, you should increase the stack slop by calling the routine SetStackSlop in the Error_Handling unit. This unit is not in the Pascal Runtime library, so you will have to make sure its code is available at runtime by using the user library feature in Set Options, or by using the Librarian utility to include its code in your application.

Suppose you need 6Kb of stack slop for a portion of your program. This can be set as follows:

```
var
  default_slop : integer ;
begin
  default_slop := GetStackSlop ;
  SetStackSlop(6*512 {words});
  { put code that needs large slop factor here }
  SetStackSlop(default_slop); { restore default slop }
end;
```

Each separate UCSD Pascal process has its own stack slop. Many programs do not use processes, so they only need to worry about one stack slop. If your application uses processes, and you are doing ToolBox calls from them, be sure that you keep in mind that different processes have different slop factors. In particular, the default slop factor for a subsidiary process is forty (40) words. You set the slop factor for a process by calling SetStackSlop from within that process.

When your program is started by the UCSD Pascal runtime support software, it is running as the "main task", and the stack slop is set to a default value of 5Kb. This amount of slop allows the Macintosh Operating System to save the screen image bits for the portion of the screen image that is obscured by "disk swap boxes." A disk swap box appears when your program or the runtime support software attempts to access a file on a volume that is mounted, but not physically present in the appropriate disk drive. After you supply the requested disk, the Macintosh Operating System will restore the affected portion of the screen image, provided there was enough space to save it.

If your program **uses** the Error_Handling unit to set the stack slop below the default value of 5Kb, the disk swap boxes will still appear, but will remain visible on the screen until the next time your program or the runtime support software calls the Event Manager routine GetNextEvent. GetNextEvent will fill the affected area of the screen with the appropriate background pattern. Usually, you would set the stack slop to less than 5Kb only if there is a critical need to maximize your program's utilization of memory. For example, the UCSD Pascal compiler sets the stack slop to its minimum value of 2Kb so as to maximize the capacity of its symbol table.

The SetStackSlop routine will not let you set the stack slop below the minimum of 2K bytes. (Otherwise your program would probably crash, as discussed below.) A convenient way to set the stack slop to its minimum setting, without placing the "magic" 2K byte number in your program is to pass zero (0) for the argument to SetStackSlop.

NOTE: Once you set the stack slop below the default setting of 5Kb, the saving of the screen contents underneath disk swap boxes becomes permanently disabled (i.e. even if you later set the slop back to 5Kb, disk swap boxes wll continue to remain on the screen until GetNextEvent is called).

While your application is running as the main task, the Macintosh's "stack sniffer" is enabled. The stack sniffer detects when the stack expands into the Macintosh heap. If your application gets a stack sniffer error (a "bomb" with ID=28) you have probably failed to provide enough stack slop to your application. The stack sniffer is not enabled while you are within a subsidiary task——you are on your own if you make use of processes.

Declaring ToolBox Interface Procedures

There may be some instances when you need to use only one or two procedures from an interface unit. If the declarations of these procedures in the **interface** unit ends with an **external** then you can declare them yourself. For example, the following program calls the QuickDraw procedure InitCursor without using the QuickDraw interface unit.

```
program doint ;

 procedure InitCursor ; external (-22448) ;

begin
  InitCursor ;
end.
```

The above program will compile much faster than the program below which uses the QuickDraw unit.

```
program doint ;

Uses {$U Mac Interface}
     MacCore,
     QDTypes,
     QuickDraw (InitCursor) ;

begin
  InitCursor ;
end.
```

This technique would be particularly useful if the only procedure you needed from QuickDraw was the InitCursor procedure, which uses none of the type declarations found in MacCore or QDTypes.

EXAMPLE APPLICATION

This section presents an entire (although small) Macintosh application complete with scroll bars, grow box, menu bar and desk accessories. The source code for this example is located in the files GROW and GROW.R on the **UCSD Pascal 2** disk. In order to see the application in action you must use RMaker, the Compiler and the Set Options program as outlined in the following steps.

1. Use the RMaker utility on GROW.R. This will create the file GROW.RSRC.

2. Compile GROW. Use GROW.RSRC as the resource input file.

3. Use the utility Set Options to set the locations of the Pascal Runtime, p–Machine and Mac Library files. You must disable the "Create Default Window" option (GROW creates its own window).

The GROW.Code program puts up a single window in which you can insert and edit text. The window can be sized and moved. The text in the window can be scrolled horizontally and vertically.

You should use the GROW program source as an example of:

- the handling of Macintosh events. Notice that window update events are generated by the Macintosh ROM. The GROW window is updated as a response to these events.

- the calling conventions for many of the ToolBox procedures.

- the relationship between resources defined in a resource file and the program code that uses those resources.

In addition the GROW program demonstrates the use of the ToolBox from UCSD Pascal. For example, in procedure Initialize the line:

```
SetRect (locate(DragRect), 4, 24, 508, 338) ;
```

initializes the rectangle DragRect. The call to locate returns the 32-bit address of DragRect. This address is passed as a parameter to the SetRect procedure.

In procedure CursorAdjust the line:

```
if FrMacBool(PtInRect(mousePt.param,locate(TRect)) then
```

tests to see whether the point specified by mousePt is in the rectangle specified by TRect. Notice the use of the param field of the mousePt variable. This field is used to pass the value of mousePt to the procedure PtInRect. FrMacBool is used to convert the Lisa Pascal Boolean, returned by PtInRect, to the UCSD Pascal representation of Boolean.

Modifying data outside of the UCSD Pascal Data Area is demonstrated by the following lines of code from the procedure GrowWnd.

```
sbs_move (derefhnd (hTE), locate (dummy), sizeof (dummy)) ;
dummy.viewrect := TRect ;
sbs_move (locate (dummy), derefhnd (hTE), sizeof (dummy)) ;
```

The above code copies the first part of the text edit record, pointed to by the handle hTE, to a local variable (dummy). Dummy is updated and put back into the text edit record.

The use of ToolBox procedure pointers is demonstrated by the following line of code in procedure DoMouseAction.

```
tc := TrackControl (whichControl, MouseEvent.where.param,
                    locate (Scrollup,1)) ;
```

The procedure Scrollup (declared earlier in the program) is being passed to the ToolBox procedure TrackControl. Scrollup is called by TrackControl to scroll the bits of the text edit window.

```
program Grow;

{ This example program is based on a program of the same name
  written by Cary Clark of Macintosh Technical Support. }

{$L-}
Uses {$U UCSD Pascal 2:Mac Interface}
  MacCore,
  QDTypes,
  TBTypes
    ({types} EvtRecPtr, EventRecord, WindowRecord, WindowPtr,
             WindowHandle, TEHandle, TEPtr, TERec),
  OsTypes
    ({types} QElemPtr, QHdrPtr),
  MacData
    ({vars } Arrow, thePort),
  QuickDraw
    ({procs} SetCursor, SetRect, PtInRect, SetPort, GetPort,
             EraseRect, GlobalToLocal, ClipRect),
  EventMgr
    ({const} everyevent, mousedown, keydown, autokey, activateEvt,
             updateEvt,
     {procs} GetMouse, GetNextEvent, StillDown),
  WindowMgr
    ({const} inDesk, inMenuBar, inContent, inDrag, inGrow, inGoaway,
             inSysWindow,
     {procs} GetNewWindow, FrontWindow, DrawGrowIcon, BeginUpdate,
             EndUpdate, FindWindow, DragWindow, TrackGoAway, SelectWindow,
             InvalRect, SizeWindow, GrowWindow),
  MenuMgr
    ({types} MenuHandle,
     {procs} InitMenus, GetMenu, AddResMenu, InsertMenu, DrawMenuBar,
             MenuKey, MenuSelect, HiliteMenu, GetItem, EnableItem,
             DisableItem) ,
  ControlMgr
    ({const} inUpButton, inDownButton, inPageUp, inPageDown, inThumb,
     {types} ControlHandle, ControlPtr, ControlRecord,
     {procs} GetNewControl, ShowControl, HideControl, DrawControls,
             FindControl, TrackControl, GetCtlValue, SetCtlValue,
             TestControl, MoveControl, SizeControl),
  TBoxUtils
    ({procs} GetCursor, HiWord, LoWord) ,
  DeskMgr
    ({procs} SystemTask, SystemClick, SystemEdit, OpenDeskAcc),
  TextEdit
    ({procs} TENew, TEIdle, TEKey, TEActivate, TEDeactivate,
             TEUpdate, TEClick, TECut, TECopy, TEPaste, TEScroll) ,
```

```
  OsUtilities
    ({procs} Delay);
{$Lt}

const
  applemenu      = 1;              { Menu ID for desk accessory menu }
  filemenu       = 1000;           { Menu ID for my File Menu }
  editmenu       = 1001;           { Menu ID for my Edit Menu }
  lastmenu       = 3;              { there are 3 menu items }
  wndwid         = 1000;           { Window ID for theWindow }
  ibeamID        = 1;              { IBeam Cursor ID }
  VScrollID      = 1000;           { Control ID for Vertical Scrolling }
  HScrollID      = 1001;           { Control ID for Horizontal Scrolling }
  UnDoItem       = 1;              { Item # for UNDO Menu Item }

var
  doneFlag:      boolean;
  MyMenus:       ARRAY [1..lastMenu] OF MenuHandle;
                                   { Handles to Menu resources }
  GrowRect:      Rect;             { Limits the size of window during grow }
  DragRect:      Rect;             { Limits the dragging of the window }
  wRecord:       WindowRecord;     { The window we operate on }
  theWindow:     WindowPtr;        { A pointer to the window }
  tRect:         Rect;             { Rectangle containing Text }
  hTE:           TEHandle;         { handle to our edit record }
  ibeamCursor:   Handle;           { Handle to IBeam Cursor System Resource }
  VScroll:       ControlHandle;    { Vertical scrolling control }
  HScroll:       ControlHandle;    { Horizontal scrolling control }
  TheOrigin:     Point;            { Current Origin in the Window }

procedure ResizeTRect; forward;

segment procedure Initialize;
var
  drvrtype:      OsType;           { Used to pass parm to AddResMenu }
  i:             integer;          { a counter }
begin
  doneFlag:= false;

  { initialize menu manager }
  InitMenus;

  { pick up handles to menu resources }
  mymenus [1]:= GetMenu(appleMenu);
  mymenus [2]:= GetMenu(filemenu);
  mymenus [3]:= GetMenu(editmenu);

  { pick up driver names of desk accessories }
  drvrtype.c:= 'DRVR';
  AddResMenu(mymenus[1],drvrtype.p);

  { insert menus into menu list }
  for i:= 1 to lastmenu do
    InsertMenu(mymenus[i],0);

  DrawMenuBar;
  SetCursor(Arrow);
  SetRect(locate(dragRect),4,24,508,338);
  SetRect(locate(growRect),100,60,512,302);
  theWindow:= GetNewWindow(wndwID, locate(wRecord), -1);
  SetPort(theWindow);

  { set text edit window size }
  ReSizeTRect;

  { set window text font }
  wRecord.port.txFont:= 2;

  { Allocate the Edit Record }
  hTE:= TENew(locate(tRect),locate(tRect));

  { get I-beam cursor resource }
```

```
  IbeamCursor:= GetCursor(ibeamID);

  { establish scrolling controls }
  vScroll:= GetNewControl(vScrollID, theWindow);
  hScroll:= GetNewControl(hScrollID, theWindow);
  theOrigin.h:= 0;
  theOrigin.v:= 0;
end   {Initialize};

procedure ReSizeTRect;
{ Resets the bounds of the non-control portion of the window. }
begin {ReSizeTRect}
  TRect:= wRecord.Port.PortRect;
  with TRect do
    begin
      left:= left + 4; right:= right - 15;
      bottom:= bottom - 15;
    end;
end;

procedure CursorAdjust;
{ Makes the cursor an I-beam if the mouse is inside the application's
  content portion and an arrow otherwise. }
var
  mousePt:      Point;          { Current Mouse Location }
begin
  GetMouse(locate(mousePt));
  if theWindow = FrontWindow
    then
      if FrMacBool(PtInRect(mousePt.param, locate(TRect)))
        then SetCursor(DeRefHnd(iBeamCursor))
        else SetCursor(Arrow);
end;

procedure GrowWnd(where: Point);
var
  hw: LongInt;
  height, width:integer;
  cRect: Rect;              { Rectangle used for movement calcs }
  dummy: Record             { Dummy Record for updating Textedit record }
    destRect:Rect;
    viewRect:Rect;
  end;
begin

  { Grow the entire window }
  hw:= GrowWindow(theWindow, where.param, locate(growRect));
  height:= HiWord(hw); width:= LoWord(hw);

  { remove scroll bars from update region }
  cRect:= wRecord.Port.PortRect;
  cRect.left:= cRect.right - 16;
  InvalRect(locate(cRect));
  cRect:= wRecord.Port.PortRect;
  cRect.top:= cRect.bottom - 16;
  InvalRect(locate(cRect));

  { now draw the window }
  SizeWindow(theWindow,width,height,MacTrue);

  { move the scroll bars }
  With wRecord.port.PortRect do
    begin
      HideControl(vScroll);
      MoveControl(vScroll,right-15,top-1);
      SizeControl(vScroll,16,bottom-top-13);
      ShowControl(vScroll);
      HideControl(hScroll);
      MoveControl(hScroll,left-1,bottom-15);
      SizeControl(hScroll,right-left-13,16);
      ShowControl(hScroll);
    end;
```

```
  { adjust text edit rectangle }
  ResizeTRect;
  abs_move(derefhnd(hTE),locate(dummy),sizeof(dummy));
  dummy.viewrect:= TRect;
  abs_move(locate(dummy),derefhnd(hTE),sizeof(dummy));

  { add scroll bars to update region }
  cRect:= wRecord.Port.PortRect;
  cRect.left:= cRect.right - 16;
  InvalRect(locate(cRect));
  cRect:= wRecord.Port.PortRect;
  cRect.top:= cRect.bottom - 16;
  InvalRect(locate(cRect));
end; {GrowWnd}

procedure DrawWindow;
{ Erase the current contents of theWindow and redraw it. }
begin
  ClipRect(locate(wRecord.port.portrect));
  EraseRect(locate(wRecord.port.portrect));
  DrawGrowIcon(theWindow);
  DrawControls(theWindow);
  TEUpdate(locate(TRect),hTE);
end;

procedure ScrollBits;
var
  OldOrigin: Point;
  dh, dv: integer;
begin
  with wRecord do
    begin
      oldOrigin:= TheOrigin;
      TheOrigin.h:= 4*GetCtlValue(hScroll);
      TheOrigin.v:= 4*GetCtlValue(vScroll);
      dh:= oldOrigin.h - theOrigin.h;
      dv:= oldOrigin.v - theOrigin.v;
      TEScroll(dh,dv,hTE);
    end;
end   {ScrollBits};

procedure ScrollUp(theControl: ControlHandle; theCode: integer);
begin
  if theCode = inUpButton
    then
      begin
        SetCtlValue(theControl, GetCtlValue(theControl)-1);
        ScrollBits;
      end;
end;

procedure ScrollDown(theControl: ControlHandle; theCode: integer);
begin
  if theCode = inDownButton
    then
      begin
        SetCtlValue(theControl, GetCtlValue(theControl)+1);
        ScrollBits;
    end;
end;

procedure PageScroll(code: integer; theControl: ControlHandle;
                     amount: integer);
var
  pt:   Point;
begin
  repeat
    GetMouse(locate(pt));
    if TestControl(theControl,pt.param) = code
      then
        begin
          SetCtlValue(theControl,GetCtlValue(theControl)+amount);
          ScrollBits;
```

```
        end;
  until not FrMacBool(StillDown);
end;

procedure DoCommand(menu_command: LongInt);
{ Execute a command from the menu bar. }
var
  theMenu:      integer;        { the menu selected }
  theItem:      integer;        { the item in themenu }
  name:         String[255];    { Name of the desk accessory selected }
  refNum:       integer;        { Reference number of the desk accessory }
  ticks:        LongInt;
begin
  theMenu:= HiWord(menu_command);
  theItem:= LoWord(menu_command);
  case theMenu of

    applemenu:
      begin
        { open Desk Accessory with item's name }
        GetItem(myMenus[1],theItem,locate(name));
        refNum:= OpenDeskAcc(locate(name));
      end;

    filemenu: doneFlag:= true;

    editmenu:
      { process edit command if not System's }
      if not FrMacBool(SystemEdit(theItem-1)) then
        begin
          { Delay is used to keep menu lit }
          Delay(30, ticks);
          Case theItem of
            3: TECut(hTE);
            4: TECopy(hTE);
            5: TEPaste(hTE);
          end;
        end;

    end; {case}

  { unhilite the menu selected }
  HiliteMenu(0);
end; {DoCommand}

procedure DoMouseAction(MouseEvent: EventRecord);
var
  code:         integer;        { where mouse was pressed }
  whichWindow:  WindowPtr;      { Window where mouse was pressed }
  mycontrol:    integer;        { Part of control where mouse was pressed }
  whichControl: ControlHandle;  { Control where mouse was pressed }
  tc:           integer;        { Code returned by TrackControl }
begin
  code:= FindWindow(MouseEvent.where.param,locate(whichwindow));
  case code of

    inMenuBar:
      DoCommand(MenuSelect(MouseEvent.where.param));

    inSysWindow:
      SystemClick(locate(MouseEvent),whichWindow);

    inDrag:
      DragWindow(theWindow, MouseEvent.where.param, locate(dragRect));

    inGoAway:
      doneFlag:=
        FrMacBool(TrackGoAway(whichWindow,MouseEvent.where.param));

    inGrow:
      if theWindow = FrontWindow
        then GrowWnd(MouseEvent.where)
        else SelectWindow(theWindow);
```

```
    inContent:
      if theWindow <> FrontWindow
        then SelectWindow(theWindow)
        else
          begin
            GlobalToLocal(locate(MouseEvent.where));
            if FrMacBool(PtInRect(MouseEvent.where.param,locate(TRect)))
              then
                if BAnd(MouseEvent.modifiers,512) <> 0
                  then TEClick(MouseEvent.where.param,MacTrue,hTE)
                  else TEClick(MouseEvent.where.param,MacFalse,hTE)
              else
                begin
                  mycontrol:=
                    FindControl(MouseEvent.where.param,theWindow,
                                locate(whichcontrol));

                  Case mycontrol of
                    inUpButton:
                      tc:=
                       TrackControl(whichControl, MouseEvent.where.param,
                                    locate(ScrollUp,1));
                    inDownButton:
                      tc:=
                       TrackControl(whichControl, MouseEvent.where.param,
                                    locate(ScrollDown,1));

                    inPageUp:
                      PageScroll(mycontrol, whichcontrol, -10);

                    inPageDown:
                      PageScroll(mycontrol, whichcontrol, 10);
                    inThumb:
                      begin
                        tc:=
                         TrackControl(whichControl, MouseEvent.where.param,
                                      Abs_Nil);
                        ScrollBits;
                      end;
                    end; {case}
                end;

          end;

      end; {case}
nd; {DoMouseAction}

rocedure CheckEvents;
  Handle one event from the event queue. }
ar
  myevent:       EventRecord;
  theChar:       Char;
  saveport:      GrafPtr;
egin
  if FrMacBool(GetNextEvent(everyevent, locate(myevent))) then
  case myevent.what of

    mousedown: DoMouseAction(myEvent);

    keydown, autokey:
      if theWindow = FrontWindow then
        begin
          theChar:= Chr(myEvent.message mod 256);
          if BAnd(myEvent.modifiers,256) <> 0
            then DoCommand(MenuKey(theChar))
            else TEKey(theChar,hTE);
        end;

    activateEvt:
      begin
        DrawGrowIcon(theWindow);
        if Band(myevent.modifiers,1) = 1
```

```
            then
              begin
                SetPort(theWindow);
                TEActivate(hTE);
                ShowControl(vScroll);
                ShowControl(hScroll);
              end
            else
              begin
                TEDeactivate(hTE);
                HideControl(vScroll);
                HideControl(hScroll);
              end;
        end;

      updateEvt:
        begin
          GetPort(locate(saveport));
          SetPort(theWindow);
          BeginUpdate(theWindow);
          DrawWindow;
          EndUpdate(theWindow);
          SetPort(saveport);
        end;

      end;
end     {CheckEvents};

begin {Grow}
  Initialize;
  repeat
    CursorAdjust;          { adjust cursor shape to location }
    SystemTask;            { allow desk accessories to run }
    TEIdle(hTE);           { blink insertion point }
    CheckEvents;           { check for events }
  until DoneFlag;
end.
```

RMAKER Input for the GROW Program

The following text defines the resources used by the GROW program. The first two lines define the output resource file and the file type/Creator. The INCLUDE statement pulls in the resources that are required for all UCSD Pascal programs. The rest of the text defines resources that are specific to the GROW program.

```
GROW.RSRC
APPLPROG

INCLUDE UCSD Pascal 1:Empty Program

TYPE MENU

  ,1
 \14

  ,1000
File
Quit
```

```
  ,1001
Edit
Undo/Z
(-
Cut/X
Copy/C
Paste/V

TYPE WIND
  ,1000
UCSD Pascal Sample
50 40 300 450
Visible GoAway
0
0

TYPE CNTL
  ,1000
vertical scroll bar
-1 395 236 411
Visible
16
0
0 50 0

TYPE CNTL
  ,1001
horizontal scroll bar
235 -1 251 396
Visible
16
0
0 50 0
```

6
RMAKER

This chapter describes RMaker, the utility program that is used to produce resource files for UCSD Pascal programs. The use of resources is described in *Inside Macintosh.* The sections of this chapter are organized as follows:

ABOUT RMAKER describes the function of the RMaker utility.

RMAKER INPUT FILES describes the structure of RMaker input files, including suggested file naming conventions.

DEFINED RESOURCE TYPES describes the syntax for predefined resource types. This section will tell you the syntax for defining menus, dialog boxes, alert boxes and other ToolBox resources.

CREATING YOUR OWN TYPES describes how you use the predefined type GNRL to create your own resource types.

USING RMAKER describes how to run the RMaker utility and how to create resource files for input to the UCSD Pascal compiler.

ABOUT RMAKER

RMaker is the resource compiler supplied with **The MacAdvantage: UCSD Pascal**. It is very similar to the RMaker program in the Lisa Workshop, but some changes have been made to the syntax. Be careful if you are converting resource files from one system to the other.

RMaker takes a text file as input, and produces a resource file. The text file contains an entry for each resource to be defined, as described in the section DEFINED RESOURCE TYPES. The input text file also specifies the location and type of the output resource file.

The output from RMaker can be used as an input to the UCSD Pascal compiler. The compiler will copy the resources from the resource file specified to the UCSD Pascal program's resource fork. You can also use RMaker to append new resources to the resource fork of an existing UCSD Pascal program.

RMAKER INPUT FILES

An RMaker input file is a text file, as created using the Editor. By convention, RMaker input files have the extension .R. If you follow this convention you will easily be able to tell which text files on your disk are resource text files.

RMaker ignores all comment lines and blank lines between resource definitions. It also ignores leading and embedded spaces (except in lines defined to be strings). Comment lines begin with an asterisk. To put comments at the end of other RMaker lines, precede the comment with two consecutive semicolons (;;).

Creating New Resource Files

The first non–blank and non–comment line of the input file specifies the name of the resource file to be created. The file should have the extension .RSRC. The line following the file name should either specify the file type and creator bytes for the Finder, or be blank. For example, the first two lines below designate the file NewResFile.Rsrc as the output file. The file is an application (type APPL) with a creator of PROG. The standard file type and creator for all UCSD Pascal programs is 'APPLPROG'. If you do not specify the type and creator, they default to 0 (a null string).

```
NewResFile.Rsrc
APPLPROG

* The following include statement will read in the
* resources that are required by all UCSD Pascal programs.

INCLUDE UCSD Pascal 1:Empty Program

* Program specific resources go here
```

The RMaker output file NewResFile.Rsrc, created by the above input file, can be used as input to the UCSD Pascal compiler.

Appending to an Existing Resource File

The other type of resource input file starts with an exclamation point, followed by the name of the existing resource file that you wish to change. For example

```
!MyProgram.Code          ;; must be followed by a blank line.

* New resource definitions go here
```

tells RMaker to add new resources to the UCSD Pascal program called MyProgram.Code.

WARNING: You may not follow a file name with a comment (the above example is illegal.)

Include Statements

The rest of the resource input text file consists of INCLUDE statements and TYPE statements.

INCLUDE statements are used to read in existing resource files. An INCLUDE statement looks like this:

```
NewResFile.Rsrc
APPLPROG

* The following include statement will read in
* the resources that are required by all UCSD Pascal
* programs.

INCLUDE UCSD Pascal 1:Empty Program

* Program specific resources go here
```

Typically you will use an INCLUDE statement to include the standard UCSD Pascal resources into a resource file that contains resources specific to your application program. Standard UCSD Pascal resources are in the file Empty Program on the disk **UCSD Pascal 1.**

Type Statements

TYPE statements consist of the word "TYPE" followed by the resource type and, below that, one or more resource definitions. The resource type must be capitalized to match a predefined resource type.

The following statement creates three resources of type 'STR '.

```
TYPE STR
  ,1
This is a string
  ,2
Another String
  ,3
Another string resource
```

It is not necessary for all resources of a given type to be declared together. However, all resources of a type must have unique resource ID's. If you specify a resource ID that is already in use, the new resource replaces the old one.

A resource definition looks like this:

```
[resource name] ,resource ID [(resource attribute byte)]
type-specific data
```

The square brackets indicate that the resource name and resource attribute bytes are optional. Don't place these brackets in your input file. The comma before the resource ID is mandatory. Attribute byte numbers are given in decimal. Attribute byte values are defined in the Resource Manager chapter of *Inside Macintosh*. The default attribute byte value is 0. Here are some sample resource definitions:

```
TYPE STR

NewStr ,4 (32)          ;; 32 means resource is purgeable
This resource has a name and an attribute byte!!

   ,5 (32)
This one has only an attribute byte.

MyNewStr,6
This one has only a name (the attribute byte is 0).
```

The type–specific data is different for each resource type. As you have probably guessed, the type specific data for a 'STR ' resource is simply a string. The next section describes the type specific data for the resource types defined by RMaker.

DEFINED RESOURCE TYPES

RMaker has 11 defined resource types: ALRT, BNDL, CNTL, DITL, DLOG, FREF, GNRL, MENU, STR , STR# and WIND. The format of the type–specific data for each type is shown by example, below. The type GNRL is used to define your own resource types. It is explained later.

Syntax of RMaker Lines

There are just a few general rules that apply to lines read by RMaker.

- Leading and embedded blanks are ignored, except when necessary to separate multiple numbers on a line, or when they are part of a string.
- Blank lines should not be placed inside a resource definition, unless required (the exceptions are pointed out below).
- Numbers are decimal, unless specified otherwise.
- RMaker is sensitive to line breaks. Thus if a type description shows four values on a single line, you must put four values on a single line.

Two special symbols can be used in resource definitions: the continuation symbol (++) and the enter ASCII symbol (\).

```
++    goes at the end of a line that is continued
      on the next line.

\     precedes two hexadecimal digits.  That ASCII
      character is entered into the resource
      definition.
```

Look at the description of the 'STR ' type for examples of these special symbols.

The use of most of the TYPEs listed below are described in the appropriate chapter in *Inside Macintosh.* For example, the use of the type DLOG is described in the Dialog Manager chapter of *Inside Macintosh.*

ALRT--Alert Resource

```
TYPE ALRT
   ,128                    ;; resource ID
50 50 250 250              ;; top left bottom right
1                          ;; resource ID of item list
7FFF                       ;; stages word in hexadecimal
```

BNDL--Application Bundle Resource

The BNDL resource is used to implement the Macintosh Finder interface to an application program. It allows the application to define its own desktop icons and associate documents with specific programs. The BNDL resource is discussed more fully in the section APPLICATION INTERFACE TO THE FINDER in the chapter GENERAL OPERATIONS.

```
TYPE BNDL
   ,128              ;; resource ID
MPNT 0               ;; bundle owner
ICN#                 ;; resource type
0 128 1 129          ;; ID 0 maps to resource ID 128, 1 to 129
FREF                 ;; resource type
0 128 1 129          ;; ID 0 maps to resource ID 128, 1 to 129
                     ;; Must be followed by a blank line.
```

NOTE: The number of mappings from local ID to resource ID is variable. Simply include multiple mappings on a single line.

NOTE: If the BNDL resource is present in an RMaker input file, the resulting output file will have its bundle bit set.

CNTL--Control Resource

```
TYPE CNTL
   ,130                    ;; resource ID
Stop                       ;; title
244 40 260 80              ;; top left bottom right
Invisible                  ;; see note
0                          ;; ProcID (control definition ID)
0                          ;; RefCon (reference value)
0 1 0                      ;; minimum maximum value
```

NOTE: Controls can be defined to be Visible or Invisible. Only the first character (V or I) is significant.

DITL--Dialog or Alert Item List Resource

```
TYPE DITL
   ,129                    ;; resource ID
5                          ;; 5 items in list

StaticText                 ;; static text item (see note)
20 20 32 100               ;; top left bottom right
Whoopie                    ;; message
                           ;; blank lines are optional here.
EditText                   ;; editable text item (see note)
20 120 32 200              ;; top left bottom right
Default message            ;; message

radioButton                ;; radio button item (see note)
40 40 60 150               ;; top left bottom right
Hello                      ;; message

CheckBox Disabled          ;; disabled item (see note)
75 40 95 150               ;; top left bottom right
GoodBye                    ;; message

Button                     ;; button item (see note)
75 160 95 200              ;; top left bottom right
Hi!                        ;; message
```

NOTE: Five types of dialog items are defined: Static text, Editable text, Radio Buttons, CheckBoxes, and Buttons. These items are assumed to be enabled. Otherwise you may specify Disabled. Only the first character of these item definition words are significant (S,E,R,C,B,D).

DLOG--Dialog Resource

```
TYPE DLOG
   ,3                      ;; resource ID
This is a dialog box.      ;; message
100 100 190 250            ;; top left bottom right
Visible GoAway             ;; box status (see note)
0                          ;; procID (dialog definition ID)
0                          ;; refCon (reference value)
200                        ;; resource ID of item list
```

NOTE: A dialog box can be Visible or Invisible. GoAway and NoGoAway determine whether or not the box can be closed. Only the first characters (V,I,G,N) are significant.

FREF——File Reference Resource

The FREF resource is used to associate file types with icons. Used in conjunction with the BNDL resource, the FREF resource allows applications to define their own desktop icons. For more information see the section APPLICATION INTERFACE TO THE FINDER in the chapter GENERAL OPERATIONS.

```
TYPE FREF
   ,128                    ;; resource ID
APPL 0                     ;; File type, local ID of icon
                           ;; Blank lines ok between resource
                           ;; definitions.
   ,129                    ;; resource ID
TEST 127 myFile            ;; File type, local ID of icon, file name
```

If there is no file name, it can be omitted.

MENU——Menu Resource

```
TYPE MENU
   ,3                      ;; resource ID
Transfer                   ;; menu title
Edit                       ;; item 1
Asm                        ;; item 2
Link                       ;; item 3
(-                         ;; item 4 (draw a line)
Exec                       ;; item 5
                           ;; MUST be followed by an empty line!!
```

WARNING: An empty line must follow a MENU resource definition. The line must not have comments (the example above is illegal) or spaces.

STR ——String (space required)

```
TYPE STR                   ;; 'STR ' (space required)
   ,1                      ;; resource ID
This is a string           ;; and a string

   ,23                     ;; resource ID
This is a string ++        ;; and a long string
that shows the ++
line continuation characters.

   ,25 (32)                ;; resource ID, attribute byte
I've got attributes!       ;; and a string

   ,27                     ;; resource ID
```

```
Testing, \31, \32, \33  ;; 'Testing, 1, 2, 3' the hard way
```

STR#--String List Resource

This resource type allows you define a number of strings using one resource identifier. The procedure GetIndString in the OsUtilities unit (listed in Appendix A) can be used to index into a string list.

```
TYPE STR#
   ,1                    ;; resource ID
4                        ;; number of strings
This is string one       ;; and the strings...
And string two
Third string
Bench warmer
```

WIND--Window Resource

```
TYPE WIND
   ,128
Wonder Window            ;; title
40 80 120 300            ;; top left bottom right
Invisible GoAway         ;; window status (see note)
0                        ;; ProcID (window definition ID)
0                        ;; RefCon (reference value)
```

NOTE: A Window can be Visible or Invisible; GoAway and NoGoAway determine whether or not the window has a close box. Only the first character of each option (V,I,G,N) is significant.

CREATING YOUR OWN TYPES

There are two ways to create your own resource types. The first is to equate a new type to an existing type. For example, you can create a resource of type ERRM like this:

```
TYPE ERRM = STR                ;; type ERRM is just like STR
   ,17 (32)                    ;; resource ID, attribute byte
Bad input file name            ;; the error message
```

In the example, we have defined type ERRM to be an STR type. This allows us to avoid resource identifier conflicts at runtime with other resources of type STR .

The other way to create your own type is to equate the new type to GNRL, and then to specify the precise format of the resource. A set of element type designators lets you define the type of each element that is to be placed in the resource.

Here are the element type designators:

```
.P      pascal string
.S      string without length byte

.I      decimal integer
.L      decimal long integer
.H      hexadecimal

.R      Read resource from file.  Followed by three
        parameters: file name type ID
```

For example, to define a resource of type CHRG consisting of the integer 57 followed by the Pascal string 'Finance charges', you could use the following type statement:

```
TYPE CHRG = GNRL               ;; define type CHRG
   ,200                        ;; resource ID
  .I                           ;; a decimal integer
57
  .P                           ;; a pascal string
Finance charges
                               ;; MUST be followed by a blank line.
```

A more practical example: An application that has its own icon must define an icon list, and reference it using FREF (described above). Such an icon list can be defined as follows:

```
TYPE ICN# = GNRL              ;; icon list for an application
   ,128                       ;; resource ID
  .H                          ;; enter 2 icons in hexadecimal
0001 0002 0003 0004           ;; each is 32 bits by 32 bits
   . . .
007D 007E 007F 0080           ;; for 128 words total
                              ;; MUST be followed by a blank line.
```

The .R type designator is used to include an existing resource as part of a new resource type. For example, to read an existing FONT resource into a new resource of type FONT, use the following resource definition:

```
TYPE FONT = GNRL              ;; define a new type
   ,268                       ;; resource ID
  .R System FONT 268          ;; read from the System file
                              ;; the FONT resource with ID=268
```

USING RMAKER

Once you have created the input file to RMaker, the hard work is done. Simply select and open the utility RMaker. The standard file selection window is automatically opened. Select the file you want to compile, and off it goes.

By default, the standard file selection window displays all the text files on the disk. If you want to display only the .R files, Cancel the selection window, select .R Filter from the File menu, then select Compile from the File menu to redisplay the file selection window.

When RMaker is compiling a file, the name of the source file is displayed in the upper left of the window, and the name of the output file is displayed in the upper right. As the file is compiled, the current size of the resource data, the size of the resource map, and the total size are tracked on the right half of the screen. In addition, as each line is compiled, it is displayed on the screen. When RMaker is finished, the Quit button in the lower left hand corner of the window will blink.

If there are no errors in the RMaker input file, a resource file with the specified name is created.

WARNING: The TRANSFER menu is not supported. Trying to transfer out of RMaker could cause unpredictable results.

UCSD Pascal Compiler Input

Most of the time you will want to generate resource files that can be used as input to the UCSD Pascal compiler. UCSD Pascal programs require a minimum set of resources. These resources are in the file Empty Program on the **UCSD Pascal 1** disk. A typical application resource text file would be:

```
program.rsrc                ;; Output file name
APPLPROG                    ;; Type , Creator

INCLUDE UCSD Pascal 1:Empty Program

* Your program's resource TYPEs go here.
```

Note the use of the volume prefix on the file Empty Program. The volume prefix is not needed if Empty Program is on the same disk as RMaker (the default volume). Volume prefixes must follow Macintosh file naming conventions, as defined in the chapter GENERAL OPERATIONS.

Errors in the Input File

If an error occurs, the line containing the error is the last line on the screen. RMaker then displays a box with an error message in it. These are the possible error messages. A brief description accompanies the error messages that are not self–explanatory.

- **An Input/Output error has occured.**
- **Can't open the output file.**
- **Can't create the output file.**
- **Syntax error in source file.**
- **Bad type or item declaration.**
- **Bad ID Number.**
- **Bad Attributes Parameter.**
- **Can't load INCLUDE file.**
- **Bad format resource designator in GNRL type.** This is any error in a user—defined resource type.
- **Out of memory.**
- **Can't add to the file —— disk protected or full.**
- **Bad bundle definition.**
- **Unknown type.** Specified resource type is undefined.
- **Bad Object definition.** This can happen if the specified file is of the wrong type.
- **Bad item type.**
- **Bad format number.**

7
LIBRARIAN

The Librarian is a utility program that allows you to manipulate code segments within library files. Libraries are a useful means of grouping the separate code pieces needed by a program or group of programs. Libraries generally contain routines relating to a certain area of application; they can be used for functional groupings much as units can. Thus, you might want to maintain a math library, a data file–management library, and so forth——each of these libraries containing routines general enough to be used by many programs over a long period of time.

Maintaining units in well organized libraries is more convenient than maintaining a larger number of separate files. It allows you

- to manipulate an entire collection of units easily.
- to reduce the number of library files you must specify in the Library Files list for a program (see the Set Options utility).
- to reduce the number of files that are open when the program is executing (each library file will be an open file).
- to think of your application in a more organized way.

Individual programs may also take advantage of the library construct. If a program uses several units suitable for compiling separately, but the units logically belong within the program, you may want to construct a single library containing the program and all of those units.

Library files created by the Librarian have the same structure as code files created by the compiler. Thus, a library file which contains a single unit is equivalent to the code file produced by

the compiler for that unit.

NOTE: The Librarian is useful for determining what units and segments are in the Pascal Runtime and Debug Runtime library files. However, it cannot be used to change these files. Changing them will make them unusable.

This chapter uses the term compilation unit to refer to a program or unit and all the segments declared inside it. The segment for the program or unit is called the host segment of the compilation unit. Segment routines declared inside the host are called subsidiary segments. Information in the host segment called segment references referes to units used by the compilation unit. The segment references contain the names of all segments referenced by a compilation unit. When a program is executed, the runtime system searches all the library files specified in the program's Library Files list to find the referenced segments.

Some routines called from hosts exist in units in the Runtime Support Library and therefore appear in segment references, even though there is no explicit **uses** declaration for them. For example, writeln resides in the Runtime Support Library **unit** PASCALIO, so the name PASCALIO appears in the segment references of any host that calls writeln.

USING THE LIBRARIAN

When the Librarian is executed, it first asks you for the name of an output file. This can be any legal Macintosh file name including a volume prefix. The UCSD Pascal Compiler appends .CODE to the end of every code file name to avoid confusing code files with source files. We recommend that you use the same convention when creating library files. The Librarian removes an old file with the same name as the output file.

The Librarian then asks you for the name of an input file. If the name you enter cannot be found, the Librarian appends .CODE to the end of the name and looks again. If you do not want to specify an input file at this time, press <Return> in place of a

file name.

Here is a screen display from the middle of a run of Librarian. Pascal Runtime has been specified as the input file, and three segments have been copied to the output file:

```
Librarian: N(ew, 0-9(slot-to-slot, E(very, S(elect, ?

Input file? PASCAL RUNTIME
  0 u KERNEL     2015      10 u REALOPS    2092
  1 s USERPROG   1460      11 u LONGOPS    1364
  2 u CONCURRE    431      12 u ASSOCIAT    207
  3 u FILEOPS     854      13 s CREATEEN    877
  4 u EXTRAIO     221      14 s PEDBUILD   1455
  5 u PASCALIO    994
  6 u HEAPOPS     234
  7 u EXTRAHEA    786
  8 u STRINGOP    234
  9 u OSUTIL      340
Output file? NEW.CODE
Output file is 20 blocks long.
  0 u KERNEL     2015
  1 s USERPROG   1460
  2 u CONCURRE    431
```

The screen display consists of the prompt line, the question line, the input display, the output file line, the output file size line, and the output display. The input and output displays each show a list of code segments entries. Each entry consists of the slot number, the segment code, the segment name, and the segment length (in words). The segment codes are as follows: "p" refers to a main program unit, "u" refers to a unit, and "s" refers to a subsidiary segment. Some Librarian commands have you specify a segment by its slot number.

To build a library, you copy segments from various input files to the output file. Normally, the Librarian will not allow you to transfer more than one segment with the same name to the output file. However, the 0—9(slot—to—slot command allows you to override this restriction.

LIBRARIAN COMMANDS

This section describes the Librarian commands.

- The **N(ew** command displays a prompt asking for a new input file. This file becomes the file from which segments may be copied. The segments contained in the input file are displayed in the input display.

- The **A(bort** command stops the Librarian without saving the output file.

- The **Q(uit** command stops the Librarian and saves the output file. Just prior to terminating, the Librarian asks you to enter a copyright notice. The Q(uit command also copies resources to the resource fork of the output file.

  When the Librarian displays the prompt "Notice?" at the top of the screen, you should enter a copyright notice and press <Return>. The copyright notice is placed in the segment dictionary of the output file. Pressing <Return> without entering a copyright notice exits the Librarian without writing a copyright notice.

  After the copyright notice is processed, the Librarian copies resources to the resource fork of the output file. The source for the resources is determined as follows:

  If you have transferred one or more segments of type program to the output file, the Librarian attempts to copy the resources for the file in which the last such segment was located.

  If, instead, you have transferred only segments of type unit (or subsidiary segments), the Librarian attempts to copy the resources from the last input file that was specified.

  If you enter Q(uit prior to copying any segments to the output file, the Librarian attempts to copy the resources from the file "Empty Program" to the output file.

If the Librarian can't find a source from which to copy the resources or the copy was unsuccessful, an error message will be displayed. If you are trying to create a program that you intend to execute and this error occurrs, you will have to rebuild the program, making sure a source for the resources is available. If you are just building a library file, the occurrance of this error is not critical, since the library file should still be useable.

- The **T(og** command toggles a switch that determines whether or not **interface** sections of units are copied to the output file. The **interface** sections are required if you reference the library file in a **uses** statement while compiling a program. Since the **interface** sections make your library file bigger, you should exclude them from the library file when development of you application is complete (i.e., no more compilations will be done using it) to save disk space.

- The **R(efs** command lists the names of each entry in the segment reference lists of all segments currently in the output file. The list of names also includes the names of all compilation units currently in the output file, even though their names may not occur in any of the segment references. To refresh the output display, press <Space>.

- The **I(nput** command "scrolls" the display of segments in the input display if there are more segments than will fit on the screen. Type I(nput multiple times to cycle through the input display.

- The **O(utput** "scrolls" the display of segments in the output display if there are more segments that will fit on the screen. Type O(utput multiple times to cycle through the output display.

The remaining five commands transfer code segments from the input file to the output file.

- The **0—9(slot—to—slot** command transfers a segment the segment from a specified slot in the intput file to a specified slot in the output file. When you enter the first digit, Librarian displays the prompt: "From slot # ?". Terminate the entry with <Space>. Librarian displays the prompt "To slot #?". Enter the number of the slot that the segment is to be copied to in the output file. To abort the command, press <Return> with an empty output slot number.

  NOTE: You may not use the <Backspace> key to correct typing errors. To abort the command after specifying a input slot, press <Return> in response to the second question. You cannot abort the command after the specifying the output slot.

  This command will allow you to copy a segment with a duplicate name into the output file.

- The **E(very** command copies all of the code segments in the input file to the output file. Each segment is copied to the first available output file slot, provided that its name does not conflict with the name of a segment already in the output file.

- The **S(elect** command causes the Librarian to loop through each segment in the input file, asking you whether you would like to have it transferred to the output file. For each code segment not already in the output file, the Librarian asks: "Copy from slot #?". Press Y to copy the segment. Press N to skip the segment. Press E to copy the rest of the code segments in the input file (as in the E(very command). Press <Space> or <Return> to abort the S(elect command. Each segment is copied to the first available slot.

- The **C(omp—unit** command causes the Librarian to ask: "Copy what compilation unit?". You should enter the name of a compilation unit. The compilation unit named is transferred to the output file, along with any segment procedures that it contains.

- The **F(ill** command does the equivalent of a C(omp—unit command for all the compilation units referenced by the segment references in the output file.

8
DEBUGGER

This chapter describes the Symbolic Debugger and the Performance Monitor. The Debugger is a tool for detecting and correcting errors in programs that you develop. The Performance Monitor is a mechanism for gathering performance information and for extending the capabilities of the Debugger.

The Debugger gives you the following program diagnostic capabilities:

- setting and removing breakpoints.
- single stepping p–code.
- displaying and altering memory and p–Machine registers.
- disassembling p–code.

To use the Debugger effectively, you must be familiar with the p–Machine architecture and understand the p–code operators, stack usage, and variable and parameter allocation. These topics are discussed in the P–MACHINE ARCHITECTURE chapter. Other useful information will be found in *The UCSD Pascal Handbook* and the MACINTOSH INTERFACE chapter.

A compiled listing of your program is helpful when using the Debugger. The listing helps you to determine p–code offsets and variable offsets.

WARNING: The Debugger is a low–level tool, and as such, you must use it with caution. If you use the Debugger incorrectly, your program can fail.

GENERAL INFORMATION

This section discusses general information about using the Debugger. The individual Debugger commands are covered in the next section, DEBUGGER COMMANDS.

Installation

To use the Debugger, you must be using the Debug Runtime file as your Runtime Support Library, and you must have the Startup in Debugger option enabled in your program's startup options. Both of these may be configured by using the Set Options utility, as described in the GENERAL OPERATIONS chapter.

Set Options also allows you to select whether the Debugger will communicate via an external terminal, connected to the modem port, or the .DBGTERM device (the lower eight lines of the Macintosh Screen). If the Create .DBGTERM Device and Create Default Window options are simultaneously enabled, the screen I/O window will appear smaller on the screen in order not to overlap the .DBGTERM screen region.

If you have properly installed the Debugger, when you start your program it will immediately enter the Debugger, displaying the following prompt:

```
UCSD Pascal Debugger [1R0.0]
(
```

Command Format

The Debugger prompts you for input by printing a left parenthesis character '('. There are no menus explaining the Debugger commands because they would detract from the information displayed on the screen by the Debugger. However, when you enter a command, the Debugger may display several short prompts that ask you for information.

Many of the Debugger commands require you to enter two characters (such as 'LP' for List P–code, or 'LR' for List Register). To abort a command after entering the first character, press <Space>.

Here is a sample of a debugging session:

```
UCSD Pascal Debugger [1R0.0]
(BS) Set break#? 0  Segname? EXAMPLE   Procname or #? 1  Offset#? 0
(BL)
(b0) S=EXAMPLE P#1 O#0
(R )
Hit break#0 at  S=EXAMPLE  P#1  O#0
(VG) Varname or offset#? 1
(g ) S=EXAMPLE P#1 O#1 000012E6:01 04 01 56 01 A8 00 00 ---V----
(
```

Most lines of Debugger interaction are prefaced with a command code or response code surrounded by parentheses. If the code is in upper–case letters, it is a command that you entered. If the code is in lower–case letters, it is a Debugger response line. There is a table of the possible Debugger response codes and their meanings in the Summary of Response Codes section.

When the Debugger prompts you with questions, you type the response and terminate it by pressing <Space> or <Return>. When you are asked for the name of a segment or an identifier, you may type only eight characters of the name. Most numeric input is in decimal radix (base 10). However, when you are requested for an address the Debugger expects hexadecimal notation to be used.

If you make a mistake when typing a response to a question, you may use the <Backspace> key to fix the entry. However, you cannot edit a response after you have gone on to the next question. Terminate the command and reenter it.

Some commands display more information than will fit on the display device. If so, the Debugger will print out a "screen full" of output, then ask you to type <Space> to continue. If you would like to terminate the output, press any other character.

Entering and Exiting

There are several ways to enter the Debugger. (You can tell that you are in the Debugger by the presence of the left parentheses prompt.) When the Debugger is enabled, the Runtime Support Library will enter the Debugger in the following situations:

- upon starting your program.
- upon execution of the Debugger procedure in the Error_Handling unit.
- upon encountering a break point.
- upon completing a single step operation.
- upon detecting an execution error.
- upon execution of the halt intrinsic.
- upon recognition that the break button has been pressed.
- upon recognition that the Debug "button" in an execution error dialog box has been pressed.

You exit the Debugger by executing one of the following commands:

- The Quit command puts the debugger in a dormant state.
- The Resume command continues program execution from where it left off.
- The Step and Trace commands execute a single p—code, then automatically reenter the Debugger.

If any display options are enabled, then the Debugger will print the enabled options just after reentering the Debugger. See Configuring the Display for details.

Debugger State

When the Debugger is reentered after a Resume, Step or Trace command, it remembers its previous state, including the condition of the break points, its memory locked state, and its display modes. However, if the Debugger is entered after it has been made dormant by the Quit command, it starts up in a "fresh" state, with no break points set.

Two other features of the debugger state are its *current activation record* and its *current address.*

The current activation record determines the environment for displaying variables. On reentry to the Debugger, it corresponds to the most recent activation record. However, it can be changed for a series of commands by using the Chain Down and Chain Up commands.

The current address corresponds to the last address that was displayed by a memory examine command. The slash (/) and back slash (\) commands alter memory at the current address, and the plus (+) and minus (—) commands display memory in the vicinity of the current address.

Symbolic Debugging

The Debugger becomes a Symbolic Debugger with a little cooperation from the UCSD Pascal compiler. If you compile your program with the symbolic debugging compiler options around portions of your program (see the PASCAL LANGUAGE chapter), you can then access variables by name rather than by data offset, and you can access code by line number rather than by p–code offset. Also, break points may be specified by procedure name and line number, and disassembled code will display procedure names rather than numbers.

Having a current compiled listing of your program is still essential for serious debugging efforts.

To use symbolic debugging, it is necessary that the code being debugged is compiled with $D+ compiler options. The $D+ option instructs the compiler to output symbolic Debugger information for those portions of a program that are compiled with $D+ turned on.

Once you have debugged your program, you should recompile it without the symbolic debugging flags, because the symbolic debugging information increases the size of your code file.

When you use symbolic debugging, you may specify locations in your code by line number. The line number corresponds to the line number in a compiled listing of your program. Of course, you can also specify locations in your code by offset. When you specify variable or procedure names symbolically, you may only type the first eight characters of the symbol's name.

The example debugging sessions in the rest of this chapter are a mixture of symbolic and non–symbolic debugging examples. When you are running the Debugger, the Debugger will make it clear to you what the permissible command options are by the content of its questions. Each question is explicit about what type of response it expects.

DEBUGGER COMMANDS

The following sections describe each of the Debugger commands.

Resuming Execution

You resume program execution by using one of the following commands:

Q The Quit command puts the Debugger in a dormant state, disabling all break points, and continues program execution.

R The Resume command continues program execution from where it left off.

Using Break Points

The Debugger allows you to maintain up to five break points within a program at one time. A break point is a location within p—code that will cause the Debugger to be entered when the p—code is about to be executed.

You specify a break point by its break point number. Break points are numbered 0 through 4. The location of the break point is specified by the segment name, procedure number and code offset. If symbolic debugging is enabled, you may specify the procedure by name and the location within the procedure by line number. A compiled listing of your program is indispensible for specifying breakpoints in code, because both code offsets and line numbers are printed in the listing. Line numbers can be determined "on the fly" by using the File command to examine a compiled listing that is stored in a file on disk.

The following commands manipulate break points:

BS The Breakpoint Set command enables one break point. You are asked to specify (1) the number of the breakpoint to set, (2) the segment name, (3) the

procedure name or number, and (4) the code offset or line number.

BR The Breakpoint Remove command disables one break point. You are asked to specify the number of the break point to remove.

BL The Breakpoint List command lists the break points that are currently in effect.

Here is an example of using the break point commands:

```
(BS) Set break #? 0 Segname? EXAMPLE Procname or #? 1 Offset#? 0
(BL)
(b0) S=EXAMPLE  P#1         O#0
(BS) Set break #? 1 Segname? EXAMPLE Procname or #? 2 Offset#? 25
(BR) Remove break#? 0
(BL)
(b1) S=EXAMPLE  P#2         O#25
```

When you have resumed execution with the Resume command and a break point is encountered, the Debugger is reentered and the following message will appear on the screen:

```
Hit break#0 at  S=EXAMPLE  P#2         O#25
(
```

This message means that breakpoint 0 was encountered in segment EXAMPLE, procedure number 0, code offset 25.

Single Stepping

The Debugger allows you to execute the p—code in your program a single instruction at a time by using the single step commands. Single stepping is most effective when used in conjunction with the enable command, which allows you to set the display options. (See Configuring the Display.)

Using one of the single step commands causes one or more p—codes to be executed. Nothing is left on the screen to indicate that a single step operation has been performed. However, you

may notice that the left parenthesis prompt disappears for a moment, the reappears. Most often you will want to run single stepping with the P–code option enabled so you can tell where you are in the program you are debugging. This option will cause each p–code to be printed on the screen before it is executed.

Here are the single step commands:

S This command causes a single p–code to execute. Execution of a procedure call instruction will cause the Debugger to "step into" the procedure.

T This command causes a single p–code to execute unless the p–code is a call instruction, whereupon the Debugger will execute the entire call and stop on the p–code following the call. This command "steps over" procedure calls, i.e., it allows you to single step through a procedure without worrying about what goes on within the procedures that it calls.

Here is an example of single stepping with the P–code display option enabled. Note that the Trace command has been used to "step over" procedure calls. The format of the p–code disassembly is discussed, below, in Disassembling P–code.

```
(EP)
(cd) S=EXAMPLE  P#1        0#1        SRO      3
(cd) S=EXAMPLE  P#1        0#3        SLDC     5
(cd) S=EXAMPLE  P#1        0#4        SRO      2
(cd) S=EXAMPLE  P#1        0#6        CPG      2
(cd) S=EXAMPLE  P#1        0#8        RPU      0
```

Disassembling P–code

The P–code command allows you to look at disassembled p–code for portions of your program.

P This command disassembles a section of p–code. You are asked to specify (1) a segment name, (2) the procedure name or offset, (3) the start offset or line number, and (4) the end offset or line number.

Here is an example disassembly:

```
(P )   Segname? EXAMPLE   Procname or #? SETCIFD
       First# 10 Last# 12 Start Line#? 10  End Line#? 12
(cd) S=EXAMPLE  P#2          0#0            SLDO    3
(cd) S=EXAMPLE  P#2          0#1            SLDO    2
(cd) S=EXAMPLE  P#2          0#2            LEQI
(cd) S=EXAMPLE  P#2          0#3            BNOT
(cd) S=EXAMPLE  P#2          0#4            SSTL    1
(cd) S=EXAMPLE  P#2          0#5            SLDL    1
(cd) S=EXAMPLE  P#2          0#6            FJP     6
(cd) S=EXAMPLE  P#2          0#8            SLDO    3
(cd) S=EXAMPLE  P#2          0#9            SLDO    2
(cd) S=EXAMPLE  P#2          0#10           MPI
(cd) S=EXAMPLE  P#2          0#11           SRO     1
(cd) S=EXAMPLE  P#2          0#13           RPU     1
```

Each p–code instruction is presented, along with the "coordinates" of the instruction (i.e., the segment, procedure and offset). The p–code names correspond to the instructions described in the P–MACHINE ARCHITECTURE chapter. P–code operands are in decimal radix (base 10).

Disassembling p–code is useful in analyzing the exact cause of certain runtime errors, and discovering the exact p–code coordinates for a break point. If you press <Return> to the line number prompts, the Debugger assumes the first and last line numbers for the procedure you have indicated. (Similarly, when not using symbolic debugging, pressing <Return> in response to the start and stop offset prompts causes the Debugger to assume a starting offset of zero and an ending offset equal to the offset of the last p–code instruction in the procedure.)

Examining and Modifying Memory

The commands described in this section allow you to examine and modify memory. The Address Data and Address Code commands ask you to specify a memory address in hexadecimal (base 16) notation.

For the Address Data command you must specify a 16–bit address in the range 0000–FFFF. This address is interpreted as an address within the Pascal Data Area. If you enter the value of any Pascal pointer variable, the Debugger will display the memory that it points to.

For the Address Code command you must specify a 32–bit absolute address in the range of legal memory address for your Macintosh. This address is interpreted as a 68000 absolute address. Note that the 16–bit addresses mentioned above do not correspond to the same absolute address. If you enter the value of a Macintosh absolute pointer, the Debugger will display the memory that it points to.

For more information on the relationship between Pascal pointers and absolute addresses, refer to the MACINTOSH INTERFACE and PASCAL LANGUAGE chapters.

A memory examine command displays memory to the screen in the following format:

```
(AD) Data address? 500
(a )              00000500:2D F8 2D FD 2E 03 2E F6 ----.-.-
```

The address is followed by two representations of the eight bytes stored at that address. First they are displayed in hexidecimal, then in ASCII. If the byte does not have a printable ASCII representation, it is represented as a dash (–).

After you have entered a memory examine command, you can examine words in the immediate vicinity by using the plus (+) and minus (–) commands. The slash (/) and back slash (\) commands allow you to alter the memory displayed by the previous memory examine command.

Here are the memory examine and modify commands:

AD The Address Data command displays eight bytes starting at the specified address, and sets the current address to this address. The address is a 16–bit pointer within the Pascal Data Area.

AC The Address Code command displays eight bytes starting at the specified address, and sets the current address to this address. The address is a 32–bit absolute address.

+ The plus command increments the current address by eight bytes, and displays the eight bytes at the current address.

– The minus command decrements the current address by eight bytes, and displays the eight bytes at the current address.

/ The slash command allows you to alter in hexadecimal the memory at the current address. You alter the bytes by typing two hexadecimal characters for each byte and using <Space> or <Return> to skip to the next byte.

\ The back slash command allows you to alter in ASCII the memory at the current address. You alter the bytes by typing a character for each byte. If you wish to skip a byte, press <Return>.

Here is an example of displaying and modifying memory contents:

```
(AD) Data address? 1000
(a )              00001000:52 42 67 02 76 09 E3 4B RBg-v--K
(a )              00001008:70 63 B6 40 6F 08 96 40 pc-•o--•
(/ )                       B7 FF
(\ )                                        abdefghi
```

Examining and Altering Variables

The following commands allow you to examine the areas of memory where variables are stored. The display is not formatted based on the type of the variables. Instead, you must interpret the variables in hexadecimal or ASCII form.

These commands may be used in conjunction with the slash (/) and back slash (\) commands to alter the value of variables. They may also be used in conjunction with the plus (+) and minus (–) commands to display memory in the vicinity of the initial memory examine command.

To use the variable examine commands you must be somewhat familiar with the storage of variables in the p–Machine. See P–MACHINE ARCHITECTURE for details.

- Local variables refer to the variables declared in the procedure that corresponds to the current activation record.

- Global variables refer to the variables declared at the outermost level of the program or unit that contains the procedure that corresponds to the current activation record.

- Intermediate variables refer to variables declared in procedures that are nested lexically between the global level and the procedure that corresponds to the current activation record.

- External variables refer to variables stored at the global level of a unit accessible to the procedure that corresponds to the current activation record.

The current activation record is normally the activation record that the p–code instruction that is about to be executed is located within. You may change the current activation record by changing the frame of reference with the Chain Up or Chain Down commands. See Changing the Frame of Reference.

Each of these commands asks for a variable by name or by offset. The offset is a word offset of the variable within an activation record. You can determine the offset of a variable by using a compiled listing. Variable offsets are numbered starting with 1. If you specify an offset that is out of range, the Debugger gives you an error message and aborts the command.

Here are the variable examine commands:

VL The Var Local command displays eight bytes of the local variables of the current activation record. You are asked to specify either a variable name or a variable offset. This command sets the current address.

VG The Var Global command displays eight bytes of the

global variables in the context of the current activation record. You are asked to specify either a variable name or a variable offset. This command sets the current address.

VI The Var Intermediate command displays eight bytes of the variables for an activation record at an intermediate lexical level with respect to the current activation record. You are asked to specify (1) a lexical offset and (2) a variable name or variable offset. A lexical offset of one refers to the lexical parent of the current activation record. This command sets the current address.

VE The Var Extended command displays eight bytes of the global variables for a unit that is accessible to your current procedure. You are asked to specify (1) a segment number, and (2) a variable name or offset. This command sets the current address.

VS The Var Segment command displays eight bytes of the variables for a segment that is part of the Runtime Support Library or part of your program. The segment in question does not have to be accessible to your current procedure. You are asked to specify (1) a segment name, (2) a procedure name or number, and (3) a variable name or offset. This command sets the current address.

VP The Var Procedure command displays eight bytes of the variables for a specified procedure. You are asked to specify (1) a segment name, (2) a procedure name or number, and (3) a variable name or offset. The procedure must currently by in the call chain. This command sets the current address.

Here is an example of examining and modifying variables:

```
(VL)  Varname or offset#? 1
(l  ) S=EXAMPLE P#3 O#3 0000E776:00 00 00 00 4E 1A 00 00 ----N---
(l  ) S=EXAMPLE P#3 O#7 0000E77E:65 94 00 4E FE 12 0A CD e--N----
(VG)  Varname or offset#? A
(g  ) S=EXAMPLE P#1 V=A 00001A00:00 01 3F 3F 3F 3F 3F 3F --??????
(VI)  Delta lex level? 1  Offset#? 3
(i  ) S=EXAMPLE P#2 O#3 0000E8FE:00 65 00 00 3F 3F 3F 3F -e--????
(VE)  Seg#? 1  Offset#? 5
(e  ) S=KERNEL  P#1 O#5 00000AE4:11 65 00 00 01 04 06 3F -e-----?
(/  )                                              66
```

```
(VS)  Segname? FILEOPS    Offset#? 5
(e )  S=FILEOPS P#1 O#5 00000530: out of range
(VP)  Segname? KERNEL     Procname or #? 31  Offset#? 1
(p )  S=KERNEL  P#31O#1 0000FFE2:3F 3F 3F 3F 3F 3F 3F 3F ????????
```

Changing the Frame of Reference

It is possible to change the current activation record, thereby changing the frame of reference from which the global, local, intermediate, and external variables are viewed. This is accomplished by "chaining up" and "chaining down" the call chain. The following commands examine the call chain and cause the Debugger to move up and down the call chain:

CL — The Chain List command prints out the activation records on the entire call chain.

CD — The Chain Down command sets the current activation record to be the caller of the current procedure. If there are no more procedures in the call chain, this command does nothing.

CU — The Chain Up command sets the current activation record to be the procedure that was called from the current procedure in the actual call chain. If the current procedure is the last procedure that was called, this command does nothing.

Here is an example of using these commands:

```
(CL)
(ms) S=EXAMPLE   P#2  O#3          stat=00006530 dyn =00006530
                                   env =00000996 ipc =0024
(ms) S=EXAMPLE   P#1  O#6          stat=00006542 dyn =0000FFCE
                                   env =00000996 ipc =0036
(ms) S=KERNEL    P#31 O#25         stat=0000004E dyn =0000FFD8
                                   env =000002B6 ipc =0BA5
(ms) S=EXAMPLE   P#1  O#8          stat=0000004E dyn =0000FFE4
                                   env =000002B6 ipc =0C10
(VL)  Offset#? 1
(I )  S=EXAMPLE P#1 O#1 00006520:00 00 00 00 00 00 00 00 --------
(CD)
(ms) S=EXAMPLE   P#1  O#6          stat=00006542 dyn =0000FFCE
                                   env =000002B6 ipc =0C10
(VL)  Offset#? 1
(I )  S=KERNEL P#31 O#6 0000FFE2:65 94 99 4E FE 12 9A CD e--N----
```

The call chain display shows a list of the procedure activations currently on the stack. Each activation is listed with the p–code coordinates of where the activated procedure will return, and four values from the MSCW record, which are used to restore the state of the caller upon leaving the procedure activation. For more information on the MSCW, see the P–MACHINE ARCHITECTURE chapter.

Displaying Registers

The following commands display the p–Machine registers and related information. These commands are closely related to the display options described in the next section.

LR The List Registers command displays the contents of the following p–Machine registers: MP, SP, EREC, SEG, IPC, CURTASK, READYQ. These registers may not be modified. CURTASK is called TIB in the display.

LP The List P–code command displays the p–code that is about to be executed.

LM The List MSCW command displays the Mark Stack Control Word of the current procedure.

LS The List Stack command displays the top portion of the stack, where expression evaluation is taking place. If more than eight words of partial results are currently on top of the stack, the command displays eight words and indicates the number of words not displayed in square brackets.

LA The List Address command displays the eight bytes at the current address.

LE The List Every command is a combination of all the other List options.

Here is an example of each list option, with a description of what is displayed:

```
(LR)
(rg) mp  =0000FFD8 sp  =0000FFD4 erec=00000286 seg =0000839C
     tib =00000030 rdyq=00000030 ipc =0BAB      ior =0000
```

The Register display shows eight p–Machine registers. MP, SP, EREC, TIB and RDYQ are Pascal pointers within the Pascal Data Area. SEG is an absolute handle to the segment base. IPC is a byte offset from the beginning of the segment. IOR is a signed integer value. SEG does not correspond directly to a p–Machine register; it is actually a field of the SIB. TIB is the CURTASK register.

```
(LP)
(cd) S=KERNEL   P#31          0#31          NFJ      11
```

The P–code display shows the current p–code in disassembled form and the coordinates of the p–code.

```
(LM)
(ms) S=KERNEL   P#31 0#31 stat=00005BBA dyn =0000FFD8
                          env =000002B6 ipc =0BAB
```

The MSCW display shows the coordinates of the current p–code and a representation of the current mark stack record. STAT, DYN and ENV are pointers in the Pascal Data Area. IPC is a byte offset from the beginning of the segment.

```
(LS)
(st) [0  ] FED0 0001 0002 0001
```

The Stack display shows the contents of the partial expression results stored on the runtime stack. The rightmost word is the top of the stack. Only the region of the stack between the SP and MP registers is shown. If more than eight words are on the stack, the number in brackets indicates how many words are not shown.

```
(a ) S=HEAPOPS P#3 0#23 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa
```

The Address display shows the eight bytes of memory at the current address.

```
(ms) S=KERNEL  P#31  O#31  stat=00005BBA dyn =0000FFD8
                           env =000002B6 ipc =0BAB
(rg) mp  =0000FFD8 sp  =0000FFD4 erec=000002B6 seg =0000839C
     tib =00000030 rdyq=00000030 ipc =0BAB     ior =0000
(a )               00000000:00 00 00 01 00 04 00 01 --------
(st) [0  ] 0000 0000
(cd) S=KERNEL  P#31        O#31        NFJ     11
```

The Every display prints all the options of the List command.

Configuring the Display Options

Normally, when the Debugger is entered it does so quietly, printing only the right parenthesis prompt. You may configure the display to show register and other information on entry to the Debugger. Each option of the List command described in the last section may be enabled or disabled within the display.

The following commands affect the display:

E — The Enable command is a two character command that enables a given option in the display. The options are: Register, P—code, MSCW, Stack, Address, and Every.

D — The Disable command is a two character command that disables the indicated option in the display. The options are: Register, P—code, MSCW, Stack, Address, and Every.

Miscellaneous Commands

ML — The Mem Lock command causes the Debugger segment to be locked in memory. This will remain in effect until the next Mem Swap command or Quit command.

MS — The Mem Swap command causes the Debugger segment to be swappable.

Z — The Zseg command prints a formatted listing of the

current environment. The format of its display is described below.

I The Interaction command calls a Debugger Interaction Procedure, if you have supplied one to the Pascal Runtime Library by using the Error_Handling unit. This command allows you to expand the capabilities of the Debugger by writing your own command routine. See GENERAL OPERATIONS for more information on using the Error_Handling unit.

F The File command allows you to examine a portion of a text file between two line numbers that you select.

This is a sample of the output generated by the Zseg command:

```
LONGOPS  evec 000002AA  sib  0000047E
1 KERNEL     erec 000002B6  sib 00000346  seg 0000839C  res=1
2 LONGOPS    erec 0000031E  sib 0000047E  seg 00000000  res=0
3 PASCALIO   erec 000002DE  sib 000003BE  seg 00008340  res=0
4 OSUTIL     erec 000002E6  sib 000003D6  seg 0000837C  res=0
CONCURRE evec 000002A0  sib  00000466
1 KERNEL     erec 000002B6  sib 00000346  seg 00000000  res=0
2 CONCURRE   erec 00000316  sib 00000466  seg 000045EC  res=0
3 EXTRAHEA   erec 000002C6  sib 00000376  seg 00008378  res=0
FILEOPS  evec 00000292  sib  0000044E
1 KERNEL     erec 000002B6  sib 00000346  seg 0000839C  res=1
2 FILEOPS    erec 0000030E  sib 0000044E  seg 00008380  res=0
3 PASCALIO   erec 000002DE  sib 000003BE  seg 00008340  res=0
4 OSUTIL     erec 000002E6  sib 000003D6  seg 0000837C  res=0
5 STRINGOP   erec 000002BE  sib 0000035E  seg 00000000  res=0
press <space> to continue, <esc> to abort
```

The Zseg command prints the segment reference list for each unit in your program's execution environment. Each unit in your program's environment is indicated by a line without a line number, along with the location of the segment's EVEC and SIB. Following the header line for a segment, each segment referenced in its EVEC is indicated. The number at the left of each line is the local segment number of the referenced segment. For each referenced segment, a pointer to its EREC and SIB, and a handle to the segment itself is printed. The RES field at the end of the line corresponds to the seg_res field of its SIB, and indicates whether or not the segment is swappable.

The Interaction command allows you to expand the capabilities of the Debugger by calling a Debugger Interaction Procedure that your program provides to the Debugger via the Error_Handling unit. Here is an example of a program that installs a Debugger Interaction Procedure.

```
program dbgtest;

uses {$U UCSD Pascal 2:Errorhandl.CODE} error_handling;

const
  max_friend = 100;

type
  friend_rec = record
                 name: string;
                 age: integer;
               end;

var
  num_friends: integer;
  my_friends: array[1..max_friend] of friend_rec;
  df: interactive;

procedure my_debug_command;
var
  n: integer;
begin
  write(df,'entry #?');
  readln(df,n);
  writeln(df,n);
  if (n > 0) and (n <= num_friends)
    then
      begin
        writeln(df,'   name=',my_friends[n].name);
        writeln(df,'    age=',my_friends[n].age);
      end;
end;

begin
  reset(df,'.DBGTERM');
  num_friends:= 0;
  set_pm_interactive(my_debug_command);
  ...
end.
```

In order to make use of the Interaction command, you must have the Performance Monitor option enabled in your program. See GENERAL OPERATIONS for information on using the Set Options utility. After the call to SET_PM_INTERACTIVE, the Debugger is augmented with a new command. Here is a sample Debugger session using the above program:

```
Program Begin
Seg Fault on DBGTEST  at Seg KERNEL   P#31  O#23
Seg Fault on ERRORHAN at Seg DBGTEST  P#1   O#28
Seg Fault on DEBUGGER at Seg ERRORHAN P#6   O#25
```

```
Seg Fault on SEGDEBUG at Seg DEBUGGER P#2    O#1
Hit break#0 at  S=DBGTEST  P#1  O#100
(I ) entry #?1
   name=bill
    age=100

(
```

The output before the break point is Performance Monitor output. See PERFORMANCE MONITOR, below, for more information on the Performance Monitor.

The File command allows you to examine a portion of a text file. This command is especially useful if you are using symbolic debugging and have a compiled listing stored in a file on disk. The File command allows you to examine the compiled listing in a range of line numbers that you specify. Here is an example:

```
(F ) Filename? PMTEST First Line#? 10  Last Line#? 12
                 name: string;
                 age: integer;
               end;
```

Summary of Commands

AC Displays bytes at an absolute address.

AD Displays bytes in Pascal Data Area.

BL Lists current break points.

BR Removes a break point.

BS Sets a break point.

CD Chains down one activation record.

CL Lists the call chain.

CU Chains up one activation record.

D Disables a display option: A,E,M,P,R,S.

E Enables a display option: A,E,M,P,R,S.

F	Displays a portion of a text file.
I	Calls the Debugger Interaction Procedure.
LA	Lists memory at the current address.
LE	Lists every option.
LM	Lists current activation record.
LP	Lists current p–code instruction.
LR	Lists p–Machine registers.
LS	Lists the top of the evaluation stack.
ML	Memlocks the Debugger.
MS	Memswaps the Debugger.
P	Disassembles p–code.
Q	Quits the Debugger.
R	Resumes program execution.
S	Steps a single p–code.
T	Steps a single p–code without entering procedures.
VE	Displays external variables.
VG	Displays global variables.
VI	Displays intermediate variables.
VL	Displays local variables.
VP	Displays variables of indicated procedure.
VS	Displays global variables of indicated segment.

Z	Displays program environment.
+	Displays next eight bytes.
–	Displays previous eight bytes.
/	Modifies memory in hexadecimal.
\	Modifies memory in ASCII.

Summary of Response Codes

(a)	Memory address.
(b0)	Break point 0.
(b1)	Break point 1.
(b2)	Break point 2.
(b3)	Break point 3.
(b4)	Break point 4.
(c)	Absolute memory address.
(cd)	Code.
(e)	External variable.
(g)	Global variable.
(i)	Intermediate variable.
(l)	Local variable.
(ms)	MSCW record.
(p)	Procedure variable.
(rg)	Registers.

(st) Stack.

EXAMPLES OF DEBUGGER USAGE

Suppose the following program is to be debugged:

```
 1  2  1:d   1    program not_debugged;
 2  2  1:d   1    var
 3  2  1:d   1      i,j,k: integer;
 4  2  1:d   4      b1,b2: boolean;
 5  2  1:0   0    begin
 6  2  1:1   0      i:= 1;
 7  2  1:1   3      j:= 1;
 8  2  1:1   6      if k <> 1
 9  2  1:1   7        then writeln('What''s wrong?');
10  2   :0   0    end.
```

First we enter the Debugger and set a break point at the beginning of the **if** statement:

```
UCSD Pascal  Debugger  [1R0.0]
(BS) Set break #? 0  Segname? NOTDEBUG
                     Procname or #? 1  Offset  #? 6
(EP)
(R )
```

After setting the break point we enable p–code (EP) and resume (R). When the program reaches offset 6, the Debugger break point is encountered. We single–step twice:

```
Hit break #0 at S=NOTDEBUG P#1 O#6
(cd)  S=NOTDEBUG  P#1       O#6       SLD0    1
(cd)  S=NOTDEBUG  P#1       O#7       SLDC    1
(cd)  S=NOTDEBUG  P#1       O#8       NFJ     18
```

We see that our first single–step did a short load global 1.

NOTE: The allocation of memory offsets to variables is a bit confusing. Normally, the offset line in the listing indicates the word offset of a variable. However, if more than one variable is allocated at once in a list, the variables are allocated in reverse order. Thus, K has offset 1, J as offset 2, and I has offset 3.

The second single–step did a short load constant 1 onto the stack. Now we are about to do an integer comparison and jump. But this is where our error shows up, so we decide to look at what is on the stack before doing this comparison:

```
(LS)
(st)  [0 ]  C514  0001
```

We list the stack and then see two words on the stack. We discover a 1 on top of the stack followed by a word of what appears to be garbage. This leads us to suspect that K was not initialized. Looking over the listing, we realize that this is the case.

Symbolic Debugging Example

To use symbolic debugging, some part of a Pascal compilation unit must be compiled with the {$D+} compiler–time option. After this code has been generated, it is possible to reference variables and procedures by name rather than offset. The following example is a small Pascal program that has been compiled with the $D+ option.

```
 1   0     0:d    1      {$D+}
 2   2     1:d    1       program example;
 3   2     1:d    1       var a,b,c:integer;
 4   2     1:d    4
 5   2     1:d    4         procedure set_c_if_d;
 6   2     2:d    1         var d:boolean;
 7   2     2:0    0         begin
 8   2     2:1    0           d:=a>b;
 9   2     2:1    5           if d then
10   2     2:2    8             c:=a+b;
11   2     1:0    0         end;
12   2     1:0    0
13   2     1:0    0       begin
14   2     1:1    0         a:=0;
15   2     1:1    3         b:=5;
16   2     1:1    6         set_c_if_d;
17   2      :0    0       end.
```

The following listing is an example of a debug session.

```
UCSD Pascal Debugger [1R0.0]
(BS) Segname=EXAMPLE Procname or # = SETCIFD
     symbolic seg not in mem Line#? 8
(R )
```

```
Hit break#0 at S=EXAMPLE P=SETCIFD L#8
(BS) Segname=EXAMPLE Procname or # = SETCIFD
       First#8 Last#10 Line#? 9
(R )
Hit break#1 at S=EXAMPLE P=SETCIFD L#9
(VL) Varname or offset#? D
(I ) S=EXAMPLE P=SETCIFD V=D
                    0000E7B2:00 00 94 48 BE E7 00 00 190C-H-
(Q )
```

The first time the Debugger is entered, the program example isn't in memory and hence the symbolic segment isn't in memory. However, a break point can still be set symbolically providing you know on which line number to stop. For the second break point, the symbolic segment is in memory; because of this, its first and last line numbers are given.

Notice the variable D was accessed symbolically, and its contents are displayed.

If you try to access symbolically when the actual code segment is in memory and its symbolic segment counterpart isn't present, the system displays the error message 'symbolic seg not in mem'. Use the Zseg command in the symbolic Debugger to find out if symbolic information is available for a particular segment.

PERFORMANCE MONITOR

If you are using the Debug Runtime version of the Pascal Runtime Library, you have available to you a performance monitor that can help you detect performance bottlenecks in your program that are related to segment swapping.

You enable the performance monitor by using the Set Options utility. (See GENERAL OPERATIONS.) The output generated by the performance monitor is displayed on the same device that you have enabled for the debugger.

Here is a sample of the performance monitor output:

```
Program Begin
Seg Fault on DBGTEST  at Seg KERNEL   P#31  O#23
```

```
Seg Fault on ERRORHAN at Seg KERNEL   P#1   O#28
Seg Fault on DEBUGGER at Seg KERNEL   P#6   O#25
Seg Fault on SEGDEBUG at Seg KERNEL   P#2   O#1
```

Information is displayed about each fault (segment fault, stack fault or heap fault) that occurs while the performance monitor is running. This information can help you to arrange the segments of your program to avoid faulting. The performance monitor also displays a message indicating when the program has begun and when the program has ended.

The performance monitor can be controlled from the Error_Handling unit. See GENERAL OPERATIONS for details.

9
MEMORY MANAGEMENT

OVERVIEW

This chapter describes the memory management activities performed by the Runtime Support Library. The discussions in this chapter are intended to give you enough of a basic understanding of these memory management activities, so that you should have little difficulty writing sophisticated programs that utilize the Macintosh's memory well. Also, the reasons for some of the "DON'Ts" in regard to using the Macintosh Interface should become clearer.

The chapter begins with a section containing a general discussion of the machine's memory configuration while a UCSD Pascal program is running, and the overall memory management strategies used by the Runtime Support Library.

Next, there is a section which describes the memory management activities that take place when special events called "faults" occur. Understanding the various kinds of faults and how the Runtime Support Library responds to them is important due to their adverse effect on a program's performance.

The final section of this chapter is your guide to the composition of the Runtime Support Library. It details the duties performed by the major routines within the Runtime Support Library units. This information should help you in understanding which Runtime Support Library units will be brought into memory when you use certain UCSD Pascal constructs.

MEMORY ORGANIZATION

All of the memory management activities done by the Runtime Support Library affect the Application Heap Zone and the stack. Figure 9–1 shows the organization of the region of the Macintosh's memory which is dedicated to the Application Heap Zone and the stack.

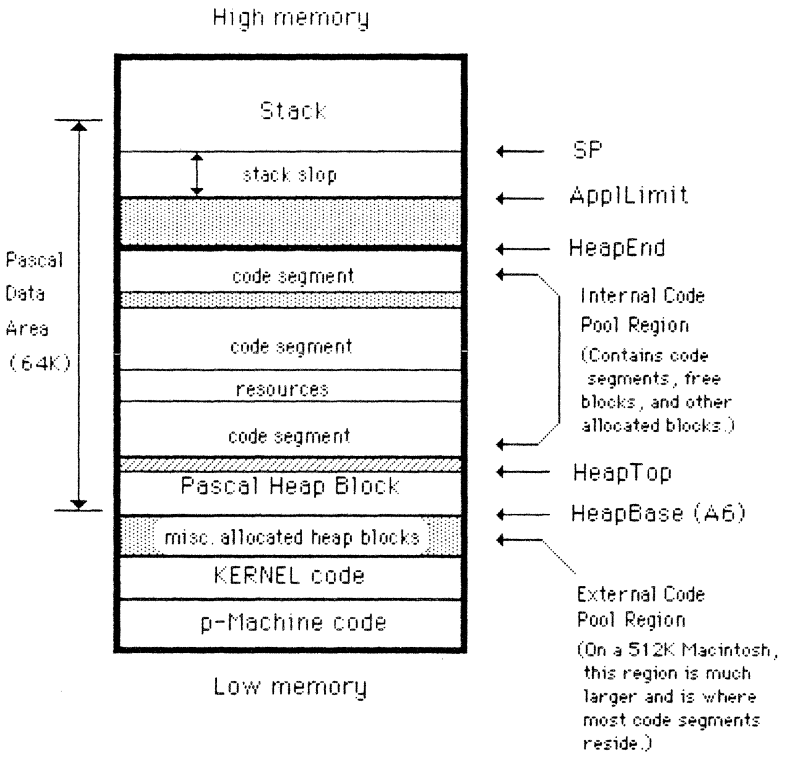


Figure 9–1. Application Heap Zone Organization:

Organization At Program Startup

When a UCSD Pascal program executes, it runs under the control of the p–Machine component of the Runtime Support Package. A small bootstrap routine contained in one of the UCSD Pascal program's resources performs the initial setup of the Application Heap Zone, then it reads in the p–Machine emulator (PME) from the p–Machine file, and transfers control to it.

The primary memory configuration task done by this bootstrap routine is the establishment of the Pascal Data Area. As illustrated in Figure 9–1, this is a 64K region of memory extending from the top of the stack (where the 68000 register A7 points) to the base of a nonrelocatable block in the Application Heap Zone called the Pascal Heap Block. The establishment of the Pascal Data Area involves the proper positioning of the Pascal Heap Block. To create this nonrelocatable heap block, the bootstrap program extends the Application Heap Zone.

The Pascal Data Area is the region of the Macintosh's memory that can be addressed using a UCSD Pascal pointer variable. The PME keeps the 68000 register A6 pointed at the base of the Pascal Heap Block. (As shown in Figure 9–1, this address is given the name HeapBase.) Thus, UCSD Pascal pointer values are actually 16–bit byte offsets off of A6. Because of the limited range of these pointer values, the bootstrap program is careful that the Pascal Data Area is not larger than 64K bytes.

On a Macintosh with 128K bytes of memory, the size of the Pascal Data Area may be smaller than 64K bytes. This is because the bootstrap program must also reserve enough space below the Pascal Heap Block for the p–Machine code and the KERNEL **unit** from the Runtime Support Library. Furthermore, if you have one of the Macintosh debuggers supplied by Apple Computer (e.g. MacsBug) installed, there is a substantial drain on the size of the Pascal Data Area. If you are using a Macintosh with 512K bytes of memory, or the MacWorks software on a Lisa, you will always end up with a 64K Pascal Data Area.

The Pascal Heap Block is where the UCSD Pascal intrinsics new and varnew allocate variables. It is expanded as necessary to accomodate the allocation requests of your program. It is also possible for the Runtime Support Library to shrink the size of Pascal Heap Block whenever there is surplus space in it created by a call to the release, dispose, or vardispose intrinsics. Throughout this chapter, the term "Pascal heap" refers to the heap contained in the Pascal Heap Block.

The PME code and the KERNEL **unit** must always be present and cannot move around in memory. Since it must be possible to expand the size of the Pascal Heap Block when warranted by the UCSD Pascal program's demands for additional variables in the Pascal heap, these position locked pieces of code are positioned below the heap block. This way, they are out of the way of the Pascal heap. They are not allocated inside the Pascal heap, since that would "waste" approximately 16K of the 64K p–Machine data space. (The p–Machine occupies approximately 12K; KERNEL occupies almost 4K.)

Code Segments And Their Location

Any code file created by the UCSD Pascal compiler contains one or more code segments. A code segment is a section of executable code which is brought into memory as a whole unit. Every "compilation unit" (a separately compiled UCSD Pascal **program** or **unit**) results in a "principal segment" of code. In addition, there may be "subsidiary segments," if the **program** or **unit** contains **segment** routines.

A code segment may contain either p–code or native code (or both). Each segment consists of a collection of routines (procedures, functions, and so forth), together with descriptive information, and (usually) a pool of constants. The information embedded within a principal code segment includes references to other compilation units (if any) that it utilizes. The code and information in a segment are contiguous since the code segment is the "unit of movement" for code. There may be up to 255 routines within a segment, numbered 1 through 255.

At compile time, segments are assigned a name and a number. The name is eight characters long. It is used by the Runtime Support Library to resolve intersegment references during the construction of a program's execution environment, and during the maintainance of code files using the Librarian utility. A segment's number is used to reference the segment at runtime.

The segments of a running program compete for space in memory with each other. The segments also compete with the stack and the Pascal heap for space in the Pascal Data Area. The principal constraint (as far as code segments are concerned) is that both the calling and called segment must be present in memory for an intersegment call to succeed.

When a code segment which is not in memory is referenced, it is read from the disk on which it resides into a purgeable and relocatable heap block within the Application Heap Zone. The Runtime Support Library keeps a handle to the code segment within its execution environment data structures. In terms of the memory organization shown in Figure 9–1, code segments are located in one of two regions of the Application Heap Zone: the Internal Code Pool Region, or the External Code Pool Region. A segment remains in memory until it is purged by the Macintosh Memory Manager in order to satisfy an allocation request, or until a fault occurs which causes the Runtime Support Library to purge the block containing the segment.

On a Macintosh with 128K bytes of memory, the External Code Pool Region is very small, and usually becomes clogged with small nonpurgeable blocks. This means that most code segments are located within the Internal Code Pool Region. On a Macintosh with 512K bytes of memory, the situation is dramatically different, since the External Code Pool Region is quite large and most code segments will be located there.

Tasks And Their Stacks

A task is a routine that is executed concurrently with other routines. In UCSD Pascal, a task is known as a **process**. The "main task" is the thread of execution that is the UCSD Pascal program as it is started by the Runtime Support Package. The program may have subsidiary tasks which it starts itself.

The stack used by the main task (your program) is the standard Macintosh application program stack. As shown in Figure 9–1, this stack grows downward in memory and can extend down to where the Macintosh global variable ApplLimit points. As described in *Inside Macintosh*, ApplLimit is the lower limit on the stack, and HeapEnd marks the end of the Application Heap Zone. The Runtime Support Library manages the settings of ApplLimit and HeapEnd for you. In order to maximize your program's utilization of the Macintosh's memory, ApplLimit and HeapEnd are usually quite close to each other. This means that the shaded unused memory region between ApplLimit and HeapEnd shown in Figure 9–1 is usually nonexistent. Instead, most of this unused memory will show up as unallocated blocks of memory inside the Application Heap Zone.

During execution, each subsidiary task uses its own stack instead of the main task stack. The stacks for all subsidiary tasks are allocated within the Pascal heap. The size of a subsidiary task's stack is specified when it is started as a parameter to the start intrinsic. As described in the P–MACHINE ARCHITECTURE chapter, the p–Machine has a Task Information Block (TIB) for each task that has been started. One field in a TIB is called SPLOW, and is the lower limit on the stack pointer when that task is being executed. For the main task, SPLOW always points to the same memory location as the Macintosh ApplLimit variable. The Macintosh's "stack sniffer" (a vertical retrace process that checks that the stack has not encroached on the Application Heap Zone) is enabled while the main task is executing, and is disabled whenever a subsidiary task is executing.

Another field of importance in a TIB is the TASKSLOP field. This field specifies the amount of unused stack space that must be available for use after a call to a procedure has occurred, and any local variables have been allocated on the stack. For the main task, this unused stack space is the "stack slop" area shown in Figure 9–1.

TASKSLOP can be adjusted by a task via an entry point in the Error_Handling unit. (See the GENERAL OPERATIONS chapter.) For the main task, the default setting for TASKSLOP is 2560 (5K bytes), and the minimum setting is 1024 (2K bytes). For subsidiary tasks, the default and minimum settings are both 40 words. (Adjusting the main task's stack slop setting can affect the behavior of your program. This is described in the MACINTOSH INTERFACE chapter.)

Because of the sizeable amount of stack space that they can require, it often isn't practical to call Macintosh Toolbox routines or do I/O operations on disk files from within a subsidiary task.

Controlling Segment Residence

As mentioned previously, a code segment is loaded into a purgeable and relocatable heap block within the Application Heap Zone. Using the memlock and memswap intrinsics, your program can control the residency of a code segment. (See *The UCSD Pascal Handbook* for a discussion on how to use memlock and memswap. The discussion here is aimed at explaining how memlock and memswap are implemented on the Macintosh.)

The memlock instrinsic increments a residency counter for a segment, and memswap decrements this same counter. The transition of the counter's value from zero to one causes the Runtime Support Library to do a Macintosh Memory Manager HNoPurge operation on the heap zone block containing the code segment. As described in *Inside Macintosh*, this has the effect of making the heap block nonpurgeable, but still moveable. Conversely, a transition of the counter's value from one to zero results in an HPurge operation being done, which makes the heap block purgeable again.

In addition to the controlling of code segment residency through memlock and memswap, some additional residency controls are applied by the PME when a Macintosh Interface routine written in assembly language or an in-line **procedure** is called.

When handling a call to an assembly language routine, the PME does a memlock operation on the calling segment, and if necessary, an HLock operation on the heap block containing the code segment in which the assembly language routine resides. This insures that the calling code segment cannot be purged, and that the code for the assembly language routine cannot be moved during the execution of that routine. After the assembly language routine returns to the PME, a memswap operation is done on the calling segment, and an HUnLock operation is done on the called segment.

Similiarly, when an in-line **procedure** is called (i.e. an RCALL p-code is executed), the PME does a HLock operation on the current code segment before executing the Macintosh trap instruction in order to prevent it from being moved or purged.

Actually, the mechanism used by the PME for doing HLock and HUnLock operations on code segments is more complicated than described in the preceeding paragraphs. Because it is possible for a call to a Macintosh Toolbox routine to result in the activation of an "action procedure," situations can arise where HLock operations on a given segment must be nested, and then undone so that the segment remains locked until the initial Toolbox call is completed. To implement this sort of thing, a counter used for HLock/HUnLock operations is also kept for every segment. This counter is used in much the same way as the other residency counter used by memlock and memswap to control when HPurge and HNoPurge operations should be done.

FAULT HANDLING

When memory space is required by the stack or the Pascal heap, or entry into a nonresident code segment is attempted, a fault is issued. When this happens, a **process** called the Faulthandler within the Runtime Support Library KERNEL **unit** is activated. This Faulthandler **process** is started at bootstrap time. Most of the time it is idle, since it does a wait operation on a Pascal semaphore variable. When the semaphore is signaled (either by the PME or another **unit** within the Runtime Support Library), the Faulthandler immediately begins executing, since it is the highest priority task.

In a special tricky maneuver, the Faulthandler switches from its tiny subsidiary task stack to the main task stack. This allows the Faulthandler to take advantage of the stack slop space (which is guaranteed to be at least 2K bytes in size) for its operations. This is one reason why the Error_Handling unit will not let you set the stack slop below 2K bytes: there must be enough slop for the Faulthandler to do its job. This arrangement of having the Faulthandler use the main task's stack slop area works especially well, since it allows the Faulthandler to economize on space in its own stack, without having to forgo the ability to have the screen image preserved when it is marred by "disk swap boxes". (The Faulthandler causes a disk swap box to appear on the screen whenever it attempts to read in a code segment which resides on a mounted disk that is not physically present in a disk drive.) It is also convenient that the manner of disappearance for all disk swap boxes (whether they appear due to segment faults or normal disk file I/O operations) can be controlled through the adjustment of the size of the main task stack slop area. (See the discussion in the MACINTOSH INTERFACE chapter on how setting the stack slop influences the disappearance of "disk swap boxes.")

The PME detects two kinds of faults: segment faults and stack faults. A segment fault occurs when a reference to a nonresident segment happens. Stack faults occur when there isn't enough unused stack space between the stack pointer and the stack limit. The Runtime Support Library causes a third type of fault, called a heap fault, when there isn't sufficient space in the Pascal Heap

Block to satisfy an allocation request in the Pascal heap.

The Memory Collector

An important part of the memory management software within the Runtime Support Library is the Application Heap Zone "grow zone" function called the Memory Collector. (See *Inside Macintosh* for a complete introduction to grow zone functions.) Like any grow zone function, the Memory Collector can be activated any time the Macintosh Memory Manager is called upon to allocate some memory in the Application Heap Zone. The Faulthandler also calls the Memory Collector directly prior to making a call to the Macintosh Memory Manager. This is done primarily to insure that the Application Heap Zone is always extended as far as possible toward the stack, and that heap blocks containing code segments are purged only when there is no other possible means of obtaining the required memory. (The normal inclination of the Macintosh Memory Manager is to purge things from the heap zone first, then expand the zone toward the stack if necessary.)

Through a set of variables in the KERNEL's data, the Memory Collector can tell what kind of fault the Faulthandler is attempting to handle, and it tailors its actions to suit that particular situation. First the Memory Collector calculates the amount of unused space in the main task stack and in the Pascal heap. If the handling of a heap fault is currently in progress, the Memory Collector tries to gain the needed space by expanding the Application Heap Zone. (In terms of Figure 9–1, this is done by repositioning ApplLimit and HeapEnd higher in memory.) When a stack fault is in progress, the Memory Collector takes away any excess space in the Pascal heap by shrinking the Pascal Heap Block. When neither a stack or heap fault is being handled (i.e. when a segment fault occurs), the needed bytes of space are collected from both the stack and the Pascal heap in proportion to the amount of unused space in each.

Effect On Other Heap Zones

As described in the Memory Manager documentation in *Inside Macintosh*, you can set aside a heap block within the Application Heap Zone and initialize it as a heap zone in its own right. You can then establish such a heap zone as the "current" zone, and allocate heap blocks within that zone.

Creating additional heap zones in this fashion can be done without interfering with the activities of the Faulthandler. In fact, establishing such a zone is one way of preventing the Faulthandler from utilizing a region of memory for code segments, or anything else. This is because the Faulthandler makes the current zone the Application Heap Zone prior to making any Macintosh Memory Manager requests. After it has handled the fault, the Faulthandler restores the setting of the current zone to what it was when it started its activities.

Segment Faults

Segment faults are handled by first calling the Memory Collector, then doing a NewHandle or ReAllocHandle Memory Manager request. (ReAllocHandle is used to bring in a code segment that was previously faulted in and subsequently purged.) Since the Memory Collector only collects space by giving up space in the Pascal heap or the stack and doesn't actually purge any heap blocks from the Application Heap Zone, the Faulthandler relies on the Macintosh Memory Manager to purge whatever purgeable heap blocks it has to in order to find enough space for the segment being faulted in. The Faulthandler does a memlock operation on the currently executing segment to prevent it from being purged by the Macintosh Memory Manager. This is necessary because the p-code instruction on which the fault occurred must be re-executed by the PME. A fatal runtime error is reported if it isn't possible to free up enough contiguous memory for the code segment, or if an I/O error is detected when the attempt is made to read the segment into memory.

Heap Faults

The handling of a heap fault begins with an attempt to expand the Pascal Heap Block. This is done by calling the Macintosh Memory Manager ResrvMem and SetPtrSize routines. If this initial attempt fails, the Memory Collector is given a chance to take space away from the stack (through an expansion of the Application Heap Zone), and the enlargement of the Pascal Heap Block is reattempted. If this second attempt fails, the fatal runtime error "Heap Expansion Error" is reported.

It is possible to get a heap expansion error even in situations where there is plenty of space available to the Application Heap Zone. This happens when there is a locked or nonrelocatable heap block in the way of the expansion of the Pascal Heap Block. This is why it is a bad idea to allocate nonrelocatable heap blocks or lock relocatable blocks in a program which intends to allocate variables in the Pascal heap.

Stack Faults

A stack fault within a subsidiary task results in a fatal runtime error, since the stack for a subsidiary task cannot be expanded. A stack fault within the main task occurs when the value (SP – TASK_SLOP) is less than SPLOW (ApplLimit). Stack faults are handled as follows. First the desired new setting for ApplLimit is calculated and compared to HeapEnd. If this new ApplLimit setting (termed "NewApplLimit" in the remainder of this discussion) is greater than HeapEnd, then all the Faulthandler has to do is set ApplLimit to the value NewApplLimit. Otherwise, the Faulthandler must attempt to shrink the Application Heap Zone before setting ApplLimit to its new value.

The shrinking of the Application Heap Zone is the most elaborate task done by the Faulthandler. First the Faulthandler scans the blocks in the zone above the Pascal Heap Block. During this scan, it calculates the maximum amount of space that it can turn into stack space, and records the location of the highest immovable block in the zone. If the highest immovable block is

the Pascal Heap Block, the Faulthandler calls the Memory Collector in the hope that the amount of space it can reclaim from the Pascal Heap Block will result in enough stack space once all the heap blocks are compacted up against the Pascal Heap Block. (Basically, this strategy results in code segments being purged only when absolutely necessary, but at the possible expense of additional heap faults.)

Next, the Faulthandler begins compacting the Application Heap Zone. If the Memory Collector wasn't called, or if it was unable to free up enough space, the Faulthandler purges any blocks that it can during the compaction process, until it judges that enough space has been freed. Throughout the compaction process, any nonpurgeable blocks are moved downward in memory against the highest immovable block in the zone, and adjacent free blocks are combined.

After the compaction and purging process is complete, a new value for HeapEnd is established. If this lowest possible HeapEnd is still above NewApplLimit, a stack overflow runtime error is reported. Otherwise, ApplLimit is set so as to give the stack the space that it needs, plus half of any surplus space reclaimed from the Application Heap Zone.

The compaction and purging process described here does not affect the contents of the External Code Pool Region. The Macintosh Memory Manager does all of the management of that region of the Application Heap Zone.

RUNTIME SUPPORT LIBRARY

The following tables identify the Runtime Support Library routines that the Pascal compiler generates calls to. The first table summarizes the routines in each unit. The second table is indexed by the names of the UCSD Pascal intrinsics that result in calls to Runtime Support Library routines.

Unit	Proc #	Proc Name	Pascal Construct
CONCURRE	3	SStartP	start
	4	SStopP	<exit code>
	6	SExitProcess	exit
EXTRAHEA	2	SDispose	dispose
	3	SVarNew	varnew
	4	SMemLock	memlock
	5	SMemAvail	memavail
	6	SVarAvail	varavail
	7	SMemSwap	memswap
EXTRAIO	2	FBlockIO	blockread,blockwrite
FILEOPS	2	FOpen	reset,rewrite
	3	FClose	close,<exit code>
	4	FInit	<entry code>
	5	FSeek	seek
	6	FReset	reset
HEAPOPS	2	SMark	mark
	3	SRelease	release
	4	SNew	new
KERNEL	15	Moveleft	moveleft
	16	MoveRight	moveright
	17	SExit	exit
	20	Time	time
	21	Fillchar	fillchar
	22	Scan	scan
	23	IOCheck	<after I/O operation>
	29	SAttach	attach
	30	IOResult	ioresult
	32	PwrOfTen	pwroften
	35	Halt	halt
	37	Idsearch	idsearch
	38	Treesearch	treesearch
LONGOPS	2	Decops	<long integer arithmetic>, trunc,str

	3	FReadDec	read,readln
	4	FWriteDec	write,writeln
OSUTIL	3	IntToStr	str,PASCALIO
	4	Int2ToStr	str,PASCALIO
	5	GotIntStr	PASCALIO
	6	Upcase	PASCALIO,EXTRAHEA
PASCALIO	3	FGet	get
	4	FPut	put
	5	FEof	eof
	6	FEoln	eoln
	7	FReadInt	read,readln
	8	FWriteInt	write,writeln
	9	FReadChar	read,readln
	10	FReadString	read,readln
	11	FWriteString	write,writeln
	12	FWriteBytes	write,writeln
	13	FReadln	readln
	14	FWriteln	writeln
	15	FWriteChar	write,writeln
	16	FPage	page
	17	RdInt2	read,readln
	18	WrInt2	write,writeln
	19	ReadBytes	EXTRAIO
	20	WriteBytes	EXTRAIO
	21	ReadTextChar	REALOPS
REALOPS	2	Sin	sin
	3	Cos	cos
	4	Log	log
	5	Ln	ln
	6	ATan	atan
	7	Exp	exp
	8	Sqrt	sqrt
	9	FReadReal	read,readln
	10	FWriteReal	write,writeln
STRINGOPS	2	SConcat	concat
	3	SInsert	insert
	4	SCopy	copy
	5	SDelete	delete
	6	SPos	pos

Intrinsic	Param Type	Routine Called
arctan		REALOPS,6
atan		REALOPS,6
attach		KERNEL,29
blockread		EXTRAIO,2
blockwrite		EXTRAIO,2
close		FILEOPS,3
concat		STRINGOPS,2
copy		STRINGOPS,4
cos		REALOPS,3
delete		STRINGOPS,5
dispose		EXTRAHEA,2
eof		PASCALIO,5
eoln		PASCALIO,6
exit	<proc/func>	KERNEL,17
exit	**program**	KERNEL,17
exit	**process**	CONCURRE,6
exp		REALOPS,7
fillchar		KERNEL,21
get		PASCALIO,3
halt		KERNEL,35
idsearch		KERNEL,37
insert		STRINGOPS,3
ioresult		KERNEL,30
ln		REALOPS,5
log		REALOPS,4
mark		HEAPOPS,2
memavail		EXTRAHEA,5
memlock		EXTRAHEA,4
memswap		EXTRAHEA,7
moveleft		KERNEL,15
moveright		KERNEL,16
new		HEAPOPS,4
page		PASCALIO,16
pos		STRINGOPS,6
put		PASCALIO,4
pwroften		KERNEL,32
read	char	PASCALIO,9
read	integer	PASCALIO,7
read	integer2	PASCALIO,17
read	Long Integer	LONGOPS,3

read	PA of char	PASCALIO,10
read	real	REALOPS,9
read	string	PASCALIO,10
readln		PASCALIO,13
readln	char	PASCALIO,9
readln	integer	PASCALIO,7
readln	integer2	PASCALIO,17
readln	Long Integer	LONGOPS,3
readln	PA of char	PASCALIO,10
readln	real	REALOPS,9
readln	string	PASCALIO,10
release		HEAPOPS,3
reset	(named)	FILEOPS,2
reset	(nameless)	FILEOPS,6
rewrite		FILEOPS,2
scan		KERNEL,22
seek		FILEOPS,5
sin		REALOPS,2
sqrt		REALOPS,8
start		CONCURRE,3
str	integer	OSUTIL,3
str	integer2	OSUTIL,4
str	Long Integer	LONGOPS,2
time		KERNEL,20
treesearch		KERNEL,38
trunc	Long Integer	LONGOPS,2
varavail		EXTRAHEA,6
varnew		EXTRAHEA,3
write	char	PASCALIO,15
write	integer	PASCALIO,8
write	integer2	PASCALIO,18
write	PA of char	PASCALIO,12
write	real	REALOPS,10
write	string	PASCALIO,11
writeln		PASCALIO,14
writeln	char	PASCALIO,15
writeln	integer	PASCALIO,8
writeln	integer2	PASCALIO,18
writeln	PA of char	PASCALIO,12
writeln	real	REALOPS,10
writeln	string	PASCALIO,11
<after I/O>		KERNEL,23
<entry code>		FILEOPS,4

<exit code>	CONCURRE,4
<exit code>	FILEOPS,3
<Long integer arith>	LONGOPS,2

10
P–MACHINE ARCHITECTURE

OVERVIEW

Object code produced by the UCSD Pascal compiler is p–code rather than 68000 machine ("native") code. This p–code is object code for the p–Machine, which is an idealized machine. This chapter describes the p–Machine in general and the p–codes that are produced by the compiler. The information contained in this chapter is most useful when you are debugging a UCSD Pascal program.

p–Code is designed to be compact, so that programs in p–code are much shorter than equivalent programs in native code. p–Code is also designed to be easily generated by a compiler.

Emulative Execution

The "p" in p–code and p–Machine stands for pseudo. The p–Machine emulator program is written in 68000 native code for the Macintosh. It is responsible for executing p–code instructions and interfacing with the Macintosh operating system to obtain system services. The p–Machine emulator is also referred to as the PME.

At runtime, the user's program (or a portion of it) is in main memory. The PME fetches each p–code instruction in sequence, and performs the appropriate action.

STACK ENVIRONMENT

UCSD Pascal programs manipulate data in the stack and the heap. The stack is used for static variables, bookkeeping information about procedure and function calls, and evaluation of expressions. The heap is used for dynamic variables, including the structures that describe a program's environment. It is also used to store private stacks for subsidiary processes and to store code segments that are position–locked.

The stack is an integral part of the p–Machine architecture. Most p–code instructions affect the stack in one way or another. Each time a procedure is called, an activation record is created on the stack which contains some housekeeping information about the calling environment. Space for the procedure's variables is allocated along with some extra space for expression evaluation.

The heap is also an integral part of the system, but is primarily supported by the Runtime Support Library, rather than the p–Machine. The heap contains global data for programs and units (data not declared inside of a named procedure). The global data is allocated when a program is started and remains in memory until the program is terminated. The heap also contains SIBs, ERECs, and EVECs.

Activation Records

An activation record is created for each invocation of an active routine (procedure or function). Figure 10—1 shows the structure of an activation record.

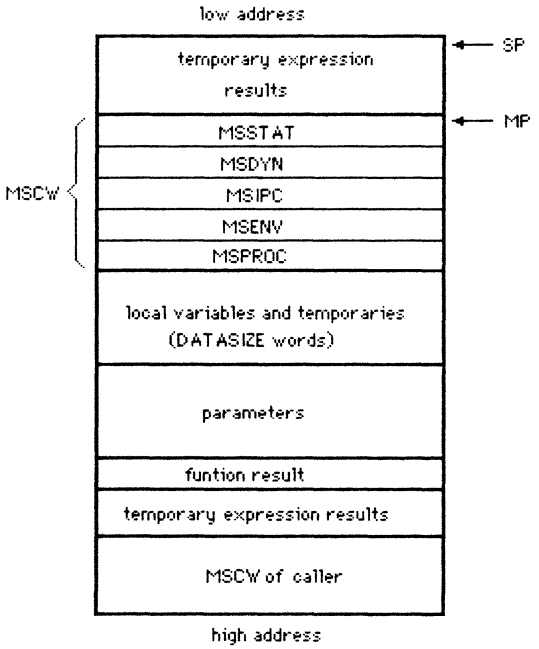


Figure 10—1. Activation Record.

The parts of an activation record are:

1. Mark Stack Control Word (MSCW). This area contains five words of housekeeping information:

   a. MSSTAT — pointer to the activation record of the lexical parent.

   b. MSDYN — pointer to the activation record of the caller.

   c. MSIPC — segment relative byte pointer to point of call in the caller.

   d. MSENV — EREC pointer of the caller.

   e. MSPROC — procedure number of caller.

2. Local and temporary variables. This area is DATA_SIZE words long. The DATA_SIZE value is taken from the code segment that contains the procedure being called. See the CODE FILE FORMAT section for more information.

3. Parameters. This area (which may be empty) contains:

   a. Addresses — for VAR parameters, and record and array value parameters.

   b. Values — for other value parameters.

4. Function value. This area is present only for functions, and is the size of the function result (one, two, or four words).

CODE FILE FORMAT

A code file is composed of a segment dictionary and at least one code segment.

The first block of the code file contains the first record of that file's segment dictionary. A segment dictionary consists of a linked list of dictionary records; if the dictionary is longer than one record, subsequent records are embedded in the code file.

These are each one block long, and are located between code segments.

A single dictionary record can describe up to 16 distinct segments. The information describing a segment is contained in six arrays; the information describing a segment is found by using a single index value to select a component from each of these arrays. Entries in the segment dictionary describe only segments whose code bodies are included in the code file.

The Segment Dictionary

The following Pascal declarations describe a segment dictionary record:

```
CONST Max_Dic_Seg = 15; {maximum seg dict record entry}

TYPE  Seg_Dic_Range = 0..Max_Dic_Seg; {range for seg dict entries}

   Segment_Name = PACKED ARRAY [0..7] OF CHAR; {segment name}

   {segment types}
        Seg_Type = (No_Seg,     {empty dictionary entry}
                    Prog_Seg,   {program outer segment}
                    Unit_Seg,   {unit outer segment}
                    Proc_Seg,   {program or unit}
                    Seprt_Seg); {native code segment}

   {machine types}
        M_Type = (M_Psuedo, M_6809, M_PDP_11, M_8080, M_Z_80,
                            M_GA_440, M_6502, M_6800, M_9900,
                            M_8086, M_Z8000, M_68000, M_HP87);

   {p-machine versions}
        Version = (Unknown, II, II_1, III, IV, V, VI, VII);

   {segment dictionary record}
        Seg_Dict = RECORD
          Disk_Info:
            ARRAY [Seg_Dic_Range] OF {disk info entries}
              RECORD
                Code_Addr: integer;  {segment starting block}
                Code_Leng: integer;  {words in segment}
              END {of RECORD};
          Seg_Name:
            ARRAY [Seg_Dic_Range] OF Segment_Name;
          Seg_Misc:
            ARRAY [Seg_Dic_Range] OF {misc entries}
              PACKED RECORD
                Seg_Type: Seg_Types;     {segment type}
                Filler: 0..31;           {reserved for future use}
                Has_Link_Info: boolean;  {need to be linked?}
                Relocatable: boolean;    {segment relocatable?}
              END {of PACKED RECORD};
          Seg_Text:
            ARRAY [Seg_Dic_Range] OF integer; {interface text}
          Seg_Info:
            ARRAY [Seg_Dic_Range] OF {segment information entries}
              PACKED RECORD
                Seg_Num: 0..255;         {local segment number}
```

```
            M_Type: M_Types;              {machine type}
            Filler: 0..1;                 {reserved for future use}
            Major_Version: Versions;      {p-Machine version}
          END {of PACKED RECORD};
      Seg_Famly:
        ARRAY [Seg_Dic_Range] OF {segment family entries}
          RECORD
            CASE Seg_Types OF
              Unit_Seg, Prog_Seg:
                (Data_Size: integer;      {data size}
                 Seg_Refs: integer;       {segments in comp unit}
                 Max_Seg_Num: integer;    {num segments in file}
                 Text_Size: integer);     {# of blks interface text}
              Seprt_Seg, Proc_Seg:
                (Prog_Name: Segment_Name); {host unit name}
          END {of Seg_Famly};
      Next_Dict: integer; {block num of next dictionary record}
      Filler: ARRAY [1..2] OF integer;
      Checksum: integer;              {see QuickStart in Chapter 6}
      Ped_Block: integer;             {see QuickStart in Chapter 6}
      Ped_Blk_Count: integer;         {see QuickStart in Chapter 6}
      Part_Number: PACKED ARRAY [0..7] of 0..15;
      Copy_Note: string[77]; {copyright notice}
      Dict_Byte_Sex: integer; {machine sex (Sex = 1)}
END {of SEG_DICT};
```

DISK_INFO contains information about the segment's location within the file. Segment code always starts on a block boundary. CODE_ADDR is the number of the block where the segment code starts (relative to the start of the code file). CODE_LENG is the number of 16–bit words in the segment. This size includes the relocation list but doesn't include the segment reference list. All unused entries in this array are zero.

SEG_NAME contains the first eight characters of the program, unit, segment, or assembly procedure name. Unused entries are filled with blanks.

SEG_MISC contains miscellaneous information about the segment. SEG_TYPE indicates the type of segment. PROG_SEG and UNIT_SEG are outer segments of programs and units, respectively. PROC_SEG is a segment routine within either a unit or a program.

SEG_TEXT contains the starting block of the segment's INTERFACE text section, relative to the start of the code file. The INTERFACE text section can appear anywhere within the code file that contains the code segment it describes. The SEG_TEXT array entry, in conjunction with the TEXT_SIZE field in the SEG_FAMLY record, indicates the address and

length of the INTERFACE section in blocks. The INTERFACE text section always starts on a block boundary. Only segments with a SEG_TYPE of UNIT_SEG may have INTERFACE sections. All other segments and unused entries are zero–filled.

SEG_INFO contains further information about the segment. SEG_NUM is the segment number. M_TYPE tells what kind of object code is in the segment. If there is any native code in the segment, then M_TYPE will have one of the processor–specific M_TYPE's. If the segment consists exclusively of p–code, then its M_TYPE is M_PSUEDO. MAJOR_VERSION gives the version of the p–Machine on which the code file is intended to run.

SEG_FAMLY contains information about the code segment's compilation unit. The information contained in this array depends on whether SEG_TYPES indicates a principal or a subsidiary segment.

If the segment is a subsidiary segment, then SEG_FAMLY contains the first eight significant characters of the parent compilation unit's name, stored in PROG_NAME.

If the segment is a principal segment, then the information in SEG_FAMLY consists of four fields:

- DATA_SIZE is the number of words in this segment's base data segment. The variables of principal segments are referenced from any location, including their own outer routine bodies, via global loads and stores (rather than local operations). Therefore, the DATA_SIZE field associated with the body of a code segment is 0, so that no superfluous memory will be allocated in an unused local data area.

- SEG_REFS is the size in words of the segment reference list for this segment.

- MAX_SEG_NUM is the total number of segment numbers assigned to this compilation unit. MAX_SEG_NUM includes all segments with assigned numbers, regardless of whether the segment body is contained in this file or not.

- TEXT_SIZE is the number of blocks of INTERFACE text within the compilation unit. TEXT_SIZE is used in conjunction with the SEG_TEXT array to specify the INTERFACE text for a compilation unit of type UNIT_SEG; it is zero–filled for all other compilation unit types.

If the segment is unused (SEG_TYPES = NO_SEG), then SEG_FAMLY is zero–filled.

NEXT_DICT contains the block number of the next segment dictionary record, relative to the start of the code file. In the last record of the segment dictionary, NEXT_DICT is zero.

PART_NUM contains the SofTech Microsystems internal part number for the file.

FILLER is reserved for future use and should always be zero–filled.

COPY_NOTE is reserved for a copyright message, which can be created with either the Librarian utility or via a compiler directive.

DICT_BYTE_SEX indicates the byte sex of the segment dictionary. It is a full word that contains the value 1, with the same byte sex as the rest of the dictionary record. On the Macintosh, the segment dictionary and all code segments are most–significant–byte–first sex.

Code Segment Structure

The beginning (low address) of a code segment contains the following information about the segment:

- segment–relative pointer to the procedure dictionary
- segment–relative pointer to the relocation list
- the 8–character name of the segment (four words)
- byte sex indicator
- segment–relative pointer to the constant pool
- real size indicator
- part number (two words)

Figure 10–2 illustrates a code segment as it would be loaded into memory.

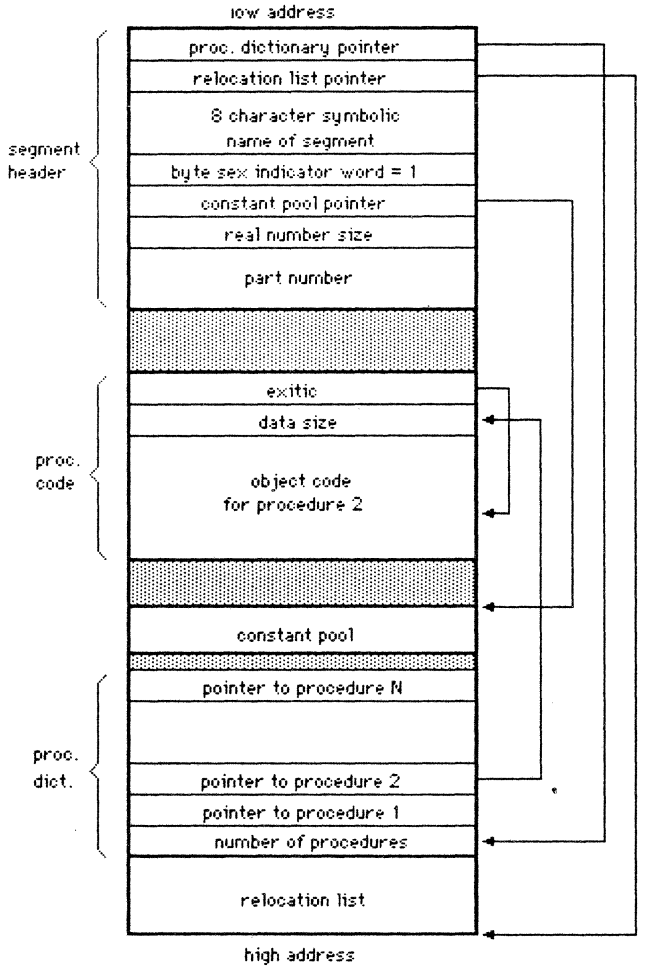


Figure 10–2. Executable Code Segment Format.

The Routine Dictionary

The first word in a code segment points to word 0 of the segment's routine dictionary (also called the procedure dictionary). The routine dictionary is a list of pointers to the code for each routine in the segment. Each routine dictionary pointer is a segment—relative word pointer.

Routines within a segment are numbered 1 through 255. A routine's number is a negative index into the routine dictionary; the n'th word in the dictionary contains a pointer to the code for routine n.

The first word (word 0) of the dictionary contains the number of routines in the segment.

Routine Code

The code of a routine consists of two words: DATA_SIZE and EXIT_IC, followed by the executable object code. The object code may be entirely p—code, entirely native code, or a mixture of the two.

DATA_SIZE is the number of words of local data space that must be allocated when the procedure is called. DATA_SIZE doesn't include parameters; the routine's parameters are assumed to already be on the stack. The first executable instruction starts at the word immediately following the DATA_SIZE word. If the first executable instruction is native code, DATA_SIZE is negative. No local data space is allocated for assembly language procedures.

If this first instruction is a p—code instruction, then EXIT_IC is a segment—relative byte pointer to the code that must be executed when the procedure is exited. Otherwise, EXIT_IC is undefined at runtime.

The Constant Pool/Real Constants

Multi–word constants are stored together in a single constant pool for the entire segment. The constant pool begins immediately after the last body of procedure code in the segment.

The location of the constant pool is contained in the constant pool pointer, a segment–relative word pointer that immediately follows the byte sex indicator word at the beginning of the segment; it points to the low address of the constant pool. If the constant pool pointer is equal to 0, the segment doesn't contain a constant pool.

Constants are referenced by word offsets relative to the beginning (low address) of the constant pool.

The constant pool is divided into two subpools: the real pool and the main pool.

The first word of the constant pool points to the beginning of the real pool. This is a word pointer relative to the start of the constant pool; if there are no real constants in the code segment, this word will be 0. The first word of the real pool contains the number of real constants in the real pool.

Figure 10–3 shows the format of a constant pool with an embedded real subpool.

Real constants are compiled to a processor–independent ("canonical") format and are converted, at segment load time, into a processor–specific internal format.

The real size at compilation time is embedded in every code segment (even though it may not reference any reals). The REAL_SIZE word at the base of the segment contains this value.

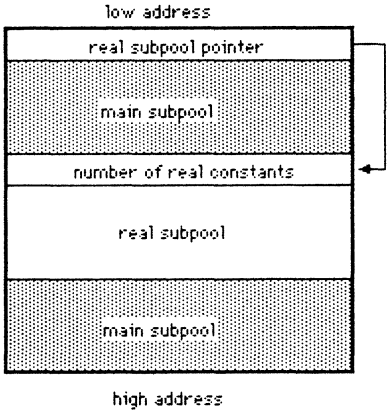


Figure 10—3. Constant Pool

A real constant is represented by a four to six word record. The first word contains a signed integer representing the exponent value. The following words contain the mantissa digits. A mantissa word representing significant mantissa digits contains an integer whose absolute value is between 0 and 9999; its value corresponds to four mantissa digits. The first mantissa word is signed and, thus, contains the mantissa sign. The second and succeeding mantissa word may contain a negative value; in this case, it doesn't contain any significant digits and is disregarded when constructing the internal representation of the real constant. It serves as a terminator word for the constant conversion routines. The decimal point is defined to lie to the right of the four digits in the last valid (used) mantissa word. The digits in the last mantissa word are left—justified. For example, if the real value is 1.1, the first mantissa word contains 1100 decimal (or 044C hexadecimal).

Real constants are converted to native machine format when a code segment is loaded into memory.

The Relocation List

The last (high address) body of information in a code segment is the relocation list. The second pointer at the beginning of the code segment points to the last (highest address) word in the relocation list. This pointer is a segment relative word pointer; if there is no relocation list, it is equal to 0.

The relocation list contains all the information necessary to fix any absolute addresses used by code within the segment, whenever the segment is loaded or moved in memory. Such absolute addresses are needed only by native code. Segments containing exclusively p–code are completely position–independent; no relocation list is needed.

A relocation list consists of 0 or more relocation sublists. Each sublist contains code offsets for objects that must be relocated, and specifies the type of relocation that must be done. Sublists can occur in any order, and more than one sublist can have the same type of relocation.

The following code fragment shows the format of the heading of a sublist:

```
Loc_Types=(Reloc_End, {signals end of entire relocation list}
       Seg_Rel,    {relative to address of base of this segment}
       Base_Rel,   {relative to data segment given in DATASEGNUM}
       Interp_Rel, {relative to PME's interp-relative table}
       Proc_Rel); {relative to address of 1st instruction in proc}

List_Header=PACKED RECORD
          List_Size: integer;    {number of pointers in sublist}
          Data_Seg_Num: 0..255;  {local segment number for Base_Rel}
          Reloc_Type: Loc_Types; {relocation type of sublist entries}
        END;
```

Each sublist contains a LIST_HEADER and 0 or more segment relative byte pointers to the objects which must be relocated. The RELOC_TYPE field in the LIST_HEADER defines what kind of relocation will be applied to all objects designated by the sublist.

The DATA_SEG_NUM field in the LIST_HEADER is used only in sublists with a RELOC_TYPE of BASE_REL, and in all other cases should be zeroed. It specifies the local segment number of the data segment to which all the sublist's pointers are relative.

The LIST_SIZE field in the LIST_HEADER contains the number of pointers in the sublist.

Figure 10—4 illustrates a relocation list with multiple sublists.

The relocation list is intended to be used from high address down to low address. Each sublist in turn is processed from high to low until a sublist with a relocation type of Reloc_End is encountered. The DATA_SEG_NUM should be 0 for the terminating entry; LIST_SIZE is left out for the terminating entry.

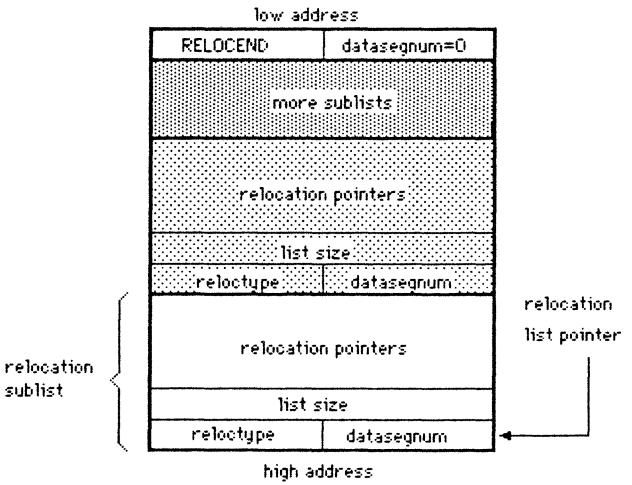


Figure 10—4. Relocation List.

Segment Reference List

In the p–Machine, each code segment is associated at runtime with an "environment vector" that defines the mapping of each segment number to the segment or unit that it designates. Each compilation unit has its own independent (that is, local) series of segment numbers, and its own environment vector. In this way, a particular unit may be referenced by more than one unit, and each unit that references it may use a different segment number.

When a compilation unit references one or more other compilation units, the principal segment of the compilation contains a segment reference list. This list defines the connection between the segment numbers that appear in the object code (they are created by the compiler), and the names of the units to which they refer. Only principal segments contain segment reference lists.

The segment reference list, when present, is located above the relocation list (it grows toward higher memory addresses). The list is used by the operating system at associate time. It doesn't occupy any space in memory during the program's execution (since the segment length field doesn't include it).

The segment reference list associates the name of each compilation unit (which doesn't change) with the number by which that compilation unit is referenced.

The following Pascal declaration describes a record in the segment reference list:

```
Seg_Rec= PACKED RECORD
  Seg_Name: PACKED ARRAY [0..7] OF CHAR; {referenced segment name}
  Seg_Num: PACKED ARRAY [0..1] of 0..255 {0 is SEG_NUM; 1 is unused}
END;
```

The SEG_REFS entry in the segment dictionary (described below) contains the number of words in the segment reference list. The CODE_LENG field in the segment dictionary can be used as a segment relative word pointer to the start of the segment reference list. The segment reference list consists of one

or more SEG_RECs, starting directly above the relocation lists (or procedure dictionary) and continuing towards higher memory addresses. A SEG_REC consists of SEG_NAME, which contains the name of the segment; SEG_NUM[0], which contains the number by which the segment is referenced within this current code segment; and SET_NUM[1], a filler byte.

The segment reference list is terminated by a SEG_REC with a blank–filled SEG_NAME and SEG_NUM of 0.

SEG_RECs with a SEG_NAME of *** are generated so that the Runtime Support Library can execute the initialization and termination code sections of a unit.

When the initialization/termination section of a unit (which is procedure 1) is compiled, the following instruction is emitted between the initialization and termination parts:

```
CXG <***'s Seg_Num>, 1
```

where CXG is the p–code representation of a global procedure call. A local segment number is reserved for the "***" segment reference, and the Runtime Support Library creates a linear list that links together the units of a program that require initialization. At the end of this list is the outer body of the main program. The Runtime Support Library invokes the program by calling the first initialization code on this list, which calls the next, and so forth up to the body of the main program. When the main program terminates, the calling chain is "popped," and termination sections are executed in the reverse order.

CODE SEGMENT ENVIRONMENT

At program startup time, the Runtime Support Library creates a runtime "environment" that describes each code segment and its references to other code segments. A segment's runtime environment is defined by three data structures: the environment record (EREC), the environment vector (EVEC), and the segment information block (SIB).

The Segment Information Block (SIB)

A segment information block (SIB) is a record that contains information about a code segment of a running program. The SIB contains information about the current state of a segment and about the segment's location in memory and on disk. SIBs are created at program startup time for each code segment that the program references and each segment in the Runtime Support Library. SIBs are permanently allocated on the heap for the duration of program execution.

The following Pascal record definition describes a SIB:

```
SIB = record
        seg_base: mem_handle;
        seg_residency: integer;
        seg_locks: integer;
        seg_name: packed array [0..7] of char;
        seg_file: integer;
        seg_addr: integer;
        seg_leng: integer;
        seg_data_size: integer;
      end;
```

seg_base — This field is a handle (absolute pointer to an absolute pointer) that points to the base of the segment in memory. If seg_base is 0 or derefhnd(seg_base) returns 0, then the segment is not currently in memory.

seg_residency — This field contains the memory residency status of the segment. When equal to –1 the segment is position locked. A zero value indicates that the segment is swappable. A value greater than zero is a count of the number of outstanding

memlock operations that have been applied to the segment.

seg_locks — This field contains the count of the number of conceptual HLock operations that have been done on the segment. (HLock causes a Macintosh heap block to become position–locked in memory.) A value of –1 indicates that the segment is position–locked and HLock operations are inappropriate for the segment.

seg_name — This field contains the first eight characters of the segment's name, space filled.

seg_file — This field contains the Macintosh file reference number of the open file that the segment is stored in.

seg_addr — This field contains the block number of the segment within the file whose reference number is seg_file.

seg_leng — This field contains the number of words that the segment occupies, including the relocation list but excluding the segment reference list.

seg_data_size — This field contains the number of words in the segment's global data. This field only applies to **unit** and **program** segments.

The Environment Record (EREC)

A code segment enviroment is a mapping from local segment numbers to the ERECs of the segments they represent. Within the p–code instruction set, segments are referred to by local segment number (an integer in the range 1..255).

The segment environment is represented by two data structures: the environment record (EREC) and the environment vector (EVEC). The EVEC describes the mapping from local segment numbers to the ERECs of those segments. It is implemented as a

word array of pointers to ERECs, indexed by the local segment number. Entry zero of the EVEC is a count of the number of segments in the environment.

The following Pascal record describes ERECs and EVECs:

```
evecp = ^evec;
erecp = ^erec;

evec = record
         vec_length: integer;
         map: array[1..1] of erecp;
       end;

erec = record
         env_data: memptr;
         env_vect: evecp;
         env_sib: sibp;
         env_next: erecp;
       end;
```

env_data — This field points to the segment's global data. The global data is allocated on the heap at program startup time.

env_vect — This field points to the environment's EVEC, which provides the mapping from local segment numbers to ERECs.

env_sib — This field points to the segment's SIB.

env_next — This field is used by the Runtime Support Library to keep track of ERECs.

TASK ENVIRONMENT

The p–Machine supports the implementation of the concurrent tasks of UCSD Pascal. Each task has its own set of the p–Machine registers and its own private stack space in which to save local data. The main task, which is the thread of execution for the user program, uses the Macintosh system stack for its stack. Other tasks use stacks of fixed size allocated within the heap. All tasks share a common heap for dynamic variable allocation.

The main data structure for the implementation of concurrency is the Task Information Block, or TIB, which saves a task's private set of p–Machine registers when it is dormant. A system of task queues is used to handle synchronization of waiting tasks and tasks that are ready to run.

The Semaphore

The Pascal semaphore data type is implemented as a two word construct described by the following Pascal record structure:

```
sem = record
        sem_count: integer;
        sem_wait_q: tib_p;
      end;
```

The sem_count field contains the current value of the semaphore count. The sem_wait_q field points to the queue of tasks that are currently waiting on the semaphore.

The Task Information Block (TIB)

The Task Information Block (TIB) data structure contains all the information necessary to awaken a task that has been dormant. TIBs are linked into queues of waiting and ready–to–run tasks.

The following Pascal record describes a TIB:

```
tib = packed record
        regs: packed record
                wait_q: tib_p;
                prior: byte;
                flags: byte;
                sp_low: mem_ptr;
                sp_upr: mem_ptr;
                sp: mem_ptr;
                mp: mscw_p;
                task_link: tib_p;
                ipc: integer;
                env: e_rec_p;
                procnum: byte;
                m_depend: byte;
                hang_p: sem_p;
                tibioresult: integer;
              end;
        main_task: boolean;
        system_task: boolean;
```

```
    reserved: 0..16383;
    start_mscw: mscw_p;
    task_slop: integer;
end;
```

wait_q — This field points to the next task on a queue.

prior — This field contains the task priority, a number between 0 and 255. Higher numbers represent higher priority.

flags — This field is reserved for future use.

sp_low — This field points to the lower address bound for the stack pointer of the task. In the main task, sp_low is compared with the SP register to determine whether a stack fault should be generated.

sp_upr — This field points to the upper address bound for the stack pointer of the task.

sp — This field is used to save the tasks stack pointer register (SP) when the task is dormant.

mp — This field is used to store the task's mark stack pointer register (MP) when the task is dormant.

task_link — This field is used by the Runtime Support Library to link together all TIBs.

ipc — This field is used to store the task's instruction pointer register (IPC) when the task is dormant. The value is a byte offset within the current segment.

env — This field is used to store the task's enviromnent record register (EREC) when the task is dormant.

procnum — This field is used to store the task's procedure number register when the task is dormant.

m_depend — This field is reserved for future use.

hang_p	This field points to the semaphore that the task is waiting on, or has the value nil if the task is not waiting on a semaphore.
tibioresult	This field is used to store the task's ioresult register when the task is dormant.
main_task	This field is true if this is the main task and false otherwise.
system_task	This field is true if this task is part of the Runtime Support Library and false if this task was started by a user program.
start_mscw	This field points to the first MSCW record in the task's stack.
task_slop	This field is used to store the stack slop value for this task when the task is dormant.

Task Queues

Two p–Machine registers figure in the maintenance of the task enviromnent.

CURTASK	This register points to the TIB of the currently executing task, which is also linked into the ready queue.
READYQ	This register points to the queue of tasks that are ready to run.

Tasks that are waiting to run are linked onto a queue in priority order (tasks with high priority toward the front of the queue). A task queue is a list of TIBs linked through their WAIT_Q fields. The queue is terminated by a pointer to nil. If the task is ready to run, it is linked on the queue pointed to by the READYQ register. If the task is waiting on a semaphore it is linked onto that semaphore's wait queue (the SEM_WAIT_Q field).

Task Switching

Tasks are synchronized through the use of the Pascal intrinsics signal and wait. These, in turn, are implemented by the p–Code instructions SIGNAL and WAIT. See P–CODE DESCRIPTIONS for their operational details.

Both signal and wait can cause a "task switch" to occur. Task switch is the term used to describe the shutting down of one task and the revival of another task. The signal intrinsic causes a task switch when it causes a task of higher priority than the current task to be put into the ready queue. The wait intrinsic causes a task switch to occur when it hangs the current task on a semaphore.

During the operation of a task switch, the p–Machine saves the current state of the p–Machine registers in the TIB of the task that it is shutting down, and restores the registers from the TIB of the task that it is awakening.

FAULTS AND EXECUTION ERRORS

This section describes faults and execution errors, which are exception conditions that may occur during a program's execution.

Faults

A fault is a special condition recognized by the PME during execution of a p–code that requires runtime support assistance to fix. After handling the problem, control returns to p–code execution at the point at which the fault was detected. The p–code where the fault was detected is reexecuted.

Two types of faults may be issued by the PME: segment faults and stack faults. A segment fault is issued when a segment that must be accessed is not in memory. A stack fault is issued if not enough room is available on the stack for a p–code to perform its

operation. Stack height checking is done only on p—codes that will place multiple words on the stack, except in the case of real number operations, which do no stack checking.

When the fault is detected, the p—Machine is to be returned to the state it was in prior to execution of the p—code. This is so that the p—code may be reexecuted on return from the fault.

The following p—codes may issue a segment fault:

CAP, CSP, CXL, SCXGn, CXG, CXI, CFP, RPU, SIGNAL (if a task switch occurs), WAIT (if a task switch occurs)

The following p—codes may issue a stack fault:

LDC, LDM, ADJ, SRS, CLP, CGP, SCIPn, CIP, CXL, SCXGn, CXG, CXI, CFP

Execution Errors

An execution error is a special error condition that the PME may recognize during execution of a p—code. When an execution error is detected, the system reports the error to the user. After an execution error has been detected, the user may choose either to continue execution or reinitialize the system.

Each p—code that can cause an execution error will leave the p—Machine in a consistent state on detection of the error. The IPC will point to the next p—code, putting "dummy" results on the stack; that way the p—code won't be reexecuted on return.

Below is a list of the execution errors, along with the execution error number, the p—codes that may issue the error and a description of what the error means.

Fatal Runtime Support Error Execution Errror 0

p–Codes <none>

This error should not occur. It indicates a corrupt Runtime Support Library.

Value Range Error Execution Errror 1

p–Codes CHK, CSTR, REDU, RED2, SRS

A value range error is issued if an array index or scalar is out of bounds. This is detected only with one of the special check instructions. Generation of these range checks is suppressed by the $R– compiler directive.

Exit from Uncalled Procedure Execution Errror 3

p–Codes <EXIT>

This error occurs when an attempt is made to exit a procedure that is not currently active.

Stack Overflow Execution Errror 4

p–Codes LDC, LDM, ADJ, SRS, CLP, CGP, SCIPn, CIP, CXL, SCXGn, CXG, CXI, CFP

A stack overflow error occurs when there is no room left in memory to expand the stack by the desired amount.

Integer Overflow Execution Errror 5

p—Codes ADI2, SBI2, INC2, DEC2, MPI2, ADIU, SBIU, INCU, DECU, MPIU, <long integer routines>, ABS2, NEG2

An integer overflow error is issued when an integer2 operation result value is too large to represent in an integer2 variable. It can also occur when converting from real, long integer or integer2 to integer, where the resulting integer is too large to fit into 16 bits.

Divide by Zero Execution Errror 6

p—Codes DVI, MODI, DVR, DVI2, MDI2, DVIU, MDIU, <long integer routines>

This error occurs whenever division or the remainder function is attempted with a 0 denominator.

Invalid Memory Reference Execution Errror 7

p—Codes <none>

This error occurs when a memory reference is made through a pointer variable that currently contains nil. This condition is not always detected.

Program Interrupted by User Execution Errror 8

p—Codes <none>

This error occurs if the user presses the break button and the debugger is not enabled.

Runtime Support I/O Error Execution Errror 9

p–Codes <none>

This error occurs if an I/O error occurs during program startup.

I/O Error Execution Errror 10

p–Codes <IOCHECK>

This error occurs when the IOCHECK standard procedure detects the IORESULT is nonzero. Calls to IOCHECK that follow I/O operations can be suppressed with the $I– compiler directive.

Unimplemented Instruction Execution Errror 11

p–Codes <any unimplemented p–code>

This error occurs when an attempt is made to execute an illegal or reserved p–code. This error may not always be detected.

Floating Point Error Execution Errror 12

p–Codes LDCRL, LDRL, STRL, FLT, TNC, RND, ABR, NGR, ADR, SBR, MPR, DVR, EQREAL, LEREAL, GEREAL, RFLT, FLT2, RFLT2, FLTU, RFLTU, TRUNC, ROUND, TRNC2, ROND2, <POWEROFTEN>

This error occurs when the result of a floating point calculation is not a legal floating point number. This may happen on floating point overflow.

String Overflow Execution Errror 13

p—Codes CSP, ASTR, <long integer routines>

This error occurs when a string assignment is made to a string that is too small to hold the source string.

Programmed Halt Execution Errror 14

p—Codes <HALT>

This error occurs upon execution of the halt intrinsic in a user program.

Illegal Heap Operation Execution Errror 15

p—Codes <VARNEW>

This error occurs when a varnew of 0 or fewer words is attempted. It can also occur when calls to mark and release are not properly paired.

Breakpoint Execution Errror 16

p—Codes BPT

This error occurs when a breakpoint p—code is executed. This error will result in entering the debugger if the debugger is currently in an active state.

Incompatible Real Number Size Execution Errror 17

p–Codes <none>

This error occurs if you attempt to run a program compiled with the $R2 compiler option.

Set Too Large Execution Errror 18

p–Codes SRS

This error occurs when an attempt is made to create a set that is larger than the largest allowed set size (4080 members).

Segment Too Large Execution Errror 19

p–Codes CAP, CXL, SCXGn, CXG, CXI, CFP, RPU, SIGNAL, WAIT

This error occurs if an attempt is made to load a segment that is more than 32K bytes in size.

Heap Expansion Error Execution Errror 20

p–Codes <heap operations>

This error occurs if there is no room for the heap to expand. This is most likely to occur due to the presence of a nonrelocatable Macintosh heap block immediately above the Pascal heap in memory.

Insufficient Memory to Load Segment Execution Errror 21

p–Codes CAP, CXL, SCXGn, CSG, CXI, CFP, RPU, SIGNAL, WAIT

This error occurs if there is not enough room in memory to load a required code segment.

P–MACHINE REGISTERS

Like other processors, the p–Machine has registers which are a fundamental part of its architecture. Since the p–Machine is emulated by a program on the host 68000 processor, only some of these registers correspond to actual 68000 processor registers.

Unlike most processors, the p–Machine doesn't allow its registers to be used in a general fashion. All registers have specific uses. The p–Machine stack takes the place of general purpose registers——all temporary data is stored there.

Here is a list of the p–Machine registers, along with a description of how they are used.

CURPROC The CURPROC register contains the procedure number of the currently executing procedure. It changes whenever a procedure call is made. There is a maximum of 255 procedures per segment, so CURPROC will have a value in the range 1 through 255.

CURTASK The CURTASK register is a pointer to the TIB of the currently executing task. It changes whenever a task switch occurs.

EREC The EREC register is a pointer to the EVEC of the current environment. It changes whenever a call or return is made to a procedure in a different segment. The EREC contains pointers

to the global data, EVEC, and SIB. The pointer to the global data (called BASE) is kept in the 68000 A1 register. A pointer to the base of the current segment is kept in the 68000 A2 register.

EVEC — The EVEC register is a pointer to the EVEC (environment vector) of the current environment. It changes whenever a call or return is made to a procedure in a different segment. The EVEC is a redundant register, because it is a field of the EREC. The EVEC register is used to find the EREC of a different segment in order to access its data or to call a procedure in that segment.

IORESULT — IORESULT contains the error code resulting from the last I/O operation. This is the only register that may be accessed directly from a program (via the ioresult intrinsic).

IPC — The IPC register (interpreter program counter) is a pointer to the next p–code that will be executed. This register is located in the 68000 A4 register. IPC changes during each p–code execution. Whenever the IPC register is saved temporarily (for instance, in an MSCW) it is stored as a byte offset from the base of the current segment.

MP — The MP register points to the current activation record (MSCW). This register is located in the 68000 A0 register. It changes whenever a procedure call or return is made. All variables (except those that have been dynamically allocated on the heap) are accessed from an MSCW. Local variables are accessed from MP, global variables from BASE (see EREC, above), and intermediate variables from an intermediate MSCW.

READYQ — The READYQ register points to the TIB at the head of the queue of tasks ready to be run. It may change on a SIGNAL or WAIT p–code.

SP — The SP register points to the word that is on the top of the p–Machine stack. The SP register corresponds to the 68000 stack pointer register in A7. SP changes on nearly every p–code, whenever an item is pushed on or popped off the stack.

P–CODE DESCRIPTIONS

Introduction

The p–codes generated by the compiler are described in this section. Instructions for the p–machine consist of an opcode, which is one or two bytes long, followed by zero to three parameters.

The following example illustrates the format that is used in this chapter to describe the p–codes. (The format of the description is the same for all p–codes.)

LDCB UB — **Load Constant Byte**
[:word] — 80

The constant UB with high byte 0 is pushed onto the stack. LDCB is used to load a constant in the range 0 through 255.

The top line of each p–code description contains the p–code mnemonic, any in–line parameters, and the title of the p–code. (An in–line parameter follows the p–code byte in the p–code stream.) There will be zero to three in–line parameters for each p–code. The symbol for each in–line parameter defines its type. Here the format is UB, meaning unsigned byte. UB and the other parameter formats are discussed below.

The second line of each p–code description contains the stack values on the left in brackets and the p–code hexadecimal instruction value on the right. The stack values consist of two lists of operand types separated by a colon. The list to the left of the colon contains the type of each operand that will be on the stack before the instruction is executed. Following the colon, the type of each operand that the instruction places on the stack as a result is listed. When multiple operands are listed, the operand on the right of each list is at the top of the stack. For this example, the LDCB instruction uses no operands from the stack, but leaves a word result on the stack. The operand types are discussed below.

NOTE: Most p–Machine instructions don't have specific in–line parameters but instead deal with operands that are on the stack.

Finally, there is a brief description of the p–code function. The terms TOS, TOS–1, etc. are used in this description to refer to operands on the stack. TOS is the operand at the top of the stack. TOS–1 is the stack operand just below the operand at TOS.

NOTE: The TOS, TOS–1, etc. terminology only represents the position of an operand relative to other operands; it does not necessarily indicate the displacement on the stack. For example, an operand at TOS–1 would be four words below a floating point operand at TOS, two words below an integer2 operand at TOS, or one word below an integer operand at TOS.

Instruction Parameters

The parameters to a p–code instruction contain information about the size and number of the instruction's operands. (In some cases, the parameter may be an operand itself, as in the case of LDCB, shown above.)

The parameter formats are:

B — Big. This is a parameter with variable length. If bit 7 (MSB) of the first byte is 0, the remaining 7 bits represent a positive integer in the range 0 through 127. If bit 7 of the first byte is 1, then bit 7 is cleared; the first byte is the high–order byte of a 16–bit word, and the following byte is the low–order byte of that word. The big format may represent positive integers in the range 0 through 32767.

DB — Don't Care Byte. Represents a positive integer in the range 0 through 127. Bit 7 is always 0. When converted to a 16–bit value, the most significant byte is zeroed.

DW — Doubleword. This is a 4–byte parameter. It is a 32–bit two's complement value that represents an integer2 in the range −2147483648..2147483647. The doubleword is always represented most significant word first, and each of these words is least significant byte first.

PD — Packed Descriptor. This is a one byte packed field descriptor. The size of the packed field minus 1 (in bits) is stored in the high order nibble of the byte. The bit number of the rightmost bit of the packed field is stored in the low order nibble.

SB — Signed byte. Represents a two's complement 8–bit integer in the range −128 through 127. When converted to a 16–bit two's complement value, the most significant byte is a sign extension (all bits equal bit 7 of the low byte (SB)).

UB — Unsigned byte. Represents a positive integer in the range 0 through 255. When converted to a 16–bit value, the most significant byte is zeroed. When more than one UB parameter is needed in an instruction, they will be referred to by the description as UB1, UB2, etc.

W Word. This is a 2–byte parameter. It is a 16–bit two's complement value that represents an integer in the range –32768 through 32767. The word is always represented as least significant byte first in the code stream.

Dynamic Operands

This section describes the stack–oriented dynamic operands of p–Machine instructions.

activation Activation record for a procedure. See the STACK ENVIRONMENT section for more details.

addr Addr represents a 16–bit p–Machine pointer within the Pascal Data Area.

abs–ptr An 32–bit absolute memory address. It is stored in memory as most significant word first; both word are stored as most significant byte first.

block Block represents a group of 0 or more words. (Used in instructions with variable length operands.)

bool Bool represents a 16–bit quantity treated as a logical value. If bit 0 is 0, the value is FALSE. If bit 0 is 1, the value is TRUE.

byte–ptr A 16–bit byte offset from the base of the Pascal data area.

dword A 32–bit p–Machine doubleword.

func Function result. The actual type depends on the function type. The func operand is null for procedures.

int Int represents a 16–bit two's complement integer.

int2	Same as a doubleword, but interpreted as a signed two's complement integer value.
nil	Nil represents a constant that references an invalid address.
offset	Offset represents a byte offset into a code segment.
pack–ptr	Pack–ptr represents three words that designate a bit field within a 16–bit word. TOS is the number of the rightmost bit of the field, TOS–1 is the number of bits in the field, and TOS–2 is the address of the word.
param	Parameters for a procedure. The number of parameters and their types depend on the code that put them onto the stack.
proc–ptr	Pointer to a procedure.
real	Real represents a 64–bit floating point quantity.
set	A set represents 0 through 255 words of bit flags, preceded by a word that contains the number of words in the set.
uint	A 16–bit unsigned integer value in the range 0..65535.
word	Word represents a 16–bit quantity that may be treated in any way—as an integer, boolean, address, and so forth.
word–ptr	A 16–bit byte offset from the base of the Pascal data area. It must point to a word memory boundary (even address).

Constant Loads

Constant p–codes are used to place constant values from the intruction stream onto the stack.

LCO B **Load Constant Offset**
[:offset] 82

B is a word offset into the constant pool of the current segment. The address of the indicated constant is converted into a segment relative byte offset. The computed offset is pushed onto the stack.

LDCB UB **Load Constant Byte**
[:word] 80

The constant UB with high byte 0 is pushed onto the stack. LDCB is used to load a constant in the range 0 through 255.

LDCD DW **Load Constant Doubleword**
[:dword] FF 00

The doubleword constant DW is pushed onto the stack.

LDCI W **Load Constant Integer**
[:word] 81

The constant word W is pushed onto the stack.

LDCN **Load Constant NIL**
[:nil] 98

A NIL value is pushed onto the stack. The value zero is used to represent NIL.

SLDCn **Short Load Constant**
[:word] 00..1F

The constant word whose value is encoded in the opcode is pushed onto the stack. The value n is the value of the opcode itself. SLDCn is used to load a constant between 0 and 31.

SLDCD0 **Short Load Doubleword Constant Zero**
[:dword] 41

A doubleword containing the value zero is pushed onto the stack.

Local Loads and Stores

The local load and store p–codes are used to transfer data between the stack and the local activation record.

LDL B **Load Local**
[:word] 87

The word with word offset B in the local activation record is pushed onto the stack.

LDLD B **Load Local Doubleword**
[:dword] 58

The doubleword at offset B in the local activation record is pushed onto the stack.

LLA B **Load Local Address**
[:addr] 84

The address of the variable with offset B in the local activation record is pushed onto the stack.

SLDLn **Short Load Local**
[:word] 20..2F

The word with word offset n in the local activation record is pushed onto the stack. SLDLn is used to load one of the first 16 local words. The value of n is 1..32.

SLDLDn **Short Load Local Doubleword**
[:dword] 42..47

The doubleword at offset n in the local activation record is pushed onto the stack. SLDLDn is used to load any of the doubleword data containers whose first word is one of the first 6 local words. The value of n is 1..6.

SLLAn **Short Load Local Address**
[:addr] 60..67

The address of the variable with offset n in the local activation record is pushed onto the stack. SLLAn is used to load the address of local variables with offsets between 1 and 8.

SSTLn **Short Store Local**
[word:] 68..6F

TOS is stored in the word with offset n in the local activation record. SSTLn is used to store in one of the first eight local words. The value of n is 1..8

STL B **Store Local**
[word:] A4

TOS is stored in the word with offset B in the local activation record.

STLD B — **Store Local Doubleword**
[dword:] — 5D

The doubleword operand at TOS is stored into the doubleword located at word offset B in the local activation record.

Global Loads and Stores

The global load and store p–codes are used to transfer data between the stack and the global data storage of the current code segment.

LAO B — **Load Global Address**
[:addr] — 86

The address of the variable with offset B in the global activation record is pushed onto the stack.

LDO B — **Load Global**
[:word] — 85

The word with offset B in the global activation record is pushed onto the stack.

LDOD B — **Load Global Doubleword**
[:dword] — 5A

The doubleword at word offset B in the global activation record is pushed onto the stack.

SLDOn — **Short Load Global**
[:word] — 30..3F

The word with offset n in the global activation record is pushed onto the stack. SLDOn is used to load global words with offsets between 1 and 16. The value of n is 1..16.

SLODDn **Short Load Global Doubleword**
[:dword] 48..4F

The doubleword at word offset n in the global activation record is pushed onto the stack. SLODDn is used to load any of the doubleword data containers whose first word is one of the first 8 global words.

SRO B **Store Global**
[word:] A5

The word at TOS is stored in the word with offset B in the global activation record.

SROD B **Store Global Doubleword**
[dword:] 5F

The doubleword at TOS is stored into the doubleword at word offset B in the global activation record.

Intermediate Loads and Stores

The intermediate load and store p–codes are used to transfer data between the stack and a specific activation record in the stack.

LDA DB,B **Load Intermediate Address**
[:addr] 88

DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The address of the variable with offset B in the indicated activation record is pushed onto the stack.

LOD DB,B **Load Intermediate**
[:word] 89

DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The word with offset B in the indicated activation record is pushed onto the stack.

LODD DB,B **Load Intermediate Doubleword**
[:dword] 59

DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The doubleword with offset B in the indicated activation record is pushed onto the stack.

SLODn B **Short Load Intermediate**
[:word] AD..AE

The word with offset B in the activation record of the parent (SLOD1) or grandparent (SLOD2) of the local activation record is pushed onto the stack.

STR DB,B **Store Intermediate**
[word:] A6

DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The word at TOS is stored into the word with offset B in the indicated activation record.

STRD DB,B **Store Intermediate Doubleword**
[word:] 5E

DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The doubleword at TOS is stored into the doubleword with offset B in the indicated activation record.

Extended Loads and Stores

The extended load and store p–codes are used to transfer data between the stack and the global data storage of a code segment that is not the current segment.

LAE UB,B **Load Extended Address**
[:addr] 9B

The address of the variable with offset B in the global activation record of local segment UB is pushed onto the stack.

LDE UB,B **Load Extended Word**
[:word] 9A

The word at offset B in the global data segment of local code segment UB is pushed onto the stack.

LDED UB,B **Load Extended Doubleword**
[:dword] 5B

The doubleword at offset B in the global data segment of local code segment UB is pushed onto the stack.

STE UB,B **Store Extended Word**
[word:] D9

The word at TOS is stored into the word with offset B in the global activation record of local segment UB.

STED UB,B **Store Extended Doubleword**
[dword:] F6

The doubleword at TOS is stored into the doubleword with word offset B in the global data segment for the local code segment UB.

Indirect Loads and Stores

The indirect load and store p–codes are used to transfer data between the stack and an address specified by an operand on the stack.

IND B **Index and Load Word**
[addr:word] E6

The word offset specified by B is added to the word pointer at TOS. The word pointed to by the resulting word pointer is pushed onto the stack.

INDD B **Load Indirect Doubleword**
[addr:dword] 5C

The word offset specified by B is added to the word pointer at TOS. The doubleword pointed to by the resulting word pointer is pushed onto the stack.

SINDn **Short Index and Load Word**
[addr:word] 78..7F

The word offset n is added to the word pointer at TOS, and the word pointed to by the resulting word pointer is pushed onto the stack. The value of n is 0..7.

SINDDn **Short Index and Load Doubleword**
[addr:dword] 50..57

The word offset n is added to the word pointer at TOS, and the doubleword pointed to by the resulting word pointer is pushed onto the stack. The value of n is 0..7.

STO **Store Word Indirect**
[addr,word:] C4

The word at TOS is stored into the word pointed to by the word pointer at TOS–1.

STOD **Store Doubleword Indirect**
[addr,dword:] F5

The doubleword at TOS is stored into the doubleword pointed to by the word pointer at TOS–1.

Multiple Word Loads and Stores

The multiple word load and store p–codes are used to transfer multiple word data between the stack and memory.

LDC UB1,B,UB2 **Load Constant**
[:block] 83

If less than UB2+STACK_SLOP words are available on the stack, a stack fault is issued.

B is a word offset into the constant pool of the currently executing segment. UB2 words starting at that offset are pushed onto the stack, preserving the order of the words. If UB1, the mode, is 2, and the current segment is of opposite byte sex from the host processor, the bytes of each word are swapped as they are loaded.

LDCRL B — **Load Constant Real**
[:real] — F2

The real constant at offset B in the constant pool of the currently executing segment is loaded onto the stack.

LDM UB — **Load Multiple**
[addr,block] — D0

If less than UB+STACK_SLOP words are available on the stack, a stack fault is issued.

TOS is a pointer to a block of UB words. The block is pushed onto the stack, preserving the order of the words.

LDRL — **Load Real**
[addr:real] — F3

TOS is the address of a real variable. TOS is replaced with the indicated real.

STM UB — **Store Multiple**
[addr,block:] — 8E

TOS is a block of UB words. The block is stored at address TOS–1, preserving the order of the words.

STRL — Store Real

[addr,real:] F4

TOS is a real value. TOS–1 is an address. TOS is stored at the address TOS–1.

Parameter Copying

These instructions are generated by the compiler to copy multiple word parameters which are passed to a procedure by value.

CAP B — Copy Array Parameter

[addr,addr:] AB

TOS is the address of a parameter descriptor for a packed array of characters. The parameter description is a two–word record. The first (low) word is either NIL or a pointer to an EREC. If the first word is NIL, the second word is the address of the parameter. If the first word points to an EREC, the second word is an offset relative to the segment indicated by the EREC. This offset was created with an LCO instruction.

A segment fault is issued if the parameter descriptor indicates a nonresident segment. Otherwise, the array (which is B words long), is copied to the destination at address TOS–1.

CSP UB — Copy String Parameter

[addr,addr:] AC

TOS is the address of a parameter descriptor for a packed array of characters. The parameter description is a two–word record. The first (low) word is either NIL, or a pointer to an EREC. If the first word is NIL, the second word is the address of the parameter. If the first word points to an EREC, the second word is an offset relative to the segment indicated by the EREC. This offset was created with an LCO instruction.

A segment fault is issued if the parameter descriptor indicates a nonresident segment. Otherwise, the dynamic length of the designated string is compared to UB (the declared size of the destination formal parameter). If the string is larger than the destination size, a string overflow execution error is issued. Otherwise, the string is copied to the address TOS–1.

Byte Load and Store

These instructions transfer a byte of data between the stack and a storage area designated by an address on the stack.

LDB — Load Byte

[byte–ptr:word] A7

TOS is a byte pointer. TOS is replaced by the indicated byte with the high byte 0.

STB — Store Byte

[byte–ptr,word:] C8

The low byte of TOS is stored in the location pointed to by byte pointer TOS–1.

Packed Field Loads and Stores

The packed field p–codes are used to transfer packed data between the stack and an address specified by an operand on the stack.

LDP — Load Packed

[pack–ptr:word] C9

The packed field pointer TOS is replaced with the field it designates. Before being pushed onto the stack, the field is right–justified and zero–filled.

SSTP PD **Short Stored Packed**
[addr,word:] 40

The word operand at TOS contains a right justified value which is stored into a field within the word pointed to by the word pointer at TOS–1. The packed field descriptor PD specifies the size and location of the field within the word.

STP **Stored Packed**
[pack–ptr,word:] CA

TOS contains right–justified data. TOS–1 is a packed field pointer. TOS is masked to the field width indicated in TOS–1, then stored into the field described by TOS–1.

UPACK PD **Unpack Field from Top of Stack**
[word:word] AF

The field of the word operand at TOS described by the packed field descriptor parameter PD replaces the word operand at TOS. The value of the packed field is right justified in the result word.

Structure Indexing and Assignment

These instructions are used to index into and copy array and record structures.

AMOVE **Absolute Move Left**
[abs–ptr,abs–ptr,int2:] FF 35

This instruction moves a number of bytes of memory starting at where the absolute address value at TOS–2 points into the successive memory locations starting at where the absolute address value at TOS–1 points. The number of bytes to move is contained in the integer2 operand at TOS.

INC B **Increment**
[addr:addr] E7

The word pointer TOS is indexed by B words, and the resulting pointer is pushed.

INCBI **Increment Pointer with Integer Byte Offset**
[addr,word:addr] FE

The integer operand at TOS (containing a byte offset) is added to the pointer operand at TOS–1, and the resulting pointer value replaces the operands on the stack.

INCB2 **Increment Pointer with Integer2 Byte Offset**
[addr,dword:addr] FF 0D

The integer2 operand at TOS (containing a byte offset) is added to the pointer operand at TOS–1, and the resulting pointer value replaces the operands on the stack.

IXA B **Index Array**
[addr,word:addr] D7

The operand at TOS–1 is a word pointer which locates the base of an array. The word operand at TOS is an index into the array, where the value 0 selects the first element in the array. The value B specifies the size (in words) of the array elements. The operands are replaced on the stack by a word pointer which points to the selected array element.

IXA2 B **Index Array Integer2**
[addr,dword:addr] FF 0B

The operand at TOS–1 is a word pointer which locates the base of an array. The doubleword operand at TOS is an index into the array, where the value 0 selects the first element in the array. The value B specifies the size (in words) of the array elements. The operands are replaced on the stack by a word pointer which

points to the selected array element.

IXP UB1,UB2 **Index Packed Array**
[addr,word:pack–ptr] D8

This operation performs an indexing operation for an array in which multiple elements are packed into a word, and pushes a packed field pointer onto the stack which points to the selected array element. The parameter UB1 specifies the number of array elements that are packed into a word. The parameter UB2 specifies the size of an array element in bits. The word pointer operand at TOS–1 locates the base of the packed array. The integer2 operand at TOS is the index into the array, where the value zero selects the first array element.

IXP2 UB1,UB2 **Index Packed Array Integer2**
[addr,dword:pack–ptr] FF 0C

This operation performs an indexing operation for an array in which multiple elements are packed into a word, and pushes a packed field pointer onto the stack which points to the selected array element. The parameter UB1 specifies the number of array elements that are packed into a word. The parameter UB2 specifies the size of an array element in bits. The word pointer operand at TOS–1 locates the base of the packed array. The integer2 operand at TOS is the index into the array, where the value zero selects the first array element.

MOV UB,B **Move**
[addr,word:] C5

TOS is either the address of a word block (if UB=0) or the offset of a constant word block in the current segment (if UB<>0). B words are moved from the source designated by TOS to the destination address TOS–1. IF UB=2, and the current segment has opposite byte sex from the host processor, the bytes of each word are swapped as the words are moved.

Logical Operators

These instructions perform logical operations on stack data.

BNOT — Boolean Not

[word:bool] 9F

The one's complement of the word at TOS is masked to one bit, and the result is pushed on the stack. BNOT produces a 1 (TRUE) or a 0 (FALSE) on the stack, regardless of how many bits were set in TOS.

GEUSW — Greater Than or Equal Unsigned

[word,word:bool] B5

The boolean result of the unsigned comparison TOS–1 >= TOS is pushed onto the stack.

LAND — Logical AND Word

[word,word:word] A1

The word operands at TOS and TOS–1 are removed from the stack, ANDed together, and the resultant word is pushed onto the stack.

LANDD — Logical AND Doubleword

[dword,dword:dword] FF 27

The doubleword operands at TOS and TOS–1 are removed from the stack, ANDed together, and the resultant doubleword is pushed onto the stack.

LEUSW — Less than or Equal Unsigned

[word,word:bool] B4

The boolean result of the unsigned comparison TOS–1 <= TOS is pushed onto the stack.

LNOT — Logical NOT Word

[word:word] E5

The word operand at TOS is removed from the stack, one's complemented, and pushed onto the stack.

LNOTD — Logical NOT Doubleword

[dword:dword] FF 29

The doubleword operand at TOS is removed from the stack, one's complemented, and pushed onto the stack.

LOR — Logical OR Word

[word,word:word] A0

The word operands at TOS and TOS–1 are removed from the stack, ORed together, and the resultant word is pushed onto the stack.

LORD — Logical OR Doubleword

[dword,dword:dword] FF 28

The doubleword operands at TOS and TOS–1 are removed from the stack, ORed together, and the resultant doubleword is pushed onto the stack.

LXORD — Logical Exclusive OR Doubleword

[dword,dword:dword] FF 2B

The doubleword operands at TOS and TOS–1 are removed from the stack, XORed together, and the resultant doubleword is pushed onto the stack.

LXORW — Logical Exclusive OR Word

[word,word:word] FF 2A

The word operands at TOS and TOS–1 are removed from the stack, XORed together, and the resultant word is pushed onto the stack.

Shift Operators

These instructions perform shifting operations on stack data.

ASRD — Arithmetic Shift Right Doubleword

[dword,int:dword] FF 26

The doubleword operand at TOS–1 is shifted to the right by the number of bits in the value of the word at TOS. The sign of the doubleword operand is propagated into the vacated bit positions. If the count is negative, zero, or greater than 32, the result of the operation is undefined.

ASRW — Arithmetic Shift Right Word

[word,int:word] FF 23

The word operand at TOS–1 is shifted to the right by the number of bits in the value of the word at TOS. The sign of the word operand is propagated into the vacated bit positions. It the count is negative, zero, or greater than 16, the result of the operation is undefined.

LSLD **Logical Shift Left Doubleword**
[dword,int:dword] FF 24

The doubleword operand at TOS–1 is shifted to the left by the number of bits in the value of the word at TOS. If the count is negative, zero, or greater than 32, the result of the operation is undefined.

LSLW **Logical Shift Left Word**
[word,int:word] FF 21

The word operand at TOS–1 is shifted to the left by the number of bits in the value of the word at TOS. If the count is negative, zero, or greater than 16, the result of the operation is undefined.

LSRD **Logical Shift Right Doubleword**
[dword,int:dword] FF 25

The doubleword operand at TOS–1 is shifted to the right by the number of bits in the value of the word at TOS. If the count is negative, zero, or greater than 32, the result of the operation is undefined.

LSRW **Logical Shift Right Word**
[word,int:word] FF 22

The word operand at TOS–1 is shifted to the right by the number of bits in the value of the word at TOS. If the count is negative, zero, or greater than 16, the result of the operation is undefined.

Integer Arithmetic

These instructions peform arithmetic operations on data in the stack.

ABI — Absolute Value Integer

[int:int] E0

TOS is replaced by the absolute value of TOS. If TOS is −32768, the result will be −32768.

ABS2 — Absolute Value Integer2

[int2:int2] FF 06

The integer2 value at TOS is replaced with its absolute value. An Integer Overflow execution error occurs if the initial value is −2147483648.

ADI — Add Integers

[int,int:int] A2

TOS is replaced by TOS−1 + TOS. The result should be computed as if it were an unsigned operation on 32−bit operands, and only the lowest 16 bits were retained for the result. Thus, overflow or underflow will "wrap around" to the opposite sign.

ADI2 — Add Integer2

[int2,int2:int2] F7

The integer2 operands at TOS and TOS−1 are replaced on the stack by the sum of the two operands. An Integer Overflow execution error is reported if the sign bits of the operands are equal and the sign bit of the result has the opposite sign.

CHK — Check Subrange Bounds

[int,int,int:int] CB

TOS is an upper–bound. TOS–1 is a lower–bound. If it isn't the case that TOS–1 <= TOS–2 <= TOS, a value range execution error is issued. TOS–2 remains on the stack.

CHK2 — Integer2 Range Check

[int2,int2,int2:int2] FF 0F

This operator performs a check on the range of the integer2 operand at TOS–2. If it isn't true that TOS–1 <= TOS–2 <= TOS, a Value Range Error execution error is reported. The integer2 operands at TOS and TOS–1 are removed from the stack. The value at TOS–2 remains as the new TOS.

DECI — Decrement Integer

[int:int] EE

TOS is decremented by 1. If TOS is –32768, the result will be 32767.

DEC2 — Decrement Integer2

[int2:int2] FF 04

The integer2 value at TOS is decremented and the result is pushed onto the stack. An Integer Overflow execution error is reported if the initial value is –2147483648.

DVI — Divide Integer

[int,int:int] 8D

If TOS is 0, a divide–by–zero execution error occurs.

Otherwise, TOS is replaced by TOS–1 DIV TOS. The division operation is an integer division truncated toward 0.

DVI2 **Divide Integer2**
[int2,int2:int2] FA

The integer2 operands at TOS and TOS–1 are replaced on the stack by the integer2 result obtained by dividing the operand at TOS–1 by the operand at TOS. If the divisor equals zero, a Divide by Zero execution error is reported. The division operation is an integer division truncated toward zero.

EQUI **Equal Integer**
[int,int:bool] B0

The Boolean result of the comparison TOS–1 = TOS is pushed onto the stack.

EQI2 **Equal Integer2 Comparison**
[int2,int2:bool] FF 07

The integer2 operands at TOS and TOS–1 are replaced on the stack by the Boolean result determined by comparing the operands.

GEI2 **Greater Than or Equal Integer2 Comparison**
[int2,int2:bool] FF 0A

The integer2 operands at TOS and TOS–1 are replaced on the stack by the Boolean result obtained by the comparison TOS–1 >= TOS.

GEQI **Greater Than or Equal Integer**
[int,int:bool] B3

The Boolean result of the signed comparison TOS–1 >= TOS is pushed onto the stack.

INCI — Increment Integer

[int;int] ED

The word at TOS is incremented by 1. If TOS was initially 32767, the result will be –32768.

INC2 — Increment Integer2

[int2:int2] FB

The integer2 operand at TOS is incremented. If an overflow occurs, an Integer Overflow execution error is reported.

LEI2 — Less Than or Equal Integer2 Comparison

[int2,int2:bool] FF 09

The integer2 operands at TOS and TOS–1 are replaced on the stack by the Boolean result obtained by comparing TOS–1 <= TOS.

LEQI — Less Than or Equal Integer

[int,int:int] B2

The Boolean result of the signed comparison TOS–1 <= TOS is pushed onto the stack.

MDI2 — Modulo Integer2

[int2,int2:int2] FF 03

If the integer2 value at TOS is zero, a Divide by Zero execution error is reported.

Otherwise, the integer2 operands at TOS and TOS–1 are replaced on the stack by the value obtained by performing TOS–1 modulo TOS. The operation is undefined if the value at TOS is negative, but no execution error occurs. The result is always an integer2 value in the range 0 <= result < TOS. This result is calculated as if the value at TOS was added or

subtracted from the value at TOS–1 until the result is in the proper range.

MODI — Modulo Integers

[int,int:int] 8F

If the integer value at TOS is zero, a Divide by Zero execution error is reported.

Otherwise, the integer operands at TOS and TOS–1 are replaced on the stack by the value obtained by performing TOS–1 modulo TOS. The operation is undefined if the value at TOS is negative, but no execution error occurs. The result is always an integer value in the range 0 <= result < TOS. This result is calculated as if the value at TOS was added or subtracted from the value at TOS–1 until the result is in the proper range.

MPI — Multiply Integer

[int,int:int] 8C

TOS is replaced by TOS–1 * TOS. The result should be computed as if it were an unsigned operation on 32–bit operands and only the lowest 16 bits were retained for the result.

MPI2 — Multiply Integer2

[int2,int2:int2] F9

The integer2 operands at TOS and TOS–1 are replaced on the stack by the product of the two operands. An Integer Overflow execution error is reported if the value of the result is outside the range of values which can be represented in the integer2 format.

NEG2 — Negate Integer2

[int2:int2] FF 05

The integer2 value at TOS is replaced with its negated value found by taking the two's complement. An Integer Overflow execution error is reported if the initial value is –2147483648.

NEI2 **Not Equal Integer2 Comparison**
[int2,int2:bool] FF 08

The integer2 operands at TOS and TOS–1 are replaced on the stack by the Boolean result obtained by comparing the operands.

NEQI **Not Equal Integer**
[int,int:bool] B1

The Boolean result of the comparison TOS–1 <> TOS is pushed onto the stack.

NGI **Negate Integer**
[int:int] E1

TOS is replaced by the negative (two's complement) of TOS. If TOS was initially –32768, the result should be –32768.

SBI **Subtract Integer**
[int,int:int] A3

TOS is replaced by TOS–1 – TOS. The result should be computed as if it were an unsigned operation on 32–bit operands, and only the lowest 16 bits were retained for the result. Thus, overflow or underflow will "wrap around" to the opposite sign.

SBI2 **Subtract Integer2**
[int2,int2:int2] F8

The integer2 operands at TOS and TOS–1 are replaced on the stack by the difference obtained by subtracting TOS–1 from TOS. An Integer Overflow execution error is reported if the sign bits of the operands are not equal and the sign bit of the result has the same sign as the TOS–1 operand.

Unsigned Arithmetic

The instuctions perform operations on unsigned integer data on the stack.

ADIU — Add Integer Unsigned

[uint,uint:uint] FF 14

The unsigned operands at TOS and TOS–1 are replaced on the stack by the value of TOS–1 + TOS. An Integer Overflow execution error is reported if TOS–1 + TOS is greater than 65535.

CHKU — Unsigned Integer Rangecheck

[uint,uint,uint:uint] FF 1B

The unsigned integer operands at TOS and TOS–1 are removed from the stack and a range check on the value of the unsigned integer operand at TOS–2 is performed. A Value Out of Range execution error is reported if the following is not true: TOS–1 <= TOS–2 <= TOS.

DECU — Decrement Integer Unsigned

[uint:uint] FF 1A

The unsigned integer operand at TOS is decremented and the result replaces the operand on the stack. An Integer Overflow execution error is reported if TOS – 1 is less than zero.

DVIU — Divide Integer Unsigned

[uint,uint:uint] FF 17

The unsigned operands at TOS and TOS–1 are replaced on the stack by the value of TOS–1 div TOS. A Divide by Zero execution error is reported if TOS is zero.

INCU — Increment Integer Unsigned

[uint:uint] FF-19

The unsigned integer operand at TOS is incremented and the result replaces the operand on the stack. An Integer Overflow execution error is reported if TOS + 1 is greater than 65535.

MDIU — Modulo Integer Unsigned

[uint,uint:uint] FF 18

The unsigned operands at TOS–1 is divided by the unsigned integer at TOS and the remainder replaces both operands on the stack. A Divide by Zero execution error is reported if the original operand at TOS is zero.

MPIU — Multiply Integer Unsigned

[uint,uint:uint] FF 16

The unsigned operands at TOS and TOS–1 are replaced on the stack by the value of TOS–1 * TOS. An Integer Overflow execution error is reported if TOS–1 * TOS is greater than 65535.

SBIU — Subtract Integer Unsigned

[uint,uint:uint] FF 15

The unsigned operands at TOS and TOS–1 are replaced on the stack by the value of TOS–1 – TOS. An Integer Overflow execution error is reported if TOS–1 – TOS is less than zero.

Real Arithmetic

These instruction perform operations on floating point data on the stack.

ABR **Absolute Value of Real**
[real:real] E3

TOS is replaced by the absolute value of S.

ADR **Add Reals**
[real,real:real] C0

TOS is replaced by the value TOS–1 + TOS. The result should be 0 on underflow. A floating point execution error is issued on overflow.

DVR **Divide Reals**
[real,real:real] C3

If TOS is 0, a divide–by–zero execution error is issued.

Otherwise, TOS is replaced by the value TOS–1 / TOS. The result will be 0 on underflow. A floating point execution error is issued on overflow.

EQREAL **Equal Real**
[real,real:bool] CD

The Boolean result of the comparison TOS–1 = TOS is pushed onto the stack.

GEREAL **Greater than or Equal Real**
[real,real:bool] CF

The Boolean result of the comparison TOS–1 >= TOS is pushed onto the stack.

LEREAL **Less than or Equal Real**
[real,real:bool] CE

The Boolean result of the comparison TOS–1 <= TOS is pushed onto the stack.

MPR **Multiply Reals**
[real,real:real] C2

TOS is replaced by the value TOS–1 * TOS. The result will be 0 on underflow. A floating point execution error is issued on overflow.

NGR **Negate Real**
[real:real] E4

TOS is replaced by the inverse of TOS.

SBR **Subtract Reals**
[real,real:real] C1

TOS is replaced by the value TOS–1 – TOS. The result will be 0 on underflow. A floating point execution error is issued on overflow.

Set Operations

These instructions perform operations on set data on the stack.

ADJ UB **Adjust Set**
[set:block] C7

If less than STACK_SLOP words on the stack will be available after the completion of the adjust, a stack fault is issued.

The set operand at TOS is stripped of its length word and then expanded or compressed so that it is UB words in size. Expansion is done by adding words of zeros "between" TOS and TOS–1. Compression is done by removing high words of the set. It is legal for adjust to remove "significant" words of the set during compression.

DIF — Set Difference

[set,set:set] DD

The difference between sets TOS–1 and TOS is pushed onto the stack. The difference is computed as bit–wise (TOS–1 AND NOT TOS).

EQPWR — Equal Set

[set,set:bool] B6

The Boolean result of the comparison TOS–1 = TOS is pushed onto the stack. The sets need not be the same size—only the elements must match.

GEPWR — Greater than or Equal Set

[set,set:bool] B8

TRUE is pushed if TOS–1 is a superset of TOS. Otherwise, FALSE is pushed.

INN — Set Membership

[int,set:bool] DA

The Boolean result of the check whether TOS is contained in the set TOS–1 is pushed onto the stack.

INT — Set Intersection

[set,set:set] DC

The intersection (bit–wise AND) of sets TOS and TOS–1 is pushed onto the stack.

LEPWR — Less than or Equal Set

[set,set:bool] B7

TRUE is pushed if TOS–1 is a subset of TOS. Otherwise, FALSE is pushed.

SRS — Build a Subrange Set

[int,int:set] BC

If less than STACK_SLOP words will be available on the stack after this operation, a stack fault is issued.

The integers TOS and TOS–1 must be in the range 0 through 4079. (Refer to *The UCSD Pascal Handbook* for an explanation of set limitations in UCSD Pascal.) If not, a value range execution error is issued.

If TOS–1 > TOS, the empty set is pushed. Otherwise, a set is created containing the elements between TOS–1 and TOS, inclusive, as members. This set is pushed on the stack.

UNI — Set Union

[set,set:set] DB

The union (bit–wise OR) of the sets TOS and TOS–1 is pushed onto the stack.

Byte Array Comparisons

These instructions perform comparison operations on data structures (arrays and records).

EQBYT UB1,UB2,B **Equal Byte Array**
[word,word:bool] B9

UB1 and UB2 are mode flags. If UB1 (or UB2) is 0, then TOS (or TOS–1) is a pointer to a byte array. If UB1 (or UB2) is 1, then TOS (or TOS–1) is an offset within the current segment of a constant byte array. B is the size (in bytes) of the array.

The Boolean result of the comparison TOS–1 = TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the arrays is reached. If there is a mismatch in any character position, FALSE is pushed onto the stack. Otherwise, TRUE is pushed.

GEBYT UB1,UB2,B **Greater than or Equal Byte Array**
[word,word:bool] BB

UB1 and UB2 are mode flags that refer to TOS and TOS–1, respectively. If UB1 (or UB2) is 0, then TOS (or TOS–1) is a pointer to a byte array. If UB1 (or UB2) is 1, then TOS (or TOS–1) is an offset within the current segment of a constant byte array. B is the size (in bytes) of the array.

The Boolean result of the comparison TOS–1 >= TOS is pushed on the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the arrays is reached. If there is a mismatch and the character in TOS–1 < the character in TOS, FALSE is pushed onto the stack. Otherwise, TRUE is pushed.

LEBYT UB1,UB2,B **Less than or Equal Byte Array**
[word,word:bool] BA

UB1 and UB2 are mode flags that refer to TOS and TOS–1, respectively. If UB1 (or UB2) is 0, then TOS (or TOS–1) is a pointer to a byte array. If UB1 (or UB2) is 1, then TOS (or TOS–1) is an offset within the current segment of a constant byte array. B is the size (in bytes) of the array.

The Boolean result of the comparison TOS–1 <= TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the arrays is reached. If there is a mismatch and the character in TOS–1 > the character in TOS, FALSE is pushed onto the stack. Otherwise, TRUE is pushed.

Jumps

These instructions perform conditional and unconditional jumps within the p–code instruction stream.

EFJ SB **Equal False Jump**
[int,int:] D2

If TOS <> TOS–1, a jump is made, relative to the next instruction, by the byte offset SB.

FJP SB **False Jump**
[bool:] D4

If TOS is FALSE, a jump is made, relative to the next instruction, by the byte offset SB.

FJPL W **False Long Jump**
[bool:] D5

If TOS is FALSE, a jump is made, relative to the next instruction, by the byte offset W.

NFJ SB **Not Equal False Jump**
[int,int:] D3

If TOS = TOS–1, a jump is made, relative to the next instruction, by the byte offset SB.

TJP SB **True Jump**
[bool:] F1

If TOS is TRUE, a jump is made, relative to the next instruction, by the byte offset SB.

UJP SB **Unconditional Jump**
[:] 8A

A jump is made, relative to the next instruction, by the byte offset SB.

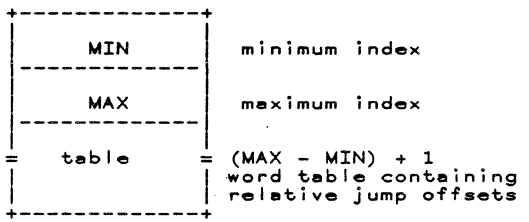
UJPL W **Unconditional Long Jump**
[:] 8B

A jump is made, relative to the next instruction, by the byte offset W.

XJP B **Case Jump**
[int:] D6

B is the offset of the case jump table within the constant pool of the current code segment. The integer value at TOS is an index into this jump table.

The case jump table is structured as follows:



If TOS is in the range MIN through MAX, inclusive, a jump is made, relative to the next instruction, by the word quantity in table entry (TOS – MIN). (The jump table is word–indexed starting at zero, and follows the MAX value in memory). If the TOS operand has a value outside of the range MIN..MAX, no jump occurs and the next p–code instruction in sequence is executed.

XJP2 B **Indexed Jump Integer2**
[int2:] FF 0E

This instruction performs the same operation as XJP, except that the index value on the stack and the MIN and MAX values in the table are integer2 values rather than integer values. The table entries are still word values.

Routine Calls and Returns

These instructions perform procedure calls and returns.

For each procedure call, the following actions occur.

If the Data_Size word for the procedure being called (procedure number at TOS) is negative, nothing is allocated on the stack and a native code call is made. Execution resumes with the following p–code.

Otherwise, DATA_SIZE words and an Mark Stack are allocated on the stack. If STACK_SLOP words are not left on the stack after the MSCW and data are allocated, a stack fault is issued. For intersegment calls, EREC and EVEC are set to reflect the new environment.

BPT **Breakpoint**
[:activation] 9E

A breakpoint execution error is issued unconditionally.

CPF **Call Formal Procedure**
[addr,addr,int:activation] 97

TOS contains a procedure number. TOS–1 contains an EREC pointer; TOS–2 contains a static link. The procedure TOS in the segment indicated by TOS–1 is called. If the segment indicated by TOS–1 is not in memory, a segment fault is issued.

CPG UB **Call Global Procedure**
[param:activation] 91

Global procedure UB in the currently executing segment is called. The static link field of the MSCW is set to the value of BASE (the global data MSCW).

CPI DB,UB **Call Intermediate Procedure**
[param:activation] 92

Intermediate procedure UB in the currently executing segment is called. The static link field of the MSCW is set to the intermediate MSCW that is DB lexical levels above the current MSCW.

CPL UB **Call Local Procedure**
[param:activation] 90

Local procedure UB in the currently executing segment is called. The static link field of the MSCW is set to the old value of MP.

CXG UB1,UB2 **Call External Global Procedure**
[param:activation] 94

The global procedure UB2 in segment UB1 is called. If segment UB1 isn't in memory, a segment fault is issued. The static link field of the MSCW is set to the new value of BASE (the global data MSCW).

If UB1 is 1 and the procedure number matches one of the standard procedure numbers, the p–code performs the standard procedure instead of the call. See the STANDARD PROCEDURES section of this chapter.

CXI UB1,DB,UB2 **Call Intermediate External Proc**
[param:activation] 95

The intermediate procedure UB2 in segment UB1 is called. If segment UB1 isn't in memory, a segment fault is issued. The static link field of the MSCW is set to the intermediate MSCW that is DB lexical levels above the current MSCW.

CXL UB1,UB2 **Call Local External Procedure**
[param:activation] 93

The local procedure UB2 in segment UB1 is called. If segment UB1 isn't in memory, a segment fault is issued. The static link field of the MSCW is set to the old value of MP.

LSL DB **Load Static Link onto Stack**
[:addr] 99

DB indicates the number of static links to traverse. A pointer to the MSCW that is DB links above the current MSCW is pushed onto the stack.

RPU B **Return from Procedure**
[activation:func] 96

Execution returns to the calling procedure.

The EREC pointer in the MSCW indicates the segment to return to. If the segment is not in memory, a segment fault is issued.

Otherwise, MP is set to the Dynamic_Link field of the MSCW. If the MSPROC field of the MSCW is positive, IPC is restored from the MSCW. Otherwise, IPC is set to the Exit_IC value found just before the procedure code in the segment. CURPROC is restored from the MSCW (negating the value, if necessary). If the EREC pointer of the MSCW differs from EREC, EREC and EVEC are set to reflect the new segment.

If the MSPROC field of the MSCW indicates that the return is to a Macintosh ROM routine, the RPU restores the processor registers and returns to the ROM. (See the description of the SETAR p–code for more details.)

SCIPn UB **Short Call Intermediate Procedure**
[param:activation] EF..F0

Intermediate procedure UB in the currently executing segment is called. The Static_Link field of the MSCW is set to the lexical parent (SCPI1) or grandparent (SCPI2) of the current MSCW.

SCXGn UB **Short Call External Global Procedure**
[param:activation] 70..77

The global procedure UB in segment n is called. If segment n isn't in memory, a segment fault is issued.

If the instruction is SCXG1 and the procedure number matches one of the standard procedure numbers, the p–code performs one of these standard procedures, instead of the call. See the STANDARD PROCEDURES section of this chapter.

Concurrency Support

SIGNAL **Signal**
[addr:] DE

The operand at TOS is the address of a semaphore. If the semaphore's wait queue is empty or the count is negative, the count is incremented by one. Otherwise, the TIB at the head of the semaphore's wait queue is put on the ready queue, and its hang_p is set to NIL. If the new task has a higher priority than the current task, a task switch occurs.

WAIT **Wait**
[addr:] DF

The operand at TOS is the address of a semaphore. If the semaphore's count is greater than zero, the count is decremented by one. Otherwise, the current TIB is put on the semaphore's wait queue, its hang_p is set to TOS, and a task switch occurs.

String Operations

The following instructions perform string assignment and comparison operations.

ASTR UB1,UB2 — **Assign String**
[addr,word:] — EB

TOS–1 is the address of the destination string variable. UB2 is the declared size of that string (the number of characters it may hold). TOS is either the address of a string variable (if UB1 is 0), or the offset of a string constant in the constant pool of the current segment.

A string overflow execution error is issued if the dynamic size of the source string is greater than the declared size of the destination string.

Otherwise, the source string is copied to the destination string.

CSTR — **Check String Index**
[:] — EC

TOS–1 is the address of a string variable. TOS is an index into that variable.

If the index is less than 1 or greater than the dynamic length of the string variable, a value range execution error is issued.

EQSTR UB1,UB2 — **Equal String**
[word,word:bool] — E8

UB1 and UB2 are mode flags that refer to TOS and TOS–1, respectively. If UB1 (or UB2) is 0, then TOS (or TOS–1) is a pointer to a string. If UB1 (or UB2) is 1, then TOS (or TOS–1) is an offset of a string within the current segment.

The Boolean result of the comparison TOS–1 = TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the shorter string is reached. The comparison begins at the second element of the strings. If there is a mismatch in any character position, FALSE is pushed on the stack. Otherwise, the lengths of the strings are compared, and the Boolean result of the comparison length(TOS–1) = length(TOS) is pushed.

GESTR UB1,UB2 **Greater or Equal String**
[word,word:bool] EA

UB1 and UB2 are mode flags that refer to TOS and TOS–1, respectively. If UB1 (or UB2) is 0, then TOS (or TOS–1) is a pointer to a string. If UB1 (or UB2) is 1, then TOS (or TOS–1) is an offset of a string within the current segment.

The Boolean result of the comparison TOS–1 >= TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the shorter string is reached. The comparison begins at the second element of the strings. If there is a mismatch in any character position and the character in TOS–1 < the character in TOS, FALSE is pushed on the stack. Otherwise, the lengths of the strings are compared, and the Boolean result of the comparison length(TOS–1) >= length(TOS) is pushed.

LESTR UB1,UB2 **Less or Equal String**
[word,word:bool] E9

UB1 and UB2 are mode flags that refer to TOS and TOS–1, respectively. If UB1 (or UB2) is 0, then TOS (or TOS–1) is a pointer to a string. If UB1 (or UB2) is 1, then TOS (or TOS–1) is an offset of a string within the current segment.

The Boolean result of the comparison TOS–1 <= TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the shorter string is reached. The comparison begins

at the second element of the strings. If there is a mismatch in any character position and the character in TOS–1 > the character in TOS, FALSE is pushed onto the stack. Otherwise, the lengths of the strings are compared, and the Boolean result of the comparison length(TOS–1) <= length(TOS) is pushed.

Operand Type Conversion Operators

The following instructions convert data on the stack from one data type to another.

ATP — **Absolute Address to Pointer**
[abs–ptr:word–ptr] FF 34

The machine absolute address value at TOS is replaced on the stack by the p–machine word pointer value that points to the same memory word.

DEREF — **Dereference Absolute Handle**
[abs–ptr:abs_ptr] FF 36

The operand at TOS is a machine absolute address that points to a doubleword containing another absolute address. This instruction replaces the pointer at TOS by a value which is equal to the low order three bytes of the doubleword that it points to.

EXTI — **Extend Integer to Integer2**
[int:int2] FD

The integer operand at TOS is replaced on the stack by the integer2 operand which contains the same value.

EXTU — **Extend Unsigned Integer to Integer2**
[uint:int2] FF 1D

The unsigned integer operand at TOS is converted to an integer2 operand.

FLT **Float Top–of–Stack**
[int:real] CC

Integer TOS is converted to a floating point number, and the result is pushed onto the stack.

FLT2 **Float Integer2**
[int2:real] FF 12

The integer2 operand at TOS is converted to a floating point number and the result replaces the integer2 operand on the stack.

FLTU **Float Unsigned Integer**
[uint:real] FF 1F

The unsigned integer operand at TOS is converted to a floating point number on top of the stack.

PTA **Pointer to Absolute Address**
[word–ptr:abs–ptr] FF 33

The p–machine word pointer value at TOS is replaced on the stack by the 32–bit machine absolute address which points to the same memory location.

OTP **Word Offset to Pointer**
[int:word–ptr] FF 2D

The integer operand at TOS contains a word memory offset, which is replaced on the stack by a word pointer which points to the memory word indicated by the memory offset. This operation is performed by shifting the word offset to the left by one bit to form a word pointer.

PTO **Pointer to Word Offset**
[word–ptr:int] FF 2C

The word pointer operand at TOS is converted to a word memory offset. The memory offset replaces the operand at TOS. This operation is performed by shifting the pointer (a byte offset) to the right by one bit to form a word offset.

RED2 **Reduce Integer2 to Integer**
[int2:int] FC

The integer2 operand at TOS is reduced to an integer. An Integer Overflow execution error is reported if the result is outside the range –32768..32767.

REDU **Reduce Integer2 to Unsigned Integer**
[int2:uint] FF 1C

The integer2 operand at TOS is removed from the stack, converted to an unsigned integer, and the result pushed onto the stack. An Integer Overflow execution error is reported if the value is negative or greater than 65535.

REXTI **Reversed Extend Integer**
[int,int2:int2,int2] FF 10

The integer operand at TOS–1 is converted to an integer2 operand and inserted into the stack below the integer2 operand at TOS. Following the operation, there are two integer2 operands on the stack.

REXTU **Reversed Extend Unsigned Integer to Integer2**
[uint,int2:int2,int2] FF 1E

The unsigned operand at TOS–1 is replaced on the stack by an integer2 of the same value. The integer2 operand at TOS remains on top of the stack.

RFLT — Reversed Float Integer

[int,real:real,real] FF 11

The integer operand at TOS–1 is converted to a floating point number and the result replaces TOS–1 on the stack. Following the operation, the real operand at TOS remains the top operand on the stack.

RFLT2 — Reversed Float Integer2

[int2,real:real,real] FF 13

The integer2 operand at TOS–1 is converted to a floating point number and the result replaces the integer2 operand at TOS–1. Following the operation, the real operand at TOS remains the top operand on the stack.

RFLTU — Reversed Float Unsigned Integer

[uint,real:real,real] FF 20

The unsigned integer operand at TOS–1 is converted to a floating point number. The real operand at TOS remains on top of the stack.

ROUND — Round Real

[real:int] BF

Real TOS is converted to an integer by rounding, and the result is pushed onto the stack. If the result is outside the range –32768 to 32767, a floating point execution error is issued.

ROND2 — Round Real to Integer2

[real:int2] FF 2F

This performs the same operation as RND, but the integer result is of type integer2. A Floating Point execution error is reported if the result is outside the range of integer2 values.

TRNC2 — **Truncate Real to Integer2**
[real:int2] FF 2E

This performs the same operation as TRUNC, but the integer result is of type integer2. A Floating Point execution error is reported if the result if outside the range of integer2 values.

TRUNC — **Truncate Real**
[real:int] BE

The floating point operand at TOS is converted to an integer by truncating, and the result is pushed onto the stack. If the result isn't in the range −32768 to 32767, a Floating Point execution error is issued.

Miscellaneous Instructions

These instructions perform miscellaneous operations that do not fit into one of the previous categories.

DUPD — **Duplicate Doubleword**
[dword:dword,dword] FF 01

The doubleword operand at TOS is replicated on the stack.

DUPR — **Duplicate Real**
[real:real,real] C6

The real operand at TOS is duplicated on top of the stack.

DUPW — **Duplicate One Word**
[word:word,word] E2

The word operand at TOS is replicated on the stack.

LEREC — Load Current EREC Pointer

[:word–ptr] AA

A word pointer addressing the EREC corresponding to the currently executing code segment is pushed onto the stack.

NOTE: EREC produces the same result as "SLDC 8; LPR" but an update of the TIB for the currently executing task does not occur.

LPR — Load Processor Register

[int:word] 9D

TOS is a register number. The value of the register indicated in TOS is pushed onto the stack. If TOS is negative, the following table indicates which register is pushed:

−1	CURTASK
−2	EVEC
−3	READYQ

If TOS is positive, the current p–Machine registers are saved in the TIB, and TOS is the word index of the register in the TIB to be pushed. If TOS is less than −3 or greater than the size of a TIB, the result of LPR is undefined.

NATIVE — Enter Native Code

[:] A8

This instruction cannot be generated on the Macintosh by the UCSD Pascal compiler.

NATINFO B — Native Code Information

[:] A9

The instruction pointer is incremented to B bytes beyond the byte starting after B in the p–code stream. The bytes after B contain information that is not used by UCSD Pascal on the

Macintosh. This instruction acts like a long form of NOP or a forward jump.

NOP — No Operation

[:] 9C

No operation is performed. Execution continues.

RCALL W — Macintosh ROM Call

[params:result] FF 32

The Macintosh ROM trap instruction contained in W is executed. The parameters that must be on the stack before this instruction and the results left on the stack by executing this instruction are dependent on the ROM call being executed.

SETAR — Set Action Routine

[word–ptr,word–ptr,int,int:abs–ptr] FF 37

To explain this p–code, mnemonic names for the stack operands will be used. STATLNK is the word pointer operand at TOS–3. ERECPTR is the word pointer operand at TOS–2. PROCNUM is the integer operand at TOS–1. SLOTNUM is the integer operand at TOS. Each of these stack operands will be removed by this instruction and an absolute pointer operand ADDR will be pushed onto the stack as the result.

This instruction establishes a p–code routine as an "action routine". STATLNK is the static link pointer required in the MSSTAT field of the MSCW built for a call to the action routine. ERECPTR is a p–Machine pointer to the EREC for the segment containing the action routine. PROCNUM is the procedure number for the routine. SLOTNUM is a number in the range 0 thru 9 which selects which p–Machine caller routine is to be used.

The p–Machine remembers which p–code routines have been associated with its caller routines in a table.

Conceptually, the mechanism involves associating each p–code action routine with a unique native code "caller" routine in the p–Machine. The 32–bit absolute address of the caller routine is returned as the result ADDR. The address ADDR can then be passed to the Macintosh operating system and ROM routines. When the Macintosh ROM decides to call an action routine, it in fact calls the native code caller routine in the p–Machine. The native code caller routine in turn forces the p–Machine to do a call to the p–code action routine to which it has been "attached". When the p–code action routine returns, the native code caller routine returns to its caller.

There are ten native code "caller" routines (numbered 0 thru 9) available for use.

NOTE: p–code action routines must reside in code segments that are loaded in memory when they are called by the p–Machine's caller routines. (The caller routines cannot handle a segment fault because there is no p–code call instruction to re–execute after the segment has been brought into memory.)

When one of the caller routines calls a p–code action routine, bit 8 of the MSPROC word of the MSCW is set to 1. This is a special flag used to indicate to the RPU instruction that control should be transferred back to the Macintosh ROM or operating system. This technique takes advantage of the fact that MSPROC is normally in one of the following ranges of hexadecimal values when an RPU is executed:

```
0001 .. 00FF    (Normal return)
```

If the RPU handler detects bit 8 of MSPROC turned on, and bit 15 isn't turned on, it causes a return to the appropriate Macintosh ROM routine.

SPR **Store Processor Register**
[int,word:] D1

TOS–1 is a register number. If TOS–1 is negative, TOS is stored in one of the following registers:

–1	CURTASK
–2	EVEC
–3	READYQ

Otherwise, the current p–Machine registers are stored in the TIB. TOS is stored in the TIB at offset TOS–1. Finally, the p–Machine registers are restored from the TIB.

SWAP **Swap**
[word,word:word,word] BD

Word TOS is swapped with word TOS–1 on the stack.

SWAPD **Swap Doublewords**
[dword,dword:dword,dword] FF 02

The two doubleword operands at TOS and TOS–1 are exchanged on the stack.

STANDARD PROCEDURES

The standard procedures are procedures that are implemented in the PME directly, either for speed or because the nature of the procedure requires that it be written in native code. A standard procedure is called via a CXG or SCXG1 instruction. The procedure executed is determined by the procedure number.

Most of the standard procedures require parameters on the stack, and some expect a function return value to be passed back. In some sense they act more like individual p–codes than procedures, because no RPU instruction is executed to return

control to the caller. For this reason, the procedure descriptions that follow are presented in a format similar to that of the p–code descriptions—showing the stack before and after execution. The first line of each description gives the name of the procedure and its parameters; the second line gives the stack values and procedure number.

Standard procedures fall into several categories: I/O support, string procedures, compiler procedures, code pool procedures, concurrent procedures, and miscellaneous procedures. The procedures in each category are discussed in the paragraphs that follow.

I/O Support

IORESULT

[zero:int] 1E

TOS is a return word. IORESULT returns the value of the p–Machine register IORESULT.

IOCHECK

[:] 17

IOCHECK tests the p–Machine register IORESULT for 0. If the register is nonzero, an I/O execution error is issued.

String

The standard string procedures are MOVELEFT, MOVERIGHT, FILLCHAR, and SCAN.

MOVELEFT

[byte–ptr,byte–ptr,int:] 0F

The integer operand at TOS is the number of bytes to move. The operand at TOS–1 is a byte pointer to the destination. The operand at TOS–2 is a byte pointer to the source. If TOS is 0 or negative, no bytes are moved. Otherwise, the bytes are moved one at a time starting from the left (low order byte).

MOVERIGHT

[byte–ptr,byte–ptr,int:] 10

The integer operand at TOS is the number of bytes to move. The operand at TOS–1 is a byte pointer to the destination. The operand at TOS–2 is a byte pointer to the source. If TOS is 0 or negative, no bytes are moved. Otherwise, the bytes are moved one at a time starting from the right (high order byte).

FILLCHAR

[byte–ptr,int,word:] 15

The operand at TOS is the character. The integer operand at TOS–1 is the length to fill. The operand at TOS–2 is the starting address for the fill. If TOS–1 is 0 or negative, no filling is done. Otherwise, memory is filled with the byte TOS for TOS–1 bytes starting at address TOS–2.

SCAN

[zero,int,bool,byte,byte–ptr,word:int] 16

The word operand at TOS is a mask field (unused). The operand at TOS–1 is a byte pointer to the array to scan. The operand at TOS–2 is the byte to look for. The boolean operand at TOS–3 is the scan kind (0 means until equal, 1 means until not equal). The integer operand at TOS–4 is the length to scan. If the length is negative, the scan proceeds to the left. The zero operand at TOS–5 is the function result word.

The array at TOS–1 is scanned in the direction indicated in TOS–4 until the character TOS–2 is found (TOS–3 = 0) or a nonmatching character is found (TOS–3 = 1) or until the length in TOS–4 is exhausted. The distance between the character where SCAN stopped and the start character is passed back as the function result.

Compiler

The standard compiler procedures are TREESEARCH and IDSEARCH.

TREESEARCH

[zero,addr,addr,addr:int] 26

The operand at TOS is a pointer to the target string, which is a packed array of eight characters. The operand at TOS–1 is a pointer to where the result of the search will be saved. The operand at TOS–2 is a pointer to the root of the identifier tree to be searched. The zero operand at TOS–3 is the return word.

TREESEARCH searches the symbol table tree TOS–2 for the target string TOS, returning a pointer to where the target was found in the variable pointed to by TOS–1. If the target wasn't found, the variable pointed to by TOS–1 will point to the leaf node of the tree that was searched last. The function result returns status information:

```
 0       target was found
 1       target is to the right
-1       target is to the left
```

Each node of the tree contains the following fields at the indicated byte offsets:

```
 0        name (8 characters)
 8        right link (pointer)
10        left link (pointer)
```

IDSEARCH

[addr,addr:] 25

The operand at TOS is the address of a buffer. The operand at TOS−1 is the address of a record that has the following fields at the indicated byte offsets:

0	SYMCURSOR
2	SY
4	OP
6	ID

IDSEARCH scans the buffer at byte offset SYMCURSOR for an identifier (string beginning with a letter, containing letters, digits, and underscores), ignoring underscores and masking lowercase to uppercase. The identifier is blank−filled to eight characters, then placed in ID for a maximum of eight characters. SYMCURSOR is updated to point to the last character past the identifier.

Finally, the identifier is looked up in a table of reserved words, and its two characteristics are filled into SY and OP. If the identifier is not found in the table, SY is set to 0 and OP is set to 15.

Here is the table of reserved words, along with the SY and OP values for each one:

ID	SY	OP
AND	39	2
ARRAY	44	15
BEGIN	19	15

CASE	21	15
CONST	28	15
DIV	39	3
DO	6	15
DOWNTO	8	15
ELSE	13	15
END	9	15
EXTERNAL	53	15
FOR	24	15
FILE	46	15
FORWARD	34	15
FUNCTION	32	15
GOTO	26	15
IF	20	15
IMPLEMEN	52	15
IN	41	14
INTERFAC	51	15
LABEL	27	15
MOD	39	4
NOT	38	15
OF	11	15
OR	40	7
PACKED	43	15
PROCEDUR	31	15
PROCESS	56	15
PROGRAM	33	15
REPEAT	22	15
RECORD	45	15
SET	42	15
SEGMENT	33	15
SEPARATE	54	15
THEN	12	15
TO	7	15
TYPE	29	15
UNIT	50	15
UNTIL	10	15
USES	49	15
VAR	30	15
WHILE	23	15
WITH	25	15

Code Pool

The standard code pool procedure is RELOCSEG.

RELOCSEG
[addr:] 04

The operand at TOS is the address of an EREC. RELOCSEG relocates the segment pointed to by the EREC. Since RELOCSEG is called after a segment is first read into memory, all necessary relocation is performed.

Concurrency

The standard concurrency procedures are: QUIET, ENABLE, and ATTACH.

QUIET
[:] 1B

QUIET must disable all p–Machine events such that no attached semaphore is signaled until the corresponding call to ENABLE is made.

ENABLE
[:] 1C

ENABLE reenables p–Machine events that have been disabled by QUIET.

ATTACH
[addr,int:] 1D

The integer operand at TOS is the number of a p–Machine event vector. It must be in the range 0 through 63. The operand at TOS–1 is the address of a semaphore.

ATTACH associates the semaphore pointed to by TOS–1 with the vector TOS such that whenever the event TOS is recognized, the semaphore is signaled. If the semaphore pointer is NIL, vector TOS must be unattached from any sempahore it was formerly attached to. If TOS is not in the range 0 through 63, no operation is performed.

Miscellaneous

Standard procedures classified as miscellaneous are TIME and POWEROFTEN.

TIME

[addr,addr:] 14

The operand at TOS is a pointer to where the high word of the time will be saved. The operand at TOS–1 is a pointer to where the low word of the time will be saved.

TIME saves the high and low words of the system clock (a 32–bit 60 Hz clock) in the indicated words. The clock value returned is the Macintosh time (number of ticks since January 1, 1904).

POWEROFTEN

[zero,zero,zero,zero,int:real] 20

The integer operand at TOS is a positive integer power. POWEROFTEN returns the real value ten to the power of TOS. If TOS is negative or TOS is greater than the largest expressible power, a floating point execution error is issued. The four words of zero are the return value.

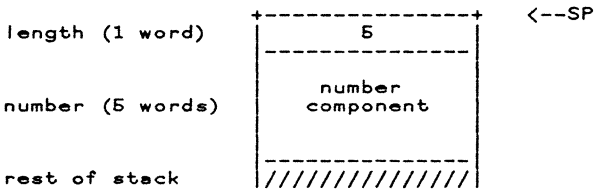
LONG INTEGERS

The long integer data type is a nonscalar data type unique to UCSD Pascal. Long integers may be up to 36 decimal digits long. Although they lack some of the flexibility of scalar types, long integers allow operations on integers outside the range of UCSD Pascal integer2 type (−2147483648..2147483647). In computations, long integers act like real numbers; however, they act more like sets in the way they are implemented and in the way they are passed as parameters.

Number Format

On the stack (when used in calculations), long integers are of variable length, and consist of a length word followed by a number component.

A long integer five words long that is on the top of stack looks like this:



When a long integer is assigned to a variable, or stored in a file on disk, only the number component is present. The length word is present only when the number is on the stack. Each long integer variable is allocated a fixed number of words. When a long integer is assigned to a variable, the number must be coerced to the storage size of the variable. If this can't be done, an Integer Overflow execution error occurs.

The storage size for long integers is from two to ten words, based on the number of digits specified in the declaration statement.

The following table shows the allocation size for each declared size.

digits	size (words)
1..4	2
5..8	3
9..12	4
13..16	5
17..20	6
21..24	7
25..28	8
29..32	9
33..36	10

The declaration size reflects the approximate number of digits that may be stored in the number. More digits than the declared number of digits may sometimes be stored in a long integer variable. As a result, the overflow value for a long integer may vary depending on the size of the long integer. The fact that more digits than the declared size may be stored in a long integer variable shouldn't be relied upon. The number of digits specified in the declaration of a long integer should be treated as the maximum number of digits that the number will ever hold.

The following paragraphs show the format of the long integer number component.

UCSD Pascal on the Macintosh stores long integers by using a sign–magnitude binary–coded decimal (BCD) format with the first word a sign word. The magnitude part of the long integer is stored in natural byte order (the most significant digits in the byte with the lowest address); the number is right–justified within the field. In the sign word, 0 means positive, and FFFF means negative.

```
Examples:

00 00   00 02 76 99     is 27699
FF FF   00 10           is -10
00 00   00 00           is 0
FF FF   00 00           is also 0
```

Long Integer Constants

Long integer constants are constructed at run time by code generated by the compiler. This code builds each constant by doing a series of calculations on integers and integer2s.

Example 1. To build the long integer constant 12, the compiler generates code to do the following:

```
ILI(12)
```

where 12 is an integer constant, and ILI is the routine to convert an integer to a long integer.

Example 2. To build −8733442, the compiler generates code to do the following:

```
-(I2LI(8733442))
```

Example 3. To build the long integer constant 123456789012345, the compiler generates code to do the following:

```
I2LI(1234567890)*I2LI(100000) + I2LI(12345)
```

Here is a listing of the actual p–code generated for the last example. The long integer routines called to do each operation are described in detail later.

```
LAO     1                 8610
LDCD    1234567890        FF00499602D2
SLDC    22                16
SCXG    LONGOPS 2         7202            I2LI
LDCD    100000            FF00000186A0
SLDC    22                16
SCXG    LONGOPS 2         7202            I2LI
SLDC    8                 08
SCXG    LONGOPS 2         7202            MPLI
LDCI    12345             813930
EXTI                      FD
SLDC    22                16
SCXG    LONGOPS 2         7202            I2LI
SLDC    2                 02
SCXG    LONGOPS 2         7202            ADLI
SLDC    5                 05
SLDC    0                 00
SCXG    LONGOPS 2         7202            ADJL
STM     5                 8E05
```

The LONGOPS Unit

LONGOPS is the UCSD Pascal **unit** that implements the long integer functions. LONGOPS contains three procedures: FREADDEC reads a long integer, FWRITEDEC writes a long integer, and DECOPS performs the long integer arithmetic functions.

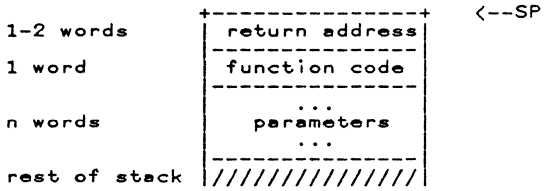
Although LONGOPS isn't part of the p–Machine, it isn't a normal UCSD Pascal **unit** either. Normally, a UCSD Pascal procedure or function must have fixed–size parameters, where the parameter's size is known at compile time. There is one procedure in LONGOPS (DECOPS) that takes variable size parameters. One way to view this is that each call to DECOPS is like the execution of a single p–code in the PME. Different functions of DECOPS take different parameters and return different results, just as different p–codes do. In fact, DECOPS

performs functions very similar to the set p—codes in the PME.

The DECOPS Routine

DECOPS is an external (assembly language) procedure in **unit** LONGOPS that performs the long integer functions.

Parameters are passed to DECOPS on the stack. On every call, the stack looks like this:



The return address is the standard return information for an assembly language routine.

The function code is a word that describes the function to be performed. The actions performed by each function are discussed below, along with the numeric value of the associated function code. Function codes are even integers between 0 and 34. Even integers are used to facilitate jumping indirectly through a word array of addresses.

The parameters vary for each long integer function. The parameter requirements for each routine are included in the description of the routine.

Below are the descriptions of each routine in DECOPS. The first line of each description contains the function name and mnemonic. The second line contains a list of the stack operands before and after the function, and the function code (in hex). The stack lists are in brackets, separated by a colon. The list to the left of the colon is the stack before the function; the list to the right of the colon is the stack contents after the function.

The rightmost operand in each list is the top of stack operand. Finally, there is a detailed description of the function, including any error conditions that should be recognized.

Here are the abbreviations used in the descriptions:

longint	Long Integer. A variable–length long integer, containing a length word.
alongint	Adjusted Long Integer. A fixed length long integer that does not contain a length word.
int	A 16–bit signed integer quantity.
bool	Boolean. A boolean quantity (1=TRUE, 0=FALSE).
uint	Unsigned Integer. A 16–bit unsigned integer in the range 0..65535
int2	Integer2. A 32–bit signed integer.
addr	Address. A 16–bit offset within the Pascal data area.

ADJL — Adjust Long Integer

[longint,int:alongint] 00

Adjusts the longint operand at TOS–1 into an adjusted long integer suitable for assignment to a variable. It does this by stripping off the size word from the longint, then expanding or contracting it until it is the number of words in length as specfied by the integer operand at TOS. If a contraction can't be done because of overflow, an Integer Overflow execution error is reported.

ADLI — Add Long Integer

[longint,longint:longint] 02

Adds the two long integer operands at TOS−1 and TOS, placing the result on the stack. If the result has more than 36 digits, an Integer Overflow execution error may be reported.

SBLI — Subtract Long Integer

[longint,longint:longint] 04

Subtracts the long integer operand at TOS from the long integer operand at TOS−1, placing the result on the stack. If the result has more than 36 digits, an Integer Overflow execution error may be reported.

NGLI — Negate Long Integer

[longint:longint] 06

The long integer operand at TOS is negated.

MPLI — Multiply Long Integer

[longint,longint:longint] 08

The long integer operands at TOS−1 and TOS are multiplied, and the result is pushed onto the stack. If the result has more than 36 digits, an Integer Overflow execution error may be reported.

DVLI — Divide Long Integer

[longint,longint:longint] 0A

The long integer operand at TOS−1 is divided by the long integer operand at TOS, and the result is pushed onto the stack. If the result has more than 36 digits, an Integer Overflow execution error may be reported. If the divisor is zero, a Divide by Zero execution error is reported.

LISTR — Long Integer to String

[longint,addr,int:] 0C

The long integer operand at TOS–2 is converted into a string, placing the result at the location pointed to by the operand at TOS–1. The integer operand at TOS is the maximum length of the string. If the long integer requires more than characters than specified by the maximum length, a String Overflow execution error is reported.

RILI — Reversed Integer to Long Integer

[int,longint:longint,longint] 0E

The integer operand at TOS–1 is converted into a long integer. The long integer at TOS is left unchanged at the top of the stack.

CMPLI — Compare Long Integers

[longint,longint,int:bool] 10

The long integer operand at TOS–2 is compared with the long integer operand at TOS–1 and the boolean result of the comparison is pushed onto the stack. The type of comparison to be performed is indicated by the integer operand at TOS as follows:

0	less than
1	less than or equal
2	greater than or equal
3	greater than
4	not equal
5	equal

ILI — Integer to Long Integer

[int:longint] 12

The integer operand at TOS is converted into a long integer and pushed onto the stack.

LII **Long Integer to Integer**
[longint:int] 14

The long integer operand at TOS is converted into an integer and pushed onto the stack. If the conversion can't be made (long integer isn't in the range −32768..32767), an Integer Overflow execution error is reported.

I2LI **Integer2 to Long Integer**
[int2:longint] 16

The integer2 operand at TOS is removed from the stack, converted into a long integer, and the result is pushed onto the stack.

RI2LI **Reversed Integer2 to Long Integer**
[int2,longint:longint,longint] 18

The integer2 operand at TOS−1 is removed from the stack, converted into a long integer, and the result is pushed onto the stack. Following the operation, the long integer operand at TOS remains at the top of the stack.

LII2 **Long Integer to Integer2**
[longint:int2] 1A

The long integer operand at TOS is removed from the stack, converted to an integer2 operand, and the result is pushed onto the stack. If the value of the long integer is outside the range of values that can be represented by an integer2 operand, an Integer Overflow execution error is reported.

ULI **Unsigned Integer to Long Integer**
[uint:longint] 1C

The unsigned integer operand at TOS is removed from the stack, converted to a long integer, and the result is pushed onto the stack.

RULI **Reversed Unsigned to Long Integer**
[uint,longint:longint,longint] 1E

The unsigned integer operand at TOS–1 is converted to a long integer on the stack. Following the operation, the long integer operand at TOS remains at the top of the stack.

LIU **Long Integer to Unsigned Integer**
[longint:uint] 20

The long integer operand at TOS is removed from the stack, converted to an unsigned integer, and the result is pushed onto the stack. If the long integer operand is negative, or greater than 65535, an Integer Overflow execution error is reported.

ABLI **Absolute Value Long Integer**
[longint:longint] 22

The long integer operand at TOS is replaced by the long integer result containing its absolute value.

APPENDIX A
MACINTOSH INTERFACE

A.1. Table of Compile Time Dependencies

The following table indicates the compile time dependencies between the Macintosh interface units. For example, if your program uses the unit ControlMgr then it must first use the units MacCore, QdTypes, and TbTypes. All of the units with a 'C' in the code column contain code and are therefore included in the Mac Library file. Some of the units that contain code reference other units that contain code. Runtime dependencies are listed after the 'C' in the code column. For example, the FileMgr references the PBIOMgr. If your program uses the FileMgr then you must make the FileMgr, PBIOMgr and MacCore units available to your program at runtime.

Unit Name		Code	Compile Time Dependencies
MacCore	(M)	C	
MacData	(D)	C / M	M Q
MacErrors	(E)		
OSTypes	(O)		M Q T
QDTypes	(Q)		M
TBTypes	(T)		M Q
ControlMgr			M Q T
DeskMgr			M Q T
DialogMgr		C / M	M Q T
EventMgr		C / M	M Q T
FileMgr		C / M P	M
FontMgr			M Q
MemoryMgr	(MM)	C / M	M
MenuMgr			M Q
OSUtilities		C / M MM	M Q T O
Packages			M
PBIOMgr	(P)	C / M	M Q T O
PrintDrvr		C / M MM P	M
PrintMgr		C / M	M Q
QuickDraw			M Q
ResMgr			M
ScrapMgr			M
Serial		C / M	M
Sound		C / M	M
TBoxUtils			M Q
TextEdit			M Q T
WindowMgr			M Q T

A.2 Identifier Cross–Reference List

The following list defines the two–letter codes used for the ToolBox Managers.

CM	ControlMgr
DS	DeskMgr
DL	DialogMgr
EM	EventMgr
FL	FileMgr
FM	FontMgr
MC	MacCore
MD	MacData
ME	MacErrors
MM	MemoryMgr
MN	MenuMgr
OT	OSTypes
OU	OSUtilities
PK	Packages
PB	PBIOMgr
PR	PrintMgr
PD	PrintDrvr
QT	QDTypes
QD	QuickDraw
RM	ResMgr
SM	ScrapMgr
SD	Serial
SN	Sound
TU	TBoxUtils
TT	TBTypes
TE	TextEdit
WM	WindowMgr

The following cross reference list contains the identifiers from the Macintosh interface units. The two–letter code to the right of each identifier indicates which unit it is in.

A5	MD	AbortErr	ME	abs_nil	MC
AbbrevDate	PK	abortEvt	EM	activateEvt	EM
abbrLen	PK	abortMask	EM	activMask	EM

AddPt	QD	BadBtSlpErr	ME	bothAxes	WM
AddReference	RM	BadCkSmErr	ME	botRight	QT
AddRefFailed	ME	BadDBtSlp	ME	bottom	QT
AddResFailed	ME	BadDCkSum	ME	bounds	QT
AddResMenu	MN	BadMDBerr	ME	boundsRect	DL
AddResource	RM	BadUnitErr	ME	boundsRect	DL
aDefItem	DL	baseAddr	QT	BoutRefNum	SD
AinRefNum	SD	baud1200	SD	bPatScale	PR
alarm	OU	baud1800	SD	bPort	PR
Alert	DL	baud19200	SD	BreakRecd	ME
AlertTemplate	DL	baud2400	SD	BringToFront	WM
AlertTHndl	DL	baud300	SD	bSpoolLoop	PR
AlertTPtr	DL	baud3600	SD	btnCtrl	DL
Allocate	FL	baud4800	SD	bUlOffset	PR
AllocPtr	MM	baud57600	SD	bUlShadow	PR
altDBoxProc	WM	baud600	SD	BUlThick	PR
amplitude	SN	baud7200	SD	bUser1Loop	PR
AngleFromSlop	TU	baud9600	SD	bUser2Loop	PR
AoutRefNum	SD	BdNamErr	ME	Button	EM
ApFile	OU	bDraftLoop	PR	bXInfoX	PR
app1Evt	EM	BeginUpdate	WM	Byte	MC
app1Mask	EM	bFileVers	PR		
app2Evt	EM	BinRefNum	SD	calcCRgns	CM
app2Mask	EM	BitClr	TU	CalcMenuSize	MN
app3Evt	EM	BitMap	QT	CalcVis	WM
app3Mask	EM	BitMapPtr	QT	CalcVisBehind	WM
app4Evt	EM	Bits16	QT	Cancel	DL
app4Mask	EM	BitSet	TU	CantStepErr	ME
AppendMenu	MN	bitsProc	QT	CaretTime	EM
appleSymbol	MN	BitTst	TU	CautionAlert	DL
applFont	FM	bJDocLoop	PR	century	PK
ApplicZone	MM	bJobx	PR	ChangedResour	RM
arcProc	QT	bkColor	QT	Chars	TE
arrow	MD	BkLim	MM	CharsHandle	TE
ascent	QT	bkPat	QT	CharsPtr	TE
ascent	FM	black	MD	CharWidth	QD
athens	FM	blackBit	QD	checkBoxProc	CM
autoKey	EM	blackColor	QD	CheckItem	MN
autoKeyMask	EM	BlockMove	MM	checkMark	MN
autoTrack	CM	blueBit	QD	CheckUpdate	WM
		blueColor	QD	chkCtrl	DL
BackColor	QD	bold	QT	ClearMenuBar	MN
BackPat	QD	bold	FM	ClipAbove	WM

DisposDialog	DL	DSBadLaunch	ME	EqualString	OU
DisposeControl	CM	DSBusError	ME	Erase	QD
DisPoseMenu	MN	DSChkErr	ME	EraseArc	QD
DisposeRgn	QD	DSCoreErr	ME	EraseOval	QD
DisposeWindow	WM	DSFPErr	ME	ErasePoly	QD
DisposHandle	MM	DSFSErr	ME	EraseRect	QD
DisposPtr	MM	DSIllInstErr	ME	EraseRgn	QD
DIUnload	PK	DSIOCoreErr	ME	EraseRoundRec	QD
DIVerify	PK	DSIrqErr	ME	errNum	FM
DIZero	PK	DskFulErr	ME	ErrorSound	DL
dkGray	MD	DSLineAErr	ME	errs	SD
DlgCopy	DL	DSLineFErr	ME	evenParity	SD
DlgCut	DL	DSLoadErr	ME	Event	OT
DlgDelete	DL	DSMemFullErr	ME	EventAvail	EM
DlgPaste	DL	DSMiscErr	ME	EventRecord	TT
DMY	PK	DSNotThe1	ME	everyevent	EM
DocumentProc	WM	DSOvFlowErr	ME	eveStr	PK
DoubleTime	EM	DSPrivErr	ME	EvQEl	OT
dQDrive	OT	DSReInsert	ME	evQElem	OT
dQDrvSize	OT	DSStknHeap	ME	evQType	OT
DQFSID	OT	DSSysErr	ME	EvtRecPtr	TT
dQRefNum	OT	DSTracErr	ME	evts	SD
dragCntl	CM	DSZeroDivErr	ME	extend	QT
DragControl	CM	dummyType	OT	ExtFSErr	ME
DragGrayRgn	WM	DupFNErr	ME	extra	FM
DragWindow	WM	duration	SN		
DrawChar	QD	duration	SN	face	FM
drawCntl	CM			family	FM
DrawControls	CM	editField	DL	FBsyErr	ME
DrawDialog	DL	editOpen	DL	fCTS	SD
DrawGrowIcon	WM	editText	DL	fdCreator	OT
DrawMenuBar	MN	Eject	FL	fdFlags	OT
DrawNew	WM	EmptyHandle	MM	fdFldr	OT
DrawPicture	QD	EmptyRect	QD	fdLocation	OT
DrawString	QD	EmptyRgn	QD	fdType	OT
DrawText	QD	enableFlags	MN	FeedCut	PR
DRemovErr	ME	EnableItem	MN	FeedFanFold	PR
driverEvt	EM	EndUpdate	WM	FeedMechCut	PR
driverMask	EM	Enqueue	OU	FeedOther	PR
DrvQEl	OT	EOFErr	ME	FFmode	SN
drvQElem	OT	EqualPt	QD	fFromUsr	PR
drvQType	OT	EqualRect	QD	FFSynthRec	SN
DSAddressErr	ME	EqualRgn	QD	fgColor	QT

GetPen	QD	GZSaveHnd	MM	iDevBytes	PR
GetPenState	QD			iFileVol	PR
getPicProc	QT	h	QT	iFMgrCtl	PD
GetPicture	TU	HandAndHand	OU	iFstPage	PR
GetPixel	QD	Handle	MC	iHRes	PR
GetPort	QD	HandleZone	MM	iLstPage	PR
GetPtrSize	MM	HandToHand	OU	inButton	CM
GetResAttrs	RM	hAxsOnly	WM	inCheckBox	CM
GetResFileAtt	RM	hAxisOnly	CM	inContent	WM
GetResInfo	RM	HeapData	MM	inDesk	WM
GetResource	RM	HFstFree	MM	inDownButton	CM
GetScrap	SM	HideControl	CM	inDrag	WM
getScroll	PK	HideCursor	QD	InfoScrap	SM
GetSoundVol	SN	HidePen	QD	inGoAway	WM
GetString	TU	HideWindow	WM	inGrow	WM
GetSysPPtr	OU	HiliteControl	CM	initCntl	CM
GetTime	OU	hilited	TT	InitCursor	QD
GetTrapAddres	OU	HiliteMenu	MN	InitIWMerr	ME
GetVCBQHdr	PB	HiliteWindow	WM	InitMenus	MN
GetVInfo	FL	hiLong	TU	InitPort	QD
GetVol	FL	HiWord	TU	InitQueue	PB
GetWindowPic	WM	HLock	MM	InitUtil	OU
GetWMgrPort	WM	HNoPurge	MM	InitZone	MM
GetWRefCon	WM	HomeResFile	RM	inMenuBar	WM
GetWTitle	WM	hotSpot	QT	inPageDown	CM
GetZone	MM	hour	OU	inPageUp	CM
GFPErr	ME	hPic	PR	inPort	TT
GlobalToLocal	QD	hPrint	PR	InsertMenu	MN
goAwayFlag	TT	HPurge	MM	InsertResMenu	MN
goAwayFlag	DL	hrLeadingZ	PK	InsetRect	QD
good	PK	hText	TT	InsetRgn	QD
gport	PR	HUnlock	MM	inSysWindow	WM
gProcs	PR	hwOverrunErr	SD	Int64Bit	TU
GrafDevice	QD			IntegerPtr	MC
GrafPort	QT	iBandH	PR	inThumb	CM
grafProcs	QT	iBands	PR	intl0Hndl	PK
GrafPtr	QT	iBandV	PR	intl0Ptr	PK
gray	MD	iconItem	DL	intl0Rec	PK
greenBit	QD	iCopies	PR	intl0Vers	PK
greenColor	QD	iCurBand	PR	intl1Hndl	PK
GrowWindow	WM	iCurCopy	PR	intl1Ptr	PK
GZCritical	MM	iCurPage	PR	intl1Rec	PK
GZProc	MM	iDev	PR	intl1Vers	PK

		minLeadingZ	PK	noConstraint	CM
MacBool	MC	minute	OU	NoDriveErr	ME
MacBoolPtr	MC	misc	OU	NoDtaMkErr	ME
MacPtr	MC	mntLeadingZ	PK	NoErr	ME
mac_false	MC	ModalDialog	DL	noGrowDocPro	WM
mac_true	MC	mode	SN	NoMacDskErr	ME
magentaBit	QD	mode	SN	noMark	MN
magentaColor	QD	mode	SN	NoNybErr	ME
MapPoly	QD	modifiers	TT	noParity	SD
MapPt	QD	monaco	FM	normalBit	QD
MapRect	QD	month	OU	noScrapErr	ME
MapRgn	QD	months	PK	NoteAlert	DL
mask	QT	MoreMast	MM	noteIcon	DL
MaxMem	MM	MoreMasters	MM	NotOpenErr	ME
MaxNRel	MM	mornStr	PK	notPatBic	QD
MaxRel	MM	mouseDown	EM	notPatCopy	QD
mChooseMsg	MN	mouseUp	EM	notPatOr	QD
mDownMask	EM	Move	QD	notPatXor	QD
mDrawMsg	MN	MoveControl	CM	notSrcBic	QD
MDY	PK	MovePortTo	QD	notSrcCopy	QD
memAdrErr	ME	MoveTo	QD	notSrcOr	QD
memAZErr	ME	MoveWindow	WM	notSrcXor	QD
memBCErr	ME	mSizeMsg	MN	noTypeErr	ME
MemError	MM	Munger	TU	NSDrvErr	ME
MemFullErr	ME	mUpMask	EM	NsVErr	ME
memPCErr	ME			null	SD
memPurErr	ME	needbits	FM	nullEvent	EM
memSCErr	ME	networkEvt	EM	nullMask	EM
memWZErr	ME	networkMask	EM	numer	FM
menuData	MN	NewControl	CM		
MenuHandle	MN	NewDialog	DL	ObscureCursor	QD
menuHeight	MN	NewHandle	MM	oddParity	SD
menuID	MN	NewMenu	MN	OffLinErr	ME
MenuInfo	MN	NewPtr	MM	OffsetPoly	QD
MenuKey	MN	NewRgn	QD	OffsetRect	QD
menuProc	MN	NewString	TU	OffsetRgn	QD
MenuPtr	MN	NewWindow	WM	OK	DL
MenuSelect	MN	newYork	FM	OpenDeskAcc	DS
menuWidth	MN	nextControl	CM	OpenDriver	FL
message	TT	nextWindow	TT	OpenErr	ME
metricSys	PK	NilHandleErr	ME	OpenPicture	QD
MFulErr	ME	nLines	TT	OpenPoly	QD
MinCBFree	MM	NoAdrMkErr	ME	OpenPort	QD

PrJobMerge	PR			ReleaseResour	RM
PrNoPurge	PD	QDProcs	QT	Rename	FL
procID	DL	QDProcsPtr	QT	resChanged	RM
ProcPtr	MC	QElem	OT	resCtrl	DL
PrOpen	PR	QElemPtr	OT	ResError	RM
PrOpenDoc	PR	QErr	ME	ResetAlrtStag	DL
PrOpenPage	PR	QFlags	OT	ResFNotFound	ME
PrPicFile	PR	QHdr	OT	resLocked	RM
PrPurge	PD	QHdrPtr	OT	ResNotFound	ME
PrSetError	PR	QHead	OT	resPreload	RM
prStl	PR	qLink	OT	resProtected	RM
PrStlDialog	PR	QTail	OT	resPurgeable	RM
PrValidate	PR	qType	OT	ResrvMem	MM
PRWrErr	ME	QTypes	OT	resSysHeap	RM
prXInfo	PR			resSysRef	RM
PScrapStuff	SM	radCtrl	DL	resUser	RM
Pt2Rect	QD	radioButProc	CM	RFNumErr	ME
PtInRect	QD	Random	QD	rgnBBox	QT
PtInRgn	QD	randSeed	MD	rgnProc	QT
PtrAndHand	OU	RcvrErr	ME	rgnSave	QT
PtrFFSynthRec	SN	RDIBadMount	PK	rgnSize	QT
PtrFTSndRec	SN	RDIFormat	PK	right	QT
PtrFTSynth	SN	RDILoad	PK	RIUDatePStrin	PK
PtrInt64Bit	TU	RDIUnload	PK	RIUDateString	PK
PtrSFReply	PK	RDIVerify	PK	RIUGetIntl	PK
PtrSFTypeList	PK	RDIZero	PK	RIUMagIDStrin	PK
PtrSWSynth	SN	RDocProc	WM	RIUMagString	PK
PtrToHand	OU	rdPend	SD	RIUMetric	PK
PtrToXHand	OU	ReadDateTime	OU	RIUSetIntl	PK
PtrZone	MM	ReadErr	ME	RIUTimePStrin	PK
PtToAngle	QD	RealFont	FM	RIUTimeString	PK
PurgeMem	MM	ReallocHandle	MM	RmveReference	RM
PurgeProc	MM	RecoverHandle	MM	RmveResource	RM
PurgePtr	MM	Rect	QT	RmvRefFailed	ME
pushButProc	CM	RectInRgn	QD	RmvResFailed	ME
putCancel	PK	rectProc	QT	rowBytes	QT
putDlgID	PK	RectPtr	QT	rPage	PR
putDrive	PK	RectRgn	QD	rPaper	PR
putEject	PK	redBit	QD	rRectProc	QT
putName	PK	redColor	QD	RSFGetFile	PK
putPicProc	QT	refCon	TT	RSFPGetFile	PK
putSave	PK	refCon	DL	RSFPPutFile	PK
PutScrap	SM	Region	QT	RSFPutFile	PK

RstFLock	FL	SetCtlMax	CM	SetWRefCon	WM
		SetCtlMin	CM	SetWTitle	WM
sanFran	FM	SetCtlValue	CM	SetZone	MM
SaveOld	WM	SetCursor	QD	SFGetFile	PK
ScalePt	QD	SetDAFont	DL	SFPGetFile	PK
ScanBT	PR	SetDateTime	OU	SFPPutFile	PK
ScanLR	PR	SetDItem	DL	SFPutFile	PK
ScanRL	PR	SetEmptyRgn	QD	SFReply	PK
ScanTB	PR	SetEOF	FL	SFTypeList	PK
scrapCount	SM	SetEventMask	EM	shadow	QT
scrapHandle	SM	SetFInfo	FL	shadow	FM
scrapName	SM	SetFLock	FL	ShieldCursor	TU
scrapSize	SM	SetFontLock	FM	ShortDate	PK
scrapState	SM	SetFPos	FL	ShowControl	CM
ScrapStuff	SM	SetGrowZone	MM	ShowCursor	QD
screenbits	MD	SetHandleSize	MM	ShowHide	WM
scrollBarProc	CM	SetItem	MN	ShowPen	QD
ScrollRect	QD	SetItemIcon	MN	ShowWindow	WM
secLeadingZ	PK	SetItemMark	MN	shrtDateFmt	PK
second	OU	SetItemStyle	MN	Size	MM
Secs2Date	OU	SetIText	DL	size	FM
SectNFErr	ME	SetMenuBar	MN	SizeControl	CM
SectRect	QD	SetMenuFlash	MN	SizeResource	RM
SectRgn	QD	SetOrigin	QD	SizeWindow	WM
SeekErr	ME	SetPenState	QD	SlopeFromAngl	TU
SelectWindow	WM	SetPort	QD	SmallBool	MC
selEnd	TT	SetPortBits	QD	sndRec	SN
SelIText	DL	SetPt	QD	sound1Phase	SN
selStart	TT	SetPtrSize	MM	sound1Rate	SN
SendBehind	WM	SetRect	QD	sound1Wave	SN
SerClrBrk	SD	SetRectRgn	QD	sound2Phase	SN
SerGetBuf	SD	SetResAttrs	RM	sound2Rate	SN
SerHShake	SD	SetResFileAttrs	RM	sound2Wave	SN
SerReset	SD	SetResInfo	RM	sound3Phase	SN
SerSetBrk	SD	SetResLoad	RM	sound3Rate	SN
SerSetBuf	SD	SetResPurge	RM	sound3Wave	SN
SerShk	SD	SetSoundVol	SN	sound4Phase	SN
SerStaRec	SD	SetStdProcs	QD	sound4Rate	SN
SerStatus	SD	SetString	TU	sound4Wave	SN
SetClip	QD	SetTime	OU	SoundDone	SN
SetCRefCon	CM	SetTrapAddress	OU	SpaceExtra	QD
SetCTitle	CM	SetVol	FL	spareFlag	TT
SetCtlAction	CM	SetWindowPic	WM	SparePtr	MM

SpdAdjErr	ME	Style	QT	TESetSelect	TE
spExtra	QT	StyleItem	QT	TESetText	TE
srcBic	QD	SubPt	QD	testCntl	CM
srcCopy	QD	supressDate	PK	TestControl	CM
srcOr	QD	SwapFont	FM	TEUpdate	TE
srcXor	QD	SWmode	SN	TextBox	TE
st0	PK	swOverrunErr	SD	TextFace	QD
st1	PK	SWSynthRec	SN	TextFont	QD
st2	PK	SysBeep	OU	textH	DL
st3	PK	SysParmType	OU	textMenuProc	MN
st4	PK	SysPPtr	OU	TextMode	QD
StageList	DL	SystemClick	DS	textProc	QT
stages	DL	SystemEdit	DS	TextSize	QD
StartSound	SN	SystemEvent	DS	TextWidth	QD
statText	DL	systemFont	FM	TFeed	PR
Status	FL	SystemMenu	DS	thePort	MD
StatusErr	ME	SystemTask	DS	thousSep	PK
StdArc	QD	SystemZone	MM	THPrint	PR
StdBits	QD			thumbCntl	CM
StdComment	QD	TEActivate	TE	THz	MM
StdGetPic	QD	TECalText	TE	TickCount	EM
StdLine	QD	TEClick	TE	time1Suff	PK
StdOval	QD	TECopy	TE	time2Suff	PK
StdPoly	QD	TECut	TE	time3Suff	PK
StdPutPic	QD	TEDeactivate	TE	time4Suff	PK
StdRect	QD	TEDelete	TE	time5Suff	PK
StdRgn	QD	TEDisPose	TE	time6Suff	PK
StdRRect	QD	TEGetText	TE	time7Suff	PK
StdText	QD	TEHandle	TT	time8Suff	PK
StdTxMeas	QD	TEIdle	TE	timeCycle	PK
StillDown	EM	TEInsert	TE	timeFmt	PK
stop10	SD	teJustCenter	TE	timeSep	PK
stop15	SD	teJustLeft	TE	title	DL
stop20	SD	teJustRight	TE	titleHandle	TT
StopAlert	DL	TEKey	TE	titleWidth	TT
stopIcon	DL	TENew	TE	Tk0BadErr	ME
StopSound	SN	TEPaste	TE	TMFOErr	ME
Str255	MC	TEPtr	TT	ToMacBool	MC
StringHandle	MC	TERec	TT	Tone	SN
StringPtr	MC	TEScrapHandle	OU	Tones	SN
StringWidth	QD	TEScrapLen	OU	top	QT
strucRgn	TT	TEScroll	TE	topLeft	QT
StuffHex	QD	TESetJust	TE	TopMem	MM

windowDefProc	TT
WindowHandle	TT
windowKind	TT
windowPic	TT
WindowPtr	TT
WindowRecord	TT
wInDrag	WM
wInGoAway	WM
wInGrow	WM
wNew	WM
wNoHit	WM
WPrErr	ME
WriteParam	OU
WriteResource	RM
WritErr	ME
wrPend	SD
WrPermErr	ME
WrUnderRun	ME
xoff	SD
XOFFHold	SD
XOFFSent	SD
xOffWasSent	SD
xon	SD
XorRgn	QD
year	OU
yellowBit	QD
yellowColor	QD
YMD	PK
ZCBFree	MM
ZeroScrap	SM
Zone	MM

A.3. Control Manager (ControlMgr)

```
unit ControlMgr ;

interface
{Uses Maccore, QdTypes, TBTypes}
{$L-}
uses {$U MACCORE.CODE} MacCore ,
     {$U QDTYPES.CODE} QDTypes (GrafPort, GrafPtr, Point, VHSelect,
                                FPoint, Rect, RectPtr),
     {$U TBTYPES.CODE} TBTypes (EvtRecPtr, EventRecord,
                                windowptr,windowhandle)  ;
{$L+}

const

  { Control Definition Ids }
  pushButProc   = 0 ;       { simple button }
  checkBoxProc  = 1 ;       { check box }
  radioButProc  = 2 ;       { radio button }
  useWFont      = 3 ;       { add to above window's font }
  scrollBarProc = 4 ;       { scroll bar }

  { Part Codes }
  inButton      = 10 ;      { simple button }
  inCheckBox    = 11 ;      { check box or radio button }
  inUpButton    = 20 ;      { up arrow of a scroll bar }
  inDownButton  = 21 ;      { down arrow of a scroll bar }
  inPageUp      = 22 ;      {
  inPageDown    = 23 ;      {
  inThumb       = 129 ;     { thumb of a scroll bar }

  { Axis constraints for DragControl }
  noConstraint  = 0 ;       { no constraint }
  hAxisOnly     = 1 ;       { horizontal axis only }
  vAxisOnly     = 2 ;       { vertical axis only }

  { Messages to control definition function }
  drawCntl      = 0 ;       { draw the control (or control part) }
  testCntl      = 1 ;       { test where mouse button was pressed }
  calcCRgns     = 2 ;       { calculate control's region (or indicator's) }
  initCntl      = 3 ;       { do any additional control initialization }
  dispCntl      = 4 ;       { take any additional disposal actions }
  posCntl       = 5 ;       { reposition control's indicator & update it }
  thumbCntl     = 6 ;       { calculate parameters for dragging indicator}
  dragCntl      = 7 ;       { drag control (or its indicator) }
  autoTrack     = 8 ;       { execute control's action procedure }

Type

  ControlHandle = MacPtr ;
  ControlPtr    = MacPtr ;

  ControlRecord = PACKED RECORD
    nextControl:        ControlHandle ;  { next control }
    contrlOwner:        MacPtr ;         { Pointer to control's window }
    contrlRect:         Rect ;           { enclosing rectangle }
    contrlHilite:       SmallBool ;      { highlite state }
    contrlVis:          SmallBool ;      { TRUE if visible }
    contrlValue:        integer ;        { current setting }
    contrlMin:          integer ;        { minimum setting }
    contrlMax:          integer ;        { maximum setting }
    contrlDefProc:      Handle ;         { control definition function }
    contrlData:         Handle ;         { data used by contrlDefProc }
    contrlAction:       ProcPtr ;        { default action procedure }
    contrlRfCon:        LongInt ;        { control's reference value }
    contrlTitle:        str255 ;         { control's Title }
  End ;

{ Initialization And Allocation ------------------------------------------}
```

```
FUNCTION  NewControl     (        theWindow:       WindowPtr ;
                                  boundsRect:      RectPtr ;
                                  title:           StringPtr ;
                                  visible:         MacBool ;
                                  value:           integer ;
                                  min,max:         integer ;
                                  procID:          integer ;
                                  refCon:          LongInt )
                         : ControlHandle ;         external(-22188); {A954}

FUNCTION  GetNewControl  (        controlID:       integer ;
                                  theWindow:       WindowPtr )
                         : ControlHandle ;         external(-22082); {A9BE}

PROCEDURE DisposeControl(         theControl:      ControlHandle) ;
                                                   external(-22187); {A955}

PROCEDURE KillControls   (        theWindow:       WindowPtr ) ;
                                                   external(-22186); {A956}

{ Control Display ------------------------------------------------------}

PROCEDURE SetCTitle      (        theControl:      ControlHandle ;
                                  title:           StringPtr ) ;
                                                   external(-22177); {A95F}

PROCEDURE GetCTitle      (        theControl:      ControlHandle ;
                                  title:           StringPtr ) ;
                                                   external(-22178); {A95E}

PROCEDURE HideControl    (        theControl:      ControlHandle ) ;
                                                   external(-22184); {A958}

PROCEDURE ShowControl    (        theControl:      ControlHandle ) ;
                                                   external(-22185); {A957}

PROCEDURE DrawControls   (        theWindow:       WindowPtr ) ;
                                                   external(-22167); {A969}

PROCEDURE HiliteControl  (        theControl:      ControlHandle ;
                                  hiliteState:     integer ) ;
                                                   external(-22179); {A95D}

{ Mouse Location -------------------------------------------------------}

FUNCTION  TestControl    (        theControl:      ControlHandle ;
                                  thePoint:        FPoint )
                         : integer ;               external(-22170); {A966}

FUNCTION  FindControl    (        thePoint:        FPoint ;
                                  theWindow:       WindowPtr ;
                                  whichControl:    MacPtr )
                         : integer ;               external(-22164); {A96C}

FUNCTION  TrackControl   (        theControl:      ControlHandle ;
                                  startPt:         FPoint ;
                                  actionProc:      ProcPtr )
                         : integer ;               external(-22168); {A968}

  Control Movement and Sizing ------------------------------------------}

PROCEDURE MoveControl    (        theControl:      ControlHandle ;
                                  h , v :          integer ) ;
                                                   external(-22183); {A959}

PROCEDURE DragControl    (        theControl:      ControlHandle ;
                                  startPt:         FPoint ;
                                  limitRect:       RectPtr ;
                                  slopRect:        RectPtr ;
                                  axis:            integer ) ;
                                                   external(-22169); {A967}
```

```
PROCEDURE SizeControl   (       theControl:     ControlHandle ;
                                w , h :         integer ) ;
                                external(-22180); {A95C}

{ Control Setting and Range ------------------------------------------}

PROCEDURE SetCtlValue   (       theControl:     ControlHandle ;
                                theValue:       integer ) ;
                                external(-22173); {A963}

FUNCTION  GetCtlValue   (       theControl:     ControlHandle )
                        : integer ;             external(-22176); {A960}

PROCEDURE SetCtlMin     (       theControl:     ControlHandle ;
                                minValue:       integer ) ;
                                external(-22172); {A964}

FUNCTION  GetCtlMin     (       theControl:     ControlHandle )
                        : integer ;             external(-22175); {A961}

PROCEDURE SetCtlMax     (       theControl:     ControlHandle ;
                                maxValue:       integer ) ;
                                external(-22171); {A965}

FUNCTION  GetCtlMax     (       theControl:     ControlHandle )
                        : integer ;             external(-22174); {A962}

{ Miscellaneous Utilities --------------------------------------------}

PROCEDURE SetCRefCon    (       theControl:     ControlHandle ;
                                data:           LongInt ) ;
                                external(-22181); {A95B}

FUNCTION  GetCRefCon    (       theControl:     ControlHandle )
                        : LongInt ;             external(-22182); {A95A}

PROCEDURE SetCtlAction  (       theControl:     ControlHandle ;
                                actionProc:     ProcPtr ) ;
                                external(-22165); {A96B}

FUNCTION  GetCtlAction  (       theControl:     ControlHandle )
                        : ProcPtr ;             external(-22166); {A96A}
```

A.4. Desktop Manager (DeskMgr)

```
unit DeskMgr ;

interface
{ Uses Maccore, OdTypes, TbTypes }
{$L-}
Uses {$U MACCORE.CODE} Maccore,
     {$U ODTYPES.CODE} ODTypes (Point,PointPtr, GrafPort, GrafPtr, Style,
                                Rect) ,
     {$U TBTYPES.CODE} TBTypes (EventRecord,EvtRecPtr,WindowPtr);
{$L+}

const

  cutCmd          = 0 ;    { Cut command }
  copyCmd         = 1 ;    { Copy command }
  pasteCmd        = 2 ;    { Paste command }
  undoCmd         = 3 ;    { Undo command }

{ Opening and Closing Desk Accessories ------------------------------}

FUNCTION  OpenDeskAcc   (       theAcc:         StringPtr )
                        : integer ;             external(-22090); {A9B6}

PROCEDURE CloseDeskAcc  (       refNum:         integer ) ;
                                                external(-22089); {A9B7}

{ Handling Events in Desk Accessories -------------------------------}

PROCEDURE SystemClick   (       theEvent:       EvtRecPtr ;
                                theWindow:      windowPtr ) ;
                                                external(-22093); {A9B3}

FUNCTION  SystemEdit    (       editCmd:        integer )
                        : MacBool ;             external(-22078); {A9C2}

{ Performing Periodic Tasks -----------------------------------------}

PROCEDURE SystemTask ;                          external(-22092); {A9B4}

{ Advanced Routines -------------------------------------------------}

FUNCTION  SystemEvent   (       theEvent:       EvtRecPtr )
                        : MacBool ;             external(-22094); {A9B2}

PROCEDURE SystemMenu    (       menuResult:     LongInt ) ;
                                                external(-22091); {A9B5}
```

A.5. Dialog Manager (DialogMgr)

```
unit DialogMgr ;

interface

{ Uses MacCore, ODTypes, TBTypes }
{$L-}
uses {$U MACCORE.CODE} MacCore ,
{$U ODTYPES.CODE} ODTypes (GrafPort, GrafPtr, Point, VHSelect,
                           FPoint, Rect, RectPtr),
{$U TBTYPES.CODE} TBTypes (EvtRecPtr, EventRecord,windowrecord,
                           windowptr,windowhandle,TEHandle,
                           TEPtr,TERec)  ;
{$L+}

const

  { Item Types }

  ctrlItem      = 4 ;    { add to following four constants }
  btnCtrl       = 0 ;    { standard button control }
  chkCtrl       = 1 ;    { standard check box control }
  radCtrl       = 2 ;    { standard
  resCtrl       = 3 ;    { control defined in control template }
  statText      = 8 ;    { static text }
  editText      = 16 ;   { editable text (dialog only) }
  iconItem      = 32 ;   { icon }
  picItem       = 64 ;   { QuickDraw Picture }
  userItem      = 0 ;    { application defined item (dialog only) }
  itemDisable   = 128 ;  { add to any of above to disable }

  { Item numbers of OK and Cancel buttons }
  OK            = 1 ;
  Cancel        = 2 ;

  { Resource IDs of Alert Icons }

  stopIcon      = 0 ;
  noteIcon      = 1 ;
  ctnIcon       = 2 ;

type

  DialogPtr     = MacPtr ;
  DialogPeek    = MacPtr ;
  DialogRecord = RECORD
    window:     WindowRecord ;   { dialog window }
    items:      Handle ;         { item list }
    textH:      TEHandle ;       { current edit Text item }
    editField:  integer ;        { editText item number minus 1 }
    editOpen:   integer ;        { internal use only }
    aDefItem:   integer ;        { default button number }
  End ;

  DialogTHndl   = MacPtr ;
  DialogTPtr    = MacPtr ;
  DialogTemplate = PACKED RECORD
    boundsRect: Rect ;           { becomes window's portRect }
    procID:     integer ;        { window definition ID }
    filler1:    SmallBool ;      { NOT USED }
    visible:    SmallBool ;      { TRUE if visible }
    filler2:    SmallBool ;      { NOT USED }
    goAwayFlag: SmallBool ;      { TRUE if has go away region }
    refCon:     LongInt ;        { window's reference value }
    itemsID:    integer ;        { resource ID of item list }
    title:      Str255 ;         { window's title }
  End ;

  StageList = PACKED ARRAY[1..4] of Byte ;
```

```
  AlertTHndl     = MacPtr ;
  AlertTPtr      = MacPtr ;
  AlertTemplate = RECORD
    boundsRect: Rect ;          { becomes window's portRect }
    itemsID:    integer ;       { resource ID of item list }
    stages:     StageList ;     { alert stage information }
  End ;

{ Initialization ------------------------------------------------------}

PROCEDURE ErrorSound    (       soundProc:      ProcPtr ) ;
                                                external(-22132); {A98C}

PROCEDURE SetDAFont     (       fontNum:        integer ) ;

{ Creating and Disposing of Dialogs -----------------------------------}

FUNCTION  NewDialog     (       dStorage:       MacPtr ;
                                boundsRect:     RectPtr ;
                                title:          StringPtr ;
                                visible:        MacBool ;
                                procID:         integer ;
                                behind:         windowPtr ;
                                goAwayFlag:     MacBool ;
                                refCon:         LongInt ;
                                items:          Handle )
                        : DialogPtr ;           external(-22147); {A97D}

FUNCTION  GetNewDialog  (       dialogId:       integer ;
                                wStorage:       MacPtr ;
                                behind:         WindowPtr )
                        : DialogPtr ;           external(-22148); {A97C}

PROCEDURE CloseDialog   (       theDialog:      DialogPtr ) ;
                                                external(-22142); {A982}

PROCEDURE DisposDialog  (       theDialog:      DialogPtr ) ;
                                                external(-22141); {A983}

PROCEDURE CouldDialog   (       dialogID:       integer ) ;
                                                external(-22151); {A979}

PROCEDURE FreeDialog    (       dialogID:       integer ) ;
                                                external(-22150); {A97A}

{ Handling Dialog Events ----------------------------------------------}

PROCEDURE ModalDialog   (       filterProc:     ProcPtr ;
                                ItemHit:        integerPtr ) ;
                                                external(-22127); {A991}

FUNCTION  IsDialogEvent (       theEvent:       EvtRecPtr )
                        : MacBool ;             external(-22145); {A97F}

FUNCTION  DialogSelect  (       theEvent:       EvtRecPtr ;
                                theDialog:      DialogPtr ;
                                itemHit:        integerPtr )
                        : MacBool ;             external(-22144); {A980}

PROCEDURE DlgCut        (       theDialog:      DialogPtr ) ;

PROCEDURE DlgCopy       (       theDialog:      DialogPtr ) ;

PROCEDURE DlgPaste      (       theDialog:      DialogPtr ) ;

PROCEDURE DlgDelete     (       theDialog:      DialogPtr ) ;

PROCEDURE DrawDialog    (       theDialog:      DialogPtr ) ;
                                                external(-22143); {A981}

  Invoking Alerts -----------------------------------------------------}
```

```
FUNCTION  Alert          (       alertID:        integer ;
                                 filterProc:     ProcPtr )
                         : integer ;             external(-22139); {A985}

FUNCTION  StopAlert      (       alertID:        integer ;
                                 filterProc:     ProcPtr )
                         : integer ;             external(-22138); {A986}

FUNCTION  NoteAlert      (       alertID:        integer ;
                                 filterProc:     ProcPtr )
                         : integer ;             external(-22137); {A987}

FUNCTION  CautionAlert   (       alertID:        integer ;
                                 filterProc:     ProcPtr )
                         : integer ;             external(-22136); {A988}

PROCEDURE CouldAlert     (       alertID;        integer ) ;
                                                 external(-22135); {A989}

PROCEDURE FreeAlert      (       alertID:        integer ) ;
                                                 external(-22134); {A98A}

{ Manipulating Items in Dialogs and Alerts ----------------------------}

PROCEDURE ParamText      (       param0:         StringPtr ;
                                 param1:         StringPtr ;
                                 param2:         StringPtr ;
                                 param3:         StringPtr ) ;
                                                 external(-22133); {A98B}

PROCEDURE GetDItem       (       theDialog:      DialogPtr ;
                                 itemNo:         integer ;
                                 kind:           integerPtr ;
                                 item:           Handle ;
                                 box:            RectPtr ) ;
                                                 external(-22131); {A98D}

PROCEDURE SetDItem       (       theDialog:      DialogPtr ;
                                 itemNo:         integer ;
                                 kind:           integer ;
                                 item:           Handle ;
                                 box:            RectPtr ) ;
                                                 external(-22130); {A98E}

PROCEDURE GetIText       (       item:           Handle ;
                                 text:           StringPtr ) ;
                                                 external(-22128); {A990}

PROCEDURE SetIText       (       item:           Handle ;
                                 text:           StringPtr ) ;
                                                 external(-22129); {A98F}

PROCEDURE SelIText       (       theDialog:      DialogPtr ;
                                 itemNO:         integer ;
                                 startSel:       integer ;
                                 endSel:         integer ) ;
                                                 external(-22146); {A97E}

FUNCTION  GetAlrtStage   : integer ;

PROCEDURE ResetAlrtStage ;
```

A.6. Event Manager (EventMgr)

```
unit EventMgr ;

interface

{Uses MACCORE, QDTypes, TBTypes}
{$L-}
Uses {$U MACCORE.CODE} Maccore,
     {$U QDTYPES.CODE} QDTypes (Point,PointPtr, GrafPort,GrafPtr, Rect) ,
     {$U TBTYPES.CODE} TBTypes (EventRecord,EvtRecPtr);
{$L+}

const
  { event codes }
  nullEvent      = 0 ;            { null }
  mouseDown      = 1 ;            { mouse down }
  mouseUp        = 2 ;            { mouse up }
  keyDown        = 3 ;            { key down }
  keyUp          = 4 ;            { key up }
  autoKey        = 5 ;            { auto-key }
  updateEvt      = 6 ;            { update }
  diskEvt        = 7 ;            { disk inserted }
  activateEvt    = 8 ;            { activate }
  abortEvt       = 9 ;            { abort }
  networkEvt     = 10 ;           { network }
  driverEvt      = 11 ;           { I/O driver }
  app1Evt        = 12 ;           { application-defined }
  app2Evt        = 13 ;           { application-defined }
  app3Evt        = 14 ;           { application-defined }
  app4Evt        = 15 ;           { application-defined }

  { event Masks }
  everyevent     = -1 ;           { all events }
  nullMask       = 1 ;
  mDownMask      = 2 ;
  mUpMask        = 4 ;
  keyDownMask    = 8 ;
  keyUpMask      = 16 ;
  autoKeyMask    = 32 ;
  updateMask     = 64 ;
  diskMask       = 128 ;
  activMask      = 256 ;
  abortMask      = 512 ;
  networkMask    = 1024 ;
  driverMask     = 2048 ;
  app1Mask       = 4096 ;
  app2Mask       = 8192 ;
  app3Mask       = 16384 ;
  app4Mask       = -32768 ;

 ype

  KeyMapPtr      = MacPtr ;
  KeyMap = PACKED ARRAY [1..128] of Boolean ;

  Accessing Events ----------------------------------------------------}

UNCTION  GetNextEvent  (       eventMask:      integer ;
                               theEvent:       EvtRecPtr )
                       : MacBool ;             external(-22160); {A970}

UNCTION  EventAvail    (       eventMask:      integer ;
                               theEvent:       EvtRecPtr )
                       : MacBool ;             external(-22159); {A971}

  Posting and Removing Events -----------------------------------------}

UNCTION  PostEvent     (       eventCode:      integer ;
```

```
                                eventMessage:   LongInt)
                        : integer ;
PROCEDURE FlushEvents   (       eventMask:      integer ;
                                stopMask:       integer ) ;
PROCEDURE SetEventMask  (       theMask:        integer ) ;
FUNCTION  OsEventAvail  (       theMask:        integer ;
                                theEvent:       EvtRecPtr )
                        : MacBool ;
FUNCTION  GetOsEvent    (       theMask:        integer ;
                                theEvent:       EvtRecPtr )
                        : MacBool ;
{ Reading the Mouse ------------------------------------------------------}
PROCEDURE GetMouse      (       mouseLoc:       PointPtr ) ;
                                                external(-22158); {A972}
FUNCTION  Button        : MacBool ;             external(-22156); {A974}
FUNCTION  StillDown     : MacBool ;             external(-22157); {A973}
FUNCTION  WaitMouseUp   : MacBool ;             external(-22153); {A977}
{ Miscellaneous Utilities ------------------------------------------------}
PROCEDURE GetKeys       (       k :             KeyMapPtr ) ;
                                                external(-22154); {A976}
FUNCTION  TickCount     : LongInt ;             external(-22155); {A975}
FUNCTION  DoubleTime    : LongInt ;
FUNCTION  CaretTime     : LongInt ;
```

A.7. File Manager (FileMgr)

```
unit FileMgr;

interface

{USES MacCore}
{$L-}
uses {$U MACCORE.CODE} MacCore ;
{$L+}

type

  finderInfo = RECORD
    fiType :     OsType ;          { The type of the file }
    fiCreator : OsType ;           { The creator of the file }
    fiFlags :    integer ;         { Hasbundle, Invisible, etc. }
    fiLocation : Record            { Point Location }
      fiv :      integer ;
      fih :      integer ;
    end ;
    fiFldr :     integer ;         { folder containing the file }
   End ;


{------------------- High Level File Manager Routines ----------------}

{ Accessing Volumes ------------------------------------------------}

FUNCTION  GetVInfo       (       DrvNum:          integer ;
                                 volName:         StringPtr ;
                          VAR    vRefNum:         integer ;
                          VAR    freeBytes:       LongInt)
                         : OsErr ;

FUNCTION  GetVol         (       volName:         StringPtr ;
                          VAR    vRefNum:         integer )
                         : OsErr ;

FUNCTION  SetVol         (       volName:         StringPtr ;
                                 vRefNum:         integer)
                         : OsErr ;

FUNCTION  FlushVol       (       volName:         StringPtr ;
                                 vRefNum:         integer)
                         : OsErr ;

FUNCTION  UnMountVol     (       volName:         StringPtr;
                                 vRefNum:         integer)
                         : OsErr ;

FUNCTION  Eject          (       volName:         StringPtr;
                                 vRefNum:         integer)
                         : OsErr ;

{ Changing File Contents -------------------------------------------}

FUNCTION  Create         (       filename:        Str255;
                                 vRefNum:         integer;
                                 creator:         OsType;
                                 filetype:        OsType)
                         : OsErr ;

FUNCTION  FSOpen         (       filename:        Str255;
                                 vRefNum:         integer;
                          VAR    refNum:          integer)
                         : OsErr ;

FUNCTION  FSRead         (       refNum:          integer;
                          VAR    Count:           LongInt;
```

```
                                buffPtr:        MacPtr)
                        : OsErr ;

FUNCTION  FSWrite       (       refNum:         integer;
                         VAR    Count:          LongInt;
                                buffPtr:        MacPtr)
                        : OsErr ;

FUNCTION  GetFPos       (       refNum:         integer;
                         VAR    filePos:        LongInt)
                        : OsErr ;

FUNCTION  SetFPos       (       refNum:         integer;
                                posMode:        integer;
                                posOff:         LongInt)
                        : OsErr ;

FUNCTION  GetEOF        (       refNum:         integer;
                         VAR    logEOF:         LongInt)
                        : OsErr ;

FUNCTION  SetEOF        (       refNum:         integer;
                                logEOF:         LongInt)
                        : OsErr ;

FUNCTION  Allocate      (       refNum:         integer;
                         VAR    Count:          LongInt)
                        : OsErr ;

FUNCTION  FSClose       (       refNum:         integer)
                        : OsErr ;

{ Changing Information About Files ------------------------------------}

FUNCTION  GetFInfo      (       filename:       Str255;
                                vRefNum:        integer;
                         VAR    fndrInfo:       FinderInfo)
                        : OsErr ;

FUNCTION  SetFInfo      (       filename:       Str255;
                                vRefNum:        integer;
                                fndrInfo:       FinderInfo)
                        : OsErr ;

FUNCTION  SetFLock      (       filename:       Str255;
                                vRefNum:        integer)
                        : OsErr ;

FUNCTION  RstFLock      (       filename:       Str255;
                                vRefNum:        integer)
                        : OsErr ;

FUNCTION  Rename        (       oldname:        Str255;
                                vRefNum:        integer;
                                newname:        Str255)
                        : OsErr ;

FUNCTION  FSDelete      (       filename:       Str255;
                                vRefNum:        integer)
                        : OsErr ;

{---------------- High Level Device Manager Routines -----------------}

FUNCTION  OpenDriver    (        name:           Str255 ;
                         VAR    refNum:         integer )
                        : OsErr ;

FUNCTION  CloseDriver   (       refNum:         integer )
                        : OsErr ;

FUNCTION  Control       (       refNum:         integer ;
                                csCode:         integer ;
                         VAR    csParam:  INTERFACE PACKED
```

```
                                        ARRAY[min..max:integer]
                                        OF CHAR         )
                    : OsErr ;

FUNCTION  Status    (       refNum:         integer ;
                            csCode:         integer ;
                     VAR    csParam:  INTERFACE PACKED
                                        ARRAY[min..max:integer]
                                        OF CHAR         )
                    : OsErr ;

FUNCTION  KillIO    (       refNum:         integer )
                    : OsErr ;
```

A.8. Font Manager (FontMgr)

```
unit FontMgr ;

interface

{Uses MACCORE, QDTypes}
{$L-}
Uses {$U MACCORE.CODE} Maccore,
     {$U QDTYPES.CODE} QDTypes (Style, Point) ;
{$L+}

const
  { Font Numbers }
  systemFont     = 0 ;          { System Font }
  applFont       = 1 ;          { application Font }
  newYork        = 2 ;
  geneva         = 3 ;
  monaco         = 4 ;
  venice         = 5 ;
  london         = 6 ;
  athens         = 7 ;
  sanFran        = 8 ;
  toronto        = 9 ;

type
  FMInPtr     = MacPtr ;
  FMOutPtr    = MacPtr ;

  FMInput = PACKED RECORD
    family:       integer ;        { font number }
    size:         integer ;        { font size }
    needbits:     smallbool ;      { TRUE if drawing }
    face:         style ;          { Character style }
    device:       integer ;        { device number }
    numer:        Point ;          { numerators of scaling factors }
    denom:        Point ;          { denominators of scaling factors }
  End ;

  FMOutput = PACKED RECORD
    errNum:       integer;         { not Used }
    fontHandle:   Handle ;         { handle to font record }
    italic:       byte ;           { italic factor }
    bold:         byte ;           { bold factor }
    ulShadow:     byte ;           { underline shadow }
    ulOffset:     byte ;           { underline offset }
    shadow:       byte ;           { shadow factor }
    ulThick:      byte ;           { underline thickness }
    ascent:       byte ;           { ascent }
    extra:        byte ;           { width of Style }
    widmax:       byte ;           { maximum character width }
    descent:      byte ;           { descent }
    unused:       byte ;
    leading:      byte ;           { leading }
    numer:        Point ;          { numerators of scaling factors }
    denom:        Point ;          { denominators of scaling factors }
  End ;

{ Getting Font Information ---------------------------------------------}

PROCEDURE GetFontName    (        fontNum:        integer ;
                                  theName:        StringPtr ) ;
                                                  external(-22273); {A8FF}

PROCEDURE GetFNum        (        fontName:       StringPtr ;
                                  theNum:         integerPtr ) ;
                                                  external(-22272); {A900}

FUNCTION  RealFont       (        fontNum:        integer ;
                                  size:           integer )
```

```
                        : MacBool ;              external(-22270); {A902}

{ Keeping Fonts in Memory ------------------------------------------------}

PROCEDURE  SetFontLock  (       lockflag:        MacBool ) ;
                                                 external(-22269); {A903}

{ Advanced Routine -------------------------------------------------------}

FUNCTION   SwapFont     (       inRec:           FMInPtr )
                        : FMOutPtr ;             external(-22271); {A901}
```

A.9. Global Types (MacCore)

```
unit MacCore;

interface

const
  abs_nil = 0 ;                    { nil value associated with MacPtr }
  mac_true = 256 ;
  mac_false = 0 ;

type
  { General purpose declarations for use with Macintosh O.S. interface }
  { procedures.                                                         }

  MacPtr =        integer2;
  Handle =        integer2;
  ProcPtr =       integer2;

  MacBool =       integer;
  SmallBool =     0..255;
  MacBoolPtr =    MacPtr ;

  LongInt =       integer2;
  LongIntPtr =    MacPtr ;

  Str255 =        string[255];
  StringPtr =     integer2;
  StringHandle =Handle ;

  IntegerPtr =    MacPtr ;

  Byte =          0..255;
  OsErr =         integer ;

  { OsType is the basic 4 character identifier used by many Macintosh   }
  { Facilities.  OsTypePtr is for passing VAR parameter addresses.      }
  { FOsType is for passing VALUE parameters.                            }

  OsTypePtr =     MacPtr ;
  FOsType=        integer2 ;
  OsType = RECORD
    case boolean of
      true  : ( c : PACKED ARRAY [1..4] OF CHAR ) ;
      false : ( p : FOsType ) ;
  End ;


{ Functions for conversion between ToolBox data representation and      }
{ UCSD Pascal data representation.                                      }

Function ToMacBool ( ub : Boolean ) : MacBool ;
Function FrMacBool ( mb : MacBool ) : Boolean ;
Function ToSmall   ( ub : Boolean ) : SmallBool ;
Function FrSmall   ( mb : SmallBool ) : Boolean ;
```

A.10. Global Data (MacData)

```
unit MacData ;

interface

{Uses MacCore, QdTypes}
{$L-}
Uses {$U MacCore.Code} MacCore ,
     {$U QdTypes.Code} QdTypes
                       (GrafPtr, PatternPtr, CursorPtr, BitMapPtr) ;
{$Lt}


Var
  thePort :      GrafPtr  ;        { pointer to quickdraw default port }
  white :        PatternPtr ;      { pointer to white pen pattern }
  black :        PatternPtr ;      { pointer to black pen pattern }
  gray :         PatternPtr ;      { pointer to gray pen pattern }
  ltGray :       PatternPtr ;      { pointer to light gray pen pattern }
  dkGray :       PatternPtr ;      { pointer to dark gray pen pattern }
  arrow :        CursorPtr ;       { pointer to arrow cursor }
  screenbits :   BitMapPtr ;       { pointer to screen bitmap }
  randSeed :     LongIntPtr ;      { pointer to random function seed }
  A5 :           MacPtr ;          { Register A5 value }
```

A.11. Error Codes (MacErrors)

```
unit macerrors ;

interface
{ Macintosh Error Codes }

const

  { General System Errors }

  NoErr          = 0 ;      { no error occurred }
  QErr           = -1 ;     { queue element not found during deletion }
  VTypErr        = -2 ;     { invalid queue element }
  CorErr         = -3 ;     { core routine number out of range }
  UnimpErr       = -4 ;     { unimplemented core routine }

  { I/O System Errors }

  ControlErr     = -17 ;
  StatusErr      = -18 ;
  ReadErr        = -19 ;
  WritErr        = -20 ;
  BadUnitErr     = -21 ;
  UnitEmptyErr   = -22 ;
  OpenErr        = -23 ;
  ClosErr        = -24 ;
  DRemovErr      = -25 ;    { Tried to remove an open driver }
  DInstErr       = -26 ;    { DrvrInstall couldn't find driver in resources }
  AbortErr       = -27 ;    { I/O call aborted by KillIO }
  NotOpenErr     = -28 ;    { driver not opened }

  { File System Errors }

  DirFulErr      = -33 ;    { directory full }
  DskFulErr      = -34;     { disk full }
  NsVErr         = -35 ;    { no such volume }
  IOErr          = -36 ;    { I/O Error }
  BdNamErr       = -37 ;    { bad name }
  FNOpnErr       = -38 ;    { File not open }
  EOFErr         = -39 ;    { End of File }
  PosErr         = -40 ;    { tried to position before start of file }
  MFulErr        = -41 ;    { memory too full to load file }
  TMFOErr        = -42 ;    { too many files open }
  FNFErr         = -43 ;    { File not found }
  WPrErr         = -44 ;    { diskette is write protected }
  FLckdErr       = -45 ;    { file is locked }
  VLckdErr       = -46 ;    { volume is locked }
  FBsyErr        = -47 ;    { file is busy }
  DupFNErr       = -48 ;    { duplicate file name }
  OpWrErr        = -49 ;    { file already open with write permission }
  ParamErr       = -50 ;    { error in user parameter list }
  RFNumErr       = -51 ;    { refnum error }
  GFPErr         = -52 ;    { get file position error }
  VolOffLinErr   = -53 ;    { volume not on line (was Ejected) }
  PermErr        = -54 ;    { permissions error (during file open) }
  VolOnLinErr    = -55 ;    { drive volume already on-line at MountVol }
  NSDrvErr       = -56 ;    { no such drive }
  NoMacDskErr    = -57 ;    { not a macintosh diskette }
  ExtFSErr       = -58 ;    { volume belongs to an external file system }
  FSDSErr        = -59 ;    { during rename old entry was deleted but
                              not be restored                          }
  BadMDBerr      = -60 ;    { bad master directory block }
  WrPermErr      = -61 ;    { write permissions error}

  { Disk, Serial Ports, and Clock specific errors }

  NoDriveErr     = -64 ;    { drive not installed }
  OffLinErr      = -65 ;    { r/w request for an offline drive }
  NoNybErr       = -66 ;    { couldn't find 5 nybbles in 200 tries }
```

```
NoAdrMkErr      = -67 ;     { couldn't find valid address mark }
DataVerErr      = -68 ;     { read verify compare failed }
BadCkSmErr      = -69 ;     { address mark checksum didn't check }
BadBtSlpErr     = -70 ;     { bad addr mark bit slip nibbles }
NoDtaMkErr      = -71 ;     { couldn't find a data mark header }
BadDCkSum       = -72 ;     { bad data mark checksum }
BadDBtSlp       = -73 ;     { bad data mark bit slip nibbles }
WrUnderRun      = -74 ;     { write underrun occured }
CantStepErr     = -75 ;     { step handshake failed }
Tk0BadErr       = -76 ;     { track 0 doesn't detect change }
InitIWMerr      = -77 ;     { unable to initialize IWM }
TwoSideErr      = -78 ;     { tried to read 2nd side on 1 side drive }
SpdAdjErr       = -79 ;     { unable to correctly adjust disk speed }
SeekErr         = -80 ;     { track number wrong on address mark }
SectNFErr       = -81 ;     { sector number never found on a track }

ClkRdErr        = -85 ;     { unable to read same clock value twice }
ClkWrErr        = -86 ;     { time written did not verify }
PRWrErr         = -87 ;     { parameter ram didn't read-verify }
PRInitErr       = -88 ;     { InitUtil found the param ram unitialized }

RcvrErr         = -89 ;     { SCC Receiver error }
BreakRecd       = -90 ;     { Break received }

{ Memory Manager Errors }

MemFullErr      = -108 ;    { not enough room in heap zone }
NilHandleErr    = -109 ;    { Handle was Nil in Handle Zone }
memWZErr        = -111 ;    { WhichZone failed (applied to free block) }
memPurErr       = -112 ;    { block was locked or non-purgable }
memAdrErr       = -110 ;    { address was odd or out of range }
memAZErr        = -113 ;    { Address in zone check failed }
memPCErr        = -114 ;    { Pointer check failed }
memBCErr        = -115 ;    { Block Check Failed }
memSCErr        = -116 ;    { size check failed }

{ Resource Manager Errors }

ResNotFound     = -192 ;    { Resource not found }
ResFNotFound    = -193 ;    { Resource file not found }
AddResFailed    = -194 ;    { Addresource failed }
AddRefFailed    = -195 ;    { Addreference failed }
RmvResFailed    = -196 ;    { RmveResource failed }
RmvRefFailed    = -197 ;    { RmveReference failed }

{ Scrap Manager Errors }

noScrapErr      = -100 ;    { No scrap exists }
noTypeErr       = -102 ;    { No object of that type in scrap }

{ APPLICATION CODE ERRORS FROM -1024 TO -4095 }

{ Dead System Alert Identifiers }

DSSysErr        = 32767 ;   { general system error }
DSBusError      = 1 ;       { bus error }
DSAddressErr    = 2 ;       { Address error }
DSIllInstErr    = 3 ;       { illegal instruction error }
DSZeroDivErr    = 4 ;       { divide by zero error }
DSChkErr        = 5 ;       { check trap error }
DSOvFlowErr     = 6 ;       { overflow trap error }
DSPrivErr       = 7 ;       { privilage violation error }
DSTracErr       = 8 ;       { trace mode error }
DSLineAErr      = 9 ;       { line 1010 trap error }
DSLineFErr      = 10 ;      { line 1111 trap error }
DSMiscErr       = 11 ;      { miscellaneous hardware exception error }
DSCoreErr       = 12 ;      { unimplemented core routine error}
DSIrqErr        = 13 ;      { uninstalled interrupt error }
DSIOCoreErr     = 14 ;      { I/O Core Error }
DSLoadErr       = 15 ;      { Segment Loader error }
DSFPErr         = 16 ;      { Floating Point error }

DSMemFullErr    = 25 ;      { out of memory }
```

```
DSBadLaunch     = 26 ;      { can't launch file }

DSStknHeap      = 28 ;      { stack has moved into application heap }
DSFSErr         = 27 ;      { file system map has been trashed }
DSReInsert      = 30 ;      { request user to reinsert off-line volume }
DSNotThe1       = 31 ;      { not the disk I wanted }
```

A.12. Memory Manager (MemoryMgr)

```
unit MemoryMgr;

interface
{Uses MacCore}
{$L-}
uses {$U MACCORE.CODE} MacCore;
{$L+}

type
  Size = LongInt;

  THz  = MacPtr ;              { Points to a Zone Record }
  Zone = RECORD
    BkLim:       MacPtr;
    PurgePtr:    MacPtr;
    HFstFree:    MacPtr;
    ZCBFree:     LongInt;
    GZProc:      ProcPtr;
    MoreMast:    integer;
    Flags:       integer;
    CntRel:      integer;
    MaxRel:      integer;
    CntNRel:     integer;
    MaxNRel:     integer;
    CntEmpty:    integer;
    CntHandles:  integer;
    MinCBFree:   LongInt;
    PurgeProc:   ProcPtr;
    SparePtr:    MacPtr;
    AllocPtr:    MacPtr;
    HeapData:    integer;
  End;

{ Initialization and Allocation ------------------------------------------}

PROCEDURE MoreMasters ;

PROCEDURE InitZone       (    growProc : ProcPtr ;
                              masterCount : integer ;
                              limitPtr, StartPtr : MacPtr ) ;

{ Heap Zone Access -------------------------------------------------------}

PROCEDURE SetZone        (    hz : THz ) ;
FUNCTION  GetZone        : THz;
FUNCTION  SystemZone     : THz;
FUNCTION  ApplicZone     : THz;

  Allocating and Releasing Relocatable Blocks ------------------------------}

FUNCTION  NewHandle      (    byteCount: Size)
                         : Handle;

PROCEDURE DisposHandle   (    g:         Handle);

FUNCTION  GetHandleSize  (    h:         Handle)
                         : Size;

PROCEDURE SetHandleSize  (    h:         Handle;
                              newSize:   Size);

FUNCTION  HandleZone     (    h:         Handle)
                         : MacPtr;

FUNCTION  RecoverHandle  (    p:         MacPtr)
                         : Handle;
```

```
PROCEDURE ReallocHandle (     h:          Handle;
                              byteCount: Size);

{ Allocating and Releasing Nonrelocatable Blocks--------------------------}

FUNCTION  NewPtr         (    byteCount: Size)
                         : MacPtr;

PROCEDURE DisposPtr      (    p:         MacPtr);

FUNCTION  GetPtrSize     (    p:         MacPtr)
                         : Size;

PROCEDURE SetPtrSize     (    p:         MacPtr;
                              newSize:   Size);

FUNCTION  PtrZone        (    p:         MacPtr)
                         : MacPtr;

{ Freeing Space on the Heap ----------------------------------------------}

FUNCTION  FreeMem        : LongInt;

FUNCTION  MaxMem         (VAR grow:      Size)
                         : Size;

FUNCTION  CompactMem     (    cbNeeded:  Size)
                         : Size;

PROCEDURE ResrvMem       (    cbNeeded:  Size);
PROCEDURE PurgeMem       (    cbNeeded:  Size);
PROCEDURE EmptyHandle    (    h:         Handle);

{ Properties of Relocatable Blocks ---------------------------------------}

PROCEDURE HLock          (    h:         Handle);
PROCEDURE HUnlock        (    h:         Handle);
PROCEDURE HPurge         (    h:         Handle);
PROCEDURE HNoPurge       (    h:         Handle);

{ Grow Zone Functions ----------------------------------------------------}

PROCEDURE SetGrowZone    (    growZone:  ProcPtr);
FUNCTION  GZCritical     : Boolean;
FUNCTION  GZSaveHnd      : Handle;

{ Utility Routines -------------------------------------------------------}

PROCEDURE BlockMove      (    srcPtr,
                              destPtr:   MacPtr;
                              byteCount: Size);

FUNCTION  TopMem         : MacPtr;
FUNCTION  MemError       : integer;
```

A.13. Menu Manager (MenuMgr)

```
unit MenuMgr ;

interface

{Uses MacCore, ODTypes}
{$L-}
Uses {$U MACCORE.CODE} MacCore ,
     {$U ODTYPES.CODE} ODTypes (FPoint, style) ;
{$Lt}

const

  noMark         = 0 ;
  checkMark      = 18 ;
  appleSymbol    = 20 ;

  mDrawMsg       = 0 ;    { draw the menu }
  mChooseMsg     = 1 ;    { tell which item was chosen and hilite it }
  mSizeMsg       = 2 ;    { calculate the menu's dimensions }

  textMenuProc   = 0 ;

type

  MenuPtr        = MacPtr ;
  MenuHandle     = Handle ;

  MenuInfo = RECORD
    menuID:              integer ;
    menuWidth:           integer ;
    menuHeight:          integer ;
    menuProc:            Handle ;
    enableFlags:         PACKED ARRAY [0..31] OF Boolean ;
    menuData:            Str255 ;
  End ;

{ Initialization and Allocation ---------------------------------------}

PROCEDURE InitMenus ;                                external(-22224); {A930}

FUNCTION  NewMenu        (       menuId:             integer ;
                                 menuTitle:          StringPtr )
                         : MenuHandle ;              external(-22223); {A931}

FUNCTION  GetMenu        (       menuId:             integer )
                         : MenuHandle ;              external(-22081); {A9BF}

PROCEDURE DisPoseMenu    (       menu:               MenuHandle ) ;
                                                     external(-22222); {A932}

PROCEDURE AppendMenu     (       menu:               MenuHandle ;
                                 data:               StringPtr ) ;
                                                     external(-22221); {A933}

PROCEDURE AddResMenu     (       menu:               MenuHandle ;
                                 theType:            FOSType ) ;
                                                     external(-22195); {A94D}

PROCEDURE InsertResMenu  (       menu:               MenuHandle ;
                                 theType:            FOSType ;
                                 afterItem:          integer ) ;
                                                     external(-22191); {A951}

{ Forming the Menu Bar ------------------------------------------------}

PROCEDURE InsertMenu     (       menu:               MenuHandle ;
                                 beforeID:           integer ) ;
                                                     external(-22219); {A935}
```

```
PROCEDURE DrawMenuBar ;                                    external(-22217); {A937}

PROCEDURE DeleteMenu    (       menuID:         integer ) ;
                                                external(-22218); {A936}

PROCEDURE ClearMenuBar  ;                                  external(-22220); {A934}

FUNCTION  GetNewMBar    (       menuBarID:      integer )
                        : Handle ;                 external(-22080); {A9C0}

FUNCTION  GetMenuBar    : Handle ;                 external(-22213); {A93B}

PROCEDURE SetMenuBar    (       menuBar:        Handle ) ;
                                                external(-22212); {A93C}

{ Choosing from a Menu ------------------------------------------------}

FUNCTION  MenuSelect    (       startPt:        FPoint )
                        : LongInt ;             external(-22211); {A93D}

FUNCTION  MenuKey       (       ch:             Char )
                        : LongInt ;             external(-22210); {A93E}

PROCEDURE HiliteMenu    (       menuID:         integer) ;
                                                external(-22216); {A938}

{ Controlling Items' Appearance ---------------------------------------}

PROCEDURE SetItem       (       menu:           MenuHandle  ;
                                item:           integer ;
                                itemString:     StringPtr ) ;
                                                external(-22201); {A947}

PROCEDURE GetItem       (       menu:           MenuHandle  ;
                                item:           integer ;
                                itemString:     StringPtr ) ;
                                                external(-22202); {A946}

PROCEDURE DisableItem   (       menu:           MenuHandle ;
                                item:           integer ) ;
                                                external(-22214); {A93A}

PROCEDURE EnableItem    (       menu:           MenuHandle ;
                                item:           integer ) ;
                                                external(-22215); {A939}

PROCEDURE CheckItem     (       menu:           MenuHandle ;
                                item:           integer ;
                                checked:        MacBool ) ;
                                                external(-22203); {A945}

PROCEDURE SetItemIcon   (       menu:           MenuHandle ;
                                item:           integer ;
                                iconNum:        integer ) ;
                                                external(-22208); {A940}

PROCEDURE GetItemIcon   (       menu:           MenuHandle ;
                                item:           integer ;
                                iconNum:        integerPtr ) ;
                                                external(-22209); {A93F}

PROCEDURE SetItemStyle  (       menu:           MenuHandle ;
                                item:           integer ;
                                chStyle:        style ) ;
                                                external(-22206); {A942}

PROCEDURE GetItemStyle  (       menu:           MenuHandle ;
                                item:           integer ;
                                chStyle:        integerPtr ) ;
                                                external(-22207); {A941}

PROCEDURE SetItemMark   (       menu:           MenuHandle ;
```

```
                                item:           integer ;
                                markChar:       char ) ;
                                                external(-22204); {A944}

PROCEDURE GetItemMark   (       menu:           MenuHandle ;
                                item:           integer ;
                                markChar:       MacPtr ) ;
                                                external(-22205); {A943}

{ Miscellaneous Utilities ------------------------------------------------}

PROCEDURE SetMenuFlash  (       menu:           MenuHandle ;
                                flashCount:     integer ) ;
                                                external(-22198); {A94A}

PROCEDURE CalcMenuSize  (       menu:           MenuHandle ) ;
                                                external(-22200); {A948}

FUNCTION  CountMItems   (       menu:           MenuHandle )
                        : integer ;             external(-22192); {A950}

FUNCTION  GetMHandle    (       menuID:         integer )
                        : MenuHandle ;          external(-22199); {A949}

PROCEDURE FlashMenuBar  (       menuID:         integer ) ;
                                                external(-22196); {A94C}
```

A.14. Operating System Types (OsTypes)

```
unit OsTypes;

interface

{Uses MacCore, ODTypes, TBTypes}
{$L-}
uses {$U MACCORE.CODE} MacCore,
     {$U ODTYPES.CODE} ODTypes (Point,VHSelect,GrafPort,GrafPtr,Rect),
     {$U TBTYPES.CODE} TBTypes (EventRecord) ;
{$L+}

type
  QElemPtr = MacPtr ;
  QHdrPtr = MacPtr ;
  ParmBlkPtr = MacPtr ;

  FInfo = RECORD
    fdType : OsType ;              { The type of the file }
    fdCreator : OsType ;           { The creator of the file }
    fdFlags : integer ;            { hasbundle, invisible, etc. }
    fdLocation : Point ;           { file's location in the folder }
    fdFldr : integer ;             { folder containing the file }
  end ;

  DrvQEl = RECORD
    qLink : QElemPtr ;
    qType : INTEGER ;
    dQDrive : INTEGER ;
    dQRefNum : INTEGER ;
    DQFSID : INTEGER ;
    dQDrvSize : INTEGER ;
  End ;

  VCB = RECORD
    qLink:       QElemPtr ;        { next queue entry }
    qType:       integer ;         { not used }
    vcbFlags:    integer ;         { bit 15=1 if dirty }
    vcbSigWord:  integer ;         { always Hex D2D7 }
    vcbCrDate:   LongInt ;         { date volume was initialized }
    vcbLsBkUp:   LongInt ;         { date of last backup }
    vcbAtrb:     integer ;         { volume attributes }
    vcbNmFls:    integer ;         { number of files in directory }
    vcbDirSt:    integer ;         { directory's first block }
    vcbBlLn:     integer ;         { length of file directory }
    vcbNmBlks:   integer ;         { number of allocation blocks }
    vcbAlBlkSiz:LongInt ;          { size of allocation blocks }
    vcbClpSiz:   LongInt ;         { number of bytes to allocate }
    vcbAlBlSt:   integer ;         { first block in block map }
    vcbNxtFNum:  LongInt ;         { next unused file number }
    vcbFreeBks:  integer ;         { number of unused blocks }
    vcbVN:       String[27] ;      { volume name }
    vcbDrvNum:   integer ;         { drive number }
    vcbDRefNum:  integer ;         { device reference number }
    vcbFSID:     integer ;         { file system identifier }
    vcbVRefNum:  integer ;         { volume reference number }
    vcbMAdr:     MacPtr ;          { location of block map }
    vcbBufAdr:   MacPtr ;          { location of volume buffer }
    vcbMLen:     integer ;         { number of bytes in block map }
    vcbDirIndex:integer ;          { used internally }
    vcbDirBlk:   integer ;         { used internally }
  End ;

  VBLTask = RECORD
    qLink:       QElemPtr ;        { next queue entry }
    qType:       integer ;         { queue type }
    vblAddr:     ProcPtr ;         { task address }
    vblCount:    integer ;         { task frequency }
    vblPhase:    integer ;         { task phase }
```

```
End ;

EvQEl = RECORD
  qLink:      QElemPtr ;      { next queue entry }
  qType:      integer ;       { queue type }
  Event:      EventRecord ;   { event record description }
End ;


ParamBlkType = (IoParam, FileParam, VolumeParam, CntrlParam) ;

ParamBlockRec = PACKED RECORD
  { 12 BYTE header used by the file and I/O system }
  qLink : QElemPtr ;          { queue link in header }
  qType : integer ;           { type byte for safety check }
  ioTrap : integer ;          { FS: the Trap }
  ioCmdAddr : MacPtr ;        { FS: address to dispatch to }

  { Common head to all variants }
  ioCompletion : ProcPtr ;    { completion routine addr }
  ioResult : OsErr ;          { result code }
  ioNamePtr : StringPtr;      { pointer to Vol:filename string }
  ioVRefNum : integer ;       { volume reference number }

  { different components for the different types of parameter blocks }
  Case ParamBlkType OF
  ioParam :
    (ioRefNum : integer ;     { refnum for I/O operation }
     ioPermssn : Byte ;       { Open : Permissions }
     ioVersNum : Byte ;       { version number }
     ioMisc : MacPtr ;        { Rename : new name }
                              { GetEOF,SetEOF : logical end of file }
                              { Open : optional ptr to buffer }
                              { SetFileType : new type }
     ioBuffer : MacPtr ;      { data buffer ptr }
     ioReqCount : LongInt ;   { requested byte count }
     ioActCount : LongInt ;   { actual byte count completed }
     ioPosMode : integer ;    { initial file positioning }
     ioPosOffset : LongInt    { file position offset }
    );
  FileParam :
    (ioFRefNum : integer ;    { reference number for file operation }
     filler1 : Byte ;
     ioFVersNum : Byte ;      { version number }
     ioFDirIndex : integer ;  { GetFileInfo directory index }
     ioFlVersNum : Byte ;     { File version number }
     ioFlAttrib : Byte ;      { GetFileInfo: in-use bit=7, lock bit=7}
     ioFlFndrInfo : FInfo ;   { Finder Info }
     ioFlNum : LongInt ;      { GetFileInfo : File Number }
     ioFlStBlk : integer ;    { start file block (0 if none)}
     ioFlLgLen : LongInt ;    { logical length (Eof) }
     ioFlPyLen : LongInt ;    { physical length }
     ioFlRStBlk : integer ;   { Start block of resource fork }
     ioFlRLgLen : LongInt ;   { file logical length of resource fork }
     ioFlRPyLen : LongInt ;   { file physical length of rsrc fork }
     ioFlCrDat : LongInt ;    { file creation time & date }
     ioFlMdDat : LongInt      { last modified time & date }
    );
  VolumeParam :
    (filler2 : LongInt ;
     ioVolIndex : integer ;   { volume index number }
     ioVCrDate : LongInt ;    { creation date and time }
     ioVLsBkUp : LongInt ;    { last backup date and time }
     ioVAtrb : integer ;      { volume attribute }
     ioVNmFls : integer ;     { number of files in directory }
     ioVDirSt : integer ;     { start block of directory }
     ioVBlLn : integer ;      { GetVolInfo : length of dir in blocks}
     ioVNmAlBlks : integer ;  { GetVolInfo : # blks (of alloc size) }
     ioVAlBlkSiz : LongInt ;  { GetVolInfo : alloc blk byte size }
     ioVClpSiz : LongInt ;    { GetVolInfo : #bytes in one alloc }
     ioAlBlSt : integer ;     { starting disk block in block map }
     ioVNxtFNum : LongInt ;   { GetVolInfo : next free file # }
     ioVFrBlk : integer       { GetVolInfo : # free blks for the vol }
```

```
      );
  CntrlParam:
    (filler3 : integer ;
     CSCode : integer ;          { word specifying status operation }
     CSParam : integer           { device control or status parameter }
    );
End ; { ParamBlockRec }

QHdr = RECORD
  QFlags : integer ;             { Miscellaneous Flags }
  QHead : QElemPtr ;             { first element on queue }
  QTail : QElemPtr ;             { last element on queue }
End ; { QHdr }

QTypes =
  ( dummyType ,
    vType ,                      { vertical retrace queue type }
    ioQType ,                    { I/O request queue type }
    drvQType ,                   { drive queue type }
    evQType ,                    { event queue type }
    fsQType ) ;                  { volume control block queue type }

QElem = RECORD
  CASE QTypes of
    vType :    ( vblQElem:      VBLTask ) ;
    ioQType : ( ioQElem:        ParamBlockRec ) ;
    drvQType :( drvQElem:       DrvQEl ) ;
    evQType : ( evQElem:        EvQEl ) ;
    fsQType : ( vcbQElem:       VCB )
End ;
```

A.15. Operating System Utilities (OsUtilities)

```
unit OsUtilities ;

interface

{$L-}
uses {$U MACCORE.CODE} MacCore,
     {$U QDTYPES.CODE} QDTypes (Point,VHSelect,GrafPort,GrafPtr,Rect),
     {$U TBTYPES.CODE} TBTypes (EventRecord) ,
     {$U OSTYPES.CODE} OSTypes (QElemPtr,QHdrPtr) ;
{$L+}

type

  SysParmType = RECORD
    valid:      LongInt ;        { validity status }
    portA:      integer ;        { modem port (
    portB:      integer ;        { printer port (
    alarm:      LongInt ;        { alarm setting }
    font:       integer ;        { default application font }
    kbdPrint:   integer ;        { auto-key thresh/rate; printer's port }
    volClick:   integer ;        { vol level; dbl-click/caret blink }
    misc:       integer ;        { mouse scaling; boot disk; menu blink }
  End ;

  SysPPtr = MacPtr ;

  DateTimeRec = RECORD
    year:       integer ;        { four-digit year }
    month:      integer ;        { 1 to 12 for January through December }
    day:        integer ;        { 1 to 31 }
    hour:       integer ;        { 0 to 23 }
    minute:     integer ;        { 0 to 59 }
    second:     integer ;        { 0 to 59 }
    dayOfWeek:  integer ;        { 1 to 7 for Sunday through Saturday }
  End ;

  ApFile = RECORD
    fvrefnum:   integer ;        { volume reference number }
    ftype:      OsType ;         { type of file }
    fversion:   integer ;        { version # in high byte }
    fname:      Str255 ;         { file name }
  End ;

{ Pointer and Handle Manipulation ---------------------------------------}

FUNCTION  HandToHand     ( VAR   theHndl:        Handle )
                         : OsErr ;

FUNCTION  PtrToHand      (       srcPtr:         MacPtr ;
                           VAR   dstHndl:        Handle ;
                                 size:           LongInt )
                         : OsErr ;

FUNCTION  PtrToXHand     (       srcPtr:         MacPtr ;
                                 dstHndl:        Handle ;
                                 size:           LongInt )
                         : OsErr ;

FUNCTION  HandAndHand    (       aHndl, bHndl :  Handle )
                         : OsErr ;

FUNCTION  PtrAndHand     (       ptr:            MacPtr ;
                                 hndl:           Handle ;
                                 size:           LongInt )
                         : OsErr ;


  String Comparison -------------------------------------------------------}
```

```
FUNCTION  EqualString   (       aStr, bStr:     StringPtr ;
                                caseSens, diacSens: Boolean )
                        : MacBool ;

PROCEDURE UprString     (       theString:      StringPtr ;
                                diacSens:       Boolean ) ;

{ Date and Time Operations ------------------------------------------}

FUNCTION  ReadDateTime  ( VAR   secs:           LongInt )
                        : OsErr ;

FUNCTION  SetDateTime   (       secs:           LongInt )
                        : OsErr ;

PROCEDURE Date2Secs     ( VAR   date:           DateTimeRec ;
                          VAR   secs:           LongInt ) ;

PROCEDURE Secs2Date     (       secs:           LongInt ;
                          VAR   date:           DateTimeRec ) ;

PROCEDURE GetTime       ( VAR   date:           DateTimeRec ) ;

PROCEDURE SetTime       ( VAR   date:           DateTimeRec ) ;

{ Parameter RAM Operations ------------------------------------------}

FUNCTION  InitUtil      : OsErr ;

FUNCTION  GetSysPPtr    : SysPPtr ;

FUNCTION  WriteParam    : OsErr ;

{ Queue Manipulation ------------------------------------------------}

PROCEDURE Enqueue       (       qElement :      QElemPtr ;
                                theQ :          QHdrPtr ) ;

FUNCTION  Dequeue       (       qElement :      QElemPtr ;
                                theQ :          QHdrPtr )
                        : OsErr ;

{ Dispatch Table Utilities ------------------------------------------}

PROCEDURE SetTrapAddress(       trapAddr:       LongInt ;
                                trapNum:        integer ) ;

FUNCTION  GetTrapAddress(       trapNum:        integer )
                        : LongInt ;

{ TextEdit / Scrap Utilities ----------------------------------------}

FUNCTION  TEScrapHandle : Handle ;

FUNCTION  TEScrapLen    : integer ;

{ Finder Interface Utilities ----------------------------------------}

PROCEDURE CountAppFiles ( VAR   Message:        integer ;
                          VAR   Count:          integer ) ;

PROCEDURE ClrAppFiles   (       index:          integer ) ;

PROCEDURE GetAppFiles   (       index:          integer ;
                          VAR   theFile:        ApFile  ) ;

{ Miscellaneous Utilities -------------------------------------------}

PROCEDURE Delay         (       numTicks:       LongInt ;
                          VAR   finalTicks:     LongInt ) ;

PROCEDURE SysBeep       (       duration:       integer ) ;
```

```
                                                    external (-22072) ; {A9C8}

PROCEDURE GetIndString  ( VAR   theString:      Str255 ;
                                StrListId:      integer ;
                                index:          integer ) ;
```

A.16. Package Manager (Packages)

```
Unit Packages ;

interface

{$L-}
Uses {$U MACCORE.CODE} MacCore ,
     {$U ODTYPES.CODE} OdTypes
       ( {Types} Point, FPoint ) ;
{$L+}

Const

  { Standard File Routine Numbers }
  RSFGetFile              = 2 ;
  RSFPGetFile             = 4 ;
  RSFPPutFile             = 3 ;
  RSFPutFile              = 1 ;

  { Disk Initilization Routine Numbers }
  RDIBadMount             = 0 ;
  RDIFormat               = 6 ;
  RDILoad                 = 2 ;
  RDIUnload               = 4 ;
  RDIVerify               = 8 ;
  RDIZero                 = 10 ;

  { International Utilities Routine Numbers }
  RIUDatePString          = 14 ;
  RIUDateString           = 0 ;
  RIUGetIntl              = 6 ;
  RIUMagIDString          = 12 ;
  RIUMagString            = 10 ;
  RIUMetric               = 4 ;
  RIUTimePString          = 16 ;
  RIUTimeString           = 2 ;
  RIUSetIntl              = 8 ;

  { Standard File Package Constants }

  putDlgID        = -3999 ;       { SFPutFile dialog template ID }
  putSave         = 1 ;           { save button }
  putCancel       = 2 ;           { Cancel button }
  putEject        = 5 ;           { Eject Button }
  putDrive        = 6 ;           { Drive Button }
  putName         = 7 ;           { EditText item for file name }

  getDlgId        = -4000 ;       { SFGetFile dialog template ID }
  getOpen         = 1 ;           { Open Button }
  getCancel       = 3 ;           { Cancel button }
  getEject        = 5 ;           { Eject button }
  getDrive        = 6 ;           { Drive button }
  getNmLst        = 7 ;           { userItem for file name list }
  getScroll       = 8 ;           { userItem for scroll bar }


  { International Utilities Package Constants }

  { DateForm Constants }
  ShortDate       = 0 ;
  LongDate        = 256 ;
  AbbrevDate      = 512 ;

  { Currency format flags }
  currLeadingZ    = 128 ;         { Mask for leading zero }
  currTrailingZ   = 64 ;          { Mask for trailing zero }
  currNegSym      = 32 ;          { Mask for for minus sign / brackets }
  currSymTrail    = 16 ;          { Mask for currency symbol location }
```

```
  { Short Date Form Constants }
  DMY             = 2 ;           { day, month, year }
  YMD             = 1 ;           { year, month, day }
  MDY             = 0 ;           { month, day, year }

  { date element format masks }
  mntLeadingZ     = 64 ;          { Mask for leading zero on month }
  dayLeadingZ     = 32 ;          { Mask for leading zero on day }
  century         = 128 ;         { Mask for century / no century }

  { time element format masks }
  hrLeadingZ      = 128 ;         { Mask for leading zero on hour }
  minLeadingZ     = 64 ;          { Mask for leading zero on minutes }
  secLeadingZ     = 32 ;          { Mask for leading zero on seconds }

  { contry codes for version numbers }
  verUS           = 0 ;
  verFrance       = 1 ;
  verBritain      = 2 ;
  verGermany      = 3 ;
  verItaly        = 4 ;

Type

  { Standard File Types }

  SFReply = PACKED RECORD
    copy:         SmallBool ;     { not used }
    good:         SmallBool ;     { ignore command if false }
    ftype:        OsType ;        { file type or not used }
    vRefNum:      integer ;       { volume reference number }
    version:      integer ;       { file version number }
    fname:        String[63] ;    { file name }
  End ;

  SFTypeList = ARRAY [0..3] of OsType ;

  PtrSFReply     = MacPtr ;
  PtrSFTypeList = MacPtr ;

  { International Resources Interface }
  intl0Hndl       = Handle ;
  intl0Ptr        = MacPtr ;
  intl0Rec        = PACKED RECORD
    thousSep:     char ;          { ASCII character for thousand seperator }
    decimalPt:    char ;          { ASCII character for decimal point }
    currSym1:     char ;          { Acurrency symbol (3 bytes) }
    listSep:      char ;          { ASCII character for list seperator }
    currSym3:     char ;
    currSym2:     char ;
    dateOrder:    Byte ;          { short Date form - DMY, YMD or MDY }
    currFmt:      Byte ;          { currency format flags }
    dateSep:      char ;          { ASCII for date seperator }
    shrtDateFmt:Byte ;            { date elements format flags }
    timeFmt:      Byte ;          { time elements format flags }
    timeCycle:    Byte ;          { indicates 12 or 24 hour cycle }
    mornStr:      PACKED ARRAY [1..4] of char ;
                                  { trailing string from 0:00 to 11:59 }
    eveStr:       PACKED ARRAY [1..4] of char ;
                                  { trailing string from 12:00 to 23:59 }
    time1Suff:    char ;          { suffix string used in 24 hr mode (8 chars) }
    timeSep:      char ;          { time seperator }
    time3Suff:    char ;
    time2Suff:    char ;
    time5Suff:    char ;
    time4Suff:    char ;
    time7Suff:    char ;
    time6Suff:    char ;
    metricSys:    Byte ;          { indicates metric or English system }
    time8Suff:    char ;
    intl0Vers:    integer ;       { vrsn: hi byte = country / lo byte = vers }
  End ;
```

```
  intl1Hndl      = Handle ;
  intl1Ptr       = MacPtr ;
  intl1Rec       = PACKED RECORD
    days:        ARRAY [1..7] of String[15] ;
                                { Sunday through Monday }
    months:      ARRAY [1..12] of String[15] ;
                                { January through December }
    dateFmt:     Byte ;         { expanded date format 0 or 255}
    supressDate:Byte ;          { 0 for day of week, 255 for no day of week }
    abbrLen:     Byte ;         { month length for short-expanded date }
    dayLeading0:Byte ;          { 255 for leading 0, 0 for no leading 0 }
    st0:         PACKED ARRAY[1..4] of char ;
    st1:         PACKED ARRAY[1..4] of char ;
    st2:         PACKED ARRAY[1..4] of char ;
    st3:         PACKED ARRAY[1..4] of char ;
    st4:         PACKED ARRAY[1..4] of char ;
    intl1Vers:   integer ;      { version word }
    localrtn:    integer ;      { routine to handle exceptions for mag comp }
  End ;

{ Standard File Package ------------------------------------------------}

PROCEDURE SFPutFile     (       where:          FPoint ;
                                prompt:         StringPtr ;
                                origName:       StringPtr ;
                                dlgHook:        ProcPtr ;
                                reply:          PtrSFReply ;
                                RSFPutFile:     integer ) ;
                                                external(-22038); {A9EA}

PROCEDURE SFPPutFile    (       where:          FPoint ;
                                prompt:         StringPtr ;
                                origName:       StringPtr ;
                                dlgHook:        ProcPtr ;
                                reply:          PtrSFReply ;
                                dlgID:          integer ;
                                filterProc:     ProcPtr ;
                                RSFPPutFile:    integer ) ;
                                                external(-22038); {A9EA}

PROCEDURE SFGetFile     (       where:          FPoint ;
                                prompt:         StringPtr ;
                                fileFilter:     ProcPtr ;
                                numTypes:       integer ;
                                typeList:       PtrSFTypeList ;
                                dlgHook:        ProcPtr ;
                                reply:          PtrSFReply ;
                                RSFGetFile:     integer ) ;
                                                external(-22038); {A9EA}

PROCEDURE SFPGetFile    (       where:          FPoint ;
                                prompt:         StringPtr ;
                                fileFilter:     ProcPtr ;
                                numTypes:       integer ;
                                typeList:       PtrSFTypeList ;
                                dlgHook:        ProcPtr ;
                                reply:          PtrSFReply ;
                                dlgID:          integer ;
                                filterProc:     ProcPtr ;
                                RSFPGetFile:    integer ) ;
                                                external(-22038); {A9EA}

{ Disk Initialization Package ------------------------------------------}

PROCEDURE DILoad        (       RDILoad:        integer ) ;
                                                external(-22039); {A9E9}

PROCEDURE DIUnload      (       RDIUnLoad:      integer ) ;
                                                external(-22039); {A9E9}

FUNCTION  DIBadMount    (       where:          FPoint ;
                                evtMessage:     LongInt ;
```

```
                                RDIBadMount:    integer )
                        : integer ;             external(-22039); {A9E9}

FUNCTION  DIFormat      (       drvNum:         integer ;
                                RDIFormat:      integer )
                        : OsErr ;               external(-22039); {A9E9}

FUNCTION  DIVerify      (       drvNum:         integer ;
                                RDIVerify:      integer )
                        : OsErr ;               external(-22039); {A9E9}

FUNCTION  DIZero        (       drvNum:         integer ;
                                volName:        StringPtr ;
                                RDIZero:        integer )
                        : OsErr ;               external(-22039); {A9E9}

{ International Utilities Package ------------------------------------}

PROCEDURE IUDateString  (       dateTime:       LongInt ;
                                DateForm:       integer ;
                                result:         StringPtr ;
                                RIUDateString:  integer ) ;
                                                external(-22035); {A9ED}

PROCEDURE IUDatePString (       dateTime:       LongInt ;
                                DateForm:       integer ;
                                result:         StringPtr ;
                                intlParam:      Handle ;
                                RIUDatePString: integer ) ;
                                                external(-22035); {A9ED}

PROCEDURE IUTimeString  (       dateTime:       LongInt ;
                                wantSeconds:    MacBool ;
                                result:         StringPtr ;
                                RIUTimeString:  integer ) ;
                                                external(-22035); {A9ED}

PROCEDURE IUTimePString (       dateTime:       LongInt ;
                                wantSeconds:    MacBool ;
                                result:         StringPtr ;
                                intlParam:      Handle ;
                                RIUTimePString: integer ) ;
                                                external(-22035); {A9ED}

FUNCTION  IUMetric      (       RIUMetric:      integer )
                        : MacBool ;             external(-22035); {A9ED}

FUNCTION  IUGetIntl     (       theID:          integer ;
                                RIUGetIntl:     integer )
                        : Handle ;              external(-22035); {A9ED}

PROCEDURE IUSetIntl     (       refNum:         integer ;
                                theID:          integer ;
                                intlParam:      Handle ;
                                RIUSetIntl:     integer ) ;
                                                external(-22035); {A9ED}

FUNCTION  IUMagString   (       a:              StringPtr ;
                                b:              StringPtr ;
                                aLen:           integer ;
                                bLen:           integer ;
                                RIUMagString:   integer )
                        : integer ;             external(-22035); {A9ED}

FUNCTION  IUMagIDString (       a:              StringPtr ;
                                b:              StringPtr ;
                                aLen:           integer ;
                                bLen:           integer ;
                                RIUMagIDString: integer )
                        : integer ;             external(-22035); {A9ED}
```

A.17. Parameter Block I/O Manager (PBIOMgr)

```
unit PbIOMgr ;

interface

{USES Maccore, QDTypes, TBTypes,OsTypes}
{$L-}
uses {$U MACCORE.CODE} Maccore,
     {$U QDTYPES.CODE} QdTypes (Point,VHSelect,GrafPort,GrafPtr,Rect),
     {$U TBTYPES.CODE} TBTypes (EventRecord),
     {$U OSTYPES.CODE} OsTypes (ParmBlkPtr, ParamBlockRec,QHdrPtr);
{$Lt}

{------------------ Low Level File Manager Routines --------------------}

{ Initializing the File I/O Queue-------------------------------------}

PROCEDURE InitQueue;                        EXTERNAL(-24542); {A022}

{ Accessing Volumes---------------------------------------------------}

FUNCTION  PBMountVol     (       paramBlock:     ParmBlkPtr)
                         : OsErr ;

FUNCTION  PBGetVInfo     (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBGetVol       (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBSetVol       (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBFlshVol      (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBUnmountVol   (       paramBlock:     ParmBlkPtr)
                         : OsErr ;

FUNCTION  PBOffLine      (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBEject        (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

{ Changing File Contents----------------------------------------------}

FUNCTION  PBCreate       (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBOpen         (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBOpenRF       (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;

FUNCTION  PBRead         (       paramBlock:     ParmBlkPtr;
                                 async:          boolean)
                         : OsErr ;
```

```
FUNCTION  PBWrite       (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBGetFPos     (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBSetFPos     (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBGetEof      (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBSetEof      (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;


FUNCTION  PBAllocate    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBFlshFile    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBClose       (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;


FUNCTION  PBGetFInfo    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBSetFInfo    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBSetFLock    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBRstFLock    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBSetFVers    (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBRename      (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

FUNCTION  PBDelete      (       paramBlock:     ParmBlkPtr;
                                async:          boolean)
                        : OsErr ;

  Accessing Queues------------------------------------------------------}

FUNCTION  GetFSQHdr     : QHdrPtr ;

FUNCTION  GetVCBQHdr    : QHdrPtr ;

FUNCTION  GetDrvQHdr    : QHdrPtr ;

---------------Low Level Device Routines-----------------------------}

FUNCTION  PBControl     (       paramBlock:     ParmBlkPtr ;
```

```
                                async:          Boolean )
                        : OsErr ;

FUNCTION   PBStatus     (       paramBlock:     ParmBlkPtr ;
                                async:          Boolean )
                        : OsErr ;

FUNCTION   PBKillIO     (       paramBlock:     ParmBlkPtr ;
                                async:          Boolean )
                        : OsErr ;
```

A.18. Print Manager (PrintMgr)

```
Unit PrintMgr ;

Interface

{$L-}
Uses {$U MACCORE.CODE} MacCore,
     {$U QDTYPES.CODE} QdTypes
       ( GrafPort, GrafPtr, Rect, RectPtr, QDProcs ) ;
{$L+}

Const

  { Printing Methods }

  bDraftLoop     = 0 ;            { Draft Printing }
  bSpoolLoop     = 1 ;            { Spooling }
  bUser1Loop     = 2 ;            { Printer Specific, method 1 }
  bUser2Loop     = 3 ;            { Printer Specific, method 2 }

  { Printer feed type constants }

  FeedCut        = 0 ;            { hand-fed , indivually cut }
  FeedFanFold    = 1 ;            { continuos-feed FanFold Paper }
  FeedMechCut    = 2 ;            { mechanically fed cut sheets }
  FeedOther      = 3 ;            { other types of paper }

  { Scan Types }

  ScanTB         = 0 ;            { Top to Bottom }
  ScanBT         = 1 ;            { Bottom to Top }
  ScanLR         = 2 ;            { Left to Right }
  ScanRL         = 3 ;            { Right to Left }

Type

  TPPrPort = MacPtr ;
  TPrPort = Record
    gport :      GrafPort ;       { GrafPort to be drawn in }
    gProcs :     QDProcs ;        { Pointers to drawing routines }
    { other fields for internal use only }
  End ;

  TPPort = RECORD
    Case Integer of
      0 : (pGPort :      GrafPtr) ;
      1 : (pPrPort :     TPPrPort) ;
  End ;

  TPrInfo = RECORD
    iDev :       integer ;        { Driver Information }
    iVRes :      integer ;        { Printer vertical resolution }
    iHRes :      integer ;        { Printer horizontal resolution }
    rPage :      Rect ;           { page rectangle }
  End ;

  TPrSt1 = PACKED RECORD
    wDev :       integer ;        { Used internally }
    iPageV :     integer ;        { Paper height }
    iPageH :     integer ;        { Paper Width }
    TFeed :      Byte ;           { Paper feed type }
    bPort :      Byte ;           { Printer or modem port }
  End ;

  TPrXInfo = PACKED RECORD
    iRowBytes : integer ;         { Bytes per row }
    iBandV : integer ;            { Vertical dots }
    iBandH : integer ;            { Horizontal dots }
    iDevBytes : integer ;         { size of bit image }
```

```
    iBands :      integer ;          { bands per page }
    BUIThick :    Byte ;             { underline thickness }
    bPatScale :   Byte ;             { used by quickdraw }
    bUIShadow :   Byte ;             { underline descender }
    bUIOffset :   Byte ;             { underline offset }
    bXInfoX :     Byte ;             { not used }
    TScan :       Byte ;             { Scan Direction }
  End ;

  TPrJob = PACKED RECORD
    iFstPage :    integer ;          { First Page to print }
    iLstPage :    integer ;          { Last Page to print }
    iCopies :     integer ;          { Number of copies to print }
    fFromUsr :    SmallBool ;        { TRUE if called from application }
    bJDocLoop :   Byte ;             { document style, Draft, Spool, etc. }
    pIdleProc :   ProcPtr ;          { The proc to call while waiting on I/O }
    pFileName :   StringPtr ;        { Spool file name }
    iFileVol :    integer ;          { spool file volume }
    bJobx :       Byte ;             { unused }
    bFileVers :   Byte ;             { spool file version }
  End ;

  THPrint = Handle ;
  TPPrint = MacPtr ;
  TPrint  = RECORD
    iPrVersion :integer ;            { Printing Manager Version Number }
    prInfo :    TPrInfo ;            { printer information }
    rPaper :    Rect ;               { paper rectangle }
    prStl :     TPrStl ;             { style information }
    prInfoPt :  TPrInfo ;            { copy of prInfo }
    prXInfo :   TPrXInfo ;           { band information }
    prJob :     TPrJob ;             { job information }
    printx :    Array [1..19] of integer ;
  End ;

  TPrStatus = PACKED RECORD
    iTotPages : integer ;            { total number of pages }
    iCurPage :  integer ;            { page being printed }
    iTotCopies :integer ;            { number of copies }
    iCurCopy :  integer ;            { current copy being printed }
    iTotBands : integer ;            { bands per page }
    iCurBand :  integer ;            { current band being printed }
    fImaging :  SmallBool ;          { TRUE if imaging }
    fPgDirty :  SmallBool ;          { TRUE if started printing page }
    hPrint :    THPrint ;            { the print record }
    pPrPort :   TPPrPort ;           { print port }
    hPic :      Handle ;             { used internally }
  End ;

{ Initialization and Termination ------------------------------------}

PROCEDURE PrOpen ;

PROCEDURE PrClose ;

{ Print Records and Dialogs -----------------------------------------}

PROCEDURE PrintDefault  (       hPrint :        THPrint ) ;

FUNCTION  PrValidate    (       hPrint :        THPrint )
                        : MacBool ;

FUNCTION  PrStlDialog   (       hPrint :        THPrint )
                        : MacBool ;

FUNCTION  PrJobDialog   (       hPrint :        THPrint )
                        : MacBool ;

PROCEDURE PrJobMerge    (       hPrintScr,
                                hPrintDst :     THPrint ) ;

{ Document Printing -------------------------------------------------}
```

```
FUNCTION  PrOpenDoc     (       hPrint :        THPrint ;
                                pPrPort :       TPPrPort ;
                                pIOBuf :        MacPtr )
                        : TPPrPort ;

PROCEDURE PrCloseDoc    (       pPrPort :       TPPrPort ) ;

PROCEDURE PrOpenPage    (       pPrPort :       TPPrPort ;
                                pPageFrame :    RectPtr ) ;

PROCEDURE PrClosePage   (       pPrPort :       TPPrPort ) ;

{ Spool Printing ------------------------------------------------------}

PROCEDURE PrPicFile     (       hPrint :        THPrint ;
                                pPrPort :       TPPrPort ;
                                pIOBuf :        MacPtr ;
                                pDevBuf :       MacPtr ;
                                prStatus :      MacPtr ) ;

{ Handling Errors -----------------------------------------------------}

FUNCTION  PrError       : integer ;

PROCEDURE PrSetError    (       iErr :          integer ) ;
```

A.19. Printer Driver (PrintDriver)

```
Unit PrintDriver ;

Interface

{$L-}
Uses {$U MACCORE.CODE} MacCore ;
{$L+}

Const

  { Printer Driver Control call parameters }

  iPrBitsCtl     = 4 ;              { bitMap Printing }
  lScreenBits    = 0 ;                { configurable }
  lPaintBits     = 1 ;                { 72 by 72 dots }
  iPrIOCtl       = 5 ;              { text streaming }
  iPrEvtCtl      = 6 ;              { screen printing }
  lPrEvtAll      = 196605 ;           { print whole screen }
  lPrEvtTop      = 131069 ;           { print top most window }
  iPrDevCtl      = 7 ;              { device control }
  lPrReset       = 65536 ;            { reset printer }
  lPrPageEnd     = 131072 ;           { start new page }
  lPrLineFeed    = 196608 ;           { start new line }
  iFMgrCtl       = 8 ;              { used by Font Mgr }

{ Initialization and Termination ------------------------------------}

FUNCTION  PrDrvrOpen     : OsErr ;

FUNCTION  PrDrvrClose    : OsErr ;

{ Printer Control ---------------------------------------------------}

FUNCTION  PrCtlCall      (       iWhichCtl :      integer ;
                                 param1,
                                 param2,
                                 param3 :        LongInt ) : OsErr ;

{ Memory Allocation Control -----------------------------------------}

PROCEDURE PrPurge ;

PROCEDURE PrNoPurge ;

{ Miscellaneous -----------------------------------------------------}

FUNCTION  PrDrvrDce      : Handle ;

FUNCTION  PrDrvrVers     : integer ;
```

A.20. Quickdraw Types (QdTypes)

```
unit QDTypes ;

interface

{Uses MacCore}
{$L-}
Uses {$U MacCore.Code} MacCore ;
{$L+}

type

  { The following Pointers are used to pass parameters by ADDRESS }
  PatternPtr     = MacPtr ;                { Pointer to Pattern Array }
  BitMapPtr      = MacPtr ;                { Pointer to BitMap }
  QDProcsPtr     = MacPtr ;                { Pointer to QDProcs Record }
  CursorPtr      = MacPtr ;                { Pointer to Cursor Record }
  FontInPtr      = MacPtr ;                { Pointer to FontInfo Record }
  PenStPtr       = MacPtr ;                { Pointer to PenState Record }
  PointPtr       = MacPtr ;                { Pointer to Point Record }
  GrafPtr        = MacPtr ;                { Pointer to Graph Port Record }

  Pattern =      packed array[0..7] of 0..255;
  Bits16 =       array[0..15] of integer;

  FPoint =       LongInt;          { fake point for on-the-stack parameters }
  VHSelect = (v,h) ;
  Point =        record case integer of
                     0: (v:      integer;
                         h:      integer);
                     1: (vh:     array[VHSelect] of integer);
                     2: (param:  LongInt);
                 end;


  RectPtr = MacPtr ;
  Rect =         record case integer of
                   0: (top:    integer;
                       left:   integer;
                       bottom: integer;
                       right:  integer);
                   1: (topLeft: Point;
                       botRight: Point);
                 end;

  StyleItem =    (bold,italic,underline,outline,shadow,condense,extend);
  Style =        set of StyleItem;

  FontInfo =     record
                   ascent:     integer;
                   descent:    integer;
                   widMax:     integer;
                   leading:    integer;
                 end;


  BitMap =       record
                   baseAddr:   MacPtr;
                   rowBytes:   integer;
                   bounds:     Rect;
                 end;

  Cursor =       record
                   data:       Bits16;
                   mask:       Bits16;
                   hotSpot:    Point;
                 end;

  PenState =     record
```

```
    pnLoc:      Point;
    pnSize:     Point;
    pnMode:     integer;
    pnPat:      Pattern;
  end;

Polygon =   record
    polySize:   integer;
    polyBBox:   Rect;
    polyPoints:array[0..0] of Point;
  end;

Region =    record
    rgnSize:    integer;    { rgnSize = 10 for rectangular }
    rgnBBox:    Rect;
    { plus more data if not rectangular }
  end;

Picture =   record
    picSize:    integer;
    picFrame:   Rect;
    { plus byte codes for picture content }
  end;

QDProcs =   record
    textProc:   ProcPtr;
    lineProc:   ProcPtr;
    rectProc:   ProcPtr;
    rRectProc:  ProcPtr;
    ovalProc:   ProcPtr;
    arcProc:    ProcPtr;
    polyProc:   ProcPtr;
    rgnProc:    ProcPtr;
    bitsProc:   ProcPtr;
    commentProc:ProcPtr;
    txMeasProc:ProcPtr;
    getPicProc:ProcPtr;
    putPicProc:ProcPtr;
  end;

GrafPort =  record
    device:     integer;
    portBits:   BitMap;
    portRect:   Rect;
    visRgn:     Handle;
    clipRgn:    Handle;
    bkPat:      Pattern;
    fillPat:    Pattern;
    pnLoc:      Point;
    pnSize:     Point;
    pnMode:     integer;
    pnPat:      Pattern;
    pnVis:      integer;
    txFont:     integer;
    txFace:     Style;
    txMode:     integer;
    txSize:     integer;
    spExtra:    LongInt;
    fgColor:    LongInt;
    bkColor:    LongInt;
    colrBit:    integer;
    patStretch:integer;
    picSave:    Handle;
    rgnSave:    Handle;
    polySave:   Handle;
    grafProcs:  MacPtr;
  end;
```

A.21. Quickdraw (QuickDraw)

```
unit QuickDraw;

interface

{Uses Maccore, ODTypes}
{$L-}
uses {$U MACCORE.CODE} MacCore,
     {$U ODTYPES.CODE} ODTypes;
{$L+}

const

  srcCopy        = 0;     { the 16 transfer modes }
  srcOr          = 1;
  srcXor         = 2;
  srcBic         = 3;
  notSrcCopy     = 4;
  notSrcOr       = 5;
  notSrcXor      = 6;
  notSrcBic      = 7;
  patCopy        = 8;
  patOr          = 9;
  patXor         = 10;
  patBic         = 11;
  notPatCopy     = 12;
  notPatOr       = 13;
  notPatXor      = 14;
  notPatBic      = 15;

{ QuickDraw color separation constants }

  normalBit      = 0;
  inverseBit     = 1;
  redBit         = 4;
  greenBit       = 3;
  blueBit        = 2;
  cyanBit        = 8;
  magentaBit     = 7;
  yellowBit      = 6;
  blackBit       = 5;

  blackColor     = 33;
  whiteColor     = 30;
  redColor       = 205;
  greenColor     = 341;
  blueColor      = 409;
  cyanColor      = 273;
  magentaColor   = 137;
  yellowColor    = 69;

  picLParen      = 0;
  picRParen      = 1;

  GrafVerb constants for the Standard Procedures }

  Frame          = 0 ;
  Paint          = 256 ;
  Erase          = 512 ;
  Invert         = 768 ;
  Fill           = 1024 ;


  GrafPort Routines ------------------------------------------------}

rocedure OpenPort      (port:      GrafPtr) ;   external(-22417); {A86F}
rocedure InitPort      (port:      GrafPtr) ;   external(-22419); {A86D}
rocedure ClosePort     (port:      GrafPtr) ;   external(-22403); {A87D}
rocedure SetPort       (port:      GrafPtr) ;   external(-22413); {A873}
```

```
procedure GetPort       (port:      MacPtr) ;    external(-22412); {A874}
procedure GrafDevice    (device:    integer);    external(-22414); {A872}
procedure SetPortBits   (bm:        BitMapPtr) ; external(-22411); {A875}

procedure PortSize      (width,
                         height:    integer);    external(-22410); {A876}

procedure MovePortTo    (leftGlobal,
                         topGlobal: integer);    external(-22409); {A877}

procedure SetOrigin     (h,v:       integer);    external(-22408); {A878}
procedure SetClip       (rgn:       Handle);     external(-22407); {A879}
procedure GetClip       (rgn:       Handle);     external(-22406); {A87A}
procedure ClipRect      (r:         RectPtr);    external(-22405); {A87B}
procedure BackPat       (pat:       PatternPtr); external(-22404); {A87C}

{ Cursor Routines ------------------------------------------------------------}

procedure InitCursor;                            external(-22448); {A850}
procedure SetCursor     (crsr:      CursorPtr);  external(-22447); {A851}
procedure HideCursor;                            external(-22446); {A852}
Procedure ShowCursor;                            external(-22445); {A853}
procedure ObscureCursor;                         external(-22442); {A856}

{ Line Routines --------------------------------------------------------------}

procedure HidePen;                               external(-22378); {A896}
procedure ShowPen;                               external(-22377); {A897}
procedure GetPen        (pt:        PointPtr);   external(-22374); {A89A}
procedure GetPenState   (pnState:   PenStPtr);   external(-22376); {A898}
procedure SetPenState   (pnState:   PenStPtr);   external(-22375); {A899}

procedure PenSize       (width,
                         height:    integer);    external(-22373); {A89B}

procedure PenMode       (mode:      integer);    external(-22372); {A89C}
procedure PenPat        (pat:       PatternPtr); external(-22371); {A89D}
procedure PenNormal;                             external(-22370); {A89E}
procedure MoveTo        (h,v:       integer);    external(-22381); {A893}
procedure Move          (dh,dv:     integer);    external(-22380); {A894}
procedure LineTo        (h,v:       integer);    external(-22383); {A891}
procedure Line          (dh,dv:     integer);    external(-22382); {A892}

{ Text Routines --------------------------------------------------------------}

procedure TextFont      (font:      integer);    external(-22393); {A887}
procedure TextFace      (face:      Style);      external(-22392); {A888}
procedure TextMode      (mode:      integer);    external(-22391); {A889}
procedure TextSize      (size:      integer);    external(-22390); {A88A}
procedure SpaceExtra    (extra:     LongInt);    external(-22386); {A88E}
procedure DrawChar      (ch:        char);       external(-22397); {A883}
procedure DrawString    (s:         StringPtr);  external(-22396); {A884}

procedure DrawText      (textBuf:   MacPtr;
                         firstByte,
                         byteCount: integer);    external(-22395); {A885}

function  CharWidth     (ch:        char)
                        : integer;               external(-22387); {A88D}

function  StringWidth   (s:         StringPtr)
                        : integer;               external(-22388); {A88C}

function  TextWidth     (textBuf:   MacPtr;
                         firstByte,
                         byteCount: integer)
                        : integer;               external(-22394); {A886}

procedure GetFontInfo   (info:      FontInfoPtr); external(-22389); {A88B}

{ Point Calculations ---------------------------------------------------------}

procedure AddPt         (src:       FPoint;
```

```
                          dst:        PointPtr);   external(-22402); {A87E}

procedure SubPt           (src:       FPoint;
                           dst:       PointPtr);   external(-22401); {A87F}

procedure SetPt           (pt:        PointPtr;
                           h,v:       integer);    external(-22400); {A880}

function  EqualPt         (pt1,pt2:   FPoint)
                          : MacBool;               external(-22399); {A881}

procedure ScalePt         (pt:        PointPtr;
                           fromRect,
                           toRect:    RectPtr);    external(-22280); {A8F8}

procedure MapPt           (pt:        PointPtr;
                           fromRect,
                           toRect:    RectPtr);    external(-22279); {A8F9}

procedure LocalToGlobal   (pt:        PointPtr);   external(-22416); {A870}
procedure GlobalToLocal   (pt:        PointPtr);   external(-22415); {A871}

{ Rectangle Calculations ------------------------------------------------}

procedure SetRect         (r:         RectPtr;
                           left,
                           top,
                           right,
                           bottom:    integer);    external(-22361); {A8A7}

function  EqualRect       (rect1,
                           rect2:     RectPtr)
                          : MacBool;               external(-22362); {A8A6}

function  EmptyRect       (r:         RectPtr)
                          : MacBool;               external(-22354); {A8AE}

procedure OffsetRect      (r:         RectPtr;
                           dh,dv:     integer);    external(-22360); {A8A8}

procedure MapRect         (r:         RectPtr;
                           fromRect,
                           toRect:    RectPtr);    external(-22278); {A8FA}

procedure InsetRect       (r:         RectPtr;
                           dh,dv:     integer);    external(-22359); {A8A9}

function  SectRect        (src1,src2: RectPtr;
                           dstRect:   RectPtr)
                          : MacBool;               external(-22358); {A8AA}

procedure UnionRect       (src1,src2: RectPtr;
                           dstRect:   RectPtr);    external(-22357); {A8AB}

function  PtInRect        (pt:        FPoint;
                           r:         RectPtr)
                          : MacBool;               external(-22355); {A8AD}

procedure Pt2Rect         (pt1,pt2:   FPoint;
                           dstRect:   RectPtr);    external(-22356); {A8AC}

{ Graphical Operations on Rectangles ------------------------------------}

procedure FrameRect       (r:         RectPtr);    external(-22367); {A8A1}
procedure PaintRect       (r:         RectPtr);    external(-22366); {A8A2}
procedure EraseRect       (r:         RectPtr);    external(-22365); {A8A3}
procedure InvertRect      (r:         RectPtr);    external(-22364); {A8A4}

procedure FillRect        (r:         RectPtr;
                           pat:       PatternPtr); external(-22363); {A8A5}

{ RoundRect Routines ----------------------------------------------------}
```

```
procedure FrameRoundRect(r:          RectPtr;
                         ovWd,ovHt: Integer);    external(-22352); {A8B0}

procedure PaintRoundRect(r:          RectPtr;
                         ovWd,ovHt: Integer);    external(-22351); {A8B1}

procedure EraseRoundRect(r:          RectPtr;
                         ovWd,ovHt: Integer);    external(-22350); {A8B2}

procedure InvertRoundRect(r:          RectPtr;
                         ovWd,ovHt: Integer);    external(-22349); {A8B3}

procedure FillRoundRect (r:          RectPtr;
                         ovWd,ovHt: Integer;
                         pat:        PatternPtr); external(-22348); {A8B4}

{ Oval Routines ------------------------------------------------------------}

procedure FrameOval      (r:          RectPtr);    external(-22345); {A8B7}
procedure PaintOval      (r:          RectPtr);    external(-22344); {A8B8}
procedure EraseOval      (r:          RectPtr);    external(-22343); {A8B9}
procedure InvertOval     (r:          RectPtr);    external(-22342); {A8BA}

procedure FillOval       (r:          RectPtr;
                          pat:        PatternPtr); external(-22341); {A8BB}

{ Arc Routines -------------------------------------------------------------}

procedure FrameArc       (r:          RectPtr;
                          startAngle,
                          arcAngle:   integer);   external(-22338); {A8BE}

procedure PaintArc       (r:          RectPtr;
                          startAngle,
                          arcAngle:   integer);   external(-22337); {A8BF}

procedure EraseArc       (r:          RectPtr;
                          startAngle,
                          arcAngle:   integer);   external(-22336); {A8C0}

procedure InvertArc      (r:          RectPtr;
                          startAngle,
                          arcAngle:   integer);   external(-22335); {A8C1}

procedure FillArc        (r:          RectPtr;
                          startAngle,
                          arcAngle:   integer;
                          pat:        PatternPtr);external(-22334); {A8C2}

procedure PtToAngle      (r:          RectPtr;
                          pt:         FPoint;
                          angle:      integerPtr);external(-22333); {A8C3}

{ Polygon Routines ---------------------------------------------------------}

function  OpenPoly       : Handle;                  external(-22325); {A8CB}
procedure ClosePoly;                                external(-22324); {A8CC}
procedure KillPoly       (poly:       Handle);      external(-22323); {A8CD}

procedure OffsetPoly     (poly:       Handle;
                          dh,dv:      integer);     external(-22322); {A8CE}

procedure MapPoly        (poly:       Handle;
                          fromRect,
                          toRect:     RectPtr);     external(-22276); {A8FC}

procedure FramePoly      (poly:       Handle);      external(-22330); {A8C6}
procedure PaintPoly      (poly:       Handle);      external(-22329); {A8C7}
procedure ErasePoly      (poly:       Handle);      external(-22328); {A8C8}
procedure InvertPoly     (poly:       Handle);      external(-22327); {A8C9}

procedure FillPoly       (poly:       Handle;
                          pat:        PatternPtr); external(-22326); {A8CA}
```

```
{ Region Calculations ------------------------------------------------------}

function  NewRgn          : Handle;                  external(-22312); {A8D8}
procedure DisposeRgn      (rgn:       Handle);       external(-22311); {A8D9}

procedure CopyRgn         (srcRgn,
                           dstRgn:    Handle);       external(-22308); {A8DC}

procedure SetEmptyRgn     (rgn:       Handle);       external(-22307); {A8DD}

procedure SetRectRgn      (rgn:       Handle;
                           left,
                           top,
                           right,
                           bottom:    integer);      external(-22306); {A8DE}

procedure RectRgn         (rgn:       Handle;
                           r:         RectPtr);      external(-22305); {A8DF}

procedure OpenRgn;                                   external(-22310); {A8DA}
procedure CloseRgn        (destRgn:   Handle);       external(-22309); {A8DB}

procedure OffsetRgn       (rgn:       Handle;
                           dh,dv:     integer);      external(-22304); {A8E0}

procedure MapRgn          (rgn:       Handle;
                           fromRect,
                           toRect:    RectPtr);      xternal(-22277); {A8FB}

procedure InsetRgn        (rgn:       Handle;
                           dh,dv:     integer);      external(-22303); {A8E1}

procedure SectRgn         (srcRgnA,
                           srcRgnB,
                           dstRgn:    Handle);       external(-22300); {A8E4}

procedure UnionRgn        (srcRgnA,
                           srcRgnB,
                           dstRgn:    Handle);       external(-22299); {A8E5}

procedure DiffRgn         (srcRgnA,
                           srcRgnB,
                           dstRgn:    Handle);       external(-22298); {A8E6}

procedure XorRgn          (srcRgnA,
                           srcRgnB,
                           dstRgn:    Handle);       external(-22297); {A8E7}

function  EqualRgn        (rgnA,rgnB: Handle)
                          : MacBool;                 external(-22301); {A8E3}

function  EmptyRgn        (rgn:       Handle)
                          : MacBool;                 external(-22302); {A8E2}

function  PtInRgn         (pt:        FPoint;
                           rgn:       Handle)
                          : MacBool;                 external(-22296); {A8E8}

function  RectInRgn       (r:         RectPtr;
                           rgn:       Handle)
                          : MacBool;                 external(-22295); {A8E9}

  Graphical Operations on Regions ------------------------------------------}

procedure FrameRgn        (rgn:       Handle);       external(-22318); {A8D2}
procedure PaintRgn        (rgn:       Handle);       external(-22317); {A8D3}
procedure EraseRgn        (rgn:       Handle);       external(-22316); {A8D4}
procedure InvertRgn       (rgn:       Handle);       external(-22315); {A8D5}

procedure FillRgn         (rgn:       Handle;
                           pat:       PatternPtr);   external(-22314); {A8D6}
```

```
{ Graphical Operations on BitMaps ------------------------------------}

procedure ScrollRect    (destRect:  RectPtr;
                         dh,dv:     integer;
                         updateRgn: Handle);     external(-22289); {A8EF}

procedure CopyBits      (srcBits,
                         dstBits:   BitMapPtr;
                         srcRect,
                         dstRect:   RectPtr;
                         mode:      integer;
                         maskRgn:   Handle);     external(-22292); {A8EC}

{ Picture Routines ---------------------------------------------------}

function  OpenPicture   (picFrame:  RectPtr)
                        : Handle;                external(-22285); {A8F3}

procedure ClosePicture;                          external(-22284); {A8F4}

procedure DrawPicture   (myPicture: Handle;
                         dstRect:   RectPtr);    external(-22282); {A8F6}

procedure PicComment    (kind,
                         dataSize:  integer;
                         dataHandle:Handle);     external(-22286); {A8F2}

procedure KillPicture   (myPicture: Handle);     external(-22283); {A8F5}

{ The Bottleneck Interface: ------------------------------------------}

procedure SetStdProcs   (procs:     QDProcsPtr); external(-22294); {A8EA}

procedure StdText       (count:     integer;
                         textAddr:  MacPtr;
                         numer,
                         denom:     FPoint);     external(-22398); {A882}

procedure StdLine       (newPt:     FPoint);     external(-22384); {A890}

procedure StdRect       (verb:      integer ;
                         r:         RectPtr);    external(-22368); {A8A0}

procedure StdRRect      (verb:      integer ;
                         r:         RectPtr;
                         ovWd,ovHt: integer);    external(-22353); {A8AF}

procedure StdOval       (verb:      integer ;
                         r:         RectPtr);    external(-22346); {A8B6}

procedure StdArc        (verb:      integer ;
                         r:         RectPtr;
                         startAngle,
                         arcAngle:  integer);    external(-22339); {A8BD}

procedure StdPoly       (verb:      integer ;
                         poly:      Handle);     external(-22331); {A8C5}

procedure StdRgn        (verb:      integer ;
                         rgn:       Handle);     external(-22319); {A8D1}

procedure StdBits       (srcBits:   BitMapPtr;
                         srcRect,
                         dxtRect:   RectPtr;
                         mode:      integer;
                         maskRgn:   Handle);     external(-22293); {A8EB}

procedure StdComment    (kind,
                         dataSize:  integer;
                         dataHandle:Handle);     external(-22287); {A8F1}

function  StdTxMeas     (count:     integer;
                         textAddr:  MacPtr;
```

```
                          numer,
                          denom:      PointPtr;
                          info:       FontInPtr)
                         : integer;                  external(-22291); {A8ED}

procedure StdGetPic      (dataPtr:    MacPtr;
                          byteCount:  integer);       external(-22290); {A8EE}

procedure StdPutPic      (dataPtr:    MacPtr;
                          byteCount:  integer);       external(-22288); {A8F0}

{ Misc Utility Routines ------------------------------------------------------}

function  GetPixel       (h,v:        integer)
                         : MacBool;                  external(-22427); {A865}

function  Random         : integer;                  external(-22431); {A861}

procedure StuffHex       (thingPtr:   MacPtr;
                          s:          StringPtr);    external(-22426); {A866}

procedure ForeColor      (color:      LongInt);      external(-22430); {A862}
procedure BackColor      (color:      LongInt);      external(-22429); {A863}
procedure ColorBit       (whichBit:   integer);      external(-22428); {A864}
```

A.22. Resource Manager (ResMgr)

```
unit ResMgr ;

interface

{Uses MacCore}
{$L-}
Uses {$U MACCORE.CODE} Maccore ;
{$L+}

const
   { Resource Attribute bits }
   resSysRef      = 128 ;          { set if system reference }
   resSysHeap     =  64 ;          { set if read into Mac System heap }
   resPurgeable   =  32 ;          { set if purgeable }
   resLocked      =  16 ;          { set if locked }
   resProtected   =   8 ;          { set if protected }
   resPreload     =   4 ;          { set if to be preloaded }
   resChanged     =   2 ;          { set if to be written to resource file }
   resUser        =   1 ;          { available for use by your application }

{ Opening and Closing Resource Files ----------------------------------}

PROCEDURE CreateResFile (       filename:       StringPtr) ;
                                                external(-22095); {A9B1}

FUNCTION  OpenResFile   (       filename:       StringPtr)
                        : integer ;             external(-22121); {A997}

PROCEDURE CloseResFile  ( refnum:       integer) ;
                                                external(-22118); {A99A}

{ Checking for errors -------------------------------------------------}

FUNCTION  ResError      : integer ;             external(-22097); {A9AF}

{ Setting the Current Resource File -----------------------------------}

FUNCTION  CurResFile    : integer ;             external(-22124); {A994}

FUNCTION  HomeResFile   (       theResource:    Handle)
                        : integer ;             external(-22108); {A9A4}

PROCEDURE UseResFile    (       refNum:         integer) ;
                                                external(-22120); {A998}
          :
{ Getting Resource Types ----------------------------------------------}

FUNCTION  CountTypes    : integer ;             external(-22114); {A99E}

PROCEDURE GetIndTypes   (       theType:        OsTypePtr ;
                                index:          integer) ;
                                                external(-22113); {A99F}

{ Getting and Disposing of Resources ----------------------------------}

PROCEDURE SetResLoad    (       load:           MacBool) ;
                                                external(-22117); {A99B}

FUNCTION  CountResources(       theType:        FOsType )
                        : integer ;             external(-22116); {A99C}

FUNCTION  GetIndResource  (     theType:        FOsType ;
                                index:          integer )
                          : Handle ;            external(-22115); {A99D}

FUNCTION  GetResource   (       theType:        FOsType ;
                                theID:          integer )
```

```
                        : Handle ;                  external(-22112); {A9A0}

FUNCTION  GetNamedResource (    theType:        FOsType ;
                                name:           StringPtr )
                            : Handle ;          external(-22111); {A9A1}

PROCEDURE LoadResource  (       theResource:    Handle ) ;
                                                external(-22110); {A9A2}

PROCEDURE ReleaseResource (     theResource:    Handle ) ;
                                                external(-22109); {A9A3}

PROCEDURE DetachResource (      theResource:    Handle ) ;
                                                external(-22126); {A992}

{ Getting Resource Information ------------------------------------}

FUNCTION  UniqueId      (       theType:        FOsType )
                        : integer ;             external(-22079); {A9C1}

PROCEDURE GetResInfo    (       theResource:    Handle ;
                                theId:          integerPtr ;
                                theType:        OsTypePtr ;
                                name:           StringPtr ) ;
                                                external(-22104); {A9A8}

FUNCTION  GetResAttrs   (       theResource:    Handle )
                        : integer ;             external(-22106); {A9A6}

{ Modifying Resources ---------------------------------------------}

PROCEDURE SetResInfo    (       theResource:    Handle ;
                                theID:          integer ;
                                name:           StringPtr ) ;
                                                external(-22103); {A9A9}

PROCEDURE SetResAttrs   (       theResource:    Handle ;
                                attrs:          integer ) ;
                                                external(-22105); {A9A7}

PROCEDURE ChangedResource (     theResource:    Handle ) ;
                                                external(-22102); {A9AA}

PROCEDURE AddResource   (       theData:        Handle ;
                                theType:        FOsType ;
                                theID:          integer ;
                                name:           StringPtr ) ;
                                                external(-22101); {A9AB}

PROCEDURE RmveResource  (       theResource:    Handle ) ;
                                                external(-22099); {A9AD}

PROCEDURE RmveReference (       theResource:    Handle ) ;
                                                external(-22098); {A9AE}

PROCEDURE AddReference  (       theResource:    Handle ;
                                theID:          integer ;
                                name:           StringPtr ) ;
                                                external(-22100); {A9AC}

PROCEDURE UpdateResFile (       refNum:         integer ) ;
                                                external(-22119); {A999}

PROCEDURE WriteResource (       theResource:    Handle ) ;
                                                external(-22096); {A9B0}

PROCEDURE SetResPurge   (       install:        MacBool ) ;
                                                external(-22125); {A993}

FUNCTION  SizeResource  (       theResource:    Handle )
                        : LongInt ;             external(-22107); {A9A5}

{ Advanced Routines -----------------------------------------------}
```

```
FUNCTION  GetResFileAttrs (     refNum:          integer )
                          : integer ;            external(-22026); {A9F6}

PROCEDURE SetResFileAttrs (     refNum:          integer ;
                                attrs:           integer ) ;
                                                 external(-22025); {A9F7}
```

A.23. Scrap Manager (ScrapMgr)

```
unit ScrapMgr ;

interface

{$L-}
Uses {$U MACCORE.CODE} MacCore ;
{$L+}

type

  PScrapStuff = MacPtr ;
  ScrapStuff = RECORD
    scrapSize:   LongInt ;
    scrapHandle:Handle ;
    scrapCount: integer ;
    scrapState: integer ;
    scrapName:   StringPtr ;
  End ;

{ Getting Scrap Information ------------------------------------------}

FUNCTION  InfoScrap      : PScrapStuff ;          external(-22023); {A9F9}

{ Keeping the Scrap on the Desk --------------------------------------}

FUNCTION  UnloadScrap    : LongInt ;              external(-22022); {A9FA}

FUNCTION  LoadScrap      : LongInt ;              external(-22021); {A9FB}

{ Reading from the Scrap ---------------------------------------------}

FUNCTION  GetScrap       (       hDest:           Handle ;
                                 theType:         FOsType ;
                                 offset:          LongIntPtr )
                         : LongInt ;              external(-22019); {A9FD}

{ Writing to the Scrap -----------------------------------------------}

FUNCTION  ZeroScrap      : LongInt ;              external(-22020); {A9FC}

FUNCTION  PutScrap       (       length:          LongInt ;
                                 theType:         FOsType ;
                                 source:          MacPtr )
                         : LongInt ;              external(-22018); {A9FE}
```

A.24. Serial Driver (Serial)

```
unit Serial ;

interface

{$L-}
Uses {$U MACCORE.CODE} MacCore ;
{$L+}

Const

  { RefNums for the serial ports }
  AinRefNum      = -6 ;          { serial port A input }
  AoutRefNum     = -7 ;          { serial port A output }
  BinRefNum      = -8 ;          { serial port B input }
  BoutRefNum     = -9 ;          { serial port B output }

  { baud rate constants }
  baud300        = 380 ;
  baud600        = 189 ;
  baud1200       = 94 ;
  baud1800       = 62 ;
  baud2400       = 46 ;
  baud3600       = 30 ;
  baud4800       = 22 ;
  baud7200       = 14 ;
  baud9600       = 10 ;
  baud19200      = 4 ;
  baud57600      = 0 ;

  { SCC channel configuration word masks }
  stop10         = 16384 ;
  stop15         = -32768 ;
  stop20         = -16384 ;
  noParity       = 8192 ;
  oddParity      = 4096 ;
  evenParity     = 12288 ;
  data5          = 0 ;
  data6          = 2048 ;
  data7          = 1024 ;
  data8          = 3072 ;

  { serial driver error masks }
  swOverrunErr   = 0 ;
  parityErr      = 16 ;
  hwOverrunErr   = 32 ;
  framingErr     = 64 ;

  { serial driver message constant }
  xOffWasSent    = 128 ;

Type

  SerShk = PACKED RECORD
    fCTS:        Byte ;            { CTS flow control enable flag }
    fXon:        Byte ;            { XON flow control enable flag }
    xoff:        char ;            { XOFF character }
    xon:         char ;            { XON character }
    evts:        Byte ;            { event enable mask bits }
    errs:        Byte ;            { errors mask bits }
    null:        Byte ;            { unused }
    flnX:        Byte ;            { input flow control enable flag }
  End ;

  SerStaRec = PACKED RECORD
    XOFFSent:    Byte ;            { XOFF Sent flag }
    cumErrs:     Byte ;            { cumulative errors report }
    wrPend:      Byte ;            { write pending flag }
    rdPend:      Byte ;            { read pending flag }
```

```
    XOFFHold:    Byte ;          { XOFF flow control hold flag }
    CTSHold:     Byte ;          { CTS flow control hold flag }
  End ;

{ Changing Serial Driver Information ----------------------------------}

FUNCTION  SerReset      (       refNum:         integer ;
                                serConfig:      integer )
                        : OsErr ;

FUNCTION  SerSetBuf     (       refNum:         integer ;
                                serBPtr:        MacPtr ;
                                serBLen:        integer )
                        : OsErr ;

FUNCTION  SerHShake     (       refNum:         integer ;
                                flags:          SerShk )
                        : OsErr ;

FUNCTION  SerSetBrk     (       refNum:         integer )
                        : OsErr ;

FUNCTION  SerClrBrk     (       refNum:         integer )
                        : OsErr ;

{ Getting Serial Driver Information ----------------------------------}

FUNCTION  SerGetBuf     (       refNum:         integer ;
                          VAR   count:          LongInt )
                        : OsErr ;

FUNCTION  SerStatus     (       refNum:         integer ;
                          VAR   serSta:         SerStaRec )
                        : OsErr ;
```

A.25. Sound Driver (Sound)

```
Unit Sound ;

interface
{$L-}
Uses {$U MACCORE.CODE} MacCore ;
{$L+}

Const

  SWmode          = -1 ;
  FTmode          =  1 ;
  FFmode          =  0 ;

Type

  Wave  = PACKED ARRAY [0..255] of Byte ;
  WavePtr = MacPtr ;

  PtrFTSndRec = MacPtr ;
  FTSoundRec = RECORD
    duration:             integer ;
    sound1Rate:           LongInt ;
    sound1Phase:          LongInt ;
    sound2Rate:           LongInt ;
    sound2Phase:          LongInt ;
    sound3Rate:           LongInt ;
    sound3Phase:          LongInt ;
    sound4Rate:           LongInt ;
    sound4Phase:          LongInt ;
    sound1Wave:           WavePtr ;
    sound2Wave:           WavePtr ;
    sound3Wave:           WavePtr ;
    sound4Wave:           WavePtr ;
  End ;

  PtrFTSynth    = MacPtr ;
  FTSynthRec = RECORD
    mode:                 integer ;
    sndRec:               PtrFTSndRec ;
  End ;

  Tone = RECORD
    count:                integer ;
    amplitude:            integer ;
    duration:             integer ;
  End ;

  Tones = ARRAY[0..5000] of Tone ;

  PtrSWSynth = MacPtr ;
  SWSynthRec = RECORD
    mode:                 integer ;
    triplets:             Tones ;
  End ;

  freeWave = PACKED ARRAY[0..30000] of Byte ;

  PtrFFSynthRec = MacPtr ;
  FFSynthRec = RECORD
    mode:                 integer ;
    count:                LongInt ;       { FIXED Point }
    waveBytes:            freeWave ;
  End ;

{ Sound Driver Procedures ------------------------------------------------}

PROCEDURE StartSound    (       synthRec:       MacPtr ;
```

```
                                numBytes:       LongInt ;
                                Async:          Boolean ) ;

PROCEDURE StopSound ;

FUNCTION  SoundDone     : Boolean ;

PROCEDURE SetSoundVol   (       level:          integer ) ;

PROCEDURE GetSoundVol   ( VAR   level:          integer ) ;
```

A.26. ToolBox Utilities (TBoxUtils)

```
unit TBoxUtils ;

interface

{$L-}
Uses {$U MACCORE.CODE} MacCore ,
     {$U QDTYPES.CODE} QDTypes (RectPtr,Rect,FPoint,Point);
{$L+}

type

  Fixed = LongInt ;
  PtrInt64Bit = MacPtr ;
  Int64Bit = RECORD
    hiLong : LongInt ;
    loLong : LongInt ;
  End ;

{ Getting Application input file names ---------------------------------}

PROCEDURE GetAppParms   (       apName :        StringPtr ;
                                apRefNum :      integerPtr ;
                                apParam :       MacPtr ) ;
                                                external(-22027); {A9F5}

{ Fixed Point Arithmetic -----------------------------------------------}

FUNCTION  FixRatio      (       numerator:      integer ;
                                denominator:    integer )
                        : Fixed ;               external(-22423); {A869}

FUNCTION  FixMul        (       a , b :         Fixed )
                        : Fixed ;               external(-22424); {A868}

FUNCTION  FixRound      (       x :             Fixed )
                        : integer ;             external(-22420); {A86C}

{ String Manipulation --------------------------------------------------}

FUNCTION  NewString     (       s:              StringPtr )
                        : StringHandle ;        external(-22266); {A906}

PROCEDURE SetString     (       h:              StringHandle ;
                                s:              StringPtr ) ;
                                                external(-22265); {A907}

FUNCTION  GetString     (       StringID:       integer  )
                        : StringHandle ;        external(-22086); {A9BA}

{ Byte Manipulation ----------------------------------------------------}

FUNCTION  Munger        (       h:              Handle ;
                                offSet:         LongInt ;
                                ptr1:           MacPtr ;
                                len1:           LongInt ;
                                ptr2:           MacPtr ;
                                len2:           LongInt )
                        : LongInt ;             external(-22048); {A9E0}

{ Operations on Bit Strings --------------------------------------------}

FUNCTION  BitTst        (       bytePtr:        MacPtr ;
                                bitnum:         LongInt )
                        : MacBool ;             external(-22435); {A85D}

PROCEDURE BitSet        (       bytePtr:        MacPtr ;
                                bitNum:         LongInt ) ;
```

```
                                                    external(-22434); {A85E}

PROCEDURE BitClr        (       bytePtr:        MacPtr ;
                                bitNum:         LongInt ) ;
                                                external(-22433); {A85F}

{ Other Operations on Long Integers ------------------------------------}

FUNCTION  HiWord        (       x:              LongInt )
                        : integer ;             external(-22422); {A86A}

FUNCTION  LoWord        (       x:              LongInt )
                        : integer ;             external(-22421); {A86B}

PROCEDURE LongMul       (       a , b :         LongInt ;
                                dest :          PtrInt64Bit ) ;
                                                external(-22425); {A867}


{ Graphics Utilities ---------------------------------------------------}

FUNCTION  GetIcon       (       iconID :        integer )
                        : Handle ;              external(-22085); {A9BB}

PROCEDURE PlotIcon      (       theRect :       RectPtr ;
                                theIcon :       Handle ) ;
                                                external(-22197); {A94B}

FUNCTION  GetPattern    (       patID :         integer )
                        : Handle ;              external(-22088); {A9B8}

FUNCTION  GetCursor     (       cursorID :      integer )
                        : Handle ;              external(-22087); {A9B9}

PROCEDURE ShieldCursor  (       shieldRect :    RectPtr ;
                                offsetPt:       FPoint ) ;
                                                external(-22443); {A855}

FUNCTION  GetPicture    (       pictureID:      integer )
                        : Handle ;              external(-22084); {A9BC}

FUNCTION  SlopeFromAngle(       angle:          integer )
                        : Fixed ;               external(-22340); {A88C}

FUNCTION  AngleFromSlope(       slope:          Fixed )
                        : integer ;             external(-22332); {A8C4}

FUNCTION  DeltaPoint    (       ptA, ptB :      FPoint )
                        : LongInt ;             external(-22193); {A94F}
```

A.27. ToolBox Types (TBTypes)

```
unit TBTypes ;

interface

{Uses MacCore, QDTypes}
{$L-}
Uses {$U MACCORE.CODE} MacCore,
     {$U QDTYPES.CODE} QDTypes (Point,VHSelect,GrafPort,GrafPtr,
                                Rect) ;
{$L+}

type

  { The following Ptrs are used for passing variables by ADDRESS. }
  EvtRecPtr      = MacPtr ;          { Pointer to EventRecord }
  WindowPtr      = MacPtr ;          { Pointer to WindowRecord }
  TEPtr          = MacPtr ;          { Pointer to TERec }

  { Event Manager Record }
  EventRecord = RECORD
    what :       integer ;
    message :    LongInt ;
    when :       LongInt ;
    where :      Point ;
    modifiers : integer ;
  End ;

  { Window Manager Record }
  WindowHandle  = MacPtr ;
  WindowRecord = packed record
    port:           GrafPort;
    windowKind:     integer;
    hilited:        SmallBool;
    visible:        SmallBool;
    spareFlag:      SmallBool;
    goAwayFlag:     SmallBool;
    strucRgn:       Handle;
    contRgn:        Handle;
    updateRgn:      Handle;
    windowDefProc:  Handle;
    dataHandle:     Handle;
    titleHandle:    Handle;
    titleWidth:     integer;
    controlList:    Handle;
    nextWindow:     MacPtr;
    windowPic:      Handle;
    refCon:         LongInt;
  end;

  { TextEdit Record }
  TEHandle       = Handle ;
  TERec = RECORD
    destRect:           Rect ;          { destination rectangle }
    viewRect:           Rect ;          { view rectangle }
    lineHeight:         integer ;       { line height }
    firstBL:            integer ;       { position of first baseline }
    selStart:           integer ;       { start of selection range }
    selEnd:             integer ;       { end of selection }
    just:               integer ;       { justification }
    length:             integer ;       { length of text }
    hText:              Handle ;        { text to be edited }
    txFont:             integer;        { text font }
    txFace:             integer ;       { character style }
    txMode:             integer ;       { pen mode }
    txSize:             integer ;       { type size }
    inPort:             GrafPtr ;       { grafPort }
    crOnly:             integer ;       { new line at Return if < 0 }
    nLines:             integer ;       { number of lines }
```

```
  lineStarts:          ARRAY [0..32000] of Integer ;
                                        { positions of line starts }
  { other fields for Mac O.S. internal use only }
End ;
```

A.28. Text Edit (TextEdit)

```
unit TextEdit ;

interface
{$L-}
uses {$U MACCORE.CODE} MacCore ,
     {$U QDTYPES.CODE} QDTypes (GrafPort, GrafPtr, Point, VHSelect,
                                FPoint, Rect, RectPtr),
     {$U TBTYPES.CODE} TBTypes (EvtRecPtr, EventRecord,windowrecord,
                                windowptr,windowhandle,TEHandle,
                                TEPtr,TERec)  ;
{$L+}

const
  teJustLeft    = 0 ;
  teJustCenter  = 1 ;
  teJustRight   = -1 ;

type

  CharsHandle   = Handle ;
  CharsPtr      = MacPtr ;
  Chars         = PACKED ARRAY [0..32000] OF char ;


{ Initialization ------------------------------------------------------------}

FUNCTION  TENew          (       destRect :      RectPtr ;
                                 viewRect :      RectPtr )
                         : TEHandle ;            external(-22062); {A9D2}

{ Manipulating Edit Records -------------------------------------------------}

PROCEDURE TESetText      (       Text :          MacPtr ;
                                 length :        LongInt ;
                                 hTE :           TEHandle ) ;
                                                 external(-22065); {A9CF}

FUNCTION  TEGetText      (       hTE :           TEHandle )
                         : CharsHandle ;         external(-22069); {A9CB}

PROCEDURE TEDisPose      (       hTE :           TEHandle ) ;
                                                 external(-22067); {A9CD}

{ Editing -------------------------------------------------------------------}

PROCEDURE TEKey          (       key :           char ;
                                 hTE :           TEHandle ) ;
                                                 external(-22052); {A9DC}

PROCEDURE TECut          (       hTE :           TEHandle ) ;
                                                 external(-22058); {A9D6}

PROCEDURE TECopy         (       hTE :           TEHandle ) ;
                                                 external(-22059); {A9D5}

PROCEDURE TEPaste        (       hTE :           TEHandle ) ;
                                                 external(-22053); {A9D8}

PROCEDURE TEDelete       (       hTE :           TEHandle ) ;
                                                 external(-22057); {A9D7}

PROCEDURE TEInsert       (       text :          MacPtr ;
                                 length :        LongInt ;
                                 hTE :           TEHandle ) ;
                                                 external(-22050); {A9DE}

{ Selection Range and Justification -----------------------------------------}
```

```
PROCEDURE TESetSelect   (       selStart :      LongInt ;
                                selEnd :        LongInt ;
                                hTE :           TEHandle ) ;
                                                external(-22063); {A9D1}

PROCEDURE TESetJust     (       j :             integer ;
                                hTE :           TEHandle ) ;
                                                external(-22049); {A9DF}

{ Mice and Carets ------------------------------------------------------}

PROCEDURE TEClick       (       pt :            FPoint ;
                                extend :        MacBool ;
                                hTE :           TEHandle ) ;
                                                external(-22060); {A9D4}

PROCEDURE TEIdle        (       hTE :           TEHandle ) ;
                                                external(-22054); {A9DA}

PROCEDURE TEActivate    (       hTE :           TEHandle ) ;
                                                external(-22056); {A9D8}

PROCEDURE TEDeactivate  (       hTE :           TEHandle ) ;
                                                external(-22055); {A9D9}

{ Text Display ---------------------------------------------------------}

PROCEDURE TEUpdate      (       rUpdate :       RectPtr ;
                                hTE :           TEHandle ) ;
                                                external(-22061); {A9D3}

PROCEDURE TextBox       (       text :          MacPtr ;
                                length :        LongInt ;
                                box :           RectPtr ;
                                style :         integer ) ;
                                                external(-22066); {A9CE}

{ Advanced Routines ----------------------------------------------------}

PROCEDURE TEScroll      (       dh , dv :       integer ;
                                hTE :           TEHandle ) ;
                                                external(-22051); {A9DD}

PROCEDURE TECalText     (       hTE :           TEHandle ) ;
                                                external(-22064); {A9D0}
```

A.29. Window Manager (WindowMgr)

```
Unit WindowMgr;

Interface
{Uses MacCore, OdTypes, TbTypes}
{$L-}
uses {$U MACCORE.CODE} MacCore ,
     {$U ODTYPES.CODE} ODTypes (GrafPort, GrafPtr, Point, VHSelect,
                                FPoint, Rect, RectPtr),
     {$U TBTYPES.CODE} TBTypes (EvtRecPtr, EventRecord,windowrecord,
                                windowptr,windowhandle)  ;
{$L+}

const

  { types of windows }

  dialogKind     = 2;
  userKind       = 8;

  { window definition procedure IDs }

  DocumentProc   = 0;
  DBoxProc       = 1;
  plainDBox      = 2 ;
  altDBoxProc    = 3 ;
  noGrowDocProc  = 4 ;
  RDocProc       = 16;

  { FindWindow result codes }

  inDesk         = 0;
  inMenuBar      = 1;
  inSysWindow    = 2;
  inContent      = 3;
  inDrag         = 4;
  inGrow         = 5;
  inGoAway       = 6;

  { ... hit test codes }

  wNoHit         = 0;
  wInContent     = 1;
  wInDrag        = 2;
  wInGrow        = 3;
  wInGoAway      = 4;

  { Window Messages }

  wDraw          = 0 ;
  wHit           = 1 ;
  wCalcRgns      = 2 ;
  wNew           = 3 ;
  wDispose       = 4 ;
  wGrow          = 5 ;
  wDrawGIcon     = 6 ;

  { Axis constraints for DragGrayRgn call }

  bothAxes       = 0 ;
  hAxsOnly       = 1 ;
  vAxsOnly       = 2 ;

{ Initialization and Allocation ------------------------------------------}
procedure GetWMgrPort    (wPort:     MacPtr);     external(-22256); {A910}
function  NewWindow      (wStorage:  MacPtr;
```

```
                        boundsRect:RectPtr;
                        title:     StringPtr;
                        visible:   MacBool;
                        theProc:   integer;
                        behind:    MacPtr;
                        goAwayFlag:MacBool;
                        refCon:    LongInt)
                       : WindowPtr ;              external(-22253); {A913}

function  GetNewWindow (windowID:  integer;
                        wStorage:  MacPtr;
                        behind:    MacPtr)
                       : WindowPtr ;              external(-22083); {A9BD}

procedure CloseWindow  (theWindow: WindowPtr) ; external(-22227); {A92D}
procedure DisposeWindow (theWindow: WindowPtr) ; external(-22252); {A914}

{ Window Display ------------------------------------------------------}

procedure SetWTitle    (theWindow: WindowPtr ;
                        title:     StringPtr ) ;external(-22246); {A91A}

procedure GetWTitle    (theWindow: WindowPtr ;
                        title:     StringPtr);  external(-22247); {A919}

procedure SelectWindow (theWindow: WindowPtr ) ;external(-22241); {A91F}
procedure HideWindow   (theWindow: WindowPtr ) ;external(-22250); {A916}
procedure ShowWindow   (theWindow: WindowPtr ) ;external(-22251); {A915}

procedure ShowHide     (theWindow: WindowPtr ;
                        showFlag:  MacBool);    external(-22264); {A908}

procedure HiliteWindow (theWindow: WindowPtr ;
                        fHilite:   MacBool);    external(-22244); {A91C}

procedure BringToFront (theWindow: WindowPtr ) ;external(-22240); {A920}

procedure SendBehind   (theWindow: WindowPtr ;
                        behindWindow: WindowPtr);external(-22239); {A921}

function  FrontWindow  : WindowPtr ;            external(-22236); {A924}

procedure DrawGrowIcon (theWindow: WindowPtr ) ;external(-22268); {A904}

{ Mouse Location ------------------------------------------------------}

function  FindWindow   (thePt:     FPoint;
                        whichWindow: WindowPtr )
                       : integer;               external(-22228); {A92C}

function  TrackGoAway  (theWindow: WindowPtr ;
                        thePt:     FPoint)
                       : MacBool;               external(-22242); {A91E}

{ Window Movement and Sizing ------------------------------------------}

procedure MoveWindow   (theWindow: WindowPtr ;
                        hGlobal,
                        vGlobal:   integer;
                        front:     MacBool);    external(-22245); {A91B}

procedure DragWindow   (theWindow: WindowPtr ;
                        startPt:   FPoint;
                        boundsRect:RectPtr);    external(-22235); {A925}

function  GrowWindow   (theWindow: WindowPtr ;
                        startPt:   FPoint;
                        sizeRect:  RectPtr)
                       : LongInt;               external(-22229); {A92B}

procedure SizeWindow   (theWindow: WindowPtr ;
                        w,h:       integer;
                        fUpdate:   MacBool);    external(-22243); {A91D}
```

```
{ Update Region Maintenance ------------------------------------------------}

procedure InvalRect     (badRect:    RectPtr);     external(-22232); {A928}
procedure InvalRgn      (badRgn:     Handle);      external(-22233); {A927}
procedure ValidRect     (goodRect:   RectPtr);     external(-22230); {A92A}
procedure ValidRgn      (goodRgn:    Handle);      external(-22231); {A929}
procedure BeginUpdate   (theWindow: WindowPtr) ; external(-22238); {A922}
procedure EndUpdate     (theWindow: WindowPtr) ; external(-22237); {A923}

{ Miscellaneous Utilities --------------------------------------------------}

procedure SetWRefCon    (theWindow: WindowPtr ;
                         data:      LongInt);     external(-22248); {A918}

function  GetWRefCon    (theWindow: WindowPtr )
                        : LongInt;                external(-22249); {A917}

procedure SetWindowPic  (theWindow: WindowPtr ;
                         pic:       Handle);      external(-22226); {A92E}

function  GetWindowPic  (theWindow: WindowPtr )
                        : Handle;                 external(-22225); {A92F}

function  PinRect       (theRect:   RectPtr;
                         thePt:     FPoint)
                        : LongInt;                external(-22194); {A94E}

function  DragGrayRgn   (theRgn:    Handle;
                         startPt:   FPoint;
                         limitRect,
                         slopRect:  RectPtr;
                         axis:      integer;
                         actionProc:ProcPtr)
                        : LongInt;                external(-22267); {A905}

{ Low-Level Routines -------------------------------------------------------}

function  CheckUpdate   (theEvent:  EvtRecPtr)
                        : MacBool;                external(-22255); {A911}

procedure ClipAbove     (window:    WindowPtr) ; external(-22261); {A90B}

procedure PaintOne      (window:    WindowPtr ;
                         clobbered: Handle);      external(-22260); {A90C}

procedure PaintBehind   (startWindow:WindowPtr ;
                         clobbered: Handle);      external(-22259); {A90D}

procedure SaveOld       (window:    WindowPtr);  external(-22258); {A90E}

procedure DrawNew       (window:    WindowPtr ;
                         update:    MacBool);     external(-22257); {A90F}

procedure CalcVis       (window:    WindowPtr);  external(-22263); {A909}

procedure CalcVisBehind (startWindow:WindowPtr ;
                         clobbered: Handle);      external(-22262); {A90A}
```

APPENDIX B
ERROR MESSAGES

B.1. Program Startup Errors

Could not open p–machine file
Could not allocate memory for p–machine
Error reading p–machine file
Could not locate MSTR resource
Could not open program data fork
Could not open Runtime Support Library file
Could not allocate stack/heap
I/O error while booting
Memory allocation error while booting
Error reading segment dictionary
Error reading library
Required unit not found
Duplicate unit
Too many library code files referenced
Too many system units referenced
No program in code file to execute
Program or unit must be linked first
Obsolete code segment
Insufficient memory to construct environment
Program environment too complicated: run QUICKSTART first
Error reading program code file
Error reading library code file
Insufficient memory to allocate data segment
Insufficient memory to load fixed position segment
Unknown environment construction error

B.2. Execution Errors

0	Fatal runtime support error
1	Value range error
2	No proc in segment table
3	Exit from uncalled proc
4	Stack overflow
5	Integer overflow
6	Division by zero
7	Invalid memory reference
8	Program interrupted by user
9	Runtime support I/O error
10	I/O Error:
11	Unimplemented instruction
12	Floating point error
13	String overflow
14	Programmed halt
15	Illegal heap operation
16	Break point
17	Incompatible real number size
18	Set too large
19	Segment too large
20	Heap expansion error
21	Insufficient memory to load code segment
98	Unknown I/O Error #
99	Unknown runtime support error

B.3. I/O Errors

0	No error
−17	Control error
−18	Status error
−19	Read error
−20	Write error
−21	Bad unit
−22	Unit empty
−23	Open error
−24	Close error
−25	Driver removal error
−26	Driver resource not found
−27	Cancelled I/O operation
−28	Driver not open
−33	Directory full
−34	Disk full
−35	No such volume mounted
−36	Data transfer error
−37	Bad file name
−38	File not open
−39	End of file
−40	File positioning error
−41	Insufficient memory for file operation
−42	Too many files open
−43	File not found
−44	Diskette is write protected
−45	File is locked
−46	Volume is locked
−47	File is in use and cannot be deleted
−48	Duplicate file name
−49	File already open with write permission
−50	Invalid file operation parameter list
−51	Invalid file reference number
−52	Error establishing file position
−53	Mounted volume not on line
−54	Invalid file open permissions
−55	Volume already on line
−56	Invalid drive number
−57	Not a Macintosh diskette
−58	Not a Macintosh volume
−59	Directory corrupted by file system
−60	Bad master directory block

−61	Write permissions error
−64	Drive not installed
−65	Drive not on line
−1024	Bad input format

B.4. Syntax Errors

1	Error in simple type
2	Identifier expected
3	unimplemented error
4	')' expected
5	': expected
6	This symbol is illegal in this context
7	Error in parameter list
8	'OF' expected
9	'(' expected
10	Error in type
11	'[' expected
12	']' expected
13	'END' expected
14	Semicolon expected
15	Integer expected
16	'=' expected
17	'BEGIN' expected
18	Error in declaration part
19	Error in <field–list>
20	'.' expected
21	'*' expected
22	'INTERFACE' expected
23	'IMPLEMENTATION' expected
24	'UNIT' expected
49	Case label out of range
50	Error in constant
51	':=' expected
52	'THEN' expected
53	'UNTIL' expected
54	'DO' expected
55	'TO' or 'DOWNTO' expected in for statement
56	'IF' expected
57	'FILE' expected
58	Error in <factor> (bad expression)
59	Error in variable
60	Must be of type 'SEMAPHORE'
61	Must be of type 'PROCESSID'
62	Process not allowed at this nesting level
63	Only main task may start processes
101	Identifier declared twice
102	Low bound exceeds high bound

103 Identifier is not of the appropriate class
104 Undeclared identifier
105 Sign not allowed
106 Number expected
107 Incompatible subrange types
108 File not allowed here
109 Type must not be real
110 <tagfield> type must be scalar or subrange
111 Incompatible with <tagfield> part
112 Index type must not be real
113 Index type must be a scalar or a subrange
114 Base type must not be real
115 Base type must be a scalar or a subrange
116 Error in type of standard procedure parameter
117 Unsatisified forward reference
118 Forward reference type identifier in variable declaration
119 Re–specified params not OK for a forward declared procedure
120 Function result type must be scalar, subrange or pointer
121 File value parameter not allowed
122 A forward declared function's result type can't be re–specified
123 Missing result type in function declaration
124 F–format for reals only
125 Error in type of standard function parameter
126 Number of parameters does not agree with declaration
127 Illegal parameter substitution
128 Result type does not agree with declaration
129 Type conflict of operands
130 Expression is not of set type
131 Tests on equality allowed only
132 Strict inclusion not allowed
133 File comparison not allowed
134 Illegal type of operand(s)
135 Type of operand must be Boolean
136 Set element type must be scalar or subrange
137 Set element types must be compatible
138 Type of variable is not array
139 Index type is not compatible with the declaration
140 Type of variable is not record
141 Type of variable must be file or pointer
142 unimplemented error
143 Illegal type of loop control variable
144 Illegal type of expression
145 Type conflict

146 Assignment of files not allowed
147 Label type incompatible with selecting expression
148 Subrange bounds must be scalar
149 Index type must be integer
150 Assignment to standard function is not allowed
151 Assignment to formal function is not allowed
152 No such field in this record
153 Illegal type of parameter for READ
154 Actual parameter must be a variable
155 Control variable cannot be formal or non–local
156 Multidefined case label
157 Too many cases in case statement
158 No such variant in this record
159 Real or string tagfields not allowed
160 Previous declaration was not forward
161 Again forward declared
162 Parameter size must be constant
163 Missing variant in declaration
164 Substition of standard proc/func not allowed
165 Multidefined label
166 Multideclared label
167 Undeclared label
168 Undefined label
169 Error in base set
170 Value parameter expected
171 Standard file was re–declared
172 Undeclared external file
173 Fortran procedure or function expected
174 Pascal function or procedure expected
175 Semaphore value parameter not allowed
176 Undefined forward procedure
182 Nested units not allowed
183 External declaration not allowed at this level
184 External declaration not allowed in INTERFACE section
185 Segment declaration not allowed in INTERFACE section
186 Labels not allowed in INTERFACE section
187 Attempt to open library unsuccessful
188 Unit not declared in previous USES
189 'USES' not allowed at this nesting level
190 Unit not in library
191 Forward declaration was not segment
192 Forward declaration was segment
193 Not enough room for this operation

194 Flag must be declared at top of program
195 Unit not importable
201 Error in real number — digit expected
202 String constant must not exceed source line
203 Integer constant exceeds range
204 8 or 9 in octal number
250 Too many scopes of nested identifiers
251 Too many nested procedures or functions
252 Too many forward references of procedure entries
253 Procedure too long
254 Too many long constants in this procedure
256 Too many external references
257 Too many externals
258 Too many local files
259 Expression too complicated
300 Division by zero
301 No case provided for this value
302 Index expression out of bounds
303 Value to be assigned is out of bounds
304 Element expression out of range
305 Maximum segment number exceeded
306 Unit name same as program name
307 Unit name declared twice
308 Invalid array bounds
309 Bounds may not be of type real
310 Only one dimension may be conformant
311 Must be a variable parameter
312 Must be a conformant array
313 Segment declaration not permitted here
314 PROCEDURE, FUNCTION or PROCESS expected
315 HOST call not permitted here
316 May not be formal procedure
317 May not be formal parameter
318 Invalid file type
319 Must be an untyped file
320 Segment entry not found
321 May not be a conformant array index bound
322 Must be a string constant
323 Must be a variable
324 Must be a declared procedure
325 May not call the main program
326 May not be an expression
327 Maximum code size exceeded

328 May not be a conformant array
329 Structured type too large
330 Too many array elements
331 Inline procedure or function not allowed here
333 Must be declared EXTERNAL to have untyped parameters
398 Implementation restriction
399 Illegal language construction or internal compiler error
400 Illegal character in text
401 Unexpected end of input
402 Error in writing code file, not enough room
403 Error in reading include file
404 Error in writing list file, not enough room
405 'PROGRAM' or 'UNIT' expected
406 Include file not legal
407 Include file nesting limit exceeded
408 INTERFACE section not contained in one file
409 Unit name reserved for system
410 Disk file read or write error
500 Assembler Error

APPENDIX C
P–CODE TABLES

C.1. Numerical Listing

0	00	SLDC:0	Short Load Word Constant
1	01	SLDC:1	Short Load Word Constant
2	02	SLDC:2	Short Load Word Constant
3	03	SLDC:3	Short Load Word Constant
4	04	SLDC:4	Short Load Word Constant
5	05	SLDC:5	Short Load Word Constant
6	06	SLDC:6	Short Load Word Constant
7	07	SLDC:7	Short Load Word Constant
8	08	SLDC:8	Short Load Word Constant
9	09	SLDC:9	Short Load Word Constant
10	0A	SLDC:10	Short Load Word Constant
11	0B	SLDC:11	Short Load Word Constant
12	0C	SLDC:12	Short Load Word Constant
13	0D	SLDC:13	Short Load Word Constant
14	0E	SLDC:14	Short Load Word Constant
15	0F	SLDC:15	Short Load Word Constant
16	10	SLDC:16	Short Load Word Constant
17	11	SLDC:17	Short Load Word Constant
18	12	SLDC:18	Short Load Word Constant
19	13	SLDC:19	Short Load Word Constant
20	14	SLDC:20	Short Load Word Constant
21	15	SLDC:21	Short Load Word Constant
22	16	SLDC:22	Short Load Word Constant
23	17	SLDC:23	Short Load Word Constant
24	18	SLDC:24	Short Load Word Constant
25	19	SLDC:25	Short Load Word Constant
26	1A	SLDC:26	Short Load Word Constant
27	1B	SLDC:27	Short Load Word Constant
28	1C	SLDC:28	Short Load Word Constant
29	1D	SLDC:29	Short Load Word Constant
30	1E	SLDC:30	Short Load Word Constant
31	1F	SLDC:31	Short Load Word Constant
32	20	SLDL:1	Short Load Local Word
33	21	SLDL:2	Short Load Local Word
34	22	SLDL:3	Short Load Local Word
35	23	SLDL:4	Short Load Local Word
36	24	SLDL:5	Short Load Local Word
37	25	SLDL:6	Short Load Local Word
38	26	SLDL:7	Short Load Local Word
39	27	SLDL:8	Short Load Local Word
40	28	SLDL:9	Short Load Local Word
41	29	SLDL:10	Short Load Local Word
42	2A	SLDL:11	Short Load Local Word
43	2B	SLDL:12	Short Load Local Word
44	2C	SLDL:13	Short Load Local Word
45	2D	SLDL:14	Short Load Local Word
46	2E	SLDL:15	Short Load Local Word
47	2F	SLDL:16	Short Load Local Word
48	30	SLDO:1	Short Load Global Word
49	31	SLDO:2	Short Load Global Word
50	32	SLDO:3	Short Load Global Word
51	33	SLDO:4	Short Load Global Word
52	34	SLDO:5	Short Load Global Word

53	35	SLDO:6	Short Load Global Word
54	36	SLDO:7	Short Load Global Word
55	37	SLDO:8	Short Load Global Word
56	38	SLDO:9	Short Load Global Word
57	39	SLDO:10	Short Load Global Word
58	3A	SLDO:11	Short Load Global Word
59	3B	SLDO:12	Short Load Global Word
60	3C	SLDO:13	Short Load Global Word
61	3D	SLDO:14	Short Load Global Word
62	3E	SLDO:15	Short Load Global Word
63	3F	SLDO:16	Short Load Global Word
64	40	SSTP	Short Store Packed
65	41	SLDCD:0	Short Load Doubleword Constant Zero
66	42	SLDLD:1	Short Load Local Doubleword
67	43	SLDLD:2	Short Load Local Doubleword
68	44	SLDLD:3	Short Load Local Doubleword
69	45	SLDLD:4	Short Load Local Doubleword
70	46	SLDLD:5	Short Load Local Doubleword
71	47	SLDLD:6	Short Load Local Doubleword
72	48	SLDOD:1	Short Load Global Doubleword
73	49	SLDOD:2	Short Load Global Doubleword
74	4A	SLDOD:3	Short Load Global Doubleword
75	4B	SLDOD:4	Short Load Global Doubleword
76	4C	SLDOD:5	Short Load Global Doubleword
77	4D	SLDOD:6	Short Load Global Doubleword
78	4E	SLDOD:7	Short Load Global Doubleword
79	4F	SLDOD:8	Short Load Global Doubleword
80	50	SINDD:0	Short Index and Load Doubleword
81	51	SINDD:1	Short Index and Load Doubleword
82	52	SINDD:2	Short Index and Load Doubleword
83	53	SINDD:3	Short Index and Load Doubleword
84	54	SINDD:4	Short Index and Load Doubleword
85	55	SINDD:5	Short Index and Load Doubleword
86	56	SINDD:6	Short Index and Load Doubleword
87	57	SINDD:7	Short Index and Load Doubleword
88	58	LDLD	Load Local Doubleword
89	59	LODD	Load Intermediate Doubleword
90	5A	LDOD	Load GLobal Doubleword
91	5B	LDED	Load External Doubleword
92	5C	INDD	Load Indirect Doubleword
93	5D	STLD	Store Local Doubleword
94	5E	STRD	Store Intermediate Doubleword
95	5F	SROD	Store Global Doubleword
96	60	SLLA:1	Short Load Local Address
97	61	SLLA:2	Short Load Local Address
98	62	SLLA:3	Short Load Local Address
99	63	SLLA:4	Short Load Local Address
100	64	SLLA:5	Short Load Local Address
101	65	SLLA:6	Short Load Local Address
102	66	SLLA:7	Short Load Local Address
103	67	SLLA:8	Short Load Local Address
104	68	SSTL:1	Short Store Local Word
105	69	SSTL:2	Short Store Local Word
106	6A	SSTL:3	Short Store Local Word
107	6B	SSTL:4	Short Store Local Word
108	6C	SSTL:5	Short Store Local Word
109	6D	SSTL:6	Short Store Local Word
110	6E	SSTL:7	Short Store Local Word
111	6F	SSTL:8	Short Store Local Word
112	70	SCXG:1	Short Call External Global Procedure
113	71	SCXG:2	Short Call External Global Procedure
114	72	SCXG:3	Short Call External Global Procedure
115	73	SCXG:4	Short Call External Global Procedure
116	74	SCXG:5	Short Call External Global Procedure
117	75	SCXG:6	Short Call External Global Procedure
118	76	SCXG:7	Short Call External Global Procedure
119	77	SCXG:8	Short Call External Global Procedure
120	78	SIND:0	Short Index and Load Word
121	79	SIND:1	Short Index and Load Word
122	7A	SIND:2	Short Index and Load Word
123	7B	SIND:3	Short Index and Load Word
124	7C	SIND:4	Short Index and Load Word
125	7D	SIND:5	Short Index and Load Word

126	7E	SIND:6	Short Index and Load Word
127	7F	SIND:7	Short Index and Load Word
128	80	LDCB	Load Constant Byte
129	81	LDCI	Load Constant Word
130	82	LCO	Load Constant Offset
131	83	LDC	Load Mulitiple Word Constant
132	84	LLA	Load Local Address
133	85	LDO	Load Global Word
134	86	LAO	Load Global Address
135	87	LDL	Load Local Word
136	88	LDA	Load Intermediate Address
137	89	LOD	Load Intermediate Word
138	8A	UJP	Unconditional Jump
139	8B	UJPL	Unconditional Long Jump
140	8C	MPI	Multiply Integers
141	8D	DVI	Divide Integers
142	8E	STM	Store Multiple Words
143	8F	MODI	Modulo Integers
144	90	CPL	Call Local Procedure
145	91	CPG	Call Global Procedure
146	92	CPI	Call Intermediate Procedure
147	93	CXL	Call Local External Procedure
148	94	CXG	Call Global External Procedure
149	95	CXI	Call Intermediate External Procedure
150	96	RPU	Return From Procedure
151	97	CPF	Call Procedure Formal
152	98	LDCN	Load Constant NIL
153	99	LSL	Load Static Link
154	9A	LDE	Load External Word
155	9B	LAE	Load External Address
156	9C	NOP	No Operation
157	9D	LPR	Load Processor Register
158	9E	BPT	Breakpoint
159	9F	BNOT	Boolean NOT
160	A0	LOR	Logical OR
161	A1	LAND	Logical AND
162	A2	ADI	Add Integers
163	A3	SBI	Subtract Integers
164	A4	STL	Store Local Word
165	A5	SRO	Store Global Word
166	A6	STR	Store Intermediate Word
167	A7	LDB	Load Byte
168	A8	NATIVE	Native Code
169	A9	NATINFO	Native Code Information
170	AA	LEREC	Load Current EREC Pointer
171	AB	CAP	Copy Array Parameter
172	AC	CSP	Copy String Parameter
173	AD	SLOD1	Short Load Intermediate Word (parent)
174	AE	SLOD2	Short Load Intermediate Word (grandparent)
175	AF	UPACK	Unpack Field From Top Of Stack
176	B0	EQUI	Equal Integer Comparison
177	B1	NEQI	Not Equal Integer Comparison
178	B2	LEQI	Less Than or Equal Integer Comparison
179	B3	GEQI	Greater Than or Equal Integer Comparison
180	B4	LEUSW	Less Than or Equal Unsigned Word Comparison
181	B5	GEUSW	Greater Than or Equal Unsigned Word Comparison
182	B6	EQPWR	Equal Set Comparison
183	B7	LEPWR	Less Than or Equal Set Comparison (subset)
184	B8	GEPWR	Greater Than or Equal Set Comparison (superset)
185	B9	EQBYT	Equal Byte Array Comparison
186	BA	LEBYT	Less Than or Equal Byte Array Comparison
187	BB	GEBYT	Greater Than or Equal Byte Array Comparison
188	BC	SRS	Subrange Set
189	BD	SWAP	Swap Words
190	BE	TRUNC	Truncate Real
191	BF	ROUND	Round Real
192	C0	ADR	Add Reals
193	C1	SBR	Subtract Reals
194	C2	MPR	Multiply Reals
195	C3	DVR	Divide Reals
196	C4	STO	Store Word Indirect
197	C5	MOV	Move Words
198	C6	DUPR	Duplicate Real

199	C7	ADJ	Adjust Set
200	C8	STB	Store Byte
201	C9	LDP	Load Packed Field
202	CA	STP	Store Packed Field
203	CB	CHK	Range Check
204	CC	FLT	Float Integer
205	CD	EQREAL	Equal Real Comparison
206	CE	LEREAL	Less Than or Equal Real Comparison
207	CF	GEREAL	Greater Than or Equal Real Comparison
208	D0	LDM	Load Multiple Words
209	D1	SPR	Store Processor Register
210	D2	EFJ	Equal False Jump
211	D3	NFJ	Not Equal False Jump
212	D4	FJP	False Jump
213	D5	FJPL	False Long Jump
214	D6	XJP	Indexed Jump
215	D7	IXA	Index Array
216	D8	IXP	Index Packed Array
217	D9	STE	Store External Word
218	DA	INN	Set Membership Test
219	DB	UNI	Set Union
220	DC	INT	Set Intersection
221	DD	DIF	Set Difference
222	DE	SIGNAL	Signal Semaphore
223	DF	WAIT	Wait On Semaphore
224	E0	ABI	Absolute Value Integer
225	E1	NGI	Negate Integer
226	E2	DUPW	Duplicate Word
227	E3	ABR	Absolute Value Real
228	E4	NGR	Negate Real
229	E5	LNOT	Logical NOT (1's complement)
230	E6	IND	Index and Load Word
231	E7	INC	Increment Word Address
232	E8	EQSTR	Equal String Comparison
233	E9	LESTR	Less Than or Equal String Comparison
234	EA	GESTR	Greater Than or Equal String Comparison
235	EB	ASTR	Assign String
236	EC	CSTR	Check String Index
237	ED	INCI	Increment Integer
238	EE	DECI	Decrement Integer
239	EF	SCPI1	Short Call Intermediate Procedure (parent)
240	F0	SCPI2	Short Call Intermediate Procedure (grandparent)
241	F1	TJP	True Jump
242	F2	LDCRL	Load Real Constant
243	F3	LDRL	Load Real
244	F4	STRL	Store Real
245	F5	STOD	Store Indirect Doubleword
246	F6	STED	Store External Doubleword
247	F7	ADI2	Add Integer2
248	F8	SBI2	Subtract Integer2
249	F9	MPI2	Multiply Integer2
250	FA	DVI2	Divide Integer2
251	FB	INC2	Increment Integer2
252	FC	RED2	Reduce Integer2 to Integer
253	FD	EXTI	Extend Integer To Integer2
254	FE	INCBI	Increment Pointer With Integer Byte Offset
0	FF 00	LDCD	Load Constant Doubleword
1	FF 01	DUPD	Duplicate Doubleword
2	FF 02	SWAPD	Swap Doublewords
3	FF 03	MDI2	Modulo Integer2
4	FF 04	DEC2	Decrement Integer2
5	FF 05	NEG2	Negate Integer2
6	FF 06	ABS2	Absolute Value Integer2
7	FF 07	EQI2	Equal Integer2 Comparison
8	FF 08	NEI2	Not Equal Integer2 Comparison
9	FF 09	LEI2	Less Than Or Equal Integer2 Comparison
10	FF 0A	GEI2	Greater Than Or Equal Integer2 Comparison
11	FF 0B	IXA2	Index Array Integer2
12	FF 0C	IXP2	Index Packed Array Integer2
13	FF 0D	INCB2	Increment Pointer With Integer2 Byte Offset
14	FF 0E	XJP2	Indexed Jump Integer2
15	FF 0F	CHK2	Integer2 Rangecheck
16	FF 10	REXTI	Reversed Extend Integer

17	FF 11	RFLT	Reversed Float Integer
18	FF 12	FLT2	Float Integer2
19	FF 13	RFLT2	Reversed Float Integer2
20	FF 14	ADIU	Add Integer Unsigned
21	FF 15	SBIU	Subtract Integer Unsigned
22	FF 16	MPIU	Multiply Integer Unsigned
23	FF 17	DVIU	Divide Integer Unsigned
24	FF 18	MDIU	Modulo Integer Unsigned
25	FF 19	INCU	Increment Integer Unsigned
26	FF 1A	DECU	Decrement Integer Unsigned
27	FF 1B	CHKU	Unsigned Integer Rangecheck
28	FF 1C	REDU	Reduce Integer2 To Unsigned Integer
29	FF 1D	EXTU	Extend Unsigned Integer To Integer2
30	FF 1E	REXTU	Reversed Extend Unsigned Integer To Integer2
31	FF 1F	FLTU	Float Unsigned Integer
32	FF 20	RFLTU	Reversed Float Unsigned Integer
33	FF 21	LSLW	Logical Shift Left Word
34	FF 22	LSRW	Logical Shift Right Word
35	FF 23	ASRW	Arithmetic Shift Right Word
36	FF 24	LSLD	Logical Shift Left Doubleword
37	FF 25	LSRD	Logical Shift Right Doubleword
38	FF 26	ASRD	Arithmetic Shift Right Doubleword
39	FF 27	LANDD	Logical AND Doubleword
40	FF 28	LORD	Logical OR Doubleword
41	FF 29	LNOTD	Logical NOT Doubleword
42	FF 2A	LXORW	Logical Exclusive OR Word
43	FF 2B	LXORD	Logical Exclusive OR Doubleword
44	FF 2C	PTO	Pointer To Word Offset
45	FF 2D	OTP	Word Offset To Pointer
46	FF 2E	TRNC2	Truncate Real to Integer2
47	FF 2F	ROND2	Round Real to Integer2
48	FF 30	--	Unassigned
49	FF 31	--	Unassigned
50	FF 32	RCALL	Macintosh ROM Call
51	FF 33	PTA	Pointer To Absolute Address
52	FF 34	ATP	Absolute Address To Pointer
53	FF 35	AMOVE	Absolute Move Left
54	FF 36	DEREF	Dereference Absolute Handle
55	FF 37	SETAR	Set Action Routine

C.2. Alphabetical Listing

ABI	224		E0	Absolute Value Integer
ABR	227		E3	Absolute Value Real
ABS2	6	FF	06	Absolute Value Integer2
ADI	162		A2	Add Integers
ADI2	247		F7	Add Integer2
ADIU	20	FF	14	Add Integer Unsigned
ADJ	199		C7	Adjust Set
ADR	192		C0	Add Reals
AMOVE	53	FF	35	Absolute Move Left
ASRD	38	FF	26	Arithmetic Shift Right Doubleword
ASRW	35	FF	23	Arithmetic Shift Right Word
ASTR	235		EB	Assign String
ATP	52	FF	34	Absolute Address To Pointer
BNOT	159		9F	Boolean NOT
BPT	158		9E	Breakpoint
CAP	171		AB	Copy Array Parameter
CHK	203		CB	Range Check
CHK2	15	FF	0F	Integer2 Rangecheck
CHKU	27	FF	1B	Unsigned Integer Rangecheck
CPF	151		97	Call Procedure Formal
CPG	145		91	Call Global Procedure
CPI	146		92	Call Intermediate Procedure
CPL	144		90	Call Local Procedure
CSP	172		AC	Copy String Parameter
CSTR	236		EC	Check String Index
CXG	148		94	Call Global External Procedure
CXI	149		95	Call Intermediate External Procedure
CXL	147		93	Call Local External Procedure
DEC2	4	FF	04	Decrement Integer2
DECI	238		EE	Decrement Integer
DECU	26	FF	1A	Decrement Integer Unsigned
DEREF	54	FF	36	Dereference Absolute Handle
DIF	221		DD	Set Difference
DUPD	1	FF	01	Duplicate Doubleword
DUPR	198		C6	Duplicate Real
DUPW	226		E2	Duplicate Word
DVI	141		8D	Divide Integers
DVI2	250		FA	Divide Integer2
DVIU	23	FF	17	Divide Integer Unsigned
DVR	195		C3	Divide Reals
EFJ	210		D2	Equal False Jump
EQBYT	185		B9	Equal Byte Array Comparison
EQI2	7	FF	07	Equal Integer2 Comparison
EQPWR	182		B6	Equal Set Comparison
EQREAL	205		CD	Equal Real Comparison
EQSTR	232		E8	Equal String Comparison
EQUI	176		B0	Equal Integer Comparison
EXTI	253		FD	Extend Integer To Integer2
EXTU	29	FF	1D	Extend Unsigned Integer To Integer2
FJP	212		D4	False Jump
FJPL	213		D5	False Long Jump
FLT	204		CC	Float Integer
FLT2	18	FF	12	Float Integer2
FLTU	31	FF	1F	Float Unsigned Integer
GEBYT	187		BB	Greater Than or Equal Byte Array Comparison
GEI2	10	FF	0A	Greater Than Or Equal Integer2 Comparison
GEPWR	184		B8	Greater Than or Equal Set Comparison (superset)
GEQI	179		B3	Greater Than or Equal Integer Comparison
GEREAL	207		CF	Greater Than or Equal Real Comparison
GESTR	234		EA	Greater Than or Equal String Comparison
GEUSW	181		B5	Greater Than or Equal Unsigned Word Comparison
INC	231		E7	Increment Word Address
INC2	251		FB	Increment Integer2
INCB2	13	FF	0D	Increment Pointer With Integer2 Byte Offset
INCBI	254		FE	Increment Pointer With Integer Byte Offset
INCI	237		ED	Increment Integer
INCU	25	FF	19	Increment Integer Unsigned
IND	230		E6	Index and Load Word

INDD	92		5C	Load Indirect Doubleword
INN	218		DA	Set Membership Test
INT	220		DC	Set Intersection
IXA	215		D7	Index Array
IXA2	11	FF	0B	Index Array Integer2
IXP	216		D8	Index Packed Array
IXP2	12	FF	0C	Index Packed Array Integer2
LAE	155		9B	Load External Address
LAND	161		A1	Logical AND
LANDD	39	FF	27	Logical AND Doubleword
LAO	134		86	Load Global Address
LCO	CL		82	Load Constant Offset
LDA	136		88	Load Intermediate Address
LDB	167		A7	Load Byte
LDC	131		83	Load Mulitiple Word Constant
LDCB	128		80	Load Constant Byte
LDCD	0	FF	00	Load Constant Doubleword
LDCI	129		81	Load Constant Word
LDCN	152		98	Load Constant NIL
LDCRL	242		F2	Load Real Constant
LDE	154		9A	Load External Word
LDED	91		5B	Load External Doubleword
LDL	135		87	Load Local Word
LDLD	88		58	Load Local Doubleword
LDM	208		D0	Load Multiple Words
LDO	133		85	Load Global Word
LDOD	90		5A	Load GLobal Doubleword
LDP	201		C9	Load Packed Field
LDRL	243		F3	Load Real
LEBYT	186		BA	Less Than or Equal Byte Array Comparison
LEI2	9	FF	09	Less Than Or Equal Integer2 Comparison
LEPWR	183		B7	Less Than or Equal Set Comparison (subset)
LEQI	178		B2	Less Than or Equal Integer Comparison
LEREAL	206		CE	Less Than or Equal Real Comparison
LEREC	170		AA	Load Current EREC Pointer
LESTR	233		E9	Less Than or Equal String Comparison
LEUSW	180		B4	Less Than or Equal Unsigned Word Comparison
LLA	132		84	Load Local Address
LNOT	229		E5	Logical NOT (1's complement)
LNOTD	41	FF	29	Logical NOT Doubleword
LOD	137		89	Load Intermediate Word
LODD	89		59	Load Intermediate Doubleword
LOR	160		A0	Logical OR
LORD	40	FF	28	Logical OR Doubleword
LPR	157		9D	Load Processor Register
LSL	153		99	Load Static Link
LSLD	36	FF	24	Logical Shift Left Doubleword
LSLW	33	FF	21	Logical Shift Left Word
LSRD	37	FF	25	Logical Shift Right Doubleword
LSRW	34	FF	22	Logical Shift Right Word
LXORD	43	FF	2B	Logical Exclusive OR Doubleword
LXORW	42	FF	2A	Logical Exclusive OR Word
MDI2	3	FF	03	Modulo Integer2
MDIU	24	FF	18	Modulo Integer Unsigned
MODI	143		8F	Modulo Integers
MOV	197		C5	Move Words
MPI	140		8C	Multiply Integers
MPI2	249		F9	Multiply Integer2
MPIU	22	FF	16	Multiply Integer Unsigned
MPR	194		C2	Multiply Reals
NATINFO	169		A9	Native Code Information
NATIVE	168		A8	Native Code
NEG2	5	FF	05	Negate Integer2
NEI2	8	FF	08	Not Equal Integer2 Comparison
NEQI	177		B1	Not Equal Integer Comparison
NFJ	211		D3	Not Equal False Jump
NGI	225		E1	Negate Integer
NGR	228		E4	Negate Real
NOP	156		9C	No Operation
OTP	45	FF	2D	Word Offset To Pointer
PTA	51	FF	33	Pointer To Absolute Address
PTO	44	FF	2C	Pointer To Word Offset
RCALL	50	FF	32	Macintosh ROM Call

RED2	252		FC	Reduce Integer2 to Integer
REDU	28	FF	1C	Reduce Integer2 To Unsigned Integer
REXTI	16	FF	10	Reversed Extend Integer
REXTU	30	FF	1E	Reversed Extend Unsigned Integer To Integer2
RFLT	17	FF	11	Reversed Float Integer
RFLT2	19	FF	13	Reversed Float Integer2
RFLTU	32	FF	20	Reversed Float Unsigned Integer
ROND2	47	FF	2F	Round Real to Integer2
ROUND	191		BF	Round Real
RPU	150		96	Return From Procedure
SBI	163		A3	Subtract Integers
SBI2	248		F8	Subtract Integer2
SBIU	21	FF	15	Subtract Integer Unsigned
SBR	193		C1	Subtract Reals
SCPI1	239		EF	Short Call Intermediate Procedure (parent)
SCPI2	240		F0	Short Call Intermediate Procedure (grandparent)
SCXG:1	112		70	Short Call External Global Procedure
SCXG:2	113		71	Short Call External Global Procedure
SCXG:3	114		72	Short Call External Global Procedure
SCXG:4	115		73	Short Call External Global Procedure
SCXG:5	116		74	Short Call External Global Procedure
SCXG:6	117		75	Short Call External Global Procedure
SCXG:7	118		76	Short Call External Global Procedure
SCXG:8	119		77	Short Call External Global Procedure
SETAR	55	FF	37	Set Action Routine
SIGNAL	222		DE	Signal Semaphore
SIND:0	120		78	Short Index and Load Word
SIND:1	121		79	Short Index and Load Word
SIND:2	122		7A	Short Index and Load Word
SIND:3	123		7B	Short Index and Load Word
SIND:4	124		7C	Short Index and Load Word
SIND:5	125		7D	Short Index and Load Word
SIND:6	126		7E	Short Index and Load Word
SIND:7	127		7F	Short Index and Load Word
SINDD:0	80		50	Short Index and Load Doubleword
SINDD:1	81		51	Short Index and Load Doubleword
SINDD:2	82		52	Short Index and Load Doubleword
SINDD:3	83		53	Short Index and Load Doubleword
SINDD:4	84		54	Short Index and Load Doubleword
SINDD:5	85		55	Short Index and Load Doubleword
SINDD:6	86		56	Short Index and Load Doubleword
SINDD:7	87		57	Short Index and Load Doubleword
SLDC:0	0		00	Short Load Word Constant
SLDC:1	1		01	Short Load Word Constant
SLDC:2	2		02	Short Load Word Constant
SLDC:3	3		03	Short Load Word Constant
SLDC:4	4		04	Short Load Word Constant
SLDC:5	5		05	Short Load Word Constant
SLDC:6	6		06	Short Load Word Constant
SLDC:7	7		07	Short Load Word Constant
SLDC:8	8		08	Short Load Word Constant
SLDC:9	9		09	Short Load Word Constant
SLDC:10	10		0A	Short Load Word Constant
SLDC:11	11		0B	Short Load Word Constant
SLDC:12	12		0C	Short Load Word Constant
SLDC:13	13		0D	Short Load Word Constant
SLDC:14	14		0E	Short Load Word Constant
SLDC:15	15		0F	Short Load Word Constant
SLDC:16	16		10	Short Load Word Constant
SLDC:17	17		11	Short Load Word Constant
SLDC:18	18		12	Short Load Word Constant
SLDC:19	19		13	Short Load Word Constant
SLDC:20	20		14	Short Load Word Constant
SLDC:21	21		15	Short Load Word Constant
SLDC:22	22		16	Short Load Word Constant
SLDC:23	23		17	Short Load Word Constant
SLDC:24	24		18	Short Load Word Constant
SLDC:25	25		19	Short Load Word Constant
SLDC:26	26		1A	Short Load Word Constant
SLDC:27	27		1B	Short Load Word Constant
SLDC:28	28		1C	Short Load Word Constant
SLDC:29	29		1D	Short Load Word Constant
SLDC:30	30		1E	Short Load Word Constant

```
SLDC:31      31      1F  Short Load Word Constant
SLDCD:0      65      41  Short Load Doubleword Constant Zero
SLDL:1       32      20  Short Load Local Word
SLDL:2       33      21  Short Load Local Word
SLDL:3       34      22  Short Load Local Word
SLDL:4       35      23  Short Load Local Word
SLDL:5       36      24  Short Load Local Word
SLDL:6       37      25  Short Load Local Word
SLDL:7       38      26  Short Load Local Word
SLDL:8       39      27  Short Load Local Word
SLDL:9       40      28  Short Load Local Word
SLDL:10      41      29  Short Load Local Word
SLDL:11      42      2A  Short Load Local Word
SLDL:12      43      2B  Short Load Local Word
SLDL:13      44      2C  Short Load Local Word
SLDL:14      45      2D  Short Load Local Word
SLDL:15      46      2E  Short Load Local Word
SLDL:16      47      2F  Short Load Local Word
SLDLD:1      66      42  Short Load Local Doubleword
SLDLD:2      67      43  Short Load Local Doubleword
SLDLD:3      68      44  Short Load Local Doubleword
SLDLD:4      69      45  Short Load Local Doubleword
SLDLD:5      70      46  Short Load Local Doubleword
SLDLD:6      71      47  Short Load Local Doubleword
SLDO:1       48      30  Short Load Global Word
SLDO:2       49      31  Short Load Global Word
SLDO:3       50      32  Short Load Global Word
SLDO:4       51      33  Short Load Global Word
SLDO:5       52      34  Short Load Global Word
SLDO:6       53      35  Short Load Global Word
SLDO:7       54      36  Short Load Global Word
SLDO:8       55      37  Short Load Global Word
SLDO:9       56      38  Short Load Global Word
SLDO:10      57      39  Short Load Global Word
SLDO:11      58      3A  Short Load Global Word
SLDO:12      59      3B  Short Load Global Word
SLDO:13      60      3C  Short Load Global Word
SLDO:14      61      3D  Short Load Global Word
SLDO:15      62      3E  Short Load Global Word
SLDO:16      63      3F  Short Load Global Word
SLDOD:1      72      48  Short Load Global Doubleword
SLDOD:2      73      49  Short Load Global Doubleword
SLDOD:3      74      4A  Short Load Global Doubleword
SLDOD:4      75      4B  Short Load Global Doubleword
SLDOD:5      76      4C  Short Load Global Doubleword
SLDOD:6      77      4D  Short Load Global Doubleword
SLDOD:7      78      4E  Short Load Global Doubleword
SLDOD:8      79      4F  Short Load Global Doubleword
SLLA:1       96      60  Short Load Local Address
SLLA:2       97      61  Short Load Local Address
SLLA:3       98      62  Short Load Local Address
SLLA:4       99      63  Short Load Local Address
SLLA:5      100      64  Short Load Local Address
SLLA:6      101      65  Short Load Local Address
SLLA:7      102      66  Short Load Local Address
SLLA:8      103      67  Short Load Local Address
SLOD1       173      AD  Short Load Intermediate Word (parent)
SLOD2       174      AE  Short Load Intermediate Word (grandparent)
SPR         209      D1  Store Processor Register
SRO         165      A5  Store Global Word
SROD         95      5F  Store Global Doubleword
SRS         188      BC  Subrange Set
SSTL:1      104      68  Short Store Local Word
SSTL:2      105      69  Short Store Local Word
SSTL:3      106      6A  Short Store Local Word
SSTL:4      107      6B  Short Store Local Word
SSTL:5      108      6C  Short Store Local Word
SSTL:6      109      6D  Short Store Local Word
SSTL:7      110      6E  Short Store Local Word
SSTL:8      111      6F  Short Store Local Word
SSTP         64      40  Short Store Packed
STB         200      C8  Store Byte
STE         217      D9  Store External Word
```

STED	246		F6	Store External Doubleword
STL	164		A4	Store Local Word
STLD	93		5D	Store Local Doubleword
STM	142		8E	Store Multiple Words
STO	196		C4	Store Word Indirect
STOD	245		F5	Store Indirect Doubleword
STP	202		CA	Store Packed Field
STR	166		A6	Store Intermediate Word
STRD	94		5E	Store Intermediate Doubleword
STRL	244		F4	Store Real
SWAP	189		BD	Swap Words
SWAPD	2	FF	02	Swap Doublewords
TJP	241		F1	True Jump
TRNC2	46	FF	2E	Truncate Real to Integer2
TRUNC	190		BE	Truncate Real
UJP	138		8A	Unconditional Jump
UJPL	139		8B	Unconditional Long Jump
UNI	219		DB	Set Union
UPACK	175		AF	Unpack Field From Top Of Stack
WAIT	223		DF	Wait On Semaphore
XJP	214		D6	Indexed Jump
XJP2	14	FF	0E	Indexed Jump Integer2

C.3. p–Code Index

The following list defines the codes used in the p–code index that follows. Each code corresponds to the name of a section of the P–MACHINE ARCHITECTURE chapter.

BAC	Byte Array Comparisons
BLS	Byte Load and Store
CL	Constant Loads
CS	Concurrency Support
DLS	Indirect Load and Stores
ELS	External Loads and Stores
GLS	Global Loads and Stores
IA	Integer Arithmetic
ILS	Intermediate Loads and Stores
JMP	Jumps
LLS	Local Loads and Stores
LO	Logical Operators
MI	Miscellaneous Instructions
MLS	Multiple Word Loads and Stores
OTC	Operand Type Conversion Operators
PC	Parameter Copying
PF	Packed Field Loads and Stores
RA	Real Arithmetic
RCR	Routine Calls and Returns
SET	Set Operators
SIA	Structure Indexing and Assignment
SO	Shift Operators
STR	String Operations
UA	Unsigned Arithmetic

The following index indicates for each p–code which section of the P–MACHINE ARCHITECTURE chapter it may be found in.

ABI	IA	ADJ	SET	ATP	OTC
ABR	RA	ADR	RA	BNOT	LO
ABS2	IA	AMOVE	SIA	BPT	RCR
ADI	IA	ASRD	SO	CAP	PC
ADI2	IA	ASRW	SO	CHK	IA
ADIU	UA	ASTR	STR	CHK2	IA

CHKU	UA	INC	SIA	LEUSW	LO
CPF	RCR	INC2	IA	LLA	LLS
CPG	RCR	INCB2	SIA	LNOT	LO
CPI	RCR	INCBI	SIA	LNOTD	LO
CPL	RCR	INCI	IA	LOD	ILS
CSP	PC	INCU	UA	LODD	ILS
CSTR	STR	IND	DLS	LOR	LO
CXG	RCR	INDD	DLS	LORD	LO
CXI	RCR	INN	SET	LPR	MI
CXL	RCR	INT	SET	LSL	RCR
DEC2	IA	IXA	SIA	LSLD	SO
DECI	IA	IXA2	SIA	LSLW	SO
DECU	UA	IXP	SIA	LSRD	SO
DEREF	OTC	IXP2	SIA	LSRW	SO
DIF	SET	LAE	ELS	LXORD	LO
DUPD	MI	LAND	LO	LXORW	LO
DUPR	MI	LANDD	LO	MDI2	IA
DUPW	MI	LAO	GLS	MDIU	UA
DVI	IA	LCO	CL	MODI	IA
DVI2	IA	LDA	ILS	MOV	SIA
DVIU	UA	LDB	BLS	MPI	IA
DVR	RA	LDC	MLS	MPI2	IA
EFJ	JMP	LDCB	CL	MPIU	UA
EQBYT	BAC	LDCD	CL	MPR	RA
EQI2	IA	LDCI	CL	NATINFO	MI
EQPWR	SET	LDCN	CL	NATIVE	MI
EQREAL	RA	LDCRL	MLS	NEG2	IA
EQSTR	STR	LDE	ELS	NEI2	IA
EQUI	IA	LDED	ELS	NEQI	IA
EXTI	OTC	LDL	LLS	NFJ	JMP
EXTU	OTC	LDLD	LLS	NGI	IA
FJP	JMP	LDM	MLS	NGR	RA
FJPL	JMP	LDO	GLS	NOP	MI
FLT	OTC	LDOD	GLS	OTP	OTC
FLT2	OTC	LDP	PF	PTA	OTC
FLTU	OTC	LDRL	MLS	PTO	OTC
GEBYT	BAC	LEBYT	BAC	RCALL	MI
GEI2	IA	LEI2	IA	RED2	OTC
GEPWR	SET	LEPWR	SET	REDU	OTC
GEQI	IA	LEQI	IA	REXTI	OTC
GEREAL	RA	LEREAL	RA	REXTU	OTC
GESTR	STR	LEREC	MI	RFLT	OTC
GEUSW	LO	LESTR	STR	RFLT2	OTC

RFLTU	OTC	SWAPD	MI
ROND2	OTC	TJP	JMP
ROUND	OTC	TRNC2	OTC
RPU	RCR	TRUNC	OTC
SBI	IA	UJP	JMP
SBI2	IA	UJPL	JMP
SBIU	UA	UNI	SET
SBR	RA	UPACK	PF
SCPI1	RCR	WAIT	CS
SCPI2	RCR	XJP	JMP
SCXGn	RCR	XJP2	JMP
SETAR	MI		
SIGNAL	CS		
SINDn	DLS		
SINDDn	DLS		
SLDCn	CL		
SLDCD0	CL		
SLDLn	LLS		
SLDLDn	LLS		
SLDOn	GLS		
SLDODn	GLS		
SLLAn	LLS		
SLOD1	ILS		
SLOD2	ILS		
SPR	MI		
SRO	GLS		
SROD	GLS		
SRS	SET		
SSTLn	LLS		
SSTP	PF		
STB	BLS		
STE	ELS		
STED	ELS		
STL	LLS		
STLD	LLS		
STM	MLS		
STO	DLS		
STOD	DLS		
STP	PF		
STR	ILS		
STRD	ILS		
STRL	MLS		
SWAP	MI		

INDEX

USUS: UCSD p–System User's Society

USUS is the society devoted to users of the p–System and UCSD Pascal. Its goal is to promote and influence the development of the p–System and UCSD Pascal, as well as to help users learn more about their systems.

USUS provides both formal and informal opportunities for members to communicate with and learn from each other. Its semiannual national meetings and quarterly newsletters feature technical presentations and discussions as well as news about the p–System and its derivatives. Electronic mail bulletin boards put you in touch with a member network that can provide up–to–the–minute information, and special interest groups zero in on specific problem areas. USUS also supports a Software Exchange Library from which members can obtain software source code for a nominal reproduction charge.

USUS stands for the UCSD p–System User's Society and is pronounced "use us." It is nonprofit and vendor independent.

As a user of UCSD Pascal, USUS is for you. USUS links you with a community of users who share your interests. The following benefits are available to USUS members:

SOFTWARE EXCHANGE LIBRARY
- Tools, games, aides
- Pascal source
- Nominally priced

INFORMATIVE NATIONAL MEETINGS
- Tutorials
- Technical presentations
- Special interest group meetings
- Low–cost software library access
- Hardware/software demonstrations
- Query "major vendors"

HELP VIA ELECTRONIC COMMUNICATIONS
- CompuServe/MUSUS SIG
  - Bulletin board
  - Data bases
  - Software library
- Telemail

USEFUL QUARTERLY NEWSLETTER
- Technical articles and updates
- SIG reports
- Software vendor directory
- Library catalog listings
- Organizational news

ACTIVIST SPECIAL INTEREST GROUPS

TECHNICAL ARCHIVE

USUS MEMBERSHIP APPLICATION

I am applying for:

☐ $25 individual membership (North America residents)
☐ $40 individual membership (For those residing outside North America; includes $15 airmail service surcharge.)
☐ $500 organizational membership

Name_____

Address_____

_____

City_____State____ Zip_____Country_____

Phone ( )_____TWX/Telex/EMail_____

Title/Affiliation_____

Option: ☐ Do not print my phone number in **USUS** rosters.
Option: ☐ Print only my name and country in **USUS** rosters.
Option: ☐ Do not release my name on mailing lists.

Computer System:

☐ Z–80	☐ 8080	☐ PDP/LSI–11
☐ 6502/Apple	☐ 6800	☐ 6809
☐ 9900	☐ 8086/8088	☐ Z8000
☐ 68000	☐ MicroEngine	☐ IBM PC
☐ Macintosh	☐ Other_____	

USUS MEMBERSHIP APPLICATION

I am interested in the following Committees/Special Interest Groups (SIGs):

- ☐ Advanced System Editor SIG
- ☐ Sage SIG
- ☐ IBM Display Writer SIG
- ☐ Application Developer's SIG
- ☐ Technical Issues Committee
- ☐ Mettings Committee
- ☐ DEC SIG
- ☐ UCSD Pascal Compatability SIG
- ☐ Publications Committee
- ☐ Graphics SIG
- ☐ Apple SIG
- ☐ Software Exchange Library
- ☐ IBM PC SIG
- ☐ Communications SIG
- ☐ Texas Instruments SIG
- ☐ Modula–2 SIG
- ☐ File Access SIG
- ☐ Word Processing SIG

I am willing to volunteer some time and/or talent in the following area(s):

_____

_____

Mail completed application with check or money order payable to **USUS** and drawn on a U.S. bank or U.S. office, to Secretary,**USUS**, P.O. Box 1148, La Jolla, CA 92038, USA.