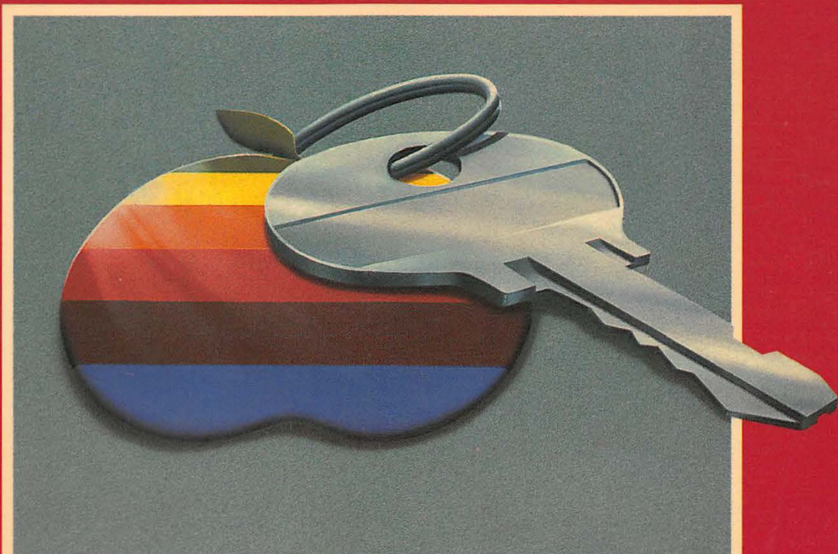
SYBEX

Programming the Macintosh™ in Assembly Language

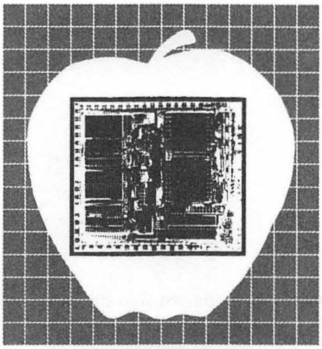


Steve Williams

BERTE

Programming the Macintosh in Assembly Language

Programming the Macintosh in Assembly Language



Steve Williams

SYBEX®

Berkeley • Paris • Düsseldorf • London

Cover art by Peni Choux
Book design by Sharon Leong

Library of Congress Card Number: 85-62542
ISBN 0-89588-263-9
Printed by Haddon Craftsmen
Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

To Jim and Sylvia Williams

CONTENTS

Chapter 2 29

68000 Architecture

Appendix A — 545

Answers to Exercises

Appendix B — 565

Macintosh Character Set

Appendix C — 571

Macro Library Summary

Appendix E — 627

Source Listings

ACKNOWLEDGMENTS

This book owes its very existence to Carole Alden, Jim Hill, and Rudolph Langer of SYBEX. Without their gentle insistence it would never have been attempted. Thanks to Henri Socha for a most thorough technical review of the book. Special thanks to my editor, Barbara Gordon, whose patience in the face of the insanity surrounding the production of this book qualifies her for sainthood. Thanks also to Bonnie Gruen, line editor; Dave Clark and Olivia Shinomoto, word processing; Sharon Leong and Ingrid Owen, book design and layout; Donna Scanlon, typesetting; Dawn Amsberry, Eileen Walsh, and Jon Strickland, proofreading.

Pacific Grove, California
Labor Day, 1985

INTRODUCTION

Ideas control the world.

—James A. Garfield

This book is a "how-to" book; its goal is to teach you how to write assembly language programs for the Apple Macintosh computer. To do this, there are three areas of information you have to learn about:

1. Assembly language programming
2. Programming for the 68000 chip inside the Macintosh
3. The Macintosh Development System software

Because this is rather a large amount of material to cover in a single volume, I decided to provide standardized software that you can use in the early phases of learning to program. In this way you can learn a small part of the system without having to learn all of the mechanics necessary for writing a complete program. You can therefore write rather substantial programs without knowing how Macintosh applications manage windows, files, and other parts of the system. By the end of the book you will understand how the supplied software works, and you can then modify or rewrite it to suit your needs.

One of the most valuable aspects of this book is the example software, which includes listings for most common Macintosh functions. These listings provide a supplement to *Inside Macintosh.* The Apple documentation gives you a lot of detailed information about each system call, but it gives you little information on how to combine these calls to produce an application. I have tried to fill this gap.

Depending on your experience with assembly language programming and the 68000 chip, you may choose to direct your attention to specific chapters. Chapter 1 covers the basic computer hardware and software concepts, and describes how assembly language programs relate to their hardware environment. It provides common definitions for the terms that will be used throughout the book.

If you have prior assembly language experience but have never used the Motorola 68000, start with Chapter 2, which covers 68000 architecture. Chapter 3 continues this discussion by detailing first the instruction classes, then the mechanics of developing a program, and finally the 68000 instruction set.

If you have prior experience with 68000 assembly language programming, read the section titled "Program Development Mechanics" in Chapter 3. This section describes how to use the Macintosh Development System software. Then go on to Chapter 4, which deals with elementary Macintosh programs, including terminal and file I/O operations.

Chapters 5 and 6 are more advanced. Chapter 5 covers advanced programming concepts required to write major applications, and Chapter 6 covers the differences between the Macintosh and conventional programming environments. Chapter 6 also includes a sample Macintosh application.

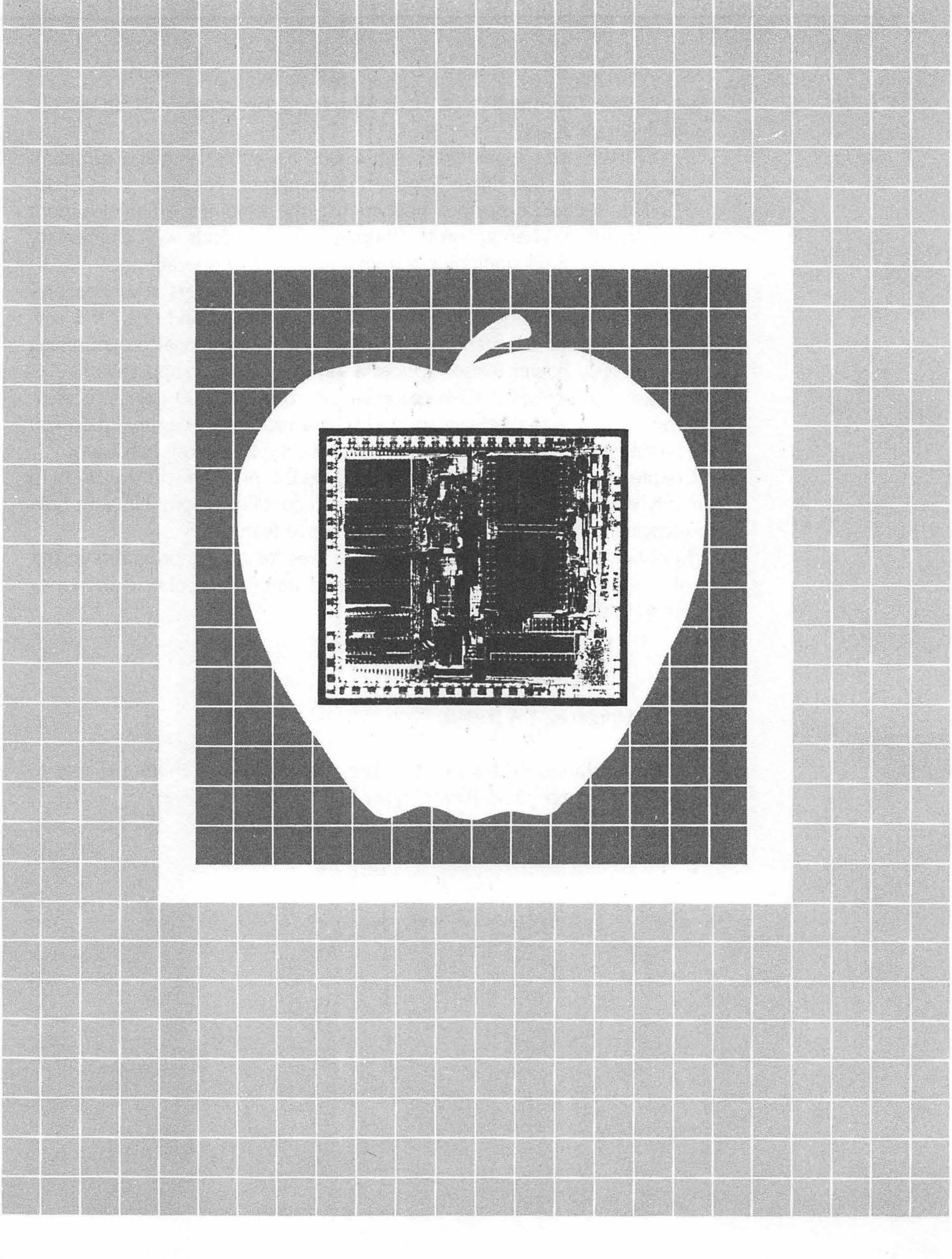
Chapter 7 describes the exception feature of the 68000 microprocessor and how it relates to the Macintosh. This information is necessary when you are writing programs that interact directly with the Macintosh hardware.

Chapter 8 describes the software used by the programs in this book to simplify writing Macintosh programs. It also contains helpful hints on how to reconfigure this software to add or remove features.

To obtain a disk containing the source files for all the programs in this book, send $19.95 plus $2.50 postage and handling in check or money order to this address:

MacDoodle Source Disk
P.O. Box 3262
Monterey, CA 93940

California residents add 6 percent sales tax. Make the check payable to *MacDoodle Source Disk*. Please allow 6–8 weeks for delivery.



Chapter **1**

Basic Concepts

The journey of a thousand miles begins with a single step.

—Lao Tzu

WHAT IS PROGRAMMING?

Machine-level computer programming has been called art, engineering, sorcery, and religion. It is all of these and more. Dealing with a computer at its own level can be a very rewarding (and frustrating) experience.

This book will introduce you to the joys and woes of this wondrous craft, using the Apple Macintosh as a vehicle. We have attempted to minimize the mathematics involved; anyone who can add, subtract, multiply, and divide can make full use of all the material presented here.

This chapter will introduce you to the concepts on which modern computing is founded: algorithms, elementary hardware operations, and the binary and hexadecimal number schemes.

As a start toward learning how to program, consider the process of starting an automobile engine:

1. Insert the key into the lock.
2. Turn the key past the ON position to the START position and hold.
3. If the motor does not crank, stop.
4. If the motor does catch, proceed to step 5. If the motor does not catch within 30 seconds, turn the key OFF and go to step 2.
5. Release the key back to the ON position, and stop.

This simple procedure has all of the same elements of a computer program:

- Step 1 is called an *initialization*. This is an action which is performed once at the start of a program.
- Steps 2 through 4 form a *loop*—a series of actions which is repeated until some condition is satisfied. (In this case, either the car starts or the battery expires.)
- Step 3 is an example of an *error condition*—some condition which causes the procedure to terminate in an abnormal fashion.
- Step 5 is the successful completion of the procedure.

Algorithms

The above procedure, in computer terminology, is called an *algorithm*. Algorithms are stepwise procedures which can be used to define the steps in programs. Any step-by-step description is an algorithm. Some examples from everyday life are kitchen recipes and directions for getting from one place to another.

You can see from these examples that not all algorithms can be made into computer programs. Even if an algorithm is suitable for transforming into a program, it must first be put into a form the computer can recognize. Computers cannot utilize even the simplest English.

Programming Languages

If you want a computer to carry out the steps you define in an algorithm, you must first translate the English description into a language that the computer can execute. Such a language is called a *programming language*. There are many such languages. The task to be performed generally dictates which language is to be used. Some examples of programming languages are:

- BASIC (Beginner's All-purpose Symbolic Instruction Code). BASIC is a very simple language to learn and to use. It is generally used for short, simple programs.
- COBOL (COmmon Business Oriented Language). COBOL is commonly used for business related software, such as payroll and other accounting applications.

- FORTRAN (FORmula TRANslation). FORTRAN is widely used in the scientific community for applications involving large numbers of calculations.
- Pascal (Named for the French mathematician Blaise Pascal). Pascal is often used in universities to teach budding computer scientists how to program.
- Assembly Language. Assembly language is the process of programming a computer at the level of individual machine instructions. This book describes the process of assembly language programming for the 68000 computer.
- Machine Language. Machine language deals with programming a computer at the instruction level, without assistance from development software. Machine language involves using a numeric language: the binary codes directly usable by the computer. This type of programming is incredibly tedious, and is only used for very specialized applications.

Flowcharts

A visual method of representing an algorithm is called *flowcharting.* A flowchart is a series of boxes which are connected by lines to show the possible paths of the algorithm. Flowcharting, like algorithm description, is not done in a computer language.

Flowcharts consist of three basic symbols:

- a square box which indicates an action to be performed
- a diamond shaped box which indicates a decision
- lines which connect the two.

There is an ANSI (American National Standards Institute) standard for flowchart symbols and flowchart layout.

A flowchart for our car-starting algorithm is shown in Figure 1.1.

A flowchart segment should fit on a single page. A flowchart for a computer program will require partitioning into several pages, interconnected by boxes. These boxes typically contain a number, indicating the mating connector on another page. Partitioning a large program flowchart into single-page segments is quite an involved process. It could require so much time that the exercise is not justified.

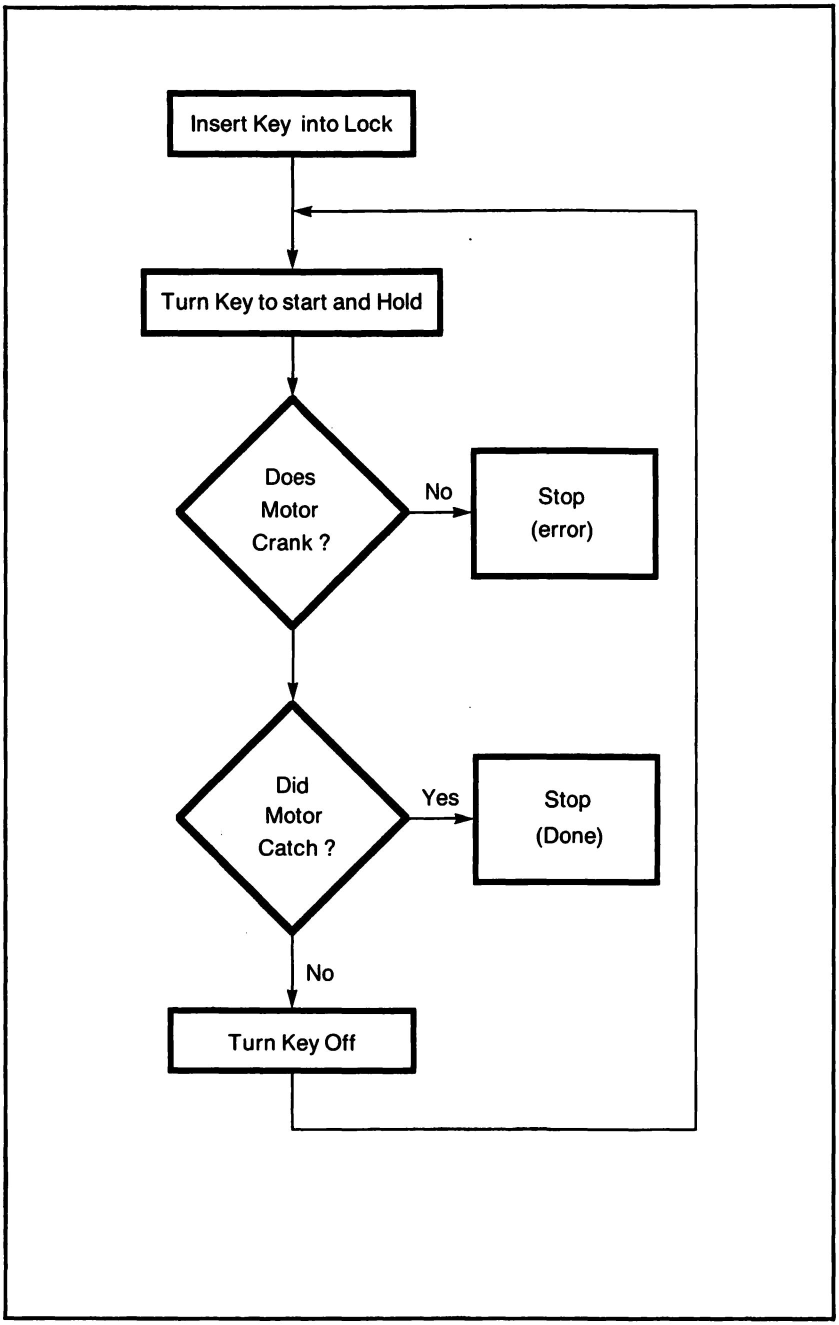


***Figure 1.1** – Flowchart for car-starting algorithm*

HOW DOES A COMPUTER WORK?

In order to learn programming, one must first understand how computer hardware functions. Figure 1.2 is a block diagram showing the major portions of a typical computer.

The *memory* contains the program which the computer follows, as well as the data on which the computer operates. In this simplified model of a computer, the box labeled *central processing unit* (CPU) is the "brains" behind the computer. The connection between the CPU and the computer's memory is known as the *memory bus. Input/output (I/O) devices* are the machines through which the computer interacts with the outside world. Examples of I/O devices are *cathode ray tube* (CRT) terminals (the computer's "TV screen"), floppy disk drives, and printers. The connection between the CPU and the I/O devices is called the *input/ouput bus.* Let us now examine each of these areas in detail.

Memory

Computer memory is a series of numbered slots, called *locations,* each of which contains a number. The number of the location is called its

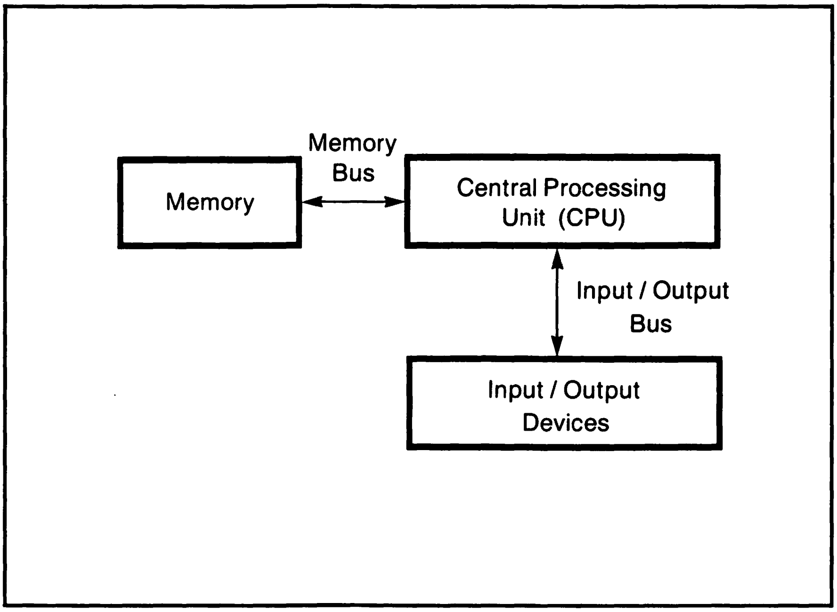


***Figure 1.2** – Computer block diagram*

address (like a street number). The number contained within the slot is called the *memory data* or *memory contents.* There are two operations associated with a computer memory:

- Change the contents of a location to a specific value. This operation is called a memory *write* or a memory *store.* The contents of the location before the write are lost.
- Obtain the present contents of a location. This operation is called a memory *read* or a memory *fetch.* A memory read does not alter the contents of the location—subsequent reads with no intervening writes will return the same value.

For example, suppose we have a four location memory with the following values:

Address	Data
0	21
1	27
2	19
3	100

Note that memory addresses are always numbered sequentially, starting with zero. If we read location 2, we will get the result 19. Writing a 6 into location 1 gives the following values:

Address	Data
0	21
1	6
2	19
3	100

Note that the memory write did not affect any other location. For example, reading location 2 again would again yield 19. Reading location 1 will yield 6. The original contents of location 1 (before the write) have been discarded, and may not be retrieved.

Input/Output Devices

Input/output devices connect the computer to the outside world. These devices typically fall into one of two categories:

1. Low-speed, character-oriented devices used to interact directly with people. Examples of this type of device are CRT terminals, printers, and plotters.
2. High-speed, block-oriented devices used for bulk storage of programs or data. Devices in this category, such as disks, tapes, etc., are usually magnetic.

Character-oriented devices typically interact with human operators. This type of device usually transfers one character at a time. The Macintosh Imagewriter printer is an example of such a device. Any other external device that connects to one of the two serial ports on the back of the Macintosh, such as the modem, or a CRT used for debugging, is also a character device.

Many computer CRT terminals send and receive characters at 960 characters per second. Although this seems amazingly fast to a human sitting at a terminal, to a computer (which can handle millions of operations per second), it is extremely slow. When interacting with humans, the computer spends a tremendous amount of time just waiting for characters from the terminal.

Block-oriented devices such as disks and tapes, on the other hand, do not normally interact with humans. These devices typically handle multi-character blocks at a time, with a short interval between characters within a block, but a comparatively long interval between blocks.

Central Processing Unit

The CPU is the heart of the computer. It executes the programs and manipulates the input/output devices.

The organization of a typical CPU is shown in Figure 1.3.

The boxes on the top of the diagram are called *registers.* A register is a single memory location within the CPU which is used to store a temporary result. Different CPUs use different numbers and types of registers. The 68000 register set will be explained in detail in Chapter 2.

The CPU registers, which can be accessed much faster than main memory, are temporary memory locations used to facilitate program execution. There are usually a limited number of these registers.

The memory address and memory data registers are used to access memory. For example, to read memory, the correct address is placed in the memory address register, and then the data can be read from the memory data register. To write to memory, the data is placed in the memory data register, and then the address is placed in the memory address register.

The program counter is a special register that is used to keep track of the next instruction to be executed. This process is detailed in the next section.

The *Arithmetic and Logic Unit* (ALU) performs all of the basic arithmetic operations, such as addition, subtraction, etc. The data on which the ALU operates can come from any of the registers with a path into the top of the ALU: the CPU registers, memory data, or the program counter. The result of the operation can be placed back into any of the registers.

Fortunately, programming the machine does not require attention to the inner operations of the ALU. The control unit supervises the movement of data through the ALU, and defines certain basic machine functions called *instructions*.

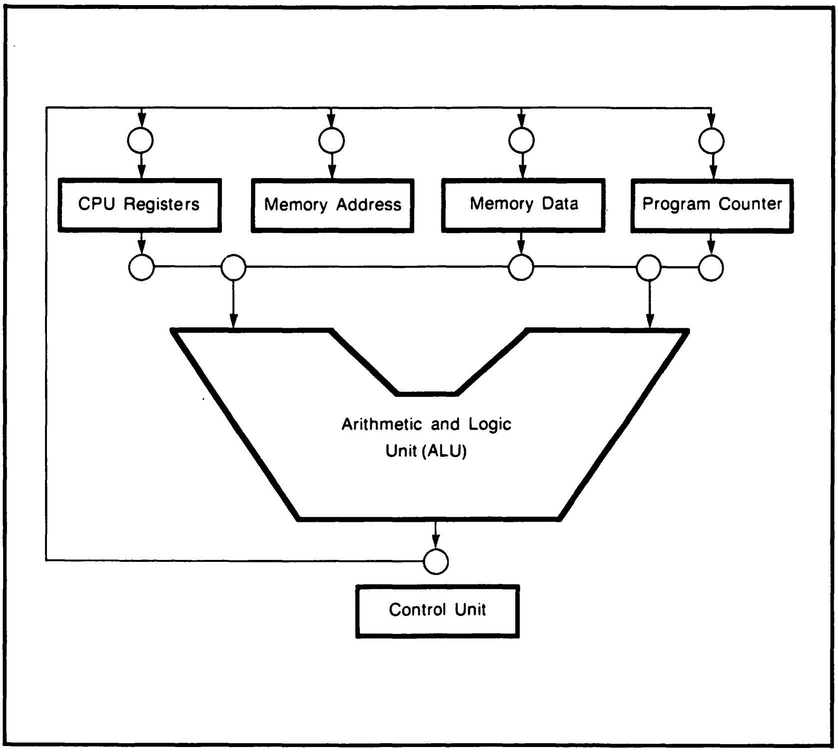


Figure 1.3 *- CPU block diagram*

Stored Program Execution

The process of executing a program works like this:

1. Fetch memory at the address indicated by the program counter, and increment the program counter to the next instruction in memory. The program counter is said to "point to" (i.e., contain the address of) the next instruction.
2. Perform the instruction.
3. Go back to step 1.

To illustrate how all this works together, we will now write a short program to add two numbers together. For simplicity, we will define a computer: Our simple computer has one register, named A. The machine can execute the following instructions:

Instruction	Meaning
1nnn	Copy memory location nnn into A
2nnn	Add memory location nnn to A
3nnn	Copy A into memory location nnn
4000	Stop

Now suppose that the computer memory contains the following:

Location	Contents	Instructions
100	1104	Load location 104 into A
101	2105	Add location 105 to A
102	3106	Store A into location 106
103	4000	Stop
104	0300	(Data)
105	0400	(Data)
106	0000	(Data)

If we then set the program counter to 100 and cause the computer to execute, the program will execute as follows:

1. The contents of memory location 104 (300) will be copied into register A.

2. The contents of location 105 (400) will be added to register A. Register A will then contain 700.

3. Register A will then be copied to memory location 106. This operation is called "Storing register A."

4. The machine will stop.

Upon completion of the above program, Register A and memory location 106 will have been altered to contain the value 700. This example is a program which adds two memory locations together and stores the result in a third memory location.

Programming at its most basic level is the process of putting the right instructions (also called *operations* or *op codes*) into the proper memory locations. The above program is an example of *machine language programming,* where the programmer deals with the actual numeric values of the instructions, and memory locations are assigned by hand. This is a tedious process at best, and programs called "assemblers" have been developed to handle the drudgery involved. Programming using an assembler is commonly called *assembly language programming.* The example program above, might look like this in assembly language:

```
    LOAD    A,X
    ADD     A,Y
    STORE   A,Z
    STOP
X:  DC      300
Y:  DC      400
Z:  DC      0
```

The words LOAD, ADD, STORE and STOP are called *mnemonics.* A mnemonic is a symbolic representation of a machine instruction. X, Y, and Z are called *labels.* A label is a way to tag a memory location without knowing what the final memory address will be. The assembler or another tool called the linker will make the final address assignment.

Finally, the abbreviation "DC" is an *assembler directive.* DC stands for "Define Constant." This directive tells the assembler to place a constant in memory at the location where the DC directive occurs.

Why is assembly language better than machine language? First, it is far more readable. Second, the task of changing an existing program is much simpler. Suppose that we wish to change our example program to add a third number, W, to X and Y. To add this to the assembly language version, we need add only two lines: an

```
ADD A,W
```

addition instruction, and a declaration for the new constant:

```
W: DC 50
```

To change the machine language version of the program, we must alter all of the instructions that reference memory, since the values to be added are now in a different place, as illustrated below:

Location	Machine Language		Assembly Language	
100	*1105		LOAD	A,X
101	*2106		ADD	A,Y
102	*2107	*	ADD	A,W
103	*3108		STORE	A,Z
104	4000		STOP	
105	0300	X:	DC	300
106	0400	Y:	DC	400
107	*0050	*W:	DC	50
108	0000	Z:	DC	0

The asterisk character (*) shows lines that have been changed. Note that no machine language location contains the same value as it did in the previous example. In large programs, altering machine code is a tremendous chore.

Data Representation

The vast majority of computers represent numbers in a form involving only two possible values: ON and OFF. This is a property of the hardware used to implement the CPU, memory, and I/O devices. This two-value representation is called *binary*, or base 2.

The Binary System

The binary system represents a number as a string of two-valued quantities. Each such quantity is called a *bit*, which stands for Binary digIT. The

ON and OFF values for a bit are 1 and 0, respectively. A bit with a value of 1 is said to be *set;* a bit with a value of 0 is said to be *clear.*

Most computers in use today organize bits in groups of eight to form a quantity known as a *byte*. The bits in a byte are numbered from right to left, starting at zero. Each bit is assigned a value twice the value of its neighbor on the right. Bit 0, the rightmost bit, has the value 1. The values associated with the eight bits in a byte are shown in Table 1.1.

Bit Number	Place Value
7	128
6	64
5	32
4	16
3	8
2	4
1	2
0	1

Table 1.1 – *Binary Bit Values*

Using this table of binary values, the binary number 0010 1111 has the value of 47. This is how the value for our example would be calculated.

The decimal value of a binary number is equal to the bit number value times the place value, as shown in Figure 1.4.

The largest number that can be represented using eight bits is 1111 1111, which is decimal 255. It is the result of adding 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1. For readability, we will write binary numbers as groups of four bits. In computerese, a four bit group, or one half of a byte, is called a *nibble*.

To represent numbers bigger than 255, two or more bytes are used. Common combinations are two bytes (16 bits), and four bytes (32 bits).

These quantities are called a *word* and a *longword,* respectively. Dealing with 16 and 32 digit numbers can be cumbersome, however, so an abbreviated form of binary representation called *hexadecimal* (or base 16) is often used.

Hexadecimal Numbers

In hexadecimal (*hex*) representation, a nibble is encoded as one hexadecimal digit.

These digits have values from 0 to 15. The values for 10 through 15 are represented by the letters A through F. Each digit in a hex number has a place value of sixteen times the value of its neighbor to the right. For instance, the number 22 hex is (2 × 16) + 2, or 34.

Converting Binary to Hex

A hex number can be derived from a binary number by first grouping the binary number into groups of four bits (nibbles), and then computing

Bit Number 7 6 5 4 3 2 1 0
Bit Value 0 0 1 0 1 1 1 1

Bit Number	Binary Value	×	Place Value	=	Decimal Value
7	0	×	128	=	0
6	0	×	64	=	0
5	1	×	32	=	32
4	0	×	16	=	0
3	1	×	8	=	8
2	1	×	4	=	4
1	1	×	2	=	2
0	1	×	1	=	1
Converted decimal value				=	47

Figure 1.4 *– Computing the decimal value of a binary number*

the proper hex digit for each group using the 8-4-2-1 values for the bits in the group. For example, the number 0010 1111 in binary is 2F in hex:

8421

$0010 = (0 \times 8) + (0 \times 4) + (1 \times 2) + (0 \times 1) = 2$

$1111 = (1 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1) = F\ (15)$

Converting Hex to Binary

Hex numbers can be converted to binary by first taking each hex digit and then expanding it into four binary bits using the hex, decimal, and binary values shown in Table 1.2.

Hex Value	Decimal Value	Binary Value
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Table 1.2 *- Hex, Decimal, and Binary Values*

When you use this table, the hex number 2F converts to the binary number 0010 1111.

Converting Hex to Decimal

To convert a hex number to a decimal number, first multiply each digit by its appropriate place value, and then add the resulting numbers. The place values for hex numbers are shown in Table 1.3.

Digit	Place Value
0	1
1	16
2	256
3	4,096
4	65,536
5	1,048,576
6	16,777,216
7	268,435,456

***Table 1.3** – Hexadecimal Place Values*

The digit numbered 0 is the rightmost digit in a hex number. Each place value is derived by multiplying the previous place value by 16, starting with a value of 1 for the rightmost digit. The digit number is also known as a "power of 16."

The hex number 54321 converts to its decimal value as shown in Figure 1.5, the hex number A25 converts to its decimal value as shown in Figure 1.6, and the hex number 1234 converts to the decimal value of 4660 as shown in Figure 1.7.

Digit	×	Place Value	=	Decimal Value
5	×	65,536	=	327,680
4	×	4,096	=	16,384
3	×	256	=	768
2	×	16	=	32
1	×	1	=	+ 1
Converted decimal value			=	344,865

Figure 1.5 – *Converting the hex number 54321 to its decimal value*

Hex Digit	×	Place Value	=	Decimal Value
A(10)	×	256		2,560
2	×	16		32
5	×	1		+ 5
Converted decimal value			=	2,597

Figure 1.6 – *Converting the hex number A25 to its decimal value*

Hex Digit	Place Value	Decimal Value
1	4096	4096
2	256	512
3	16	48
4	1	+ 4
Converted decimal value =		4660

Figure 1.7 – *Converting the hex number 1234 to its decimal value*

Converting Decimal to Hex

Decimal numbers can be converted to hex numbers using the inverse of the above technique. To convert a decimal number to hex, first find the largest place value in Table 1.3 that divides into the number to be converted, then divide the decimal value by this number. The quotient of this division is the hex digit. Divide the remainder by the next smallest place value to obtain the next hex digit (even if it's zero). Divide this remainder by the next smallest place value to get the next digit, and so on.

To convert the decimal number 123,456 to hex, we start with a hex place value of 65,536 and divide this hex value into the decimal value of 123,456. The result of 1 is the first hex digit and the remainder (57,920) is the dividend for the next hex place value.

Decimal Remainder	÷	Hex Place Value	=	Hex Digit
123,456	÷	65,536	=	1
57,920	÷	4,096	=	E (14 decimal)
576	÷	256	=	2
64	÷	16	=	4
0	÷	1	=	0

Read the answer down the Hex Digit column: 1E240. The next remainder can be calculated on a decimal calculator as: Present decimal remainder − (place value × hex digit).

For example, the remainder 576 was calculated above by first dividing 4,096 into 57,920. The answer is 14, or a hex value of E. Then the hex place value was multiplied by 14, which equals 57,344. Finally, the remainder of 576 was arrived at by subtracting 57,344 from the previous remainder, 57,920.

There are several calculators on the market which will do hex and decimal conversions. If you are going to be writing a lot of machine code, a hex calculator will pay for itself in short order.

What's in a K?

The hex number 400 translates to 1,024 in decimal. The term *K* (for "Kilo") is used in computer terminology to represent multiples of 1,024. Memory and disk device capacities are expressed in units of kilobytes. For instance, a 64 kilobyte (64K) memory contains 64 × 1,024, or 65,536 decimal (or 10000 hex) bytes of memory.

Similarly, the number 100000 hex (1,048,576 decimal) is the result of multiplying 1024 × 1024. This number is commonly abbreviated *M* (for Mega). Megabyte units are used to describe the capacities of larger memory and disk devices. A 5-megabyte (commonly written 5Mb) disk therefore contains 5 × 1,048,576, or 5,242,880 bytes.

Operations on Binary Numbers

Since decimal, binary, and hex are simply alternate ways of representing numbers, the same operations that can be performed on decimal numbers can also be performed on binary or hex numbers. One can apply techniques similar to those used for decimal numbers for the addition, subtraction, multiplication, and division of hex and binary numbers. For programming, however, it is important to understand the operations a computer is capable of performing.

A computer's ALU is capable of performing a number of very simple operations on binary numbers. These include: One's complement, AND, OR, Exclusive OR, addition, two's complement, shifts and rotates. We will now explore these in greater detail.

One's Complement

One of the simpler operations on binary numbers is to take the *one's complement:* Simply invert the values of all the bits. All 0's become 1's and vice versa.

For example, complementing the number 0011 1100 (3C hex) produces 1100 0011 (C3 hex). Complementing 0000 0000 (00 hex) yields 1111 1111 (FF hex). Complementing the complement of a number yields the original number again.

Binary AND

Performing an *AND* operation on two binary numbers produces a third binary number with 1's in each bit position where the original numbers both had a 1. ANDing 0000 1101 (0D hex) with 1001 1001 (99 hex) yields 0000 1001 (09 hex).

The AND operation is commonly used to obtain a remainder for a division by a power of two (2, 4, 8, 16, etc.). To obtain such a remainder, AND the number with the power of 2 minus 1. For example, to find the remainder when 0011 1101 (3D hex) is divided by 8, AND with 0000 0111 (07 hex). The result is 0000 0101 (05 hex). 3D hex is 3 × 16 + 13, or 61 decimal. Dividing by 8 yields 7, with a remainder of 5.

Binary OR

The *OR* operation takes two binary numbers and produces a third binary number that has a 1 where either of the original numbers had a 1. For example, ORing 1010 1010 (AA hex) with 0101 0101 (55 hex) yields 1111 1111 (FF hex). (ANDing these two numbers gives all zeros.)

Binary XOR

The *XOR* (eXclusive OR) operation takes two binary numbers and produces a third binary number which has 1's in bit positions where one (not both) of the original numbers had a 1. For example, XORing 0101 0101 (55 hex) with 1111 1111 (FF hex) yields 1010 1010 (AA hex). XORing a number with all 1's yields the 1's complement of the number. XORing a number with itself produces zero.

Binary Addition

Adding two binary numbers is similar to adding decimal numbers. You add each pair of digits, starting on the right, and carry any result over 1 to the next column. For example, adding 0011 1101 (3D hex) and 0001 0101 (15 hex) is done as follows:

Carry	0011	1010	
	0011	1101	(3D hex)
	+0001	0101	(15 hex)
Sum	0101	0010	(52 hex)

This procedure can be used for binary numbers of any length.

2's Complement Arithmetic

Subtraction involves a bit of magic. Negative numbers are stored in a form known as *two's complement.* The two's complement of a number is obtained by taking the one's complement (as explained above), and adding 1 to it. For instance, the two's complement of 0000 0001 is:

Original Number:	0000 0001	(01 hex)
One's Complement:	1111 1110	(FE hex)
Add One:	+0000 0001	(01 hex)
Two's Complement:	1111 1111	(FF hex)

Adding the two's complement of a number is the same as subtracting the number. As an example, consider adding 1111 1111 (FF hex) to 0000 0010 (02 hex). The leftmost bit of a two's complement number will be a 0 if the number is positive (zero or greater), and a 1 if the number is negative (less than 0). For this reason, the leftmost bit is often called the *sign bit*.

Carry	11111	1100	
	0000	0010	(02 hex)
	+1111	1111	(FF hex)
Sum	0000	0001	(01 hex)

Note that the carry out of the high order bit position is discarded. This is due to the fact that all of the numbers kept in a computer have exactly the same number of bits (eight in this example). Note that the result of adding 02 hex and FF hex is 01. This is the same as subtracting 1 from 2. FF hex is the two's complement of 01 hex, as shown above.

Two's complement changes the range of numbers it is possible to represent using a given number of bits. For instance, without using the two's complement, we could represent from 0 to 255 with eight bits. Using two's complement, however, we can represent from −128 to +127.

The first case is called *unsigned* arithmetic, meaning that only positive numbers can be represented. The second case is called two's complement representation, meaning that both positive and negative numbers can be represented.

Shifts and Rotates

Two other operations commonly performed on binary numbers are shifts and rotates. These operations are similar to the old "bucket brigade" operation used in fighting fires. Bits are moved from one position to the next position. Shifts and rotates can occur in either direction.

There are two types of shifts: logical and arithmetic. In a logical shift operation, the bits are moved left or right as in Figure 1.8.

Zero bits are shifted into the bit vacated by the shift operation. The bit marked C is a special status bit in one of the CPU internal registers. This bit is called the *Carry* bit, and it receives the bit which would otherwise be lost.

An arithmetic left shift is the same as a logical left shift. An arithmetic right shift is similar to a logical right shift, except that the most significant bit is copied into itself. Both of these are shown in Figure 1.9.

The two left shifts differ only in the settings of the condition codes. These are explained in Chapter 2.

Rotates are similar to logical shifts, except that the Carry bit is shifted into the vacated bit, instead of a zero, as illustrated in Figure 1.10.

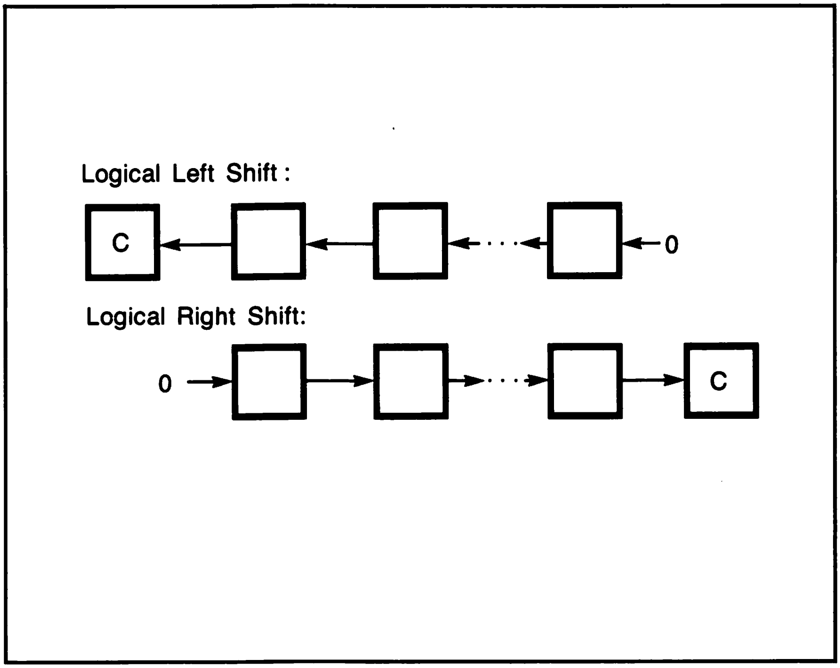


Figure 1.8 *– Logical left shift and logical right shift*

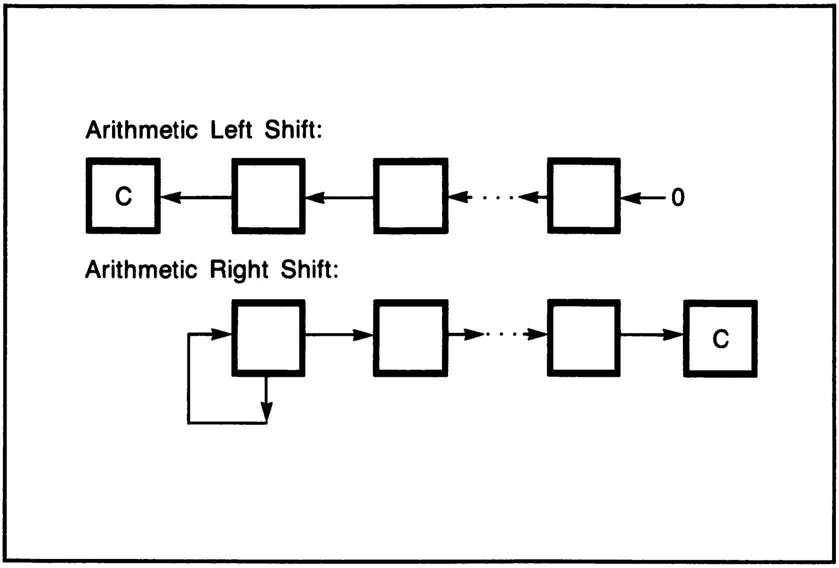


Figure 1.9 – *Arithmetic left shift and arithmetic right shift*

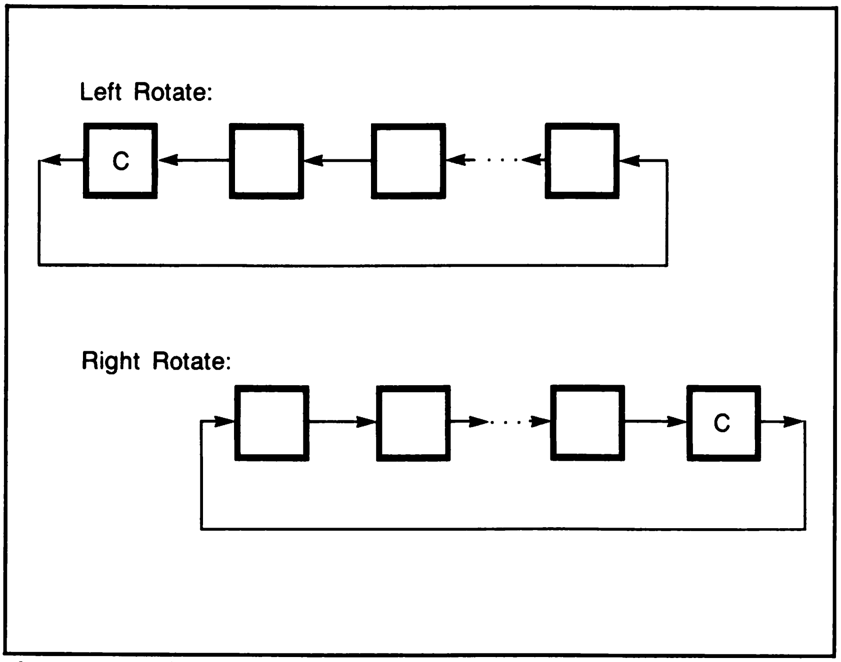


Figure 1.10 – *Left rotate and right rotate*

Examples

The following examples assume that the Carry bit is initially zero. The original number is 1010 1010 (AA hex).

Logical Shifts:

Times	Left			Right		
0	1010	1010	(AA hex)	1010	1010	(AA hex)
1	0101	0100	(54 hex)	0101	0101	(55 hex)
2	1010	1000	(A8 hex)	0010	1010	(2A hex)
3	0101	0000	(50 hex)	0001	0101	(15 hex)
4	1010	0000	(A0 hex)	0000	1010	(0A hex)
5	0100	0000	(40 hex)	0000	0101	(05 hex)
6	1000	0000	(80 hex)	0000	0010	(02 hex)
7	0000	0000	(00 hex)	0000	0001	(01 hex)
8	0000	0000	(00 hex)	0000	0000	(00 hex)

Arithmetic shifts:

Times	Left			Right		
0	1010	1010	(AA hex)	1010	1010	(AA hex)
1	0101	0100	(54 hex)	1101	0101	(D5 hex)
2	1010	1000	(A8 hex)	1110	1010	(EA hex)
3	0101	0000	(50 hex)	1111	0101	(F5 hex)
4	1010	0000	(A0 hex)	1111	1010	(FA hex)
5	0100	0000	(40 hex)	1111	1101	(FD hex)
6	1000	0000	(80 hex)	1111	1110	(FE hex)
7	0000	0000	(00 hex)	1111	1111	(FF hex)
8	0000	0000	(00 hex)	1111	1111	(FF hex)

Rotates:

Times	C	Left			Right			C
0	0	1010	1010	(AA hex)	1010	1010	(AA hex)	0
1	1	0101	0100	(54 hex)	0101	0101	(55 hex)	0
2	0	1010	1001	(A9 hex)	0010	1010	(2A hex)	1
3	1	0101	0010	(52 hex)	1001	0101	(95 hex)	0

Times	C	Left			Right			C
4	0	1010	0101	(A5 hex)	0100	1010	(4A hex)	1
5	1	0100	1010	(4A hex)	1010	0101	(A5 hex)	0
6	0	1001	0101	(95 hex)	0101	0010	(52 hex)	1
7	1	0010	1010	(2A hex)	1010	1001	(A9 hex)	0
8	0	0101	0101	(55 hex)	0101	0100	(54 hex)	1
9	0	1010	1010	(AA hex)	1010	1010	(AA hex)	0

Shifts are very useful for multiplying and dividing. A logical shift left is the same as multiplying a number by 2, and a logical shift right is the same as dividing a number by 2. This is only true for unsigned numbers.

Arithmetic shifts, on the other hand, represent multiplication and division by 2 for two's complement numbers. The one exception is that dividing (shifting right) −1 yields −1 and not zero.

Extensions

When copying an 8-bit quantity into a 16-bit quantity, or when copying a 16-bit quantity into a 32-bit quantity, there is a possibility of losing the two's complement properties of the number.

To illustrate the problem, suppose we copy FF hex (−1 as an 8-bit number) to a 16-bit number. Copying only the lower 8 bits gives us 00FF hex, which is not −1, but 255! The way to fix this situation is to copy the sign bit (most significant bit) into all the "extra" bits in the larger number. This is called *sign extension*. If we sign extend FF hex into 16 bits, we get FFFF hex, which is −1 in two's complement form.

SUMMARY

In this chapter we have learned basic concepts that are applicable to most computers on the market today. In the next chapters, we shall see how these concepts are applied to the Motorola 68000, the CPU chip used in the Macintosh.

EXERCISES

Use the following questions to help solidify your understanding of the material presented in Chapter 1. Answers to all exercise questions can be found in Appendix A.

1. Write an algorithm for converting a decimal number to hex.

2. Develop a flowchart for the algorithm in question 1.

3. Suppose the computer on page 9 has an additional instruction 5nnn, which subtracts the contents of memory location nnn from Register A. The assembly form of this instruction is *SUB A,y* where y is a label on the memory location to be subtracted from A. Modify the example of the machine-language program to compute the difference between the contents of location 105 and the contents of location 104. Store the result in location 106. (Hint: You will have to load location 105 into Register A first. Why?)

4. Move the machine-language program you wrote in question 3 to run at address 200.

5. Give the assembly language equivalent of the program for question 3.

6. Write a new assembly language program that computes the sum of the first five integers. Use the labels A through E for the memory locations that contain the numbers to be added. Store the result in a separate memory location, labeled F.

7. Convert the following decimal numbers to their hex and binary equivalents.

   273
   421
   1024
   100

8. Convert the following hex numbers to their binary and decimal equivalents.

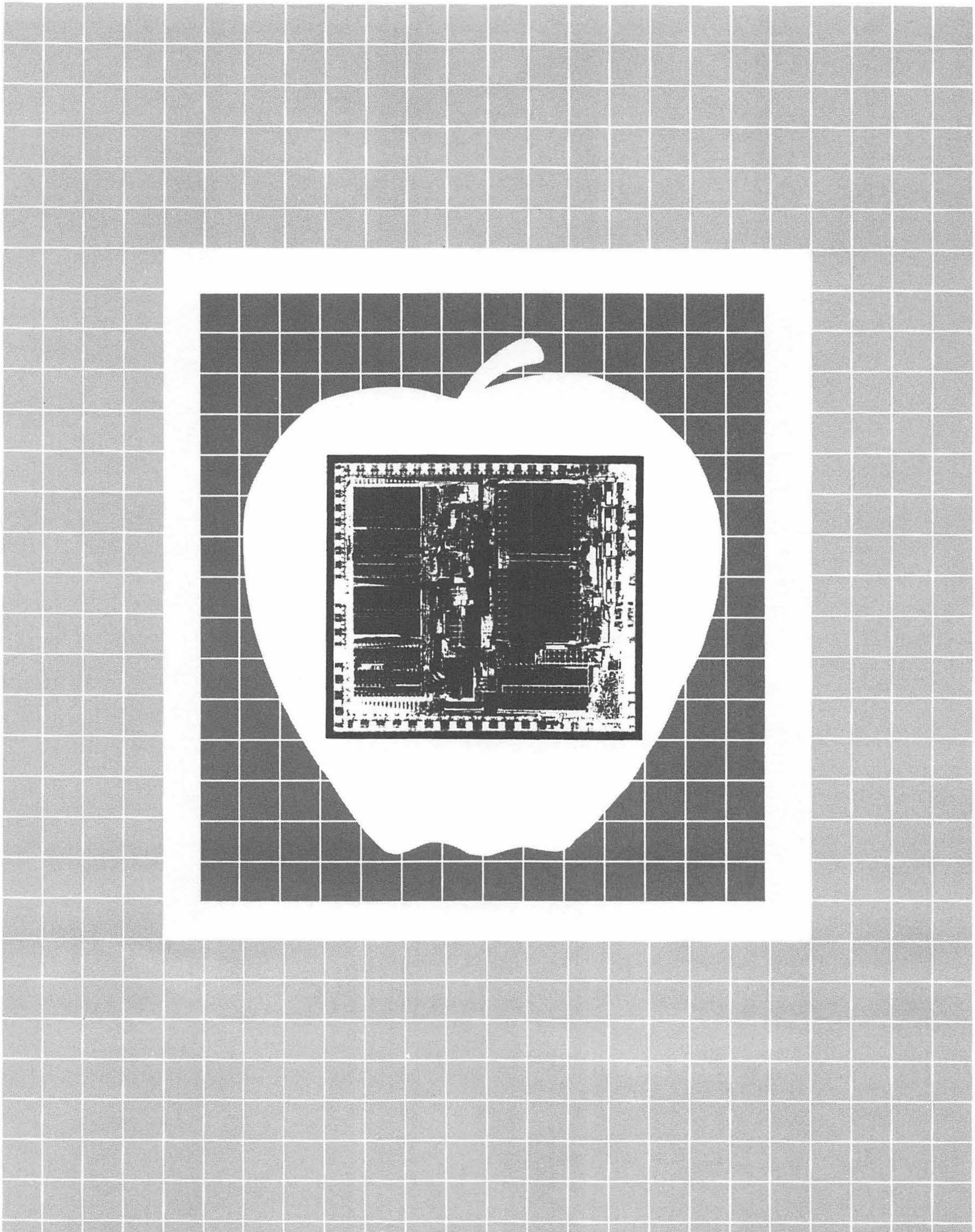
   ABE
   100
   64
   1024
   505

9. Give the one's and two's complements for each of the numbers in the previous question.

10. Perform the AND, OR, XOR, and addition operations on the following pairs of hex numbers. Use 16-bit operations. Give the carry out of the high order bit pairs for the addition operation.

    A5A5 5A5A
    FFFF 0001
    1234 4321

11. Prepare shift and rotate tables similar to those in the text for the hex quantities FF and 55. Assume that the carry bit is initially zero.



Chapter 2

68000 Architecture

An investment in knowledge always pays the best interest.

—Benjamin Franklin

INTRODUCTION

In this chapter, we will expand the general programming concepts presented in Chapter 1 to cover the architecture of the Motorola 68000, the CPU chip used by the Macintosh. The information specific to the 68000 is necessary in order to understand the instruction set contained in Chapter 3.

Most computers can be categorized by the following criteria:

- The number and type of registers that may be used by the programmer in writing software.
- How data is organized in memory, and what data types are supported with hardware instructions.
- How memory is addressed by an instruction.
- Special hardware features, such as hardware support for stacks.

We will now explore each of these areas as it relates to the 68000.

REGISTER SET

One of the first questions you should ask when learning a new computer is "How many registers does it have?" Another important consideration is whether the registers can be used interchangeably or not. If the computer has many registers that *can* be used interchangeably, you will have a great deal of flexibility in handling intermediate values in a computation.

The number of registers in a computer also has an effect on program speed and size. Computations using a register are usually smaller and faster than computations involving a memory location. (This is due primarily to the nature of computer hardware). A machine with a large number of general-purpose registers is preferred over a machine with a small number of registers or a machine whose registers are restricted in function.

The 68000 architecture trades off some generality in order to gain a larger register set. There are two types of registers: *address registers* and *data registers.* Address registers are normally used to contain memory addresses, while data registers normally contain data. The two register classes are not used interchangeably.

Address Registers

There are eight address registers, numbered A0–A7. Each address register is a 32-bit quantity. Address registers can also be used as 16-bit quantities. When a 16-bit quantity is loaded into an address register, it is sign extended to become a 32-bit quantity, as shown in Figure 2.1. The notation A0.W is used to mean the word part of address register A0. (A0.L means the entire 32 bits stored in register A0.) The .B suffix is used to denote an 8-bit quantity. Address registers may not be used as 8-bit quantities, however.

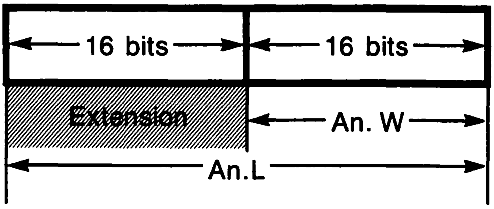


Figure 2.1 *– Address register layout*

Register A7 is a special-purpose register. A7 is the *hardware stack pointer* used by 68000 exceptions and subroutine call instructions. This register points to memory used by programs for temporary storage of data. The concept of a stack is discussed later in this chapter.

Address registers are used as temporary locations for storing memory addresses. These registers can be used in instructions that reference memory in order to specify the address at which data is located. The upper byte of the register is presently ignored in such usage (by the 68000 and 68010 chips). This limits the amount of memory that you can use to 16 megabytes (16,772,216 bytes). Future processors in the 68000 family will not ignore this byte, so it should always be set to zero for compatibility.

Data Registers

The 68000 also has eight other registers, called *data registers,* numbered D0–D7. A data register can be used as an 8-bit, 16-bit, or 32-bit quantity, as shown in Figure 2.2. Unlike address registers, loading a data register with less than 32 bits does not cause a sign extension to occur into the remaining bits in the register. These remaining bits are left unchanged.

Data registers cannot be used to address memory in an instruction. These registers are used instead as temporary locations where data may be stored. Many instructions require one or more of the operands to reside in a data register.

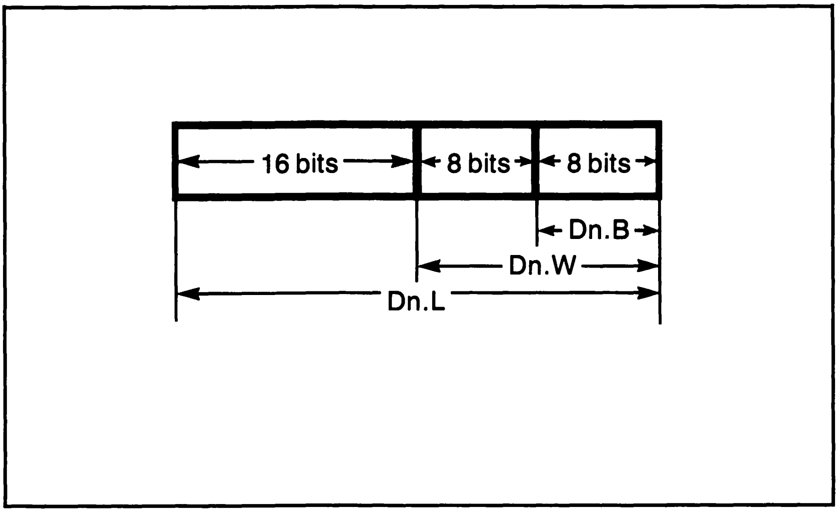


Figure 2.2 *– Data register layout*

Program Counter

A special 32-bit register called the *program counter* is used to control execution of the program in memory. The program counter always contains the memory address of the next instruction to be executed. As an instruction is executed, the program counter is advanced to point to the next instruction.

Certain instructions can be used to affect the contents of the program counter. These are:

- Instructions that alter the contents of the program counter unconditionally. These are called *unconditional branches* or *jumps.* Such an instruction is useful for programming loops, or for merging several alternative sections of the program into one common section.
- Instructions that alter the contents of the program counter based on the result of a previous instruction. These are called *conditional branches,* and enable the computer to make decisions. Using a conditional branch, either a portion of the program can be skipped or a previous portion repeated, based on the result of a previous computation.
- Instructions that cause a given section of code to be repeated a specific number of times, or until a condition is satisfied. Such instructions are called *looping primitives.*
- Instructions that are used to branch to another area of the program and then to return to the location following the original branch. This technique is known as a *subroutine call.* A subroutine call can be used to invoke a common function, such as an I/O routine, at many points in the program, using only a single copy of the instructions that perform the I/O.

As with address registers, the upper eight bits of the program counter are ignored by the 68000 and 68010 processors.

Status Register

The 68000 uses a special register, called the *status register* (SR) to store information about the status of the machine. This register is used by the conditional branch instructions to retrieve information about the last instruction.

The status register is a 16-bit quantity, organized as shown in Figure 2.3.

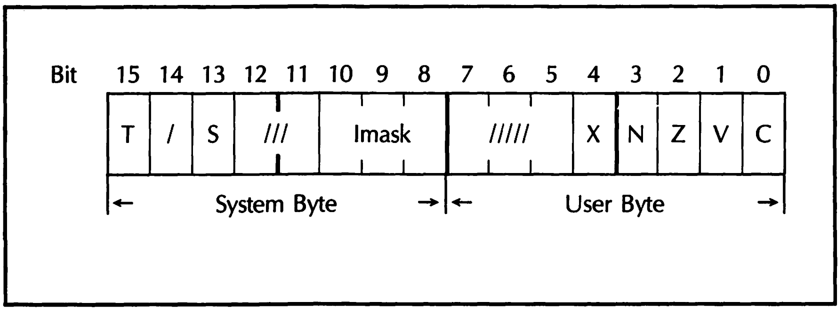


***Figure 2.3** - Status register layout*

System Byte

Bits 15–8 are called the *system byte,* because the information contained here is not normally available to applications programs. The fields in the system byte are:

1. Bit 15 is a hardware aid for debuggers. It is called the *trace bit.* If bit 15 is set, an exception will take place at the end of each instruction. Exceptions are described in Chapter 7, *Exception Processing.* This feature is used by debuggers to regain control as each instruction is executed.

2. Bit 13 is used to regulate access to certain instructions and to the system byte of the status register. It is called the *supervisor bit.* If this bit is set, access is allowed. When the supervisor bit is set, the 68000 is said to be executing in *supervisor mode.* When the bit is reset, the 68000 is said to be executing in *user mode.* User mode software is prevented by the hardware from executing certain privileged instructions that might compromise the integrity of the system software. Access to the status register's system byte is also prohibited when in user mode, ensuring that the user mode program cannot change the supervisor bit. All software on the Macintosh executes in supervisor mode, however, so all instructions can be used. Free access to the status register is permitted.

3. Bits 10–8 are called the *interrupt mask.* This feature is more fully explained in Chapter 7, which deals with 68000 exception conditions.

The system byte of the status register is of concern only to systems software programmers. We will deal more extensively with this topic in later chapters.

User Byte

The lower byte of the status register is called the *user byte*. The user byte contains a set of bits known as *condition codes*, which are bit flags used to record the outcome of the last arithmetic operation performed. The user byte can be accessed at any time regardless of machine state. The bits defined in the user byte are:

- The *C* (carry) bit carries out the high-order bit position of an arithmetic operation. For example, when two 8-bit numbers are added, the C bit is the ninth bit of the result. This bit also receives bits that are shifted out of a number during shift or rotate operations.
- The *V* (oVerflow) bit is set whenever an operation yields a result that cannot be properly represented. For example, when adding the bytes 7F hex and 01 hex, the result, 80 hex, is not properly represented in eight bits. (Remember, 80 hex is −128 decimal in two's complement notation.) The V bit would be set following such an operation.
- The *Z* (Zero) bit is set if the result of an operation is zero.
- The *N* (Negative) bit is set if the high order bit of a result is set. (In two's complement, the high order bit of a number is set if the number is less than zero.)
- The *X* (eXtended) bit is a copy of the carry bit, but it is not affected by every instruction that affects the carry bit. The purpose of this bit is to facilitate multiprecision instructions. The X bit is affected only by instructions that can be used for multiprecision operations. This allows you to intermix other instructions between multiprecision operations without having to preserve the carry bit.

The descriptions of the instruction set in Chapter 3 describe how the condition codes are used by each operation. Since the lower half of the status register contains nothing but the condition codes, it is sometimes called the *condition code register* (CCR).

DATA ORGANIZATION IN MEMORY

The 68000 instruction set supports several data formats: binary, BCD, and floating point.

Bytes, Words, and Longwords

Binary data items can be 8, 16, or 32 bits long. These data types are known as *bytes, words,* and *longwords,* respectively. Most instructions that operate on binary data support any of these three data lengths. For example, the MOVE instruction, which transfers a binary data item from one place to another, has three forms:

- MOVE.B moves a byte of data
- MOVE.W moves a word of data
- MOVE.L moves a longword of data

Note the use of the suffixes .B, .W, and .L to denote data length. If no suffix is given, .W is usually assumed.

When a word or a longword is stored in memory, the bytes are stored in order of decreasing magnitude. The most significant bits are stored at the lowest address, and the least significant bits are stored at the highest address. For example, when a 16-bit word is stored at location 1000, the most significant byte is at location 1000, and the least significant byte at 1001. When the longword 01234567 (hex) is stored at address 1000, memory appears as shown in Figure 2.4.

Address	Contents
1000	0000 0001 (01 hex)
1001	0010 0011 (23 hex)
1002	0100 0101 (45 hex)
1003	0110 0111 (67 hex)

Figure 2.4 *– Memory with longword stored at address 1000*

We emphasize this point because all computers do not store multibyte binary quantities in the same way. In particular, the 8080/Z-80, 6502, and 8086/8088 family of microprocessors store words and longwords in ascending order, so that the low byte is stored at the lowest address and the high byte at the highest address. Which order is "correct" is largely a matter of taste, but you should be aware that the difference exists.

BCD

A special form of binary numbers, called *binary coded decimal* (BCD), is often used for digital displays and input devices. The familiar displays on digital clocks and electronic calculators are universally based on BCD. Most computer-based laboratory and factory instruments also use BCD.

BCD is basically hexadecimal without the codes for 10–15 (i.e., A–F). Each nibble represents a digit in a decimal number. Thus, two decimal digits are stored in a byte. The 68000 has instructions for the addition and subtraction of BCD numbers.

BCD is especially useful in commercial applications, such as accounting. Many such applications require only addition and subtraction of numbers in character format. To convert these numbers to binary for calculation and then convert them back to character format for output requires much multiplication and division, which are comparatively slow operations for most computers. Using BCD avoids these expensive operations, and provides quick conversion to and from character format. In many applications, BCD is much more efficient than binary.

Floating Point

Scientific applications require a large range of possible values. For example, to represent Avogadro's number (a common quantity in chemical calculations, roughly 6 followed by 23 zeros) would require ten bytes of storage. To hold the result of the multiplication of two such numbers would require twice this much storage. Then there is the problem of representing fractional numbers, which none of the data representations thus far have addressed.

The clean solution to these problems is found in *floating point numbers.* Floating point numbers are a computerized form of scientific notation, which is taught in grade school mathematics. In scientific notation, a number is written as a quantity between 0 and 10 times 10 to the appropriate power. For example, the quantity 1,935,000,000 is written as 1.935×10^{12}. The quantity 0.000001349 is written as 1.349×10^{-6}.

Scientific notation is extremely useful in a computer. Allocating a fixed number of bits to the exponent and fraction parts of a number yields a very useful approximation to both very large and very small numbers. The 68000, like most microprocessors, does not directly support floating point with instructions. However, there is an additional feature which can be added to provide hardware support for floating point.

The 68000 add-on is called the 68881 Floating Point Processor. It provides hardware instructions to manipulate floating-point numbers. The 68881 uses a floating-point format known as The Institute of Electrical and Electronic Engineers (IEEE) format, named for the organization that proposed the format as a standard. IEEE format provides the following floating point format:

S	Exp	Fraction

The field labeled S is the sign bit for the entire number. If this bit is set, the number is negative. If the bit is not set, the number is positive.

The field labeled Exp is the exponent; it is seven bits long (for normal floating point) or 15 bits long (for extended floating point). The exponent field indicates the exponent of 2 by which the fractional part of the number is multiplied. To allow for negative exponents, a quantity called the *exponent bias* is subtracted from the exponent before it is used. This number is decimal 64 for 7-bit exponents and 16384 for 15-bit exponents. The 7-bit exponent range is 00 hex (-64) to 7F hex ($+63$). Thus, the range of representation is 2^{63} (approximately 9.2234×10^{18}) to 2^{-64} (approximately 5.421×10^{-20}).

The fraction part of a standard floating-point number is 24 bits, or six to seven decimal digits. This limits the number of significant digits the floating-point number can contain. A limit of six digits means that the computer cannot correctly subtract 1 from 10,000,000, for example.

The Macintosh supports a software implementation of floating point known as "SANE" (Standard Apple Numeric Environment). This format provides three forms of floating-point numbers:

1. *Single precision,* which represents numbers in 32 bits, with a 7-bit exponent and a 24-bit fraction.
2. *Double precision,* which represents numbers in 64 bits, with a 7-bit exponent and a 56-bit fraction.
3. *Extended precision,* which represents numbers in 80 bits, with a 15-bit exponent and a 64-bit fraction.

These formats are compatible with the IEEE standard used by the 68881 hardware chip. The remainder of this book will deal with integer arithmetic.

ASCII

The final form of data storage is known as *ASCII* or character format. ASCII stands for American Standard Code for Information Interchange. This code assigns a numeric value for each character. These values are used to represent characters in memory and during I/O. The current ASCII standard for the United States defines 128 characters, with values from 0–127. The characters are stored one per byte in memory. The eighth bit is used for additional characters in Europe and Japan. For a complete list of United States ASCII values, see Appendix B.

Multicharacter sequences, called *strings,* are stored in multiple consecutive bytes in memory. The 68000 provides no instructions explicitly for string manipulation; sequences of byte instructions must be used instead. There are three common types of string storage you may employ:

- Use a fixed length for each string to be stored. This has the advantage of being easy to program, but wastes memory if string length tends to vary. This technique is usually used by the FORTRAN and COBOL languages.
- Prefix each string with a character count. This is more difficult to program, but wastes less memory. If the character count is stored in a byte at the beginning of the string, then strings are limited to 255 characters. Using a word (i.e., two bytes) allows 65,535 characters in a string. This is the technique usually used by the Pascal and BASIC languages.
- Terminate the string with some flag value, usually zero. This technique is used by the C language. Problems arise, however, if strings are processed in a manner other than sequentially from beginning to end.

ADDRESSING MODES

A computer instruction must specify two things:

1. What operation to perform, such as addition or subtraction.
2. On what data to perform the operation. Data for instructions is usually found either in registers or memory.

A portion of the instruction, called the *op code,* indicates the operation to perform. The simple example in Chapter 1 used the first digit of the instruction as the op code. Data was in the machine's single register and a memory location. The memory location was identified by the address contained in the last three digits of the instruction.

Real computers are seldom so simple. In the 68000, instructions specify operands by one of three techniques:

1. Some instructions imply the use of certain operands, usually a register, such as the status register or the Program Counter (PC).
2. Some instructions work only on registers. The register number is contained in the instruction itself.
3. Most 68000 instructions specify operands with a technique called an *effective address.* This is a generalized technique for addressing the registers and memory.

Effective Address

An effective address is specified by six bits in the instruction (usually the lowest six bits). The bit values indicate how to find the data for the instruction. Figure 2.5 shows how these bits are arranged into two groups of three bits.

The mode bits determine the meaning of the entire field. Three bits give eight possible combinations. Values 0–6 mean that a register is to be used, either as the operand, or to determine the address of the operand in memory. If the mode field is 7 (i.e., all 1's), then the entire six bits of the effective address field is used to specify the mode.

In the following examples, we will illustrate the addressing modes using the *MOVE.L* instruction, which moves a longword from one operand to another. Both operands have the effective address format. We will use the D0.L data register as the destination operand, and vary the source operand to illustrate the various addressing modes. Figure 2.6 shows the format of a *MOVE.L* instruction.

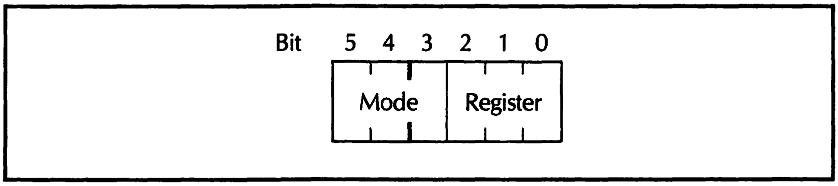


Figure 2.5 *– Format of an effective address*

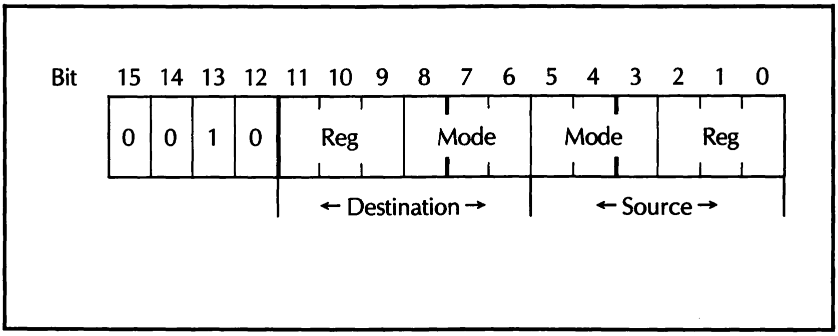


Figure 2.6 *- Format of a MOVE.L instruction*

The *Source* and *Destination* fields are used to select the source and destination operands. Since the register and mode fields are three bits wide, it is difficult to look at the hex representation of an instruction and determine the assembly language equivalent. This process is known as *disassembly.* 68000 instructions in general are difficult to disassemble by hand. Fortunately, most debuggers perform disassembly, so this problem is not as severe as it could be.

Data Register Direct Addressing

Addressing Mode Field:	000
Register Field:	000–111 (Data Register Number)
Assembler Syntax:	Dn (n is 0–7)

Description

Data register direct addressing is indicated by an effective address mode field of 000 (binary). The register field contains a number from 000 to 111 (0–7), which indicates a data register. In data register direct addressing, the data register (indicated by the register field) contains the operand.

Example

The instruction *MOVE.L D1,D0* causes the contents of data register D1 to be copied into data register D0. After the instruction executes, the two registers contain the same information. Figure 2.7 shows the format of this instruction.

When only a word or byte is transferred, the contents of the upper bytes of the data register are unchanged. Figure 2.8 shows examples of the *MOVE* instruction with a byte, a word, and a longword.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	(2001 Hex)
					← Reg →			← Mode →			← Mode →			← Reg →			
					← Destination →						← Source →						

Figure 2.7 – *Format of MOVE.L D1,D0 instruction*

Instruction	Before	After
MOVE.B D1,D0	D0=FFFFFFFF D1=01234567	D0=FFFFFF67 D1=01234567
MOVE.W D1,D0	D0=FFFFFFFF D1=01234567	D0=FFFF4567 D1=01234567
MOVE.L D1,D0	D0=FFFFFFFF D1=01234567	D0=01234567 D1=01234567

Figure 2.8 – *Moving a byte, word, or longword*

Address Register Direct Addressing

Addressing Mode Field: 001
Register Field: 000–111 (Address Register Number)
Assembler Syntax: An (n is 0–7)

Description

Address register direct addressing is indicated by an effective address mode field of 001 (binary). The register field contains a number from 000 to 111 (0–7), which indicates an address register. In address register direct addressing, the address register indicated by the register field contains the operand.

Example

The instruction *MOVE.L A1,D0* causes the contents of address register A1 to be copied into data register D0. After the instruction executes, the two registers contain the same information. Figure 2.9 shows the format of this instruction.

Transfers involving an address register are restricted to word or long size. Byte operations are not allowed. When transferring a word to an address register, bit 15 (the sign bit of a word) is extended throughout the upper word of the address register. Figure 2.10 gives several examples of address register direct addressing.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	(2009 Hex)
					← Reg →			← Mode →			← Mode →			← Reg →			
					← Destination →						← Source →						

***Figure 2.9** – Format of MOVE.L A1,D0 instruction*

Instruction	Before	After
MOVE.W A1,D0	D0=FFFFFFFF A1=01234567	D0=FFFF4567 A1=01234567
MOVE.W D0,A1	D0=01234567 A1=FFFFFFFF	D0=01234567 A1=00004567
MOVE.W D0,A1	D0=0000FFFF A1=00000000	D0=0000FFFF A1=FFFFFFFF
MOVE.L A1,D0	D0=FFFFFFFF A1=01234567	D0=01234567 A1=01234567

***Figure 2.10** – Examples of address register direct addressing*

Address Register Indirect Addressing

Addressing Mode Field: 010
Register Field: 000–111 (Address Register Number)
Assembler Syntax: (An) (n is 0–7)

Description

Address register indirect addressing is indicated by an effective address mode field of 010 (binary). The register field contains a number from 000 to 111 (0–7), which indicates an address register.

In address register indirect addressing, the address register indicated by the register field is the address of a memory location that contains the operand. The register is said to point to (contain the address of) the operand. Address register indirection is denoted by enclosing the address register name in parentheses. For example, (A0) denotes indirection on address register A0. Word or longword references require that the address contained in the address register must be even.

Example

The instruction *MOVE.L (A1),D0* causes the contents of the memory location pointed to by address register A1 to be copied into data register D0. After the instruction executes, data register D0 and the memory location contain the same information. Figure 2.11 shows the format of this instruction.

The instruction works as shown in Figure 2.12. $1000 indicates the contents of the longword in memory at address 1000 hex.

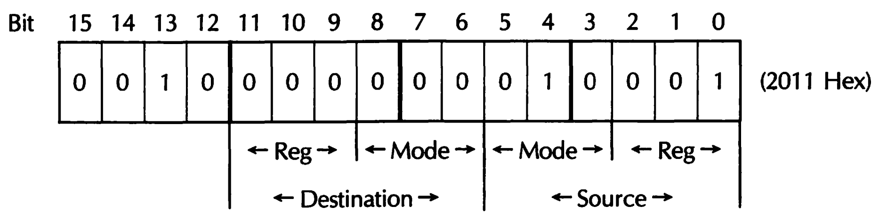


***Figure 2.11** – Format of MOVE.L (A1),D0 instruction*

Instruction	Before	After
MOVE.L (A1),D0	D0=FFFFFFFF A1=00001000 $1000=01234567	D0=01234567 A1=00001000 $1000=01234567

***Figure 2.12** - An example of the MOVE.L (A1),D0 instruction*

Address Register Indirect Addressing with Post-increment

Addressing Mode Field: 011
Register Field: 000–111 (Address Register Number)
Assembler Syntax: (An)+ (n is 0–7)

Description

Address register indirect addressing is indicated by an effective address mode field of 011 (binary). The register field contains a number from 000 to 111 (0–7), which indicates an address register. In address register indirect addressing, the address register indicated by the register field contains the address of a memory location that contains the operand. The register is said to point to (contain the address of) the operand.

The address register is incremented after the data has been obtained from memory. The increment is based on the length of the data item referenced by the instruction. Thus, for a *MOVE.B* instruction, the address register would be incremented by one. For a *MOVE.W* instruction, the address register is incremented by two. For a *MOVE.L* instruction, the address register is incremented by four.

Address register indirection with post-increment is denoted by enclosing the address register name in parentheses followed by a plus (+) symbol. For example, (A0)+ denotes post-increment indirection on address register A0. Word or longword references require that the address contained in the address register must be even.

Example

The instruction *MOVE.L (A1)+,D0* causes the memory location pointed to by address register A1 to be copied into the contents of data register

D0. After the instruction is executed, data register D0 and the memory location contain the same information. Address register A1 is incremented by 4. Figure 2.13 shows the format of this instruction.

The instruction works as shown in Figure 2.14. $1000 indicates the contents of the longword in memory at address 1000 hex.

A special case occurs when the address register specified is A7, which is the hardware stack pointer. Byte operations on address register A7 cause an increment by two rather than one. This ensures that the stack pointer always contains an even address.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1	(2019 Hex)
					← Reg →			← Mode →			← Mode →			← Reg →			
					← Destination →						← Source →						

***Figure 2.13** – Format of the MOVE.L (A1)+,D0 instruction*

Instruction	Before	After
MOVE.L (A1)+,D0	D0=FFFFFFFF A1=00001000 $1000=01234567	D0=01234567 A1=00001004 $1000=01234567

***Figure 2.14** – An example of the MOVE.L (A1)+,D0 instruction*

Address Register Indirect Addressing with Pre-decrement

Addressing Mode Field:	100
Register Field:	000–111 (Address Register Number)
Assembler Syntax:	–(An) (n is 0–7)

Description

Address register indirect addressing with pre-decrement is indicated by an effective address mode field of 100 (binary). The register field contains a number from 000 to 111 (0–7), which indicates an address register. In address register indirect addressing, the address register indicated by the register field contains the address of a memory location that contains the operand. The register is said to point to (contain the memory address of) the operand.

The address register is decremented before the data is obtained from memory. The decrement is based on the length of the data item referenced by the instruction. Thus, for a *MOVE.B* instruction, the address register is decremented by one. For a *MOVE.W* instruction, the address register is decremented by two. For a *MOVE.L* instruction, the address register is decremented by four.

Address register indirection with pre-decrement is denoted by enclosing the address register name in parentheses preceded by a minus (–) symbol. For example, –(A0) denotes pre-decrement indirection on address register A0. Word or longword references require that the address contained in the address register must be even.

Example

The instruction *MOVE.L –(A1),D0* causes address register A1 to be decremented by four. The contents of the memory location pointed to by address register A1 are copied into data register D0. After the instruction executes, data register D0 and the memory location would contain the same information. Figure 2.15 shows the format of this instruction.

The instruction works as shown in Figure 2.16. $1000 indicates the contents of the longword in memory at address 1000 hex.

A special case occurs when the address register specified is A7, which is the hardware stack pointer. Byte operations on address register A7 cause a decrement by two rather than one. This ensures that the stack pointer always contains an even address.

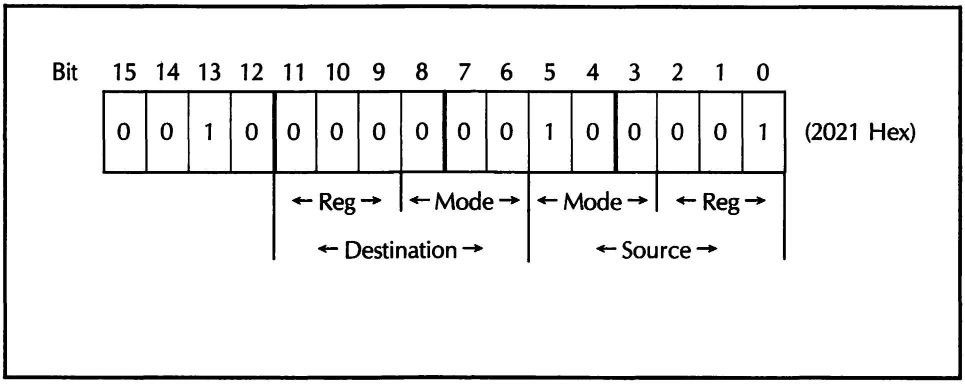


Figure 2.15 *- Format of the MOVE.L −(A1),D0 instruction*

Instruction	Before	After
MOVE.L −(A1),D0	D0= FFFFFFFF A1=00001004 $1000=01234567	D0=01234567 A1=00001000 $1000=01234567

Figure 2.16 *- An example of the MOVE.L −(A1),D0 instruction*

Address Register Indirect Addressing with Displacement

Addressing Mode Field:	101
Register Field:	000–111 (Address Register Number)
Assembler Syntax:	x(An) (x is 16 bits, n is 0–7)

Description

Address register indirect addressing with displacement is indicated by an effective address mode field of 101 (binary). The register field contains a number from 000 to 111 (0–7), which indicates an address register. In this type of addressing, the address register indicated by the register field is added to the sign-extended 16-bit number following the instruction. The result is the address of a memory location that contains the operand.

Address register indirect addressing with displacement is denoted by enclosing the address register name in parentheses preceded by a 16-bit constant. For example, 8(A0) denotes the memory location whose address is the contents of A0 plus 8. Word or longword references require that the address generated must be even.

Example

The instruction *MOVE.L 4(A1),D0* causes the contents of the memory location pointed to by address register A1 plus 4 to be copied into data register D0. After the instruction executes, data register D0 and the memory location contain the same information. Figure 2.17 shows the format of this instruction.

The instruction works as shown in Figure 2.18. $1004 indicates the contents of the longword in memory at address 1004 hex.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	(2029 Hex)
					← Reg →			← Mode →			← Mode →			← Reg →			
					← Destination →						← Source →						
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	(0004 Hex)

***Figure 2.17** – Format of the MOVE.L 4(A1),D0 instruction*

Instruction	Before	After
MOVE.L 4(A1),D0	D0=FFFFFFFF A1=00001000 $1004=01234567	D0=01234567 A1=00001000 $1004=01234567

***Figure 2.18** – An example of the MOVE.L 4(A1),D0 instruction*

Note that displacement values greater than 7FFF (hex) subtract from rather than add to the value in the address register. This is due to the sign extension.

Address Register Indirect Addressing with Index

Addressing Mode Field:	110
Register Field:	000–111 (Address Register Number)
Assembler Syntax:	x(An,Dn.L) (x is 8 bits, n is 0–7)
	x(An,Dn.W)
	x(An,An.W)
	x(An,An.L)

Description

Address register indirect addressing with index is indicated by an effective address mode field of 110 (binary). The register field contains a number from 000 to 111 (0–7), which indicates an address register. In address register indirect addressing with index, the address register indicated by the register field is added to the contents of another register, plus a sign-extended 8-bit displacement. The sum of these three quantities is the address of a memory location that contains the operand.

The second register is called the *index register,* and may be either a data register or an address register. The size of the index register may be either a word or a longword. Word quantities are sign-extended before use.

Indexed address register indirect addressing is denoted by enclosing the index and address register names in parentheses preceded by an 8-bit constant. The desired size of the index register is defined by using the .L or .W suffixes on the register name. For example, *4(A0,D0.L)* denotes the memory location whose address is the contents of data register D0 and the contents of address register A0 plus 4. Word or longword references require that the address so generated must be even.

The information concerning the index register and 8-bit displacement is contained in the 16-bit quantity that follows the instruction. This is called an *extension word,* and is in the format shown in Figure 2.19.

The bit labeled A is 1 if the index register is an address register, and 0 if the index register is a data register. Bits 14–12 contain the register number. The bit labeled L is a 1 if the index register is a long quantity, and 0 if the index is a word.

Example

The instruction *MOVE.L 4(A1,A2.L),D0* causes the contents of the memory location pointed to by the sum of address registers A2 and A1 plus 4 to be copied into data register D0. After the instruction executes, data register D0 and the memory location contain the same information. Figure 2.20 shows the format of this instruction.

Figure 2.21 shows how the instruction works. $2004 indicates the contents of the longword in memory at address 2004 hex.

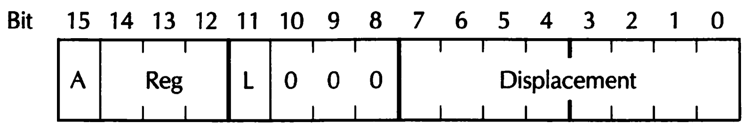


Figure 2.19 *– Format of the index register extension word*

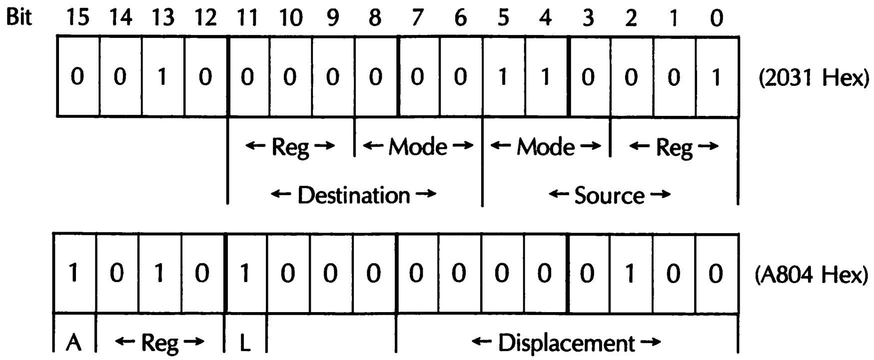


Figure 2.20 *– Format of the MOVE.L 4(A1,A2.L),D0 instruction*

Instruction	Before	After
MOVE.L 4(A1,A2.L),D0	D0=FFFFFFFF A1=00001000 A2=00001000 $2004=01234567	D0=01234567 A1=00001000 A2=00001000 $2004=01234567

***Figure 2.21** – An example of the MOVE.L 4(A1,A2.L),D0 instruction*

Absolute Short Addressing

Addressing Mode Field:	111
Register Field:	000
Assembler Syntax:	x (x is a 16-bit constant)

Description

Mode 7 with a register field of zero indicates that the word following the instruction is an absolute 16-bit address. The address is sign-extended before use, so that address specifications 8000 hex and above refer to addresses FFFF8000 and above. Remember, however, that the high byte of the address is presently discarded. The sign extension means that short addressing is useful only for the first 32,768 (32K) bytes of memory.

Example

The instruction *MOVE.L $1000,D0* causes the contents of memory location 1000 (hex) to be copied into data register D0. (Many 68000 assemblers use the $ prefix to indicate hex numbers.)

Figure 2.22 shows the format of this instruction.

Figure 2.23 shows how the instruction works.

$1000 indicates the contents of the longword in memory at the address 1000 hex.

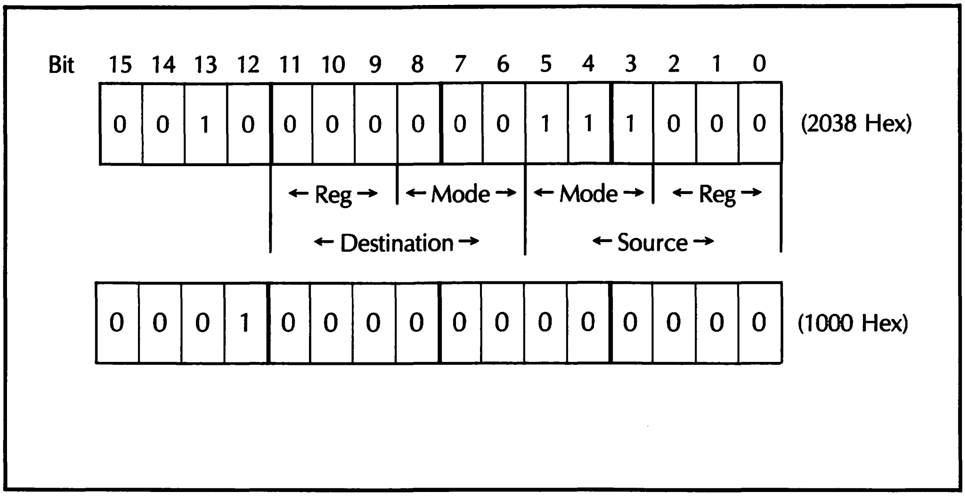


***Figure 2.22** – Format of MOVE.L $1000,D0 instruction with 16-bit address*

Instruction	Before	After
MOVE.L $1000,D0	D0=FFFFFFFF $1000=01234567	D0=01234567 $1000=01234567

***Figure 2.23** – An example of the MOVE.L $1000,D0 instruction with 16-bit address*

Absolute Long Addressing

Addressing Mode Field: 111
Register Field: 001
Assembler Syntax: x (x is a 32-bit constant)

Description

Mode 7 with a register field of one indicates that the longword following the instruction is an absolute 32-bit address. Remember that the high byte of the address is presently discarded.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	(2039 Hex)
					← Reg →			← Mode →			← Mode →			← Reg →			
					← Destination →						← Source →						
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	(0001 Hex)
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(0000 Hex)

Figure 2.24 *- Format of MOVE.L $10000,D0 instruction with 32-bit address*

Instruction	Before	After
MOVE.L $10000,D0	D0=FFFFFFFF $10000=01234567	D0=01234567 $10000=01234567

Figure 2.25 *- An example of the MOVE.L $10000,D0 instruction with 32-bit address*

Example

The instruction *MOVE.L $10000,D0* causes the contents of the memory location specified by the address of the instruction plus 102 hex to be copied into data register D0. Figure 2.24 shows the format of this instruction. Figure 2.25 shows how the instruction works.

$10000 indicates the contents of the longword in memory at address 10000 hex. Many 68000 assemblers use the prefix $ to indicate hex numbers.

Program Counter with Displacement

Register Field:	010
Assembler Syntax:	x(PC) (x is a 16-bit constant)

Description

Mode 7 with a register field of two indicates that the word following the instruction is a displacement to be added to the program counter in order to obtain a memory address. The displacement is sign-extended before the addition takes place. Thus, it is possible to address memory in a range from −32,768 to +32,767 bytes relative to the present instruction. The value used for the program counter is the address of the displacement word.

The program counter with displacement is denoted as *xxxx(PC),* where *xxxx* is a constant 16-bit number.

Example

The instruction *MOVE.L $100(PC),D0* causes the contents of the memory location specified by the address of the instruction plus 102 hex to be copied into data register D0. Figure 2.26 shows the format of this instruction.

Suppose the first word of the instruction is at location 1000 hex. Figure 2.27 shows how the instruction would work.

$1102 indicates the contents of the longword in memory at address 1102 hex. Many 68000 assemblers use the prefix $ to indicate hex numbers.

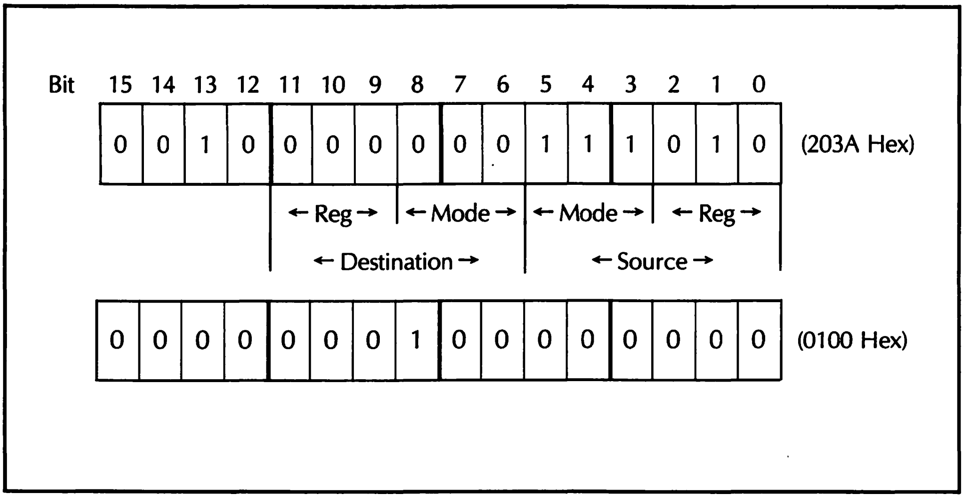


Figure 2.26 - *Format of the MOVE.L$100(PC),D0 instruction*

Instruction	Before	After
MOVE.L $100(PC),D0	D0=FFFFFFFF $1102=01234567	D0=01234567 $1102=01234567

Figure 2.27 – *An example of the MOVE.L $100(PC),D0 instruction*

Program Counter with Index

Addressing Mode Field:	111
Register Field:	011
Assembler Syntax:	x(PC,Dn.L) (x is 8 bits, n is 0–7) x(PC,Dn.W) x(PC,An.W) x(PC,An.L)

Description

Mode 7 with a register field of 3 indicates that the memory address is to be constructed using the value of the program counter, an index register, and a sign-extended 8-bit displacement. This mode is similar to the address register with index mode instruction. The same format extension word is required. Figure 2.28 shows the format of this instruction.

The program counter with index is denoted as xxx(PC,xr.s), where xxx is a constant 8-bit number, and xr.s is a register name with size specification. For example, indexing with the word contained in D0 and a displacement of 10 hex is written *$10(PC,D0.W)*.

Example

The instruction *MOVE.L $10(PC,A1.L),D0* causes the contents of the memory location at the address of the instruction plus the contents of A1 plus 12 hex to be copied into data register D0. Figure 2.29 shows the format of this instruction.

Suppose the first word of the instruction is at location 1000 hex. Figure 2.30 shows how the instruction would work.

$2012 indicates the contents of the longword in memory at address 2012 hex.

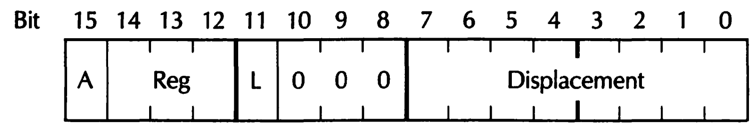


Figure 2.28 *– Format of program counter with index extension word*

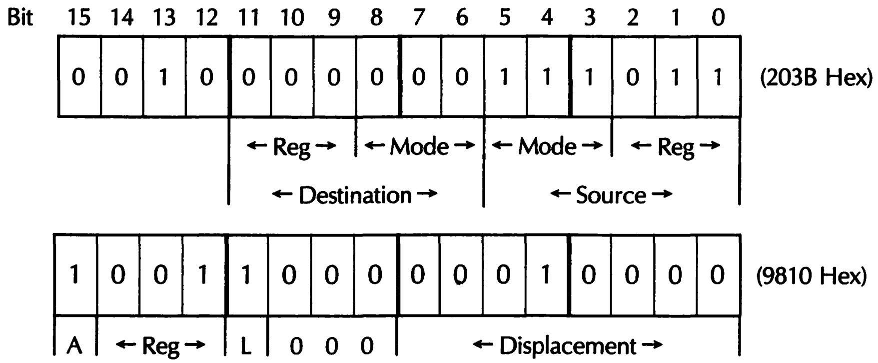


Figure 2.29 *– Format of MOVE.L $10(PC,A1.L),D0 instruction*

Instruction	**Before**	**After**
MOVE.L $10(A1.L,PC),D0	D0= FFFFFFFF $2102=01234567 A1=00001000	D0=01234567 $2102=01234567 A1=00001000

Figure 2.30 *– An example of the MOVE.L $10(PC,A1.L),D0 instruction*

Immediate Mode

Addressing Mode Field: 111 (Source only)
Register Field: 100
Assembler Syntax: #x (x is 8, 16, or 32 bits)

Description

Mode 7 with a register field of 4 indicates that the source data for an instruction is contained in the word or longword (depending on the size of the instruction) that follows the instruction. Byte data for an immediate mode instruction is contained in the low-order byte of the word following the instruction.

Immediate mode is denoted by #(constant) where (constant) is a hex or decimal number. Many assemblers allow symbols to be defined for use as immediate quantities. (See the section on assemblers in Chapter 3 for additional information.)

Example

The instruction *MOVE.L #$10002000,D0* causes the long constant 10002000 (hex) to be loaded into data register D0. The previous contents of D0 are lost. Figure 2.31 shows the format of this instruction. Figure 2.32 shows how this instruction works.

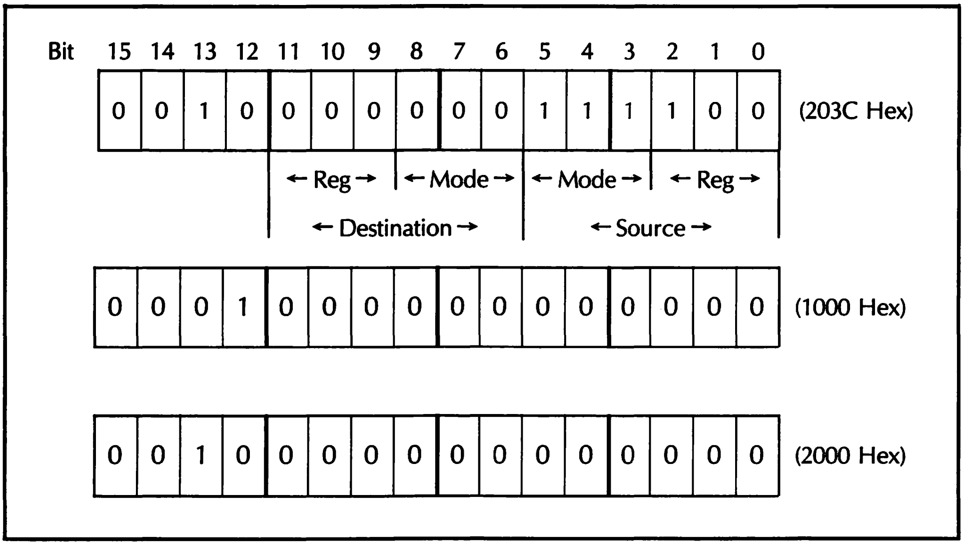


***Figure 2.31** – Format of the MOVE.L #$10002000,D0 instruction*

Instruction	Before	After
MOVE.L #$10002000,D0	D0=01234567	D0=10002000

Figure 2.32 *– An example of the MOVE.L #$10002000,D0 instruction*

Status Register Addressing

Addressing Mode Field:	111 (Destination only)
Register Field:	100
Assembler Syntax:	SR
	CCR

Description

Mode 7 with a register field of 4, when used as a destination field on some instructions, indicates that the operation is to be performed on the status register. The And Immediate (ANDI), Exclusive Or Immediate (EORI), and Or Immediate (ORI) instructions are the only operations that can use this addressing mode.

When the instruction specifies a byte length, then only the user byte of the status register is affected. When a word length instruction is used, then both the system and user bytes are affected. The System bit in the status register must be set to 1 in the latter case.

The assembler recognizes the special labels SR (for the whole status register) and CCR (for the user byte). CCR is an acronym for Condition Code Register. Only the condition codes are stored in the status register user byte.

Example

The *ORI #5,CCR* instruction sets both the carry (C) and zero (Z) flags. Figure 2.33 shows the format of this instruction. Figure 2.34 shows how the instruction works.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	(003C Hex)
											← Mode →			← Reg →			
											← Destination →						
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	(0005 Hex)

Figure 2.33 *– Format of ORI #5,CCR instruction*

Instruction	Before	After
ORI #5,CCR	CCR=0000	CCR=0005

Figure 2.34 *– An example of the ORI #5,CCR instruction*

STACKS AND STACK FRAMES

Many commercial microcomputers today (including the 68000) have a feature called a *stack*. A stack is a storage technique similar to the spring-loaded platforms used for plates in a cafeteria line. The last byte, word, or longword, placed on the stack is the first data item to be removed. This storage scheme is called *Last-In-First-Out* (LIFO). The act of placing a new data item on the stack is known as a *push*. Removing a data item is commonly called a *pop*.

How a Stack Works

Stacks are implemented on the 68000 using the pre-decrement and post-increment addressing modes. An address register (called the *stack pointer*) is used to indicate the top of the stack's position in memory. Data items are pushed onto the stack using the –(An) addressing mode, and popped using the (An)+ addressing mode, as illustrated in Figure 2.35.

The stack pointer always contains the address of the element on top of the stack. Subsequent push operations cause items to be stored at lower addresses. Pop operations cause the stack pointer to be incremented toward higher addresses. The stack is said to "grow toward lower addresses" on the 68000.

68000 Hardware Stack

Register A7 on the 68000 is called the *hardware stack pointer.* This register is used by the 68000 hardware for addressing memory that contains temporary data items. Most 68000 assemblers take the symbol SP (for Stack Pointer) as an alternative to A7 in register specifications.

There are two stack pointers on the 68000: one for when the processor is in user mode (called USP) and one for when the processor is in supervisor mode (called SSP).

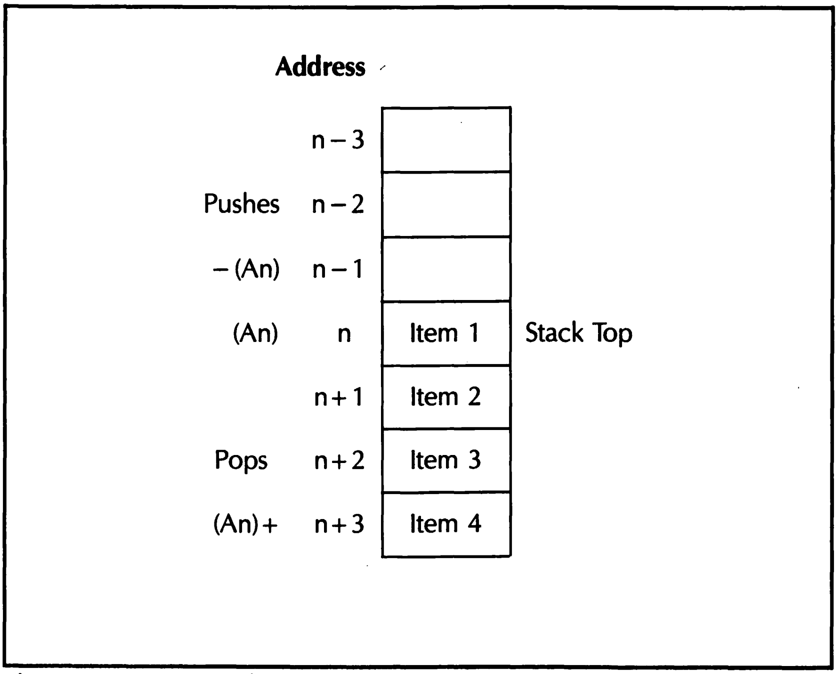


Figure 2.35 *– 68000 stack operations*

A special instruction, *MOVE USP,* allows the supervisor program to access the USP. The user mode program is not allowed access to the SSP. The present Macintosh runs in supervisor mode and does not use user mode.

Typically, the stack is used for information that will be required again later. For example, there is a common programming technique called a *subroutine.*

Subroutines

Subroutines are small programs in themselves that can be used as units of other, larger programs. Typical subroutines include I/O routines and common calculations, such as taking the square root of a number.

The 68000 supports instructions known as *subroutine calls.* These instructions place the address of the next instruction on the stack and the address of the subroutine into the program counter (PC). In this way, the next instruction to be executed becomes the first instruction in the subroutine. When the subroutine has completed processing, it executes a *return* instruction, which restores the address currently on top of the stack back into the PC. This causes the program to resume execution at the instruction that immediately follows the subroutine call.

Subroutines are powerful programming tools. A subroutine is essentially an extension to the instruction set of the machine. The programmer can treat subroutine calls as if they were sophisticated machine instructions.

Exceptions

A concept similar to the subroutine call is the 68000 exception mechanism. The 68000 allows the suspension of a program and subsequent resumption of the same program through a technique known as an *exception.* (This same technique is called an *interrupt* on other machines.)

An exception causes the status register and program counter to be pushed onto the stack. A special instruction allows restoration of the program counter/status register combination at a later point. This mechanism is described in detail in Chapter 7, Exception Programming.

Stack Implemented in Software

You can implement a stack using any address register. All that is required is to place the address of the end of the area to be used as a stack into the address register. The pre-decrement and post-increment addressing modes can then be used to push and pop data items from this software stack.

Stack Frames

Stacks are convenient for allocating temporary memory areas. The 68000 supports a hardware mechanism for allocating scratchpads called a *stack frame.*

The 68000 *LINK* and *UNLK* (unlink) instructions allocate and free temporary memory at the top of the stack. An additional address register, called the *frame pointer,* is used to point to the area allocated on the stack. References to the stack frame use the address register with displacement addressing mode. The frame pointer rather than the stack pointer is used to address the frame so that subsequent stack *PUSH* and *POP* operations will not affect the offsets of individual components of the frame.

SUMMARY

The important points that we have covered in this chapter are:

- The 68000 has sixteen registers: eight data registers and eight address registers. Data registers may be used as bytes, words, or longwords. Address registers may be used only as words or longwords. In addition, loading a word into an address register causes the word to be sign-extended to 32 bits.

- There are two special registers: the program counter (PC) and the status register (SR). The program counter contains the address of the next instruction to be executed. The status register contains machine status bits. The upper eight bits of the status register, called the system byte, may not be accessed by ordinary programs. The lower eight bits of the status register, called the condition code register (CCR), contain status bits (condition codes) that indicate the result of the last instruction executed.

- The 68000 supports three principle numeric data types: binary, BCD, and floating-point. Binary data may be used in units of 8 bits (a byte), 16 bits (a word), and 32 bits (a longword). These lengths are indicated on register and instruction names by the suffixes .B, .W, and .L. BCD is a method of storing two decimal digits per byte. Floating-point is a method of representing very large or very small numbers without requiring undue amounts of memory.

- The 68000 supports fourteen distinct methods of specifying data in an instruction. These are called *addressing modes.* They are listed in Table 2.1.

Syntax	Name
Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect post-increment
–(An)	Address register indirect pre-decrement
w(An)	Address register with displacement
b(An,Rn)	Address register with index
w(.W)	Absolute short
l(.L)	Absolute long
w(PC)	PC with displacement
b(PC,Rn)	PC with index
#x	Immediate
SR	Status register (Privileged)
CCR	Condition code register

b is a byte constant.
w is a word constant.
l is a long constant.
x can be any of these.
n is a register number, 0–7.
R is a register specifier, either A or D.

Table 2.1 *– Addressing Modes of the 68000*

- The post-increment and pre-decrement addressing modes are used to implement data structures called stacks. Stacks are organized in a last-in first-out (or LIFO) scheme in which the last data item to be put on is the first one taken off. Register A7 is used by the 68000 instructions to refer to a special stack called the hardware stack.

- Special 68000 instructions exist that allocate and free temporary scratchpad memory areas on the hardware stack. These scratchpad areas are known as *stack frames*.

Chapter 3 will build on this background to present the 68000 instruction set and the mechanics of writing 68000 assembly language programs.

EXERCISES

1. Given the following conditions:

   - D0 = 00008000
   - A0 = 00001000
   - A7 = 00010000

   What are the results of the following instructions?

   ```
   MOVE.B  D0,A0
   MOVE.W  D0,A0
   MOVE.B  D0,(A0)+
   MOVE.B  D0,-(A7)
   ```

   Give the new contents of all registers and memory locations that change. Use the same starting conditions for all the instructions.

2. The ADD binary instruction can add an effective address operand to a data register. The format of this instruction is shown in Figure 2.36.

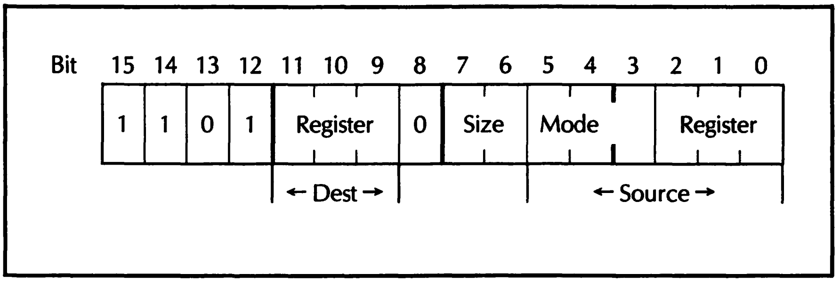
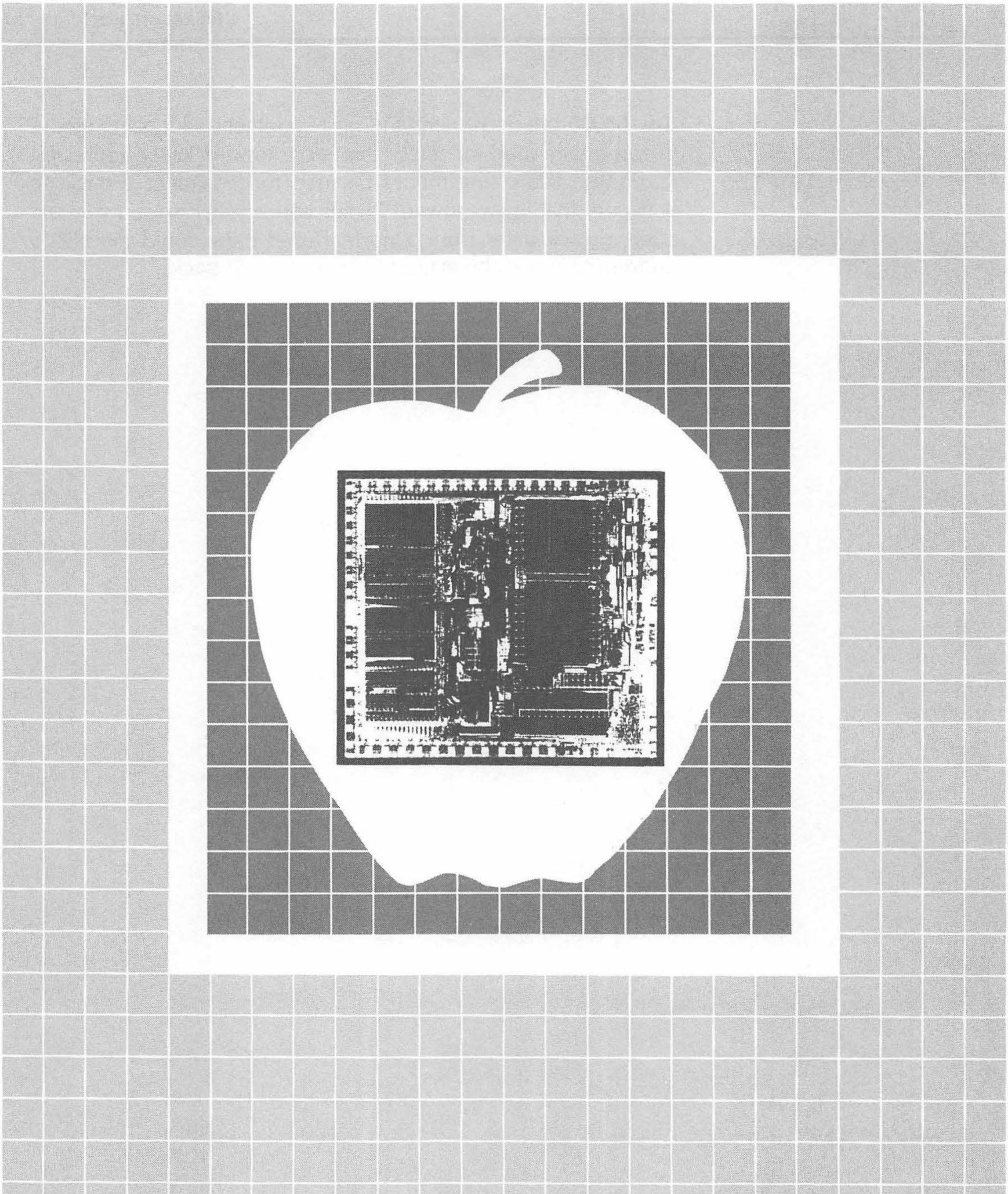


Figure 2.36 *– ADD binary instruction format*

The Size field is 00 for byte, 01 for word, and 10 for long data. Modify the examples for the addressing modes (except for the status register mode) to use this instruction.

3. The 68000 pre-decrement and post-increment addressing modes are normally used for stacks that grow toward lower addresses. Can these addressing modes be used to implement a stack (in software) that grows toward higher addresses? If so, how would such a stack differ from a normal one? How could the 68000 addressing modes be altered to support such stacks?



Chapter 3

68000 Instruction Set

Trifles make perfection, but perfection itself is no trifle.

—Michelangelo

INTRODUCTION

This chapter covers the mechanics of program generation and the details of the 68000 instruction set. It is probably not necessary to study each instruction in detail, but you should pay attention to the section on the mechanics of program generation. This information is vital to understanding the instruction descriptions.

INSTRUCTION CLASSES

The 68000 instructions fall into eight classifications:

- Data movement
- Integer arithmetic
- Logical
- Shift and rotate
- Bit manipulation
- BCD
- Program control
- System control

We will now describe each of these classes in detail.

Data Movement

Data movement instructions transport data from one location in the 68000 to another. Normally, these instructions move from one to four bytes of data between two registers, a register and memory, or between two memory locations.

This category of 68000 instructions includes:

- The EXG (EXchanGe) instruction. Exchanges the contents of two registers.
- The LEA (Load Effective Address) instruction. Calculates a memory address and places it in an address register.
- The LINK instruction. Allocates a stack frame.
- The MOVE instruction. Transfers one register or memory location to another.
- The MOVEM (Move Multiple) instruction. Transfers multiple registers to or from memory.
- The MOVEP (Move Peripheral) instruction. Transfers data to or from an 8-bit peripheral.
- The MOVEQ (Move Quick) instruction. Loads a data register with a constant.
- The PEA (Push Effective Address) instruction. Calculates a memory address, and pushes it onto the hardware stack.
- The SWAP instruction. Swaps the words in a data register.
- The UNLK (UNLinK) instruction. Deallocates a stack frame.

Integer Arithmetic Operations

Integer arithmetic instructions perform basic two's complement operations on binary data. This class of instructions includes:

- The ADD, ADDA, ADDI, ADDQ, and ADDX instructions. Used for two's complement addition.
- The CLR instruction. Moves zero into an operand.
- The CMP, CMPA, CMPI, and CMPM instructions. Compares two quantities.

- The DIVS and DIVU instructions. Perform signed and unsigned integer division.
- The EXT instruction. Sign extends a byte to a word quantity or a word to a long quantity.
- The MULS and MULU instructions. Used for signed and unsigned multiplication operations.
- The NEG and NEGX operations. Form the two's complement of a number.
- The SUB, SUBA, SUBI, SUBQ, and SUBX instructions. Used for two's complement subtraction.
- The TAS (Test and Set) instruction. Used to synchronize multiple processors.
- The TST instruction. Compares an operand to zero.

Logical Operations

The logical operation instruction group performs bit-wise Boolean operations on binary numbers. This class of instructions includes:

- The AND and ANDI instructions. Perform a Boolean AND operation on two binary integers.
- The OR and ORI instructions. Perform a Boolean OR operation.
- The EOR and EORI instructions. Perform a Boolean exclusive OR operation.
- The NOT instruction. Perform a one's complement operation.

Shift and Rotate Operations

The shift and rotate instructions perform arithmetic and logical shifts, as well as rotates with and without an auxiliary Carry bit. This group contains:

- The ASL and ASR instructions. Arithmetic left and right shift operations.
- The LSL and LSR instructions. Logical left and right shift operations.

- The ROL and ROR instructions. Left and right rotates without an auxiliary Carry bit.
- The ROXL and ROXR instructions. Left and right rotates through an auxiliary Carry bit.

Bit Manipulation Operations

The bit-manipulation instructions operate on single bits within a byte. This instruction class contains:

- The BTST instruction. Tests a single bit.
- The BSET instruction. Tests a single bit and then sets the bit.
- The BCLR instruction. Tests a single bit and then clears the bit.
- The BCHG instruction. Tests a single bit and then inverts (complements) the bit.

Binary Coded Decimal Instructions

The binary coded decimal (BCD) instructions manipulate numbers in BCD format. This group contains:

- The ABCD instruction. Performs BCD add operations.
- The SBCD instruction. Performs BCD subtract operations.
- The NBCD instruction. Performs BCD negation.

Program Control Instructions

The program control instructions alter the instruction flow through a program segment. This group contains:

- The Bcc instruction group of fifteen conditional-branch instructions. Conditionally alter the flow of instructions.
- The DBcc instruction group, consisting of sixteen looping-primitive instructions.
- The Scc instruction group of sixteen conditional-set instructions. Set a byte depending on the settings of the condition code.

- The BSR and JSR subroutine call instructions.
- The RTS subroutine return instruction.
- The JMP absolute jump instruction.
- The RTR instruction. Restores the program counter and condition codes from the stack.

System Control Operations

System control instructions alter the state of the 68000 hardware environment.

Many of these instructions are "privileged," meaning that they require that the 68000 be executing in supervisor mode. This instruction class contains:

- The MOVE USP instruction. Allows a supervisor mode program access to the user mode stack pointer.
- The RESET instruction. Resets external devices.
- The RTE instruction. Returns from an exception condition. (Exceptions are described in Chapter 7.)
- The STOP instruction. Suspends instruction processing until an external event occurs.
- The CHK and TRAPV instructions. These instructions detect catastrophic program errors.
- The TRAP group of sixteen instructions. Provide a method for a user mode program to call a supervisor mode program.

All Macintosh software runs in supervisor mode, so there is no restriction on the instructions your programs can execute. You should, however, exercise care in using privileged instructions. Improper use of these instructions may cause the system to crash, forcing you to reboot.

PROGRAM DEVELOPMENT MECHANICS

The process of writing an assembly language program for the Macintosh is illustrated in the following four steps:

1. Key in the program using EDIT (the text editor). The disk file that results is called the *assembly source (.asm) file.*

2. Transform the program into machine code using the *assembler* program. The machine-code file produced by the assembler is called an *object (.rel) file*. The assembler can also produce a *listing (.lst) file* that contains the instructions from the source file along with the machine code in the object file (produced in hexadecimal).

3. Before the program can be executed, the Macintosh Development System requires that the object output of the assembler be processed by a program called a *linker* or *linkage editor.* Many large programs are split into separate source files, assembled separately, and then combined with the linker. This process results in a file that is ready to be loaded into memory and executed. This file is called a *load file, executable file,* or a *load module.* The load file produced by the linker shows up as an "application" in the Macintosh Finder.

4. The load file produced by the above steps is then loaded into memory and executed.

This process is illustrated in Figure 3.1.

Usually, the programmer needs to repeat each of these steps many times. Often, the file must be edited many times before it assembles without errors. Then, the assembled file still may not link properly, necessitating more editing and assembling. Finally, the successfully linked file may not run properly.

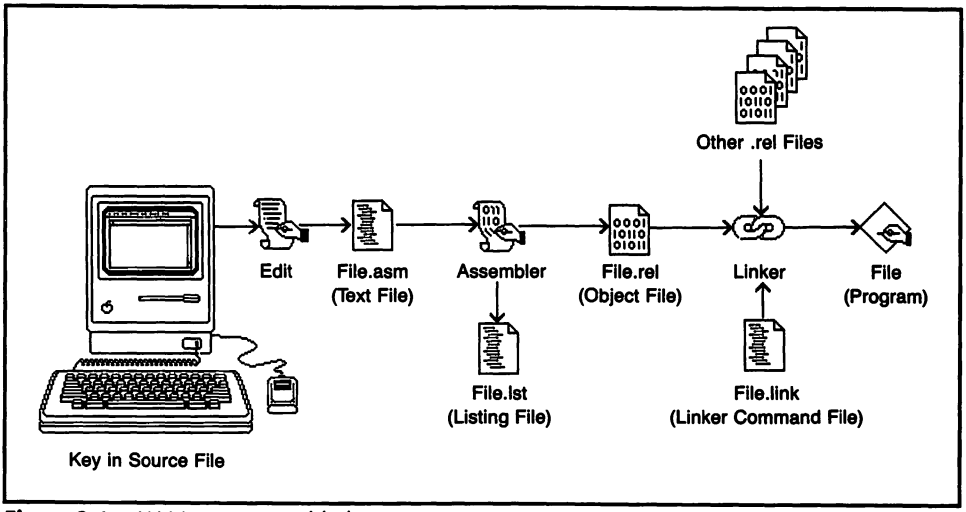


Figure 3.1 – *Writing an assembly language program*

This is the time to check for *bugs,* or errors in the program. Finding bugs in object programs is more of an art than a science. Fortunately, there are tools that make the task easier. One is the popular method of printing out values within the program by inserting a temporary printing code. The second, and more valuable tool, is the *interactive debugger.* This program allows you to stop at certain points in the program and look at the values that are currently in the registers and in memory. Many debuggers also allow you to specify labels contained in the program source file.

There are several assembly language development packages available for the Macintosh. In this book, we will use the Macintosh Development System from Apple.

Editing

Developing programs requires only a simple editor. There are many to choose from, and determining which is the best one is a matter of personal preference.

The Macintosh editor is called *EDIT;* it operates only on files known as *textfiles.* The source file for the assembler must be a textfile. You can also create textfiles using MacWrite with the Text Only option on the Save menu. Other examples of textfiles include the listing file produced by the assembler, the MAP file produced by the linker, and the information saved by MacTerminal from an interactive session with another computer. You can use EDIT to inspect, alter, or print any textfile.

EDIT is a mouse-driven, multiwindow, screen-oriented editor. You can edit multiple files, arranging and sizing the display for each file on the screen. You can even cut information from one file and paste it into another.

Backups

Maintaining backup copies of source files is essential because of the possibility of a hardware or software failure. While hardware failures are comparatively rare, they still occur from time to time. Software and media failures are all too common. Power failures can also destroy disk files. There is a bit of conventional engineering wisdom known as Murphy's Law:

If anything can go wrong, it will.

You will find that this law applies to programming as well. Most programmers keep *two* backup copies in case the machine crashes while making a backup. (If this occurs, it is possible to destroy *both* the original and the backup.) Recovery from such a disaster is extremely painful. Unfortunately, most people have to learn this lesson the hard way, as Ben Franklin said:

Experience is a dear school, but fools will learn at no other.

Don't be one of them! Back up your files!

Source File Format

The source file that you input to the assembler consists of lines of text. Each line in the source file can be classified as one of the following four things:

- A comment line. Comments in the Macintosh assembler are denoted by placing a semicolon (;) as the first character on the line. Comments can also be placed at the end of any other line.
- A 68000 instruction. Instruction lines consist of an optional label, followed by an instruction mnemonic, followed by one or more operands. Spaces or tabs are required between the label, mnemonic, and operand fields. Commas are used to separate multiple operands. The assembler does not allow spaces between operands.
- A macro invocation. *Macros* are predefined sequences of source lines that are invoked by a single source line.
- An assembler directive. Directives are a means of telling the assembler how you want your program treated. The Macintosh assembler has the following directives:

1. STRING_FORMAT. This directive controls how the assembler handles text in quotes.
2. IF, ELSE, and ENDIF. These directives include or ignore part of a program, depending on the result of an expression. This feature is called *conditional assembly*. Programs that require source changes to work in different environments often use conditional assembly to generate multiple versions of the program from a single source file.
3. INCLUDE. This directive causes the entire contents of a file to be used in the assembly at the point where the INCLUDE directive occurs.
4. MACRO and .MACRO/.ENDM. These directives define a set of source lines to be used by a macro invocation.

5. EQU. This directive allows you to create a symbolic name for a constant value.

Assembling

The Macintosh assembler takes a textfile as input and produces a machine-code (object) file and an optional program listing file. There are a set of file-name conventions that must be followed. For an application named "Program", the assembly source file is named "Program.asm". The object file produced by the assembler is called "Program.rel", and the listing file is called "Program.lst". The Macintosh ignores case differences in file names, so these names can be any combination of upper- and lowercase letters.

Linking

After obtaining an error-free assembly, you must link the object file in order to make it executable. For the linker to work, you must write a textfile, called a *linker command file*, containing the names of the files to be linked. The file names must be listed one per line, ending with a line containing a dollar sign ($) as the first character. If your application is named "Program", the linker command file is usually called "Program.link".

For example, assume you want to link five object files, named File1.rel through File5.rel. You could create a file called Files.link, containing the following:

```
file1
file2
file3
file4
file5
$
```

When the linker runs, it places the executable program in a file called File1.

EXEC

For applications involving more than one source file or long assembly and link times, you can automate the assembly and link process. The Macintosh Development System includes an application called *EXEC* that executes commands contained in a textfile. This textfile must consist of lines of text in the following format:

```
application	file	program1	program2
```

The command file must have a ".job" extension, and you must separate all of the fields with a single tab character. EXEC interprets two consecutive tab characters as a blank field. Application is the name of a program to be run, and File is the file the program should operate on. If the application completes without error, Program1 is run. If the application terminates because of errors, Program2 is run. Normally, Program1 is EXEC and Program2 is EDIT. If the application terminates with no errors, EXEC runs again and starts the next application. In an error situation, running EDIT allows you to make corrections to the file causing the error. You can then use EXEC's Resume and Redo Last option to retry the failing step.

For instance, if you want to assemble the five text files File1.asm through File5.asm and then link them using the linker command file Files.link, you could use a file named Files.job, which contains the following lines of text:

```
asm	file1.asm	exec	edit
asm	file2.asm	exec	edit
asm	file3.asm	exec	edit
asm	file4.asm	exec	edit
asm	file5.asm	exec	edit
link	files.link	exec	edit
```

If an error is detected during, say, the assembly of File3.asm, the editor is automatically invoked as if you had opened File3.asm from the Finder. After fixing the problem with the editor, you could select EXEC from the editor's Transfer menu. Once inside EXEC, you could select Resume and Redo Last from the File menu. EXEC would then resume the job stream, starting by reassembling File3.asm.

Debugging

The application or load file created by the linker can be executed in the same way as any other Macintosh application: by selecting the application icon in the Finder, in the Open Application menu of EXEC, or in the Transfer menu in the linker. But what if the file doesn't execute correctly? How do you figure out what is wrong?

When something goes wrong, use the interactive debugger. The Macintosh has three interactive debuggers: one that uses the bottom of the screen as a debugging "window," one that uses an external serial terminal, and one that uses another Macintosh. These debuggers are called

MacsBug, TermBug, and MacDB, respectively. A special version of MacsBug, called MaxBug, can be used on a Macintosh with 512K of memory. (This machine is known affectionately as a "Fat Mac.") MaxBug allows you to use the whole screen for debugging and to switch back and forth from the debugging screen to the application screen. All of these debugging programs allow you to interact with an executing program so that you can identify any problem areas.

The Macintosh debuggers all provide the following features:

- The ability to examine and change the contents of a machine register or a memory location.
- The ability to start execution at a desired location, and stop execution at one of several points (commonly called *breakpoints*).
- The ability to execute a single instruction at a time. This is commonly called *tracing*, because it is the Trace bit in the status register that provides this capability.

To use a debugger, you must first install the one you want to use in the system. Then you must activate the debugger after the program you want to debug has been loaded into memory. Installing the debugger is described below. To activate the debugger after the test program has been loaded, place an illegal instruction as the first instruction of the program. This causes the debugger to be activated as soon as the program starts execution.

You can also activate a debugger by using the *interrupt* button on the side of the Macintosh. (You should install one if you haven't already—it is very useful in assembly language programming.)

Installing the Debugger

The two one-machine debuggers, MacsBug and TermBug, are installed by copying the desired debugger to a file named MacsBug on the startup disk. When the system boots, the debugger is loaded automatically. The message "MacsBug installed" should appear below the "Welcome to Macintosh" message printed when the system boots. MacsBug (which needs no additional hardware) occupies 16K of memory. TermBug (which uses an external terminal) requires about 10K of memory. You can change the debugger name in the boot record with the application "Boot Config".

To install the MacDB (two-machine) debugger, run a serial cable between the machines. Then run MacDB in one machine and a program called a "debug nub" in the target machine. The machine running

MacDB will accept debugging commands for the application running in the target machine.

Example—Summing the First Five Integers

To illustrate how to use the Macintosh programming tools, let's write a simple program that sums the first five integers and stores the result in memory. We need to write an assembly source file, a linker control file, and an EXEC job file. These files are shown on the screen in Figure 3.2.

The first line of the Sum5.asm file generates an illegal instruction that causes the debugger to gain control. Subsequent lines accumulate the sum of the numbers 1 through 5, stored in memory locations A through E, in register D0.W. The LEA instruction causes A0 to point to memory location F. D0 is then stored there. The RTS instruction returns control to the Finder.

The files on disk before assembly and linking are shown in Figure 3.3. The disk with the assembler, linker, and EXEC utility is called "MDS1 Backup" in this example. Invoking EXEC on the file Sum5.job causes the program to be assembled and linked, producing the files shown in Figure 3.4.

The assembler listing file for Sum5 is shown in Listing 3.1. The first column of numbers contains the decimal line number of the source line

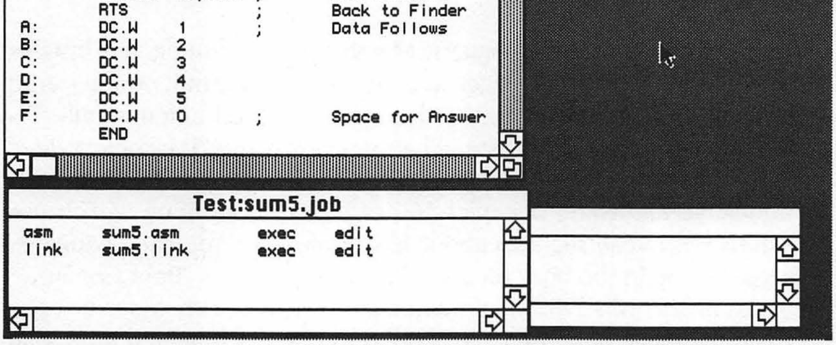


***Figure 3.2** – Creating the Sum5 source files*

in the source file. The second column of numbers contains the hexadecimal address of each instruction, relative to the first instruction in the file.

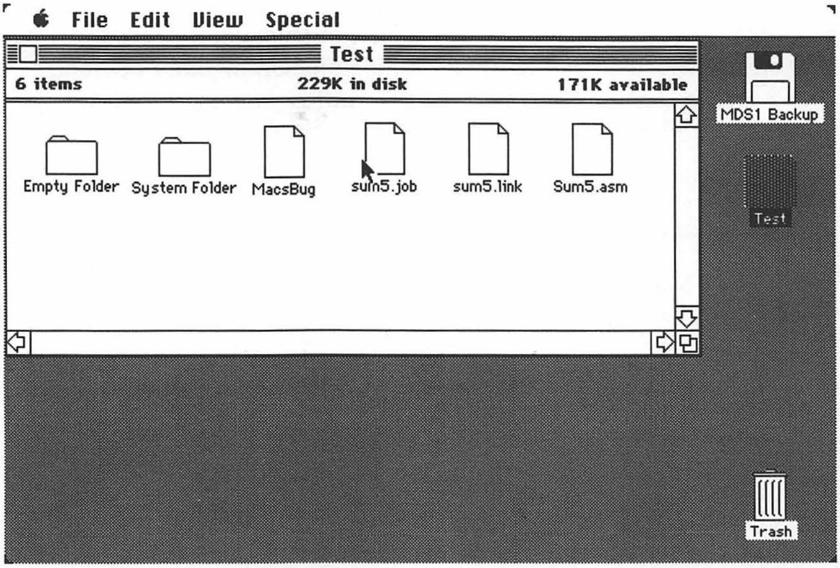


Figure 3.3 *– The Sum5 source files on disk*

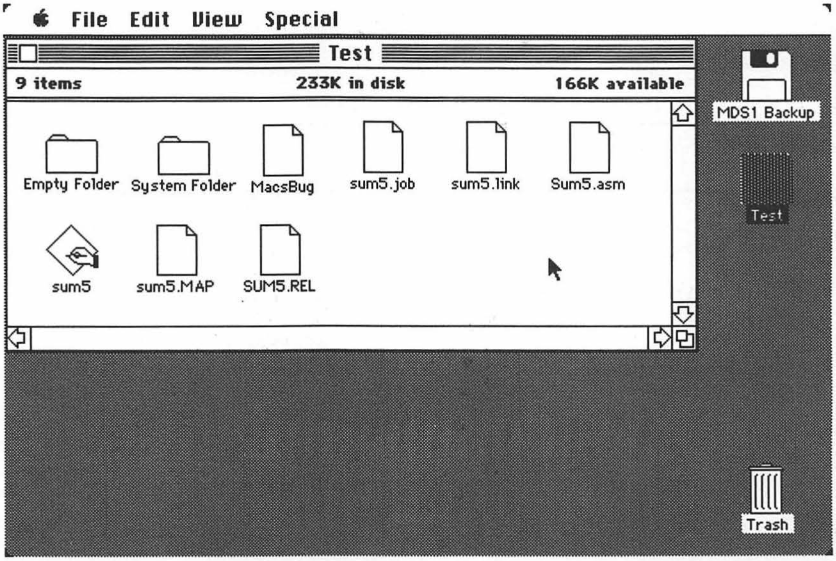


Figure 3.4 *– The Sum5 object files on disk*

The binary machine code generated appears next, followed by the line from the source file. Locations marked "xxxx" are addresses that must be processed by the linker. The Macintosh assembler uses the PC-relative addressing mode for labels on instructions or DC (Define Constant) assembler directives. This is why the LEA instruction is necessary—PC-relative addressing cannot be used on a destination memory operand.

To execute the program, we used TermBug and captured the output and keyboard input in a file. The debugging session is shown in Listing 3.2.

```
0000  FF01                  DC.W    $FF01
0002  303A xxxx(R)          MOVE.W  A,D0     ; Load First Number
0006  D07A xxxx(R)          ADD.W   B,D0     ; Add second number
000A  D07A xxxx(R)          ADD.W   C,D0
000E  D07A xxxx(R)          ADD.W   D,D0
0012  D07A xxxx(R)          ADD.W   E,D0
0016  41C0 xxxx(PX)         LEA     F,A0     ; A0 -> Answer
001A  3080                  MOVE.W  D0,(A0)  ; Store Result
001C  4E75                  RTS              ; Back to Finder
001E  0001               A: DC.W    1        ; Data Follows
0020  0002               B: DC.W    2
0022  0003               C: DC.W    3
0024  0004               D: DC.W    4
0026  0005               E: DC.W    5
0028  0000               F: DC.W    0        ; Space for Answer
```

Listing 3.1 *– The Sum5 assembly listing file*

```
1111 ERR
>TD
00004F9A:          $$$$
PC=00004F9A SR=00002004
D0=00000000 D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00015D24 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>DM 4F9A 30
004F9A  FF 01 30 3A 00 1A D0 7A 00 18 D0 7A 00 16 D0 7A   ..0:...z...z...z
004FAA  00 14 D0 7A 00 12 41 FA 00 10 30 80 4E 75 00 01   ...z..A...0.Nu..
```

Listing 3.2 *– The Sum5 debugging session*

```
004FBA  00 02 00 03 00 04 00 05 00 00 00 00 16 30 00 01   .............0..
>IL 4F9A 9
00004F9A:          $$$$
00004F9C:          MOVE.W  *+$001A,D0
00004FA0:          ADD.W   *+$0018,D0
00004FA4:          ADD.W   *+$0016,D0
00004FA8:          ADD.W   *+$0014,D0
00004FAC:          ADD.W   *+$0012,D0
00004FB0:          LEA     *+$0010,A0
00004FB4:          MOVE.W  D0,(A0)
00004FB6:          RTS
>PC
PC=00004F9A
>PC 4F9C
>T
00004FA0:          ADD.W   *+$0018,D0
PC=00004FA0 SR=0000A000
D0=00000001 D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00015D24 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>T
00004FA4:          ADD.W   *+$0016,D0
PC=00004FA4 SR=0000A000
D0=00000003 D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00015D24 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>T
00004FA8:          ADD.W   *+$0014,D0
PC=00004FA8 SR=0000A000
D0=00000006 D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00015D24 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>T
00004FAC:          ADD.W   *+$0012,D0
PC=00004FAC SR=0000A000
D0=0000000A D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00015D24 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>T
00004FB0:          LEA     *+$0010,A0
PC=00004FB0 SR=0000A000
D0=0000000F D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00015D24 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
```

Listing 3.2 *– The Sum5 debugging session (continued)*

```
>T
00004FB4:          MOVE.W  D0,(A0)
PC=00004FB4 SR=0000A000
D0=0000000F D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00004FC2 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>T
00004FB6:          RTS
PC=00004FB6 SR=0000A000
D0=0000000F D1=00000000 D2=00000000 D3=FFFFFFFF
D4=00001200 D5=00000BFF D6=0000FFFF D7=00000000
A0=00004FC2 A1=00017E44 A2=00016ED2 A3=00017E46
A4=00006124 A5=00017E24 A6=00016D92 A7=00017D20
>DM 4F9A 30
004F9A  FF 01 30 3A 00 1A D0 7A 00 18 D0 7A 00 16 D0 7A   ..0:...z...z...z
004FAA  00 14 D0 7A 00 12 41 FA 00 10 30 80 4E 75 00 01   ...z..A...0.Nu..
004FBA  00 02 00 03 00 04 00 05 00 0F 00 00 16 30 00 01   .............0..
>G
```

Listing 3.2 – *The Sum5 debugging session (continued)*

The debugger first printed out

```
1111 ERR
```

indicating that an illegal instruction with the first four bits equal to F hex caused the debugger to gain control. This is the result of executing the instruction

```
DC.W $FF01
```

at the beginning of the program.

The debugger prints a ">" (greater than) character as an input prompt. The TD (Total Display) command causes the debugger to print the instruction to be executed next, together with the contents of all the registers. The first instruction printed out as "$$$$" because $FF01 is an illegal instruction.

Next, we used the DM (Display Memory) command to list the program. With the Macintosh, you don't know where the program will be placed in memory, so you must use the address contained in the PC when the illegal instruction is attempted. This is 4F9A. From the assembly listing, we see that the program is 28 (hex) bytes long. The command

```
DM 4F9A 30
```

causes 30 (hex) bytes to be listed starting from address 4F9A. The data listed corresponds to the machine code in the assembly listing.

Next, we used the IL (Instruction List) command to disassemble the instructions. The command

```
IL 4F9A 9
```

disassembles nine instructions starting at location 4F9A. Note that all memory locations are referenced as

```
* +hex number
```

indicating PC-relative addressing.

When the 68000 encounters an illegal instruction, the PC does not advance to point to the next instruction, as it normally does. This means that in order to execute the next instruction, we must advance the program counter by hand. The debugger command

```
PC 4F9C
```

causes the program counter to point to the first real instruction. You must do this every time you use the illegal-instruction technique to debug a program. Later versions of MDS do this automatically.

Next, we use the T (Trace) command to execute each instruction in the program. Note that the status register displayed after the instruction is executed has the Trace bit (15) and the Supervisor bit (13) set.

Prior to executing the RTS instruction, we redisplay the program in hex. Note that location 4FC2 has been changed. The debugger G (Go) command returns control to the Finder.

DDT-68K

In explaining the 68000 instruction set, we will make use of the CP/M-68K debugger, DDT-68K. We also use CP/M to execute the instructions. This system allows us to use absolute addressing for memory operands, which the Macintosh does not allow. Using absolute addresses makes it much easier to determine a memory operand's location in the examples.

This debugger is called a *hex debugger* because it lacks the ability to use labels from the source program. A debugger that can use these labels is called a *symbolic debugger.* (A later version of DDT-68K provides symbolic capability.) The Macintosh single-machine debuggers MacsBug and TermBug are hex debuggers. The Fat Mac debugger, MaxBug, and the two-machine debugger, MacDB, both have symbolic capability.

We will use the following commands from DDT-68K:

- The Lxxxxx,yyyyy command. This causes memory locations xxxxx through yyyyy to be displayed as 68000 instruction mnemonics.
- The Dxxxxx,yyyyy command. This causes memory locations xxxxx through yyyyy to be displayed in hex.
- The Sxxxxx command. This causes memory location xxxxx to be displayed for possible modification. Pressing Return causes the next location to be displayed. Typing a period causes the debugger to prompt for the next command.
- The G,xxxxx command. This starts program execution at the location contained in the program counter and stops program execution immediately before the instruction at xxxxx.
- The T command. The next instruction is executed, using the Trace bit hardware mechanism.

DDT-68K Example

Suppose that we wish to execute the source file shown in Listing 3.3 under CP/M-68K. This is the same program as the Macintosh example.

```
        .text
*************************************
*       This program adds the first
*       five integers and stores the
*       result in memory.
*************************************
start:  move.w  a,d0    Load first number
        add.w   b,d0
        add.w   c,d0
        add.w   d,d0
        add.w   e,d0
        move.w  d0,f    Store answer
        rts             Return to CP/M
        .data
a:      dc.w    1       Numbers to add
b:      dc.w    2
c:      dc.w    3
d:      dc.w    4
e:      dc.w    5
f:      dc.w    0       Answer goes here
        .end
```

Listing 3.3 *– The Sum5 source file for CP/M-68K*

Note that the LEA instruction is not required. The CP/M-68K assembly listing file is shown in Listing 3.4.

The first number on each line is the number of the line in the source file, starting with 1. The assembler reports errors by line number.

The second number is the hex offset at which the present line is assembled. When added to the load address of the text (instructions), data (initialized data), or bss (uninitialized data) segment as appropriate, this number yields the absolute memory address of the instruction or data described by the line.

The third number in each line is the actual hex contents of the memory location when the program is loaded into memory. Addresses may not be relocated to their final value until load time. The assembly listing reflects addresses as they are known to the assembler.

The linker relocates all addresses to the values that they will have at execution time. Later, when the program is loaded into memory, it may be relocated yet again if the base address of the program does not match the address to which it was linked. By the time the program begins execution, however, all addresses are absolute.

The source line as it appears in the file comes next. Source lines that begin with an asterisk (*) are placed in the listing file as is. Such a line is

```
 1 00000000                        .text
 2                         *************************************
 3                         *       This program adds the first
 4                         *       five integers and stores the
 5                         *       result in memory
 6                         *************************************
 7 00000000 303900000000   start:  move.w  a,d0   Load first number
 8 00000006 D07900000002           add.w   b,d0
 9 0000000C D07900000004           add.w   c,d0
10 00000012 D07900000006           add.w   d,d0
11 00000018 D07900000008           add.w   e,d0
12 0000001E 33C00000000A           move.w  d0,f   Store answer
13 00000024 4E75                   rts            Return to CP/M
14 00000000                        .data
15 00000000 0001           a:      dc.w    1      Numbers to add
16 00000002 0002           b:      dc.w    2
17 00000004 0003           c:      dc.w    3
18 00000006 0004           d:      dc.w    4
19 00000008 0005           e:      dc.w    5
20 0000000A 0000           f:      dc.w    0      Answer goes here
21 0000000C                        .end
```

Listing 3.4 *– The CP/M-68K Sum5 assembly listing file* called a comment line. Comments serve only to help the human reader understand the program.

Other lines consist of an optional label, an opcode or directive, and one or more operands. Additional text on the line following the operands is regarded as comments by the assembler. At least one space is required between the end of the last operand and the beginning of the comment. The leading asterisk is not required for a comment at the end of a line.

A label on a line establishes a symbolic name for a memory location. A label must be the first word on a line, and must be terminated by a colon (or by a space or tab if the label starts in column 1). The label may then be referred to by instructions that reference memory. The memory locations a: through f: in Listing 3.4 illustrate this usage. These labels are tags for the memory locations referenced by the program's MOVE and ADD instructions.

The linker produces an executable file called test.68k. To run the program under the debugger, we type:

```
A>ddt test.68k

DDT-68K
Copyright 1982, Digital Research

text base     = 00000500  data base     = 00000526  bss base    = 00000532
text length   = 00000026  data length   = 0000000C  bss length  = 00000000
base page address = 00000400    initial stack pointer = 0001A2B8
```

This information indicates that the program is loaded into memory starting at 500 hex (text base). The data section of the program begins at address 526 hex (data base). From the assembly listing, we can see that the text portion is 26 bytes long. We can now use the debugger to list out the program in 68000 mnemonics with the l (el) command:

```
-l500,524
00000500 move $526,D0
00000506 add $528,D0
0000050C add $52A,D0
00000512 add $52C,D0
00000518 add $52E,D0
0000051E move D0,$530
00000524 rts
```

The data area is between addresses 526 and 532. The data base printed out by DDT (526) and data length (0c), when added together, produce the first address beyond the data area (532). We can display the data area with the Display Words (dw) command:

```
-dw526,532
00000526 0001 0002 0003 0004 0005 0000 ............
```

The Trace (t) command is used to execute each instruction. DDT shows the registers *before* the instruction is executed, as shown below:

```
PC=00000500 USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 000000D0 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
move $526,D0
-t
PC=00000506 USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 00000001 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
add $528,D0
-t
PC=0000050C USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 00000003 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
add $52A,D0
-t
PC=00000512 USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 00000006 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
add $52C,D0
-t
PC=00000518 USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 0000000A 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
add $52E,D0
-t
PC=0000051E USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 0000000F 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
move D0,$530
-t
PC=00000524 USP=0001A2B0 SSP=00002000 ST=0000=>IM=0
D 0000000F 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
A 000000A0 000000A1 000000A2 000000A3 000000A4 000000A5 000000A6 0001A2B0
rts
```

Notice how the value of register D0 changes as each number is added. To make the actions of the instructions easier to identify, we will underline registers that change in presenting examples. Just before returning to

CP/M, look at the answer in memory. (Location 530 hex corresponds to the label f: in the assembly listing.)

```
-dw530,532
00000530 000F
-g
A>
```

INSTRUCTIONS

The rest of this chapter is devoted to presenting the details of the Motorola 68000 instruction set. For ease of reference, the instructions are listed in alphabetical order. For each instruction, the following items are provided:

- A verbal description of what the instruction does.
- Which addressing modes are allowed. (Very few of the instructions allow all addressing modes.)
- What data sizes (byte, word, long) are allowed.
- Condition codes affected by executing the instruction.
- The layout of the machine code.
- Where possible, an example of how this instruction might be used. We will use the DDT-68K debugger to illustrate the results of executing the instruction.

Effective Address Operands

Most instructions that reference memory do so by means of an effective address operand. Effective address operands consist of a 3-bit mode field and a 3-bit register field. These operands are discussed in detail in Chapter 2. The notation used in the instruction descriptions for an effective address operand is <ea>. For each effective address operand in an instruction, the instruction discussion will present a table of addressing modes that are permitted with the operand.

ABCD Instruction

The ABCD (Add BCD with extend) instruction adds two bytes in BCD (Binary Coded Decimal) format. The destination operand is replaced with the sum of the source and destination bytes.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	No	No	Yes	No	No
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
No	No	No	No	No	No	No

There are only two forms of this instruction:

1. Add data register to data register (Dn addressing modes). The low-order bytes of two data registers are added and the result stored in the destination register.
2. Add memory to memory. This form of the instruction is designed for adding multiple bytes in memory. The only valid addressing mode is −(An). Since the 68000 stores BCD data with the most significant byte first, one must start at the highest address (where the least significant byte is located) and work down to add multi-byte quantities. (Hence the use of pre-decrement addressing.) Each instruction sets the X-bit if there was a carry out of the most significant BCD digit in the byte. Then the X-bit is added into the next byte.

Data Sizes: Byte only

Condition Codes Affected:

X Set by carry out of the most significant BCD digit.
N Undefined.
Z Cleared if the result is not zero. Unchanged otherwise.

V Undefined.
C Set by carry out of the most significant BCD digit.

The Z-bit is cleared if the result was not zero. Not setting the bit when the result of the present byte is zero allows the Z-bit to be accurate after a series of ABCD instructions is executed. The Z-bit must be set initially in such a case. (Comparing a register to itself is an easy way to set the Z-bit.) The N and V bits are undefined as a result of this instruction.

Assembler Syntax: ABCD Dx,Dy
ABCD – (Ax), – (Ay)

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	D. Reg			1	0	0	0	0	F	S. Reg.		

The D. Reg and S. Reg fields specify the destination and source register numbers. If the F (format) bit is 0, then the registers are data registers. If the F bit is a 1, then the registers are address registers and the predecrement addressing mode is used.

Example:
This example adds two values in data registers:

```
PC=0000050C USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000099  00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
abcd.b D1,D0
-t
PC=0000050E USP=0001598C SSP=0007BF08 ST=0011=>IM=0 EXT CRY
D 00000000  00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

Since we added 1 to 99, the result is zero, with the EXT (extend) bit set in the status register. The next example adds two two-byte BCD numbers in memory. The addresses in registers A0 and A1 point to the ends of the BCD numbers.

```
PC=0000051E USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 0000082A  0000082C 00000000 00000000 00000000 00000000 00000000 0001598C
abcd.b -(A0),-(A1)
```

Show the operands before the first add:

```
-dw828,82c
00000828 0099 0001                                              ....
-t
PC=00000520 USP=0001598C SSP=0007BF08 ST=0011=>IM=0 EXT CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000829  0000082B 00000000 00000000 00000000 00000000 00000000 0001598C
abcd.b -(A0),-(A1)
```

Now look at the operands:

```
-dw828,82c
00000828 0099 0000                                              ....
-t
PC=00000522 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000828  0000082A 00000000 00000000 00000000 00000000 00000000 0001598C
```

Now look at the results:

```
-dw828,82c
00000828 0099 0100                                              ....
```

This example adds 99 to 1 to become 100. (The second operand is destroyed.) The memory displays show exactly what happens at each step of the process. Notice how the Extend bit gives the carry between the two add operations.

ADD Instruction

The ADD (Add binary) instruction adds two operands together and stores the result in the destination operand. There are two forms of this instruction:

1. Add an effective address operand to a data register.
2. Add a data register to an effective address operand.

Addressing Modes Allowed:
All addressing modes except SR and CCR are allowed when the effective address specifies a source operand:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

When the effective address field is the destination, then the following addressing modes are allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Using a data register as a destination requires the register destination form of the instruction.

Data Sizes: byte, word, long
Using an address register as the source is valid only for word and long data lengths.

Condition Codes Affected:

- X Set by the carry-out of the most significant bit. Cleared otherwise.
- N Set if high-order bit of result was 1. Cleared otherwise.
- Z Set if result was zero. Cleared otherwise.
- C Set by the carry-out of the most significant bit. Cleared otherwise.
- V Set if operation resulted in overflow condition. Cleared otherwise.

Assembler Syntax: ADD Dn,<ea>
ADD <ea>,Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	1	Register			D	Size		Effective			Address		
											← Mode →			← Reg. →		

The Register field gives the data register that must be one of the operands. The D-bit is 0 if the Register field is the destination operand. The D-bit is 1 if the effective address is the destination.

The Size field is 00 for byte, 01 for word, and 10 for long operands.

Example:

```
PC=00000530 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 0000FFFF  0000FFFF 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
add $82C,D0
```

Examine the Memory Operand:

```
-dw82c,82e
0000082C 0001                                                     ..
-t
PC=00000536 USP=0001598C SSP=0007BF08 ST=0015=>IM=0 EXT ZER CRY
D 00000000  0000FFFF 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

Notice that the result of adding −1 and 1 is zero. (This was a word operation.) The Z-bit was set by this operation. We will now add the same two numbers with an ADD.L instruction.

```
add.l $82E,D1
```

Examine the Memory Operand:

```
-dl82e,832
0000082E 00000001   ..
-t
PC=0000053C USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00010000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

ADDA Instruction

The ADDA instruction does a binary ADD operation to an address register. In order to allow address computations to be freely intermixed with data operations, this instruction does not affect the condition codes.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

The effective address must be the source operand.

Data Sizes: word, long

The ADDA operation always affects all 32 bits of the destination address register.

Condition Codes Affected: None

Assembler Syntax: ADDA <ea>,An

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	1	Register			S	1	1	Effective			Address		
											← Mode →			← Reg. →		

The Register field gives the address register that is to be used as the destination operand. The S-bit is 1 for long operands and 0 for word operands.

Example:

```
PC=00000548 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 0000FFFF  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
adda $830,A0
-dw830,832
00000830 0001
-t
PC=0000054E USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00010000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

Notice that the operation size was a word, but the result was a long.

ADDI Instruction

The ADDI instruction adds a constant to an effective address operand. The source operand is always immediate.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Set on carry out of high-order bit.
N Set if high bit of result is set.
Z Set if result is zero.
C Set on carry out of high-order bit.
V Set on overflow condition.

Assembler Syntax: ADDI #x,<ea>

Most assemblers automatically choose the ADDI instruction if the source operand of an ADD instruction is immediate (and not 1 to 8). Therefore, ADDI need not be used; simply use ADD.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	1	1	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Word data (16 bits)	Byte data (8 bits)

Long data (32 bits, including previous word)

Size is 00 for byte operations, 01 for word operations, and 10 for long operations.

Example:

```
PC=00000560 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00007FFF  0000FFFF  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
addi.l #$10,D0
-t
PC=00000566 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 0000800F  0000FFFF  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This instruction adds 10 hex to the contents of register D0.L.

```
-t
PC=0000056A USP=0001598C SSP=0007BF08 ST=000F=>IM=0 NEG ZER OFL CRY
D 0000800F  0000FFFF  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
addi #$10,D1
-t
PC=0000056E USP=0001598C SSP=0007BF08 ST=0011=>IM=0 EXT CRY
D 0000800F  0000000F  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This instruction adds 10 hex to register D1.W. (Note that $-1 + 10 = F$.)

ADDQ Instruction

The ADDQ instruction adds a three-bit immediate value to an effective address operand. This allows adding a small number to a register or memory address using a small, fast instruction.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

When an address register is used as the destination, only word and long sizes are allowed.

Condition Codes Affected:

X Set on carry-out of high-order bit position.
N Set if high-order bit of result is set.
V Set on overflow.
Z Set if result is zero.
C Set on carry-out of high-order bit position.

No condition codes are affected if an address register is used as the destination operand.

Assembler Syntax: ADDQ #<data>,<ea>
#<data> is a constant number between 1 and 8.

Most assemblers automatically choose the ADDQ instruction if the source operand of an ADD instruction is an immediate value from 1 to 8.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	1	Data			0	Size		Effective			Address		
											← Mode →			← Reg. →		

Data is a three-bit immediate field, with 000 representing 8, 001–111 representing 1–7. Size is 00 for a byte operation, 01 for a word, and 10 for a long operation.

Example:

```
PC=00000580 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00007FFF  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
addq.l #$1,D0
-t
PC=00000582 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00008000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example adds 1 to register D0.

ADDX Instruction

The ADDX (ADD eXtended) instruction provides multiple precision ADD operands. Integers of any length can be added using the ADD and ADDX instructions. This makes it possible to represent numbers much larger than the 32-bit longword allows. The X (extended) condition code bit serves as a carry across the 32-bit boundary.

There are two forms of this instruction:

1. Add a data register to a data register.
2. Add a memory location to a memory location. The −(An) addressing mode must be used for both the source and destination in this form of the instruction.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	No	No	Yes	No	No
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
No	No	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Set on carry-out of high-order bit.
N Set if result was negative.
Z Cleared if result is not zero. Unchanged otherwise.
C Set on carry-out of high-order bit.
V Set on overflow condition.

The Z-bit is not set if the result was zero. It is cleared if the result was not zero. This property of the instruction allows the Z bit to correctly indicate the result of a multiprecision addition operation. The Z-bit must be set before the ADD begins, however. (This can be done with a MOVE to

CCR, or by comparing a register to itself. The latter instruction is two bytes shorter.)

Assembler Syntax: ADDX Dy,Dx
ADDX −(Ay),−(Ax)

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	1	Reg. Rx			1	Size		0	0	T	Reg. Ry		

The Reg. Rx and Reg. Ry fields contain the destination and source register numbers, respectively. The size field is 00 for a byte operation, 01 for a word operation, and 10 for a long operation. The T-bit (type) is 0 for the data register to data register form of the instruction. The Reg. Rx and Reg. Ry fields identify data registers in this case. The T-bit is 1 for the memory to memory form of the instruction. The Rx and Ry fields identify the address registers used by this form of the instruction.

Example:

This example adds the 64-bit quantity in (D0,D1) to the 64-bit quantity in (D2,D3). The even numbered registers contain the high order part of the number.

```
PC=00000596 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  FFFFFFFF 00000000 FFFFFFFF 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
add.l D1,D3
-t
PC=00000598 USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 00000000  FFFFFFFF 00000000 FFFFFFFE 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
addx.l D0,D2
-t
PC=0000059A USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  FFFFFFFF 00000001 FFFFFFFE 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

The quantities (0,FFFFFFFF) and (0,FFFFFFFF) when added together become (1,FFFFFFFE). Note the action of the X (extended) condition code bit. The low-order registers (D1 and D3) were added first. Although the low-order registers both contained negative numbers, the final result was positive.

AND Instruction

The AND instruction performs a bit-wise AND operation. There are two forms of this instruction:

1. AND the contents of an effective address with a data register, leaving the results in the data register.
2. AND the contents of a data register and an effective address, leaving the results in the effective address.

Addressing Modes Allowed:

Effective address as Source:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Effective address as Destination:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

- X Not affected.
- N Set if most significant bit of result is set. Cleared otherwise.
- Z Set if result is zero. Cleared otherwise.
- C Always cleared.
- V Always cleared.

Assembler Syntax: AND <ea>,Dn
AND Dn,<ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	Register			D	Size		Effective Address					
											← Mode →			← Reg. →		

The Register field specifies the Data register used by the instruction, regardless of whether the register is the source or the destination. The D-bit determines the direction of the instruction. If the D-bit is zero, then the Data register is the destination. If the D-bit is one, then the effective address operand is the destination. The Size field specifies the data size: 00 for byte, 01 for word, and 10 for long.

Example:

This example shows the machine operation when two data registers are ANDed together.

```
PC=000005AC USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D AAAAAAAA 01234567 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
and.l D1,D0
-t
PC=000005AE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00220022 01234567 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

The long quantities AAAAAAAA and 01234567 are ANDed to become 00220022.

ANDI Instruction

The ANDI instruction performs a bit-wise AND between an immediate operand (always the source) and an effective address operand (always the destination).

Addressing Modes Allowed:

Destination only. Source is always immediate.

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	Yes	Yes

Using the status register (SR) as the destination requires that the 68000 executes in Supervisor state. Attempting to execute this form of the instruction in User mode causes a privilege violation exception. (See Chapter 7 on Exception Programming.)

Data Sizes: byte, word, long

Condition Codes Affected:

X Not affected.
N Set if the high-order bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
C Always cleared.
V Always cleared.

The condition codes are cleared according to bits 4–0 of the operand if either the status register (SR) or the condition code register (CCR) is used as the destination. The normal condition code settings do not apply for these addressing modes.

Assembler Syntax: ANDI #<data>,<ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	1	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Word data: 16 bits	Byte Data: 8 bits

Long data: 16 bits plus previous word

The Size field determines the data size used by the instruction. Size is 00 for byte operation, 01 for word, and 10 for long. Byte and word operations are followed by a word of immediate data. (Byte operations use only the low eight bits of this word.) Long operations are followed by two words (32 bits) of immediate data.

Examples:

The first example ANDs an immediate quantity with a data register.

```
PC=000005BA USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 55555555  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
andi #$1234,D0
PC=000005BE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 55551014  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

The next example shows the effect of ANDing to the condition code register (CCR). First, we set all the condition codes using the MOVE to CCR instruction:

```
PC=000005BE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 55551014  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
move #$1F,CCR
-t
```

```
PC=0005C2 USP=01598C SSP=07BF08 ST=001F=>IM=0 EXT NEG ZER OFL CRY
D 55551014  00000000 00000000 00000000 00000000 00000000 00000000 00000000
 A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

Next, we AND the condition code register with 11 hex. This leaves the X and C bits set, and clears the other condition codes.

```
andi.b #$11,ccr
 -t
PC=000005C6 USP=0001598C SSP=0007BF08 ST=0011=>IM=0 EXT CRY
D 55551014  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

ASL Instruction

The ASL instruction performs an arithmetic left shift on a data register or memory operand. There are three forms of this instruction:

1. Shift a data register to the left by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Shift a data register to the left by the number of bits contained in another data register.
3. Shift a memory word left by one bit only.

Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Data size is restricted to word for the form of the instruction that is in memory.

Condition Codes Affected:

X Set according to the last bit shifted out of the operand. Unaffected if the shift count is zero. (This is possible only in the second form of the instruction.)
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
C Set according to the last bit shifted out of the operand. Unaffected if the shift count is zero.
V Set if the most significant bit is changed at any time during the shift operation. Cleared otherwise.

Assembler Syntax: ASL #<count>,Dy
ASL Dx,Dy
ASL <ea>

Machine Code Format:

Data Register as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			1	Size		T	0	0	Register		

Memory location as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	0	0	1	1	1	Effective			Address		
											← Mode →			← Reg. →		

The T-field determines the form of the register-destination form of the instruction. If T is 0, then the Immediate field contains the shift count, with 000 binary representing a count of 8. If T is 1, then the register number that contains the shift count is contained in the Immediate field.

Example:

```
PC=000005D2 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 01234567  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
asl #8,D0
-t
PC=000005D4 USP=0001598C SSP=0007BF08 ST=0013=>IM=0 EXT OFL CRY
D 01236700  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

Notice that the upper half of the register destination is unchanged when a word shift is performed. The sign bit of the word is also changed during the shift, resulting in an overflow condition.

```
PC=000005E0 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 01234567  00000010 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
asl.l D1,D0
-t
PC=000005E2 USP=0001598C SSP=0007BF08 ST=0013=>IM=0 EXT OFL CRY
D 45670000  00000010 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This is a long shift with the count specified in a register.

ASR Instruction

The ASR instruction performs an arithmetic right shift on a data register or memory operand. There are three forms of this instruction:

1. Shift a data register to the right by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Shift a data register to the right by the number of bits contained in another data register.
3. Shift a memory word right by one bit only.

Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An)+	–(An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long
Data size is restricted to word for the form of the instruction that is in memory.

Condition Codes Affected:

X Set according to the last bit shifted out of the operand. Unaffected if the shift count is zero. (That is possible only in the second form of the instruction.)
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
C Set according to the last bit shifted out of the operand. Cleared if the shift count is zero.
V Always cleared.

Assembler Syntax: ASR #<count>,Dy
ASR Dx,Dy
ASR <ea>

Machine Code Format:

Data Register as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			0	Size		T	0	0	Register		

Memory location as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	0	0	0	1	1	Effective Address					
											← Mode →			← Reg. →		

The T-field determines the form of the register-destination form of the instruction. If T is 0, then the Immediate field contains the shift count, with 000 binary representing a count of 8. If T is 1, then the register number which contains the shift count is contained in the Immediate field.

Example:

```
PC=000005EE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 01234567  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
asr #8,D0
-t
PC=000005F0 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 01230045  00000000  00000000  00000000  00000000  00000000  00000000  00000000
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This instruction shifts register D0 to the right by 8. The upper half of the register is unaffected because the operation is a word operation.

```
PC=000005FC USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 81234567  00000010  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
asr.l D1,D0
```

```
-t
PC=000005FE USP=0001598C SSP=0007BF08 ST=0008=>IM=0 NEG
D FFFF8123  00000010 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

Notice the sign extension due to the upper bit of register D0 being propagated by the right shift.

Bcc (Conditional Branch) Instructions

Branch instructions are a vital part of machine-language programs. They provide a way of interrogating the condition codes and executing two or more alternate sets of instructions based on the results of the interrogation. The Bcc group of instructions is the method by which the 68000 performs this function. There are fifteen combinations of the condition codes that may be tested. Each of these has a two-letter mnemonic that takes the place of the cc in Bcc.

The permissible branch instructions are:

BCC Branch if the C-bit (carry) is clear.

BCS Branch if the C-bit (carry) is set.

BEQ Branch on EQual. This instruction branches if the Z (Zero) bit is set.

BGE Branch on Greater than or Equal. This instruction branches if the N (negative) and V (overflow) bits are either both set or both clear. BGE is used for two's complement binary numbers.

BGT Branch on Greater Than. This instruction branches if
- The N and V bits are both set AND the Z-bit is clear, or,
- The N, V, and Z bits are *all* clear.

BHI Branch on HIgher than. This instruction branches if the C and Z bits are both clear. BHI is similar to BGT, except it is used for unsigned numbers.

BLE Branch on Less than or Equal. This instruction branches if
- The Z-bit is set, or,
- The N-bit is set AND the V-bit is clear, or,
- The N-bit is clear AND the V-bit is set.

The BLE instruction is used for two's complement binary numbers.

BLS Branch on Lower or Same. This instruction branches if either the C or Z bits are set. BLS is similar to BLE, except it is used for unsigned numbers.

BLT Branch on Less Than. This instruction branches if
- The N-bit is set and the V-bit is clear, or,
- The N-bit is clear and the V-bit is set.

BMI This instruction branches if the N-bit is set.

BNE Branch on Not Equal. This instruction branches if the Z-bit is clear.

BPL Branch on Plus. This instruction branches if the N-bit is clear.

BVC Branch on V Clear. This instruction branches if the V-bit is clear, indicating no overflow.

BVS Branch on V Set. This instruction branches if the V-bit is set, indicating overflow.

BRA Branch Always. This instruction always branches, regardless of the setting of the condition codes.

The Condition Code Summary is shown in Table 3.1.

Condition Codes				Branch Instructions That Succeed														
N	Z	V	C	BRA	BHI	BLS	BCC	BCS	BNE	BEQ	BVC	BVS	BPL	BMI	BGE	BLT	BGT	BLE
0	0	0	0	x	x		x		x		x		x		x		x	
0	0	0	1	x		x		x	x		x		x		x		x	
0	0	1	0	x	x		x		x			x	x			x		x
0	0	1	1	x		x		x	x			x	x			x		x
0	1	0	0	x		x	x			x	x		x		x			x
0	1	0	1	x		x		x		x	x		x		x			x
0	1	1	0	x		x	x			x		x	x			x		x
0	1	1	1	x		x		x		x		x	x			x		x
1	0	0	0	x	x		x		x		x			x		x		x
1	0	0	1	x		x		x	x		x			x		x		x
1	0	1	0	x	x		x		x			x		x	x		x	
1	0	1	1	x		x		x	x			x		x	x		x	
1	1	0	0	x		x	x			x	x			x		x		x
1	1	0	1	x		x		x		x	x			x		x		x
1	1	1	0	x		x	x			x		x		x	x			x
1	1	1	1	x		x		x		x		x		x	x			x

Table 3.1 - Condition Code Summary

Addressing Modes Allowed:

There is a special addressing mode for branch instructions. Branches can either have a byte or word displacement that is sign extended to a long and added to the PC (Program Counter) to perform the branch. (This is done only if the condition is satisfied.)

Data Sizes: byte, word

Condition Codes Affected: None

Assembler Syntax: Bcc.S <label> Byte displacement
Bcc.W <label> Word displacement

<label> is a label contained in the instruction area of the program (.text area on UNIX and CP/M-68K). The .S and .W suffixes are used to denote the two possible displacement sizes (short and word). Many assemblers perform this selection automatically.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	1	0	Condition				8-bit displacement							
	16-bit displacement if 8-bit displacement = 0															

The Condition is a four-bit encoding of the branch condition. The conditions are shown in Table 3.2.

The combination 0001 for condition is used to denote the BSR (Branch to Subroutine) instruction, rather than a conditional branch.

The 8-bit displacement field is an 8-bit field that gives a two's complement number to be added to the PC if the branch is successful. *The PC always* contains the address of the word that follows the first word of the branch instruction. If the 8-bit displacement is zero, then the word following the branch instruction contains a 16-bit displacement to be added to the PC.

One-word (8-bit) displacements give a branch range of -128 to $+126$ bytes away from the branch instruction. With a 16-bit displacement, this range is expanded to -32768 to $+32766$ bytes. The displacement must always be even. (Instructions must begin on a word boundary.)

Condition	Instruction	Condition	Instruction
0000	BRA	1000	BVC
0001	(NONE)	1001	BVS
0010	BHI	1010	BPL
0011	BLS	1011	BMI
0100	BCC	1100	BGE
0101	BCS	1101	BLT
0110	BNE	1110	BGT
0111	BEQ	1111	BLE

Table 3.2 *- Condition and Corresponding Instruction*

A one-word (8-bit displacement) branch to the next instruction is impossible. The displacement would have to be zero, and the next word (i.e., the first word of the next instruction) would be taken as a 16-bit branch offset. The Bcc instruction cannot be configured to yield a one word *no operation* (an instruction that does nothing).

Example:

```
PC=00000606 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
beq $60A
-l606,60e
00000606 beq $60A
00000608 moveq #$FF,D0
0000060A bne $60E
0000060C moveq #$FE,D0
0000060E nop
```

In this segment of sample code, there are two conditional branches, a BNE and a BEQ. The BEQ will not be taken, since the Z-bit is not set at the beginning. Thus, the first MOVEQ instruction will be executed. The Z-bit will remain clear as a result of this instruction. The BNE will be taken, and register D0 will have a value of FFFFFFFF.

```
-t
PC=00000608 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
moveq #$FF,D0
-t
PC=0000060A USP=0001598C SSP=0007BF08 ST=0008=>IM=0 NEG
D FFFFFFFF  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bne $60E
-t
PC=0000060E USP=0001598C SSP=0007BF08 ST=0008=>IM=0 NEG
D FFFFFFFF  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

The next example illustrates an anomaly of the 68000 instruction set. The CMP instruction is used to set the condition codes as if the two operands were subtracted. The operands remain unaffected. However, the comparison CMP D0,D1 followed by a BGT instruction will branch if *register D1 is greater than register D0* (i.e., the operands of the compare instruction must be read *backwards*).

```
PC=00000616 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D FFFFFFFF  00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
cmp.l D0,D1
-l616,620
00000616 cmp.l D0,D1
00000618 bgt $61C
0000061A addq #$2,D2
0000061C bhi $620
0000061E addq #$1,D2
00000620 nop
```

In this code segment, D0 (−1) and D1 (1) are compared. If D1 is greater than D0 (and it is), the first ADDQ instruction will be skipped. The BHI instruction is the unsigned version of a BGT. In this case, −1 (actually $2^{32} - 1$, or about 4 billion) is greater than 1, so the branch will not be taken. (Since we skipped the first ADDQ instruction, the condition codes at the time of executing the BHI are the result of the CMP.L instruction.)

Register D2 will have a value of 1.

```
-t
PC=00000618 USP=0001598C SSP=0007BF08 ST=0001=>IM=0 CRY
D FFFFFFFF 00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bgt $61C
-t
PC=0000061C USP=0001598C SSP=0007BF08 ST=0001=>IM=0 CRY
D FFFFFFFF 00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bhi $620
-t
PC=0000061E USP=0001598C SSP=0007BF08 ST=0001=>IM=0 CRY
D FFFFFFFF 00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
addq #$1,D2
-t
PC=00000620 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D FFFFFFFF 00000001 00000001 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

BCHG Instruction

The BCHG (test a Bit and CHanGe) instruction inverts a single bit in an effective address operand. The Z-bit is set according to the state of the bit *before* the inversion.

The bit number is contained either in a register or in an immediate field inside the instruction itself. The operation is restricted to long data for a data register destination, and to byte data for memory locations. Bits are numbered from 0, with bit 0 being the least significant bit in a byte (or long).

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An)+**	**–(An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, long

Condition Codes Affected:

X Not affected.

N Not affected.

Z Set if the bit was zero *before* being inverted. Cleared otherwise.

C Not affected.

V Not affected.

Assembler Syntax: BCHG Dn,<ea>
BGHG #<data>,<ea>

Machine Code Format:
Bit Number Dynamic:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	Register			1	0	1	Effective			Address		
											← Mode →			← Reg. →		

Bit Number Static:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	0	0	0	0	1	Effective			Address		
											← Mode →			← Reg. →		

0	0	0	0	0	0	0	0	////////			Bit Number				

The Register field indicates the data register in which the bit number resides for the Bit Number Dynamic form of the instruction. In the Bit Number Static case, bits 5–7 of the extension word are ignored for data register destinations. For memory locations, bits 4 and 3 are also ignored. (Bit numbers in a long range from 0–31, and in a byte from 0–7.)

Example:

```
PC=00000628 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bchg #$0,D0
-t
PC=0000062C USP=0001598C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example inverts bit 0 of register D0. The Z-bit is set *after* the instruction, indicating that the bit was clear *before* the instruction.

BCLR Instruction

The BCLR (test a Bit and CLeaR) instruction clears a single bit in an effective address operand. The Z-bit is set according to the state of the bit *before* the instruction.

The bit number is either contained in a register, or in an immediate field inside the instruction itself. The operation is restricted to long data for a data register destination, and to byte data for memory locations. Bits are numbered from 0, with bit 0 being the least significant bit in a byte (or long).

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, long

Condition Codes Affected:

X Not affected.

N Not affected.

Z Set if the bit was zero *before* being cleared. Cleared otherwise.

C Not affected.

V Not affected.

Assembler Syntax: BCLR Dn,<ea>
BCLR #<data>,<ea>

Machine Code Format:
Bit Number Dynamic:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	Register			1	1	0	Effective			Address		
											← Mode →			← Reg. →		

Bit Number Static:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	0	0	0	1	0	Effective			Address		
											← Mode →			← Reg. →		

0	0	0	0	0	0	0	0	////////			Bit Number				

The Register field indicates the data register in which the bit number resides for the Bit Number Dynamic form of the instruction. In the Bit Number Static case, bits 5–7 of the extension word are ignored for data register destinations. For memory locations, bits 4 and 3 are also ignored. (Bit numbers in a long range from 0–31, and in a byte from 0–7.)

Example:

```
PC=00000632 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bclr #$0,D0
-t
PC=00000636 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This instruction clears bit 0 of register D0. The Z-bit is not set *after* the instruction, indicating that the bit was set *before* the instruction.

BSET Instruction

The BSET (test a Bit and SET) instruction sets a single bit in an effective address operand. The Z-bit is set according to the state of the bit *before* the instruction.

The bit number is either contained in a register or in an immediate field inside the instruction itself. The operation is restricted to long data for a data register destination, and to byte data for memory locations. Bits are numbered from 0, with bit 0 being the least significant bit in a byte (or long).

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An) +**	**− (An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, long

Condition Codes Affected:

X Not affected.

N Not affected.

Z Set if the bit was zero *before* being set. Cleared otherwise.

C Not affected.

V Not affected.

Assembler Syntax: BSET Dn,<ea>
BSET #<data>,<ea>

Machine Code Format:
Bit Number Dynamic:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	Register			1	1	1	Effective			Address		
											← Mode →			← Reg. →		

Bit Number Static:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	0	0	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

0	0	0	0	0	0	0	0	////////			Bit Number				

The Register field indicates the data register in which the bit number resides for the Bit Number Dynamic form of the instruction. In the Bit Number Static case, bits 5–7 of the extension word are ignored for data register destinations. For memory locations, bits 4 and 3 are also ignored. (Bit numbers in a long range from 0–31, and in a byte from 0–7.)

Example:

```
PC=00000644 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bset #$0,D0
-t
PC=00000648 USP=0001598C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This instruction sets bit 0 of register D0. The Z-bit is set *after* the instruction, indicating that the bit was clear *before* the instruction.

BSR Instruction

The BSR (Branch to SubRoutine) instruction places the address of the next instruction to be executed on top of the stack. A displacement is then added to the PC register (Program Counter), and execution continues at that address. For the purposes of the addition, the PC points to the word that follows the first word of the BSR instruction.

Addressing Modes Allowed:

There is a special addressing mode for branch instructions. Branch instructions can either have a byte or word displacement that is sign extended to a long and added to the PC (Program Counter) to perform the branch.

Data Sizes: byte, word

Condition Codes Affected: None

Assembler Syntax: BSR.S <label> Byte displacement
BSR.W <label> Word displacement

<label> is a label contained in the instruction area of the program (.text area on UNIX and CP/M). The .S and .W suffixes are used to denote the two possible displacement sizes (short and word). Many assemblers perform this selection automatically.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	1	0	0	0	0	1	8-bit displacement							

16-bit displacement if 8-bit displacement = 0

The format of a BSR instruction is the same as that of the Bcc instructions, with a condition field (bits 11–8) of 0001.

As with the Bcc instructions, it is not possible to have a one-word BSR to the next instruction, since the 8-bit displacement in that case would be zero. The next word (which would be the first word of the next instruction) would erroneously be used as the 16-bit displacement in this case. Most assemblers will not allow this error condition to take place.

Example:

```
PC=0000064E USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
bsr $652
-t
PC=00000652 USP=00015988 SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00015988
rts
-dl15988,1598c
00015988 00000650
```

The BSR instruction at location 64E branches to location 652, after placing 650 (the address of the next instruction) on the stack (at location 15988).

```
-t
PC=00000650 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

Following the RTS instruction, execution picks up at location 650, and the return address has been popped off the stack. (Register A7 now contains 1598C—four bytes have been popped.)

BTST Instruction

The BTST (Bit TeST) instruction tests a single bit in an effective address operand. The Z-bit is set according to the state of the bit.

The bit number is either contained in a register or in an immediate field inside the instruction itself. The operation is restricted to long data for a data register destination, and to byte data for memory locations. Bits are numbered from 0, with bit 0 being the least significant bit in a byte (or long).

Addressing Modes Allowed (Bit Number Static):

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	No	No	No

Addressing Modes Allowed (Bit Number Dynamic):

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	NO

Data Sizes: byte, long

Condition Codes Affected:

X Not affected.
N Not affected.

Z Set if the tested bit is zero. Cleared otherwise.
C Not affected.
V Not affected.

Assembler Syntax: BTST Dn,<ea>
BTST #<data>,<ea>

Machine Code Format:

Bit Number Dynamic:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	Register			1	0	0	Effective			Address		
											← Mode →			← Reg. →		

Bit Number Static:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	0	0	0	0	0	Effective			Address		
											← Mode →			← Reg. →		

0	0	0	0	0	0	0	0	////////			Bit Number				

The Register field indicates the data register in which the bit number resides for the Bit Number Dynamic form of the instruction. In the Bit Number Static case, bits 5–7 of the extension word are ignored for data register destinations. For memory locations, bits 4 and 3 are also ignored. (Bit numbers in a long range from 0–31, and in a byte from 0–7.)

Example:

```
PC=00000658 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
btst #$0,D0
```

```
.

-t
PC=0000065C USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example tests the low-order bit of register D0. The bit is set, so that the Z-bit is not set by the instruction. A BNE instruction will branch in this case.

CHK Instruction

The CHK (CHecK register against bounds) instruction verifies that a data register contains a number within a certain positive range of values. This instruction is normally used by high-level language systems for range checking subscripts.

The low-order word (16 bits) of the data register is compared to an operand specified by the effective address field of the instruction. If the register is less than zero (i.e., bit 15 of the register is set) or greater than the upper bound, then a CHK exception results. (See Chapter 7 on Exception Processing for details.)

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Sizes: word

Condition Codes Affected:

X Not affected.

N Set if the data register is less than zero. Cleared if the data register is greater than the effective address operand. Undefined otherwise.

Z Undefined.

C Undefined.

V Undefined.

Assembler Syntax: CHK <ea>,Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	Register			1	1	0	Effective			Address		
											← Mode →			← Reg. →		

Register is the number of the data register to be tested.

Example:

```
PC=00000660 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
chk #$1,D0
-t
PC=00000664 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
chk #$0,D0
-t
Exception $06 at user address 00000668. Aborted.
```

The first CHK instruction in this example does nothing, as the register is equal to the limit (both are 1). In the second example, an exception is generated. The message printed here shows how CP/M-68K treats exceptions in a user program. Other operating systems behave in a similar fashion. Exception 6 is the CHK instruction. (See Chapter 7 on Exception Processing for details.)

CLR Instruction

The CLR (CLeaR) instruction sets an effective address operand to zero. An anomaly of the 68000 hardware causes memory operands to be *read* and then cleared. This usually makes no difference in program behavior, with two exceptions of initializing some memory units with parity (which may give an erroneous parity error), or accessing memory-mapped hardware.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	–(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

- X Not affected.
- N Always cleared.
- Z Always set.
- C Always cleared.
- V Always cleared.

Assembler Syntax: CLR <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	1	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Size is 00 for byte data, 01 for word, and 10 for long.

Example:

```
PC=0000066A USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
clr.l D0
-t
PC=0000066C USP=0001598C SSP=000FBF08 ST=0004=>IM=0 ZER
D 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example clears data register D0.

CMP Instruction

The CMP (CoMPare) instruction compares the contents of a data register with an effective address operand. The condition codes are set as if the effective address were subtracted from the data register. Neither operand is altered.

When used with conditional branches, this instruction creates a less than desirable effect. When the combination

```
CMP     D0,D1
BGT     X1
```

is used, the branch takes place if register D1 is greater than register D0.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	–(An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Sizes: byte, word, long

Byte mode is not allowed when address register direct mode is used as the effective address.

Condition Codes Affected:

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. (The operands are equal.) Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.
V Set on overflow in the subtract operation. Cleared otherwise.

Assembler Syntax: CMP <ea>,Dn

The CMPA instruction is used when the destination is an address register. CMPI compares an immediate source to an effective address operand. CMPM compares memory to memory. Many assemblers make this distinction automatically.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	1	1	Register			0	Size		Effective Address					
											← Mode →			← Reg. →		

The Register field is the number of the data register used as the destination. The size field is 00 for comparing bytes, 01 for words, and 10 for longs.

Example:

```
PC=00000674 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
cmp.l $932,D0
-sl932
00000932 00000001 .
-l674,684
   00000674 cmp.l $932,D0
   0000067A beq $67E
   0000067C addq.l #$1,D2
   0000067E cmp.l D0,D1
   00000680 ble $684
   00000682 addq #$2,D2
   00000684 nop
```

This program segment compares register D0 to a memory word containing 1. The BEQ instruction will branch, and the first ADDQ instruction will not be executed. The BLE instruction will not branch. (Remember that the compare operands must be read backwards.) Register D2 will have a value of 2.

```
-t
PC=0000067A USP=0001598C SSP=000FBF08 ST=0004=>IM=0 ZER
D 00000001  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
beq $67E
```

```
-t
PC=0000067E USP=0001598C SSP=000FBF08 ST=0004=>IM=0 ZER
D 00000001 00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
cmp.l D0,D1
-t
PC=00000680 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001 00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
ble $684
-t
PC=00000682 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001 00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
addq #$2,D2
-t
PC=00000684 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001 00000002 00000002 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

CMPA Instruction

The CMPA (CoMPare Address) instruction compares the contents of an address register with an effective address operand. The condition codes are set as if the effective address were subtracted from the address register. Neither operand is altered.

When used with conditional branches, this instruction creates a less than desirable effect. When the combination

```
CMPA    A0,A1
BGT     X1
```

is used, the branch takes place if the value in register A1 is greater than the value in register A0.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An)+**	**−(An)**	**x(An)**	**x(An,xr.s)**
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Sizes: word, long

Condition Codes Affected:

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. (The operands are equal.) Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.
V Set on overflow in the subtract operation. Cleared otherwise.

Assembler Syntax: CMPA <ea>,An

The CMP instruction is used when the destination is a data register. CMPI compares an immediate source to an effective address operand. CMPM compares memory to memory. Many assemblers make this distinction automatically.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	1	1	Register			S	1	1	Effective			Address		
											← Mode →			← Reg. →		

The Register field is the number of the data register used as the destination. The S (Size) bit is 0 for comparing words, and 1 for longs. Comparing a word is performed by sign-extending the source and making a 32-bit compare.

Example:

```
PC=00000694 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000936 0000094E 00000000 00000000 00000000 00000000 00000000 0001598C
cmpa.l A0,A1
-l694,69a
  00000694 cmpa.l A0,A1
  00000696 bhi $69A
  00000698 clr.l D0
  0000069A nop
```

This code segment compares address registers A0 to A1. The BHI instruction will succeed. (Remember that the compare operands must be read backwards.) Data register D0 will remain unchanged.

```
-t
PC=00000696 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000936 0000094E 00000000 00000000 00000000 00000000 00000000 0001598C
bhi $69A
-t
PC=0000069A USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000936 0000094E 00000000 00000000 00000000 00000000 00000000 0001598C
```

CMPI Instruction

The CMPI (CoMPare Immediate) instruction compares an immediate operand with an effective address operand. The condition codes are set as if the immediate quantity were subtracted from the effective address. Neither operand is altered.

When used with conditional branches, this instruction creates a less than desirable effect. When the combination

```
CMPI    #<data>,D0
BGT     X1
```

is used, the branch takes place if the value in data register D0 is greater than the immediate data.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. (The operands are equal.) Cleared otherwise.
- C Set if a borrow is generated. Cleared otherwise.
- V Set on overflow in the subtraction operation. Cleared otherwise.

Assembler Syntax: CMPI #<data>,<ea>

The CMP instruction compares an effective address operand to a data register. The CMPA instruction compares an effective address operand to an address register. The CMPM instruction compares memory to memory. Many assemblers make this distinction automatically.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	1	0	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Word data: 16 bits	Byte data: 8 bits

Long data: 32 bits, including previous word

The size field is 00 for byte compares, 01 for words, and 10 for longs.

Example:

```
PC=000006A2 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
cmp.l #$1,D0
-t
PC=000006A8 USP=0001598C SSP=000FBF08 ST=0004=>IM=0 ZER
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
beq $6AC
-t
PC=000006AC USP=0001598C SSP=000FBF08 ST=0004=>IM=0 ZER
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example compares the contents of data register D0 to the constant 1. The BEQ succeeds because D0 also contains a 1.

CMPM Instruction

The CMPM (CoMPare Memory) instruction compares two memory operands using the post-increment addressing mode. The condition codes are set as if the source operand were subtracted from the destination. Neither operand is altered.

When used with conditional branches, this instruction creates a less than desirable effect. When the combination

```
CMPM    (A0)+,(A1)+
BGT     X1
```

is used, the branch takes place if the second operand is greater than the first.

Addressing Modes Allowed: Only post-increment

Data Sizes: byte, word, long

Condition Codes Affected:

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. (The operands are equal.) Cleared otherwise.
- C Set if a borrow is generated. Cleared otherwise.
- V Set on overflow in the subtract operation. Cleared otherwise.

Assembler Syntax: CMPM (Ay)+,(Ax)+

The CMP instruction compares an effective address operand to a data register. The CMPA instruction compares an effective address operand to an address register. The CMPI instruction compares an immediate quantity to an effective address. Many assemblers make this distinction automatically.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	1	1	Reg Ax			1	Size		0	0	1	Reg Ay		

The size field is 00 for comparing bytes, 01 for words, and 10 for longs. The Ay and Ax fields specify the source and destination address registers, respectively.

Example:

```
PC=0000050C USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000514  00000524 00000000 00000000 00000000 00000000 00000000 0001598C
cmpm (A0)+,(A1)+
-l50c,510
  0000050C cmpm (A0)+,(A1)+
  0000050E beq $50C
  00000510 nop
-dw514,534
00000514 0000 0001 0002 0003 0004 0005 0006 0008        ................
00000524 0000 0001 0002 0003 0004 0005 0006 0007        ................
-g,510
PC=00000510 USP=0001598C SSP=0007BF08 ST=0009=>IM=0 NEG CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000524  00000534 00000000 00000000 00000000 00000000 00000000 0001598C
```

The example above compares two word strings in memory until it finds a pair that differ. From the *dw* command, we can see that the first pair of words that differ is at locations 522 and 532. The address registers are incremented past these locations by the post-decrement addressing modes.

DBcc Instruction

DBcc is an instruction designed for looping. The condition cc is similar to the conditions for conditional branches (see the Bcc instructions). DBcc will commonly be placed at the end of a loop. The condition is the termination condition, much like the REPEAT/UNTIL condition of the Pascal language. The loop may also be terminated on a maximum count.

The termination count is contained in the low word of a data register. The data register is decremented until it reaches −1. At this point the loop is terminated. Note that the comparison is for equal to −1. If the data register initially contains −1, then the loop will be repeated 65,536 times (assuming the termination condition is not satisfied).

The permissible instructions are:

DBCC	Terminate if the C-bit (carry) is Clear.
DBCS	Terminate if the C-bit (carry) is Set.
DBEQ	Terminate on EQual. The loop terminates if the Z (Zero) bit is set.
DBGE	Terminate on Greater than or Equal. The loop terminates if the N (negative) and V (overflow) bits are either both set or both clear. GE is used for two's complement binary numbers.
DBGT	Terminate on Greater Than. The loop terminates if • The N and V bits are both set *and* the Z-bit is clear, or, • The N, V, and Z bits are *all* clear.
DBHI	Terminate on HIgher than. The loop terminates if the C and Z bits are both clear. DBHI is similar to DBGT, except that it works on unsigned numbers.
DBLE	Terminate Less than or Equal. The loop terminates if • The Z-bit is set, or, • The N-bit is set AND the V-bit is clear, or, • The N-bit is clear AND the V-bit is set. The DBLE instruction is used for two's complement binary numbers.
DBLS	Terminate on Lower or Same. The loop terminates if either the C or Z bits are set. DBLS is similar to DBLE, except that it works on unsigned numbers.
DBLT	Terminate on Less Than. The loop terminates if • The N-bit is set and the V-bit is clear, or,

- The N-bit is clear and the V-bit is set.

DBMI Terminate on MInus. The loop terminates if the N-bit is set.

DBNE Terminate on Not Equal. The loop terminates if the Z- bit is clear.

DBPL Terminate on PLus. The loop terminates if the N-bit is clear.

DBVC Terminate on V Clear. The loop terminates if the V-bit is clear, indicating no overflow.

DBVS Terminate on V Set. The loop terminates if the V-bit is set, indicating overflow.

DBF Never terminate. The loop is terminated by count only. Many assemblers accept DBRA as an alternate to DBF.

DBT Always terminate. This instruction does not loop at all.

The condition is tested before decrementing the data register.

Addressing Modes Allowed:

DBcc instructions use a single addressing mode, where a two's complement displacement is contained in the second word of the instruction. If the loop is executed again, this displacement is sign-extended and added to the PC (Program Counter).

The PC contains the address of the displacement at the time the addition takes place.

Data Sizes: word

Condition Codes Affected: None

Assembler Syntax: DBcc Dn,<label>

<label> is a label on an instruction in the program.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	1	Condition				1	1	0	0	1	Register		
	16-bit displacement to loop beginning															

The Register field specifies the data register number to be used as the loop counter. The condition is a four-bit encoding of the branch condition. The conditions are shown in Table 3.3.

Condition	Instruction	Condition	Instruction
0000	DBT	1000	DBVC
0001	DBRA,DBF	1001	DBVS
0010	DBHI	1010	DBPL
0011	DBLS	1011	DBMI
0100	DBCC	1100	DBGE
0101	DBCS	1101	DBLT
0110	DBNE	1110	DBGT
0111	DBEQ	1111	DBLE

Table 3.3 *- Conditional Corresponding Instruction*

Example:

```
PC=000006CC USP=0001598C SSP=000FBF08 ST=0009=>IM=0 NEG CRY
D 00000001  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000952  0000095A  00000000  00000000  00000000  00000000  00000000  0001598C
moveq #$A,D0
-l6ce,6d4
   000006CE move.b (A0)+,(A1)+
   000006D0 dbeq D0,6CE
   000006D4 nop
```

This sample code segment moves a null-terminated string whose address is contained in register A0 to an area whose address is in register A1. Putting A in D0 (10 decimal) means that if a zero byte is not found, up to eleven bytes will be moved. The areas pointed to by registers A0 and A1 are (conveniently) adjacent. The debugger display command shows:

```
-d952,966
00000952 48 65 6C 6C 6F 20 00 00 00 00 00 00 00 00 00 00     Hello ..........
00000962 00 00 00 00 00                                        .....
```

The string Hello should be duplicated starting at address 95A. Notice the trailing space. Executing the program shows:

```
-g,6d4
PC=000006D4 USP=0001598C SSP=000FBF08 ST=0004=>IM=0 ZER
```

```
D 00000004  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000959  00000961 00000000 00000000 00000000 00000000 00000000 0001598C
```

The count register (D0) is now 4, indicating that six bytes were moved out of a possible eleven. Both address registers have been incremented by seven, and now point to the byte after the first zero byte. (The address registers were incremented once more than the count register was decremented because the *move.b* instruction was executed seven times, while the *dbeq* instruction was executed only six.) Looking at memory again shows:

```
-d952,966
00000952 48 65 6C 6C 6F 20 00 00 48 65 6C 6C 6F 20 00 00  Hello ..Hello ..
00000962 00 00 00 00 00                                   .....
```

The source string has been duplicated in the destination, including the trailing space and the null terminator.

DIVS Instruction

The DIVS (DIVide Signed) instruction divides a 32-bit quantity contained in a data register by a 16-bit quantity contained in an effective address operand. The low-order word of the data register is replaced by the quotient, and the high-order word by the remainder. The remainder and quotient will always have the same sign, except when the remainder is 0.

Two error conditions are possible with a DIVS instruction:

1. An attempt is made to divide by zero. The 68000 processor generates an exception condition when this is attempted (see Chapter 7 on Exception Conditions).
2. A large number is divided by a small number, and the quotient will not fit in 16 bits. This is an Overflow condition. The V-bit in the status register is set, and the contents of the data register remain unmodified.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Sizes: word

Condition Codes Affected:

X Not affected.
N Set if the quotient is negative. Cleared otherwise. Undefined on overflow conditions.
Z Set if the quotient is zero. Cleared otherwise. Undefined on overflow conditions.

C Always cleared.
V Set if overflow detected. Cleared otherwise.

Assembler Syntax: DIVS <ea>,Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	0	Register			1	1	1	Effective			Address		
											← Mode →			← Reg. →		

The Register field specifies the data register to be used as the destination (dividend). The Effective Address field specifies the source (divisor).

Example:

```
PC=000006DC USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00008887  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
divs #$2,D0
-t
PC=000006E0 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00014443  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example divides 8887 by 2 to yield a quotient of 4443 with a remainder of 1.

DIVU Instruction

The DIVU (DIVide Unsigned) instruction divides a 32-bit quantity contained in a data register by a 16-bit quantity contained in an effective address operand. The low-order word of the data register is replaced by the quotient, and the high-order word by the remainder. All quantities are considered to be unsigned positive integers.

Two error conditions are possible with a DIVU instruction:

1. An attempt is made to divide by zero. The 68000 processor generates an exception condition when this is attempted (see Chapter 7 on Exception Conditions).

2. A large number is divided by a small number and the quotient will not fit in 16 bits. This is an overflow condition. The V-bit in the status register is set and the contents of the data register remain unmodified.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Sizes: word

Condition Codes Affected:

X Not affected.
N Set if the quotient is negative. Cleared otherwise. Undefined on overflow conditions.
Z Set if the quotient is zero. Cleared otherwise. Undefined on overflow conditions.

C Always cleared.
V Set if overflow detected. Cleared Otherwise.

Assembler Syntax: DIVU <ea>,Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	0	Register			0	1	1	Effective			Address		
											← Mode →			← Reg. →		

The Register field specifies the data register to be used as the destination (dividend). The Effective Address field specifies the source (divisor).

Example:

```
PC=000006E8 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00008887  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
divu #$2,D0
-t
PC=000006EC USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00014443  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
nop
```

This example divides 8887 by 2 to yield a quotient of 4443 with a remainder of 1.

EOR Instruction

The EOR (Exclusive OR) instruction performs an exclusive OR function between a data register and memory. The data register is restricted to be the source, and the effective address is the destination.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
C Always cleared.
V Always cleared.

Assembler Syntax: EOR Dn,<ea>

The EORI instruction is used to exclusive OR immediate data with an effective address. Many assemblers allow using the EOR mnemonic for both, and produce the proper instruction based on the source operand.

Machine Code Format:

Bit 15	14	13	12	11 10 9	8	7 6	5 4 3	2 1 0
1	0	1	1	Register	1	Size	Effective	Address
							← Mode →	← Reg. →

Register is the data register number to be used as the source operand. Size is 00 for byte, 01 for word, and 10 for long data sizes.

Example:

```
PC=000006FA USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 11113333  22222222 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
eor.l D1,D0
-t
PC=000006FC USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 33331111  22222222 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example executes an exclusive OR on 11113333 with 22222222 to become 33331111. The bits with place value two in all the nibbles in the high-order word of register D0 are initially zero, so the exclusive OR operation sets these bits. The inverse is true for the bits with place value two in the nibbles of the low-order word. (If you're still not sure about how this works, convert the numbers to binary, do the EOR by hand, and then convert them back to hex.)

EORI Instruction

The EORI (Exclusive OR Immediate) instruction performs an exclusive OR function between an immediate operand and an effective address operand. The immediate operand must be the source, and the effective address is the destination.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	Yes	Yes

Data Sizes: byte, word, long

Operations specifying the status register (SR) and condition code register (CCR) are restricted to word and byte data lengths, respectively. Operations specifying the status register (SR) are privileged. A 68000 privilege-violation exception will result if this instruction is attempted with the SR addressing mode from User mode. Exceptions are covered in Chapter 7, Exception Processing.

Condition Codes Affected:

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
C Always cleared.
V Always cleared.

When the status register or the condition code register are used as the destination, the condition codes are determined by the operation itself. All bits of the register are affected. Thus, if an EORI instruction to the condition code register leaves all bits in the CCR cleared, the Z-bit is not set as it would be with other operands.

Assembler Syntax: EORI #<data>,<ea>

The EOR instruction executes an exclusive OR on a data register with an effective address. Many assemblers allow using the EOR mnemonic for both EOR and EORI, and produce the proper instruction based on the source operand.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	0	1	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Word data: 16 bits	Byte data: 8 bits

Long data: 32 bits, including previous word

Size is 00 for byte, 01 for word, and 10 for long data sizes.

Example:

```
PC=00000704 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 11113333  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
eori.l #$22222222,D0
-t
PC=0000070A USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 33331111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example executes an exclusive OR on 11113333 with 22222222 to become 33331111. The bits with place value two in all the nibbles in the high-order word of register D0 are initially zero, so the exclusive OR operation sets these bits. The inverse is true for the bits with place value two in the nibbles of the low-order word. (If you're still not sure about how this works, convert the numbers to binary, do the EOR by hand, and then convert them back to hex.)

EXG Instruction

The EXG instruction exchanges the complete contents of any two address or data registers.

Addressing Modes Allowed:
Only registers may be specified as operands.

Data Sizes: long

Condition Codes Affected: None

Assembler Syntax: EXG Rx,Ry

Machine Code Format:

Exchanging two data registers:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	Reg. Dx			1	0	1	0	0	0	Reg. Dy		

Exchanging two address registers:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	Reg. Ax			1	0	1	0	0	1	Reg. Ay		

Exchanging an address and a data register:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	Reg. Dx			1	1	0	0	0	1	Reg. Ay		

In exchanging an address and a data register, the address register must always be specified in bits 0–3 of the instruction word.

Example:

```
PC=00000718 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 11111111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 22222222  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
exg D0,A0
-t
PC=0000071A USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 22222222  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 11111111  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This instruction exchanges the contents of registers A0 and D0.

EXT Instruction

The EXT (sign EXTend) instruction extends the sign-bit of a byte into a word, or of a word into a long. The EXT instruction takes a single operand, which must be a data register.

When extending a byte to a long, bit 7 of the register is replicated into bits 15–8 of the register. Extension of a word into a long means extending bit 15 of the register into bits 16–31 of the register.

Addressing Modes Allowed: Only data registers

Data Sizes: word, long

Condition Codes Affected:

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
C Always cleared.
V Always cleared.

Assembler Syntax: EXT Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	0	0	Size		0	0	0	Register		

The size field is 10 to extend a byte to a word, and 11 to extend a word to a long.

Example:

```
PC=00000722 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00000080  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
ext D0
```

```
-t
PC=00000724 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 0000FF80  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
ext.l D0
-t
PC=00000726 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D FFFFFF80  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This example extends the byte 80 hex to a word, and then to a long. Notice that 80 (hex) is a negative number (bit 7 of the byte is set).

ILLEGAL Instruction

The ILLEGAL instruction is not really an instruction at all. It is an operation code guaranteed to cause an illegal-instruction exception on all future 68000 family machines. (Illegal instructions are often used for breakpoints by debugger software.) Exceptions are detailed in Chapter 7, Exception Processing.

Addressing Modes Allowed: None

Data Sizes: None

Condition Codes Affected: None

Assembler Syntax: None

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0	(4AFC)

JMP Instruction

The JMP (JuMP) instruction is used to transfer control to an effective address. The address to which transfer is made is the address generated by the effective address computation. For example, if address register A0 contains 1000 hex, a JMP 4(A0) instruction transfers control to the instruction located at 1004.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
No	No	Yes	No	No	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	No	No	No

Data Sizes: Unsized

Condition Codes Affected: None

Assembler Syntax: JMP <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	1	1	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
PC=0000072E USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000736  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
jmp (A0)
```

```
-t
PC=00000736 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000736  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

JSR Instruction

The JSR (Jump to SubRoutine) instruction calls a subroutine using an effective address operand. The address of the instruction following the JSR is pushed onto the stack (as a long word). The next instruction to be executed is determined by the effective address computation. For example, if address register A0 contains 1000 hex, a JSR 4(A0) instruction would call a subroutine located at address 1004.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
No	No	Yes	No	No	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	No	No	No

Data Sizes: Unsized

Condition Codes Affected: None

Assembler Syntax: JSR <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	1	1	0	1	0	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
PC=0000073A USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
jsr $652
```

```
-t
PC=00000652 USP=00015988 SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00015988
rts
```

At this point, the subroutine has been called. We can see the return address at the top of the stack (the address in register A7).

```
-sl15988
00015988 0000073E .
```

Executing the RTS yields:

```
-t
PC=0000073E USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

The return address has been popped off the stack.

LEA Instruction

The LEA (Load Effective Address) instruction places an effective address in an address register. All 32 bits of the register are affected.

The LEA instruction is normally used to write code that must be position-independent (i.e., can contain no code or data addresses in the program itself). The PC or Address register with displacement addressing modes are normally used for this type of coding.

The LEA instruction also adds a constant to an address register without altering the condition codes. By specifying the address register with displacement or index, either a constant or another register (or both) may be added to an address register in this manner.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	No	No	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	No	No	No

Data Sizes: long

Condition Codes Affected: None

Assembler Syntax: LEA <ea>,An

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	Register			1	1	1	Effective Address					
											← Mode →			← Reg. →		

The Register field specifies the address register used as the destination.

The Effective Address field specifies the address to be loaded into the address register.

Example:

```
PC=00000740 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
lea $736,A0
 -t
PC=00000746 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000736  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
lea $2(A0),A0
 -t
PC=0000074A USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000738  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example loads a constant address into address register A0 (using absolute long addressing). The second LEA instruction adds 2 to the address register.

LINK Instruction

The LINK instruction allocates a temporary area on the stack. Such an area is normally called a *stack frame*. Many block structured high-level languages, such as C, Pascal, and PL/I use this instruction for allocating variables that are local to a procedure. The variables are deallocated when the procedure is deactivated, permitting efficient memory usage.

The LINK instruction takes two operands: an address register and a 16-bit signed displacement. The address register is pushed onto the stack, and the resulting stack pointer is copied into the address register. The displacement (which is usually negative) is added to the stack pointer to allocate memory for the local variables. The stack winds up looking like Figure 3.5.

The local storage area is addressed using negative displacements from the address register, sometimes known as the *frame pointer*. In this way, local variables may be accessed without regard for intervening pushes and pops on the stack.

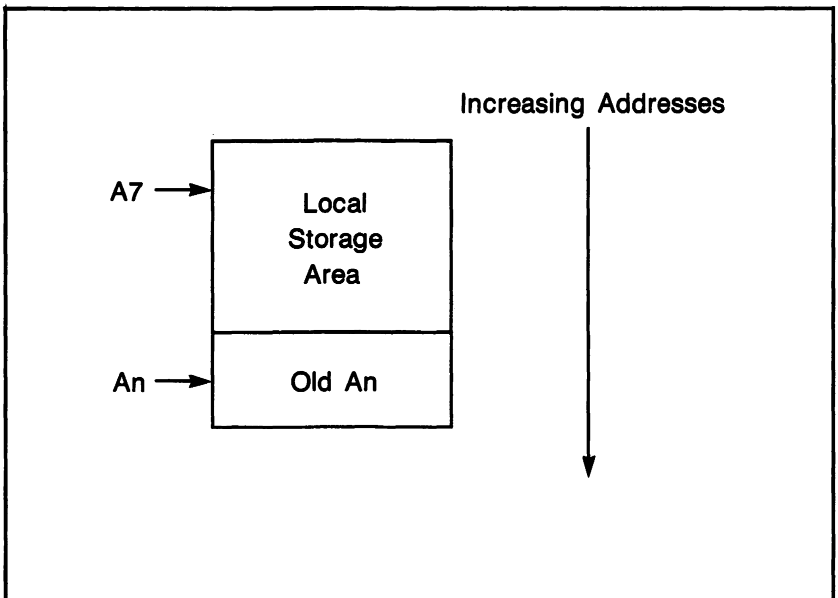


Figure 3.5 *- Example of stack layout*

The UNLK instruction deallocates the stack frame and restores the stack pointer and address register to their original contents.

Addressing Modes Allowed: None

Data Sizes: None

Condition Codes Affected: None

Assembler Syntax: LINK An, #<displacement>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	1	1	0	0	1	0	1	0	Register		

16-bit displacement to add to A7

Register is the number of the address register to be used.

Example:

```
PC=0000074C USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
link A6,#$FFF4
-t
PC=00000750 USP=0001597C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00015988 0001597C
```

This instruction pushes a total of 16 (10 hex) bytes on the stack: four for the address register and twelve due to the displacement field. (FFF4 is −12 decimal.) The long word address contained in register A6 shows the old contents of A6:

```
-sl15988
00015988 00000000
```

```
PC=00000752 USP=0001597C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00015988  0001597C
unlk A6
-t
PC=00000754 USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

The UNLK instruction reverses the action of the LINK instructions.

LSL Instruction

The LSL (Logical Shift Left) instruction performs a logical left shift on a data register or an effective address operand. There are three forms of this instruction:

1. Shift a data register to the left by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Shift a data register to the left by the number of bits contained in another data register.
3. Shift a memory word (16 bits) left by a single bit.

Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Memory operations are restricted to word length.

Condition Codes Affected:

X Set according to the last bit shifted out of the high-order bit position of the operand. Unaffected if the shift count is zero.
N Set if the high order bit of the result is set (indicating a negative result). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Set according to the last bit shifted out of the high-order bit position of the operand. Unaffected if the shift count is zero.

Assembler Syntax: LSL Dx,Dy
LSL #<data>,Dy
LSL <ea>

Machine Code Format:

Data register as destination

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			1	Size		T	0	1	Register		

Memory location as destination

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	0	1	1	1	1	Effective			Address		
											← Mode →			← Reg. →		

The Immediate field contains either the shift count or the data register number that contains the shift count. A 000 in this field represents a shift count of 8. Values 001 through 111 represent shift counts of 1–7. Size is either 00 for byte data, 01 for words, and 10 for longs. The T-bit (type) is a 0 if the shift count is contained in the instruction, and 1 if the shift count is in a data register. Register is the destination data register number.

Example:

```
PC=0000075E USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00001111  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
lsl D1,D0
-t
PC=00000760 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 00004444  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example uses the data register form of the instruction. 1111 is shifted left by two places to become 4444.

LSR Instruction

The LSR (Logical Shift Right) instruction performs a logical right shift on a data register or an effective address operand. There are three forms of this instruction:

1. Shift a data register to the right by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Shift a data register to the right by the number of bits contained in another data register.
3. Shift a memory word (16 bits) right by a single bit.

Addressing Modes Allowed: Memory form only

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Memory operations are restricted to word length.

Condition Codes Affected:

X Set according to the last bit shifted out of the low-order bit position of the operand. Unaffected if the shift count is zero.
N Set if the high order bit of the result is set (indicating a negative result). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Set according to the last bit shifted out of the low-order bit position of the operand. Unaffected if the shift count is zero.

Assembler Syntax: LSR Dx,Dy
LSR #<data>,Dy
LSR <ea>

Machine Code Format:

Data register as destination

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			0	Size		T	0	1	Register		

Memory location as destination

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	0	1	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

The Immediate field contains either the shift count or the data register number that contains the shift count. A 000 in this field represents a shift count of 8. Values 001 through 111 represent shift counts of 1–7. Size is either 00 for byte data, 01 for words, and 10 for longs. The T-bit (type) is a 0 if the shift count is contained in the instruction, and 1 if the shift count is in a data register. Register is the destination data register number.

Example:

```
PC=0000076A USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 88888888  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
lsr D1,D0
-t
PC=0000076C USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 88882222  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example shifts data register D0.W (the low word of D0) to the right by two places. The upper half of D0 is unaffected.

MOVE Instruction

The MOVE instruction copies a byte, word, or longword from one effective address operand to another. The condition codes are set according to the data that is moved during the operand.

Addressing Modes Allowed: Source operand

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Addressing Modes Allowed: Destination operand

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Use of the An (address register direct) addressing mode is restricted to instructions with word and long data sizes.

Data Sizes: byte, word, long

Condition Codes Affected:

X Not affected.
N Set if the high order bit of the result is a one. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.

V Always cleared.
C Always cleared.

Assembler Syntax: MOVE <ea>,<ea>

MOVEA moves to an address register. Most assemblers automatically use the right instruction with an address register destination.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	Size		Destination <ea>						Source <ea>					
					← Reg. →			← Mode →			← Mode →			← Reg. →		

Size is the size of the data to be transferred: 01 for bytes, 11 for words, and 10 for longs.

Example:

```
PC=00000774 USP=0001598C SSP=000FBF08 ST=0000=>IM=0
D 11111111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
move.b $92D,D0
-s92d
0000092D 99 .
-t
PC=0000077A USP=0001598C SSP=000FBF08 ST=0008=>IM=0 NEG
D 11111199  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This instruction moves a memory location into a data register. Byte moves into a data register leave the upper three bytes of the register unmodified.

MOVE to CCR Instruction

The MOVE to CCR instruction moves the low-order byte of a word operand to the condition code register (CCR), the User byte of the status register. The high-order byte of the source is ignored.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Size: word

Condition Codes Affected:

X Set from bit 4 of the source operand.
N Set from bit 3 of the source operand.
Z Set from bit 2 of the source operand.
V Set from bit 1 of the source operand.
C Set from bit 0 of the source operand.

Assembler Syntax: MOVE <ea>,CCR

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	1	0	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

Example:

PC = 00x0077C USP = 0001598C SSP = 000FBF08 ST = 0000 = >IM = 0
D 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

```
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
move #$1F,CCR
-t
PC=000780 USP=01598C SSP=0FBF08 ST=001F=>IM=0 EXT NEG ZER OFL CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This instruction sets all condition code bits in the status register.

MOVE to SR Instruction

The MOVE to SR (Status Register) instruction transfers a word operand to the CPU status register. All bits of the status register are affected. This instruction requires that the S-bit of the status register (bit 13) be set at the beginning of the instruction (i.e., the CPU must be in Supervisor state).

This instruction may be used to alter the Status register Trace bit, Supervisor bit, interrupt mask, and condition codes. Typical uses include:

- Clearing the Supervisor bit to transfer to User mode.
- Clearing the Interrupt mask to enable CPU interrupts.
- Setting bits in the Interrupt mask to partially or completely disable CPU interrupts.

Changing the condition codes is normally done with the MOVE to CCR instruction, which is not privileged, and can be executed from either supervisor or user mode.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Size: word

Condition Codes Affected:

X Set from bit 4 of the source operand.
N Set from bit 3 of the source operand.
Z Set from bit 2 of the source operand.
V Set from bit 1 of the source operand.
C Set from bit 0 of the source operand.

Assembler Syntax: MOVE <ea>,SR

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	1	1	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
-g,786
PC=00000786 USP=0001598C SSP=0007BF08 ST=2010=>SUP IM=0
EXT
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0007BF08
move #$200F,SR
-g,78a
PC=00078A USP=01598C SSP=07BF08 ST=200F=>SUP IM=0 NEG ZER OFL CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0007BF08
```

This instruction sets the N, Z, O, C, condition codes, as well as the S-bit. The S (Supervisor) bit is set before execution. A Privilege-violation exception results if this instruction is attempted from User mode. Notice that address register A7 reflects the supervisor stack pointer (SSP) rather than the user stack pointer (USP).

MOVE from SR Instruction

The MOVE from SR (Status Register) instruction transfers the entire status register to a word operand. Memory operands are read before writing. This instruction is privileged on the 68010 processor. (A privileged instruction requires that the S-bit [bit 13] in the status register be set prior to execution.)

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Size: word

Condition Codes Affected: None

Assembler Syntax: MOVE SR,<ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	0	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
PC=0000078C USP=0001598C SSP=0007BF08 ST=200F=>SUP IM=0
NEG ZER OFL CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0007BF08
move SR,D0
```

```
-t
PC=0000078E USP=0001598C SSP=0007BF08 ST=200F=>SUP IM=0
NEG ZER OFL CRY
D 0000200F  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0007BF08
```

This instruction transfers the status register to the low-order word of data register D0. Notice that address register A7 reflects the supervisor stack pointer (SSP) rather than the user stack pointer (USP).

MOVE USP Instruction

The MOVE USP instruction transfers the user stack pointer to or from an address register. This instruction is a privileged instruction. (The S-bit [bit 13] in the status register must be set.)

A MOVE USP instruction is normally used by a supervisor program to set up a stack area in a user program. The 68000 has separate user and supervisor stack pointer registers, so this special instruction is required to enable the supervisor program to access the user mode stack pointer.

Addressing Modes Allowed: Only An

Data Size: long

Condition Codes Affected: None

Assembler Syntax: MOVE USP,An
MOVE An,USP

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	1	1	0	0	1	1	0	T	Register		

To transfer the address register to the USP, the T field is 0, and 1 to transfer the USP to the address register. The Register field specifies the number of the address register to be used.

Example:

```
PC=000790 USP=01558C SSP=07BF08 ST=200F=>SUP IM=0 NEG ZER OFL CRY
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0007BF08
move.l USP,A0
-t
PC=000792 USP=01558C SSP=07BF08 ST=200F=>SUP IM=0 NEG ZER OFL CRY
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 0001558C  00000000  00000000  00000000  00000000  00000000  00000000  0007BF08
```

This example transfers the user stack pointer (USP) to address register A0. The status register S bit must be set for this privileged instruction.

MOVEA Instruction

The MOVEA (Move Address) instruction moves an effective address operand to an address register. Only word and long data sizes are allowed. All 32 bits of the address register are always affected. Word operations are sign extended to 32 bits before loading the address register.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Sizes: word, long

Condition Codes Affected: None

Assembler Syntax: MOVEA <ea>,An

Many assemblers will generate the MOVEA instruction when a MOVE instruction specifies an address register as the destination.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	Size		Register			0	0	1	Effective			Address		
											← Mode →			← Reg. →		

The Size field is 11 for word transfers, and 10 for long transfers. The Register field gives the destination address register. Notice that this instruction is really a MOVE instruction with a destination addressing mode of 001.

Example:

```
PC=00000796 USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
move.l #$1234567,A0
-t
PC=0000079C USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 01234567  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
```

This example loads address register A0 with the constant 01234567.

MOVEM Instruction

The MOVEM (MOVE Multiple) instruction provides a means for rapidly transferring a group of registers to or from memory. The size of the operation is restricted to word or long data. For word operations that transfer data to the registers, each word is sign-extended to 32 bits before loading the register. All 32 bits of the register are always affected, regardless of whether the register is an address or data register.

The order in which the registers are stored in memory is as follows:

Address (effective address)	**Register**
+0	D0
+4	D1
.	
.	
.	
+28	D7
+32	A0
+36	A1
.	
.	
.	
+60	A7
+64	(Unused)

Any combination of the registers may be loaded or stored using this instruction. The illustration above assumes that all registers are present and long data size.

This instruction is used primarily for pushing a group of registers on the stack so that they may be used temporarily and later reset to their original values, also using a MOVEM instruction. This technique is especially valuable for subroutines and exception-processing routines, where it is often not known which registers can be modified.

An anomaly of 68000 architecture causes an extra memory reference when transferring memory to registers using a MOVEM. In the diagram above, offsets 64 and 65 from the base of the register area in memory would be read (again assuming all registers were present). This is usually not significant, but it can cause problems in certain specialized cases. For example, trying to transfer registers from the very last locations in memory will cause an erroneous BUSERR to occur due to the access of the next word (which is not a valid memory address in this case). The BUSERR exception is explained in Chapter 7, Exception Processing.

Addressing Modes Allowed: Registers to memory

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
No	No	Yes	No	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Memory to Registers:

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
No	No	Yes	Yes	No	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	No	No	No

Data Sizes: word, long

Condition Codes Affected: None

Assembler Syntax: MOVEM <register list>,<ea>
MOVEM <ea>,<register list>

The register list is composed of a series of register specifications separated by a slash (the / character). For example, D0/D2/A5 specifies registers D0, D2, and A5 as operands. It is also possible to use one or more ranges in the register list. A range consists of two register specifications separated by the – character. For instance, D0–D5/A0–A2 specifies registers D0, D1, D2, D3, D4, D5, A0, A1, and A2.

Machine Code Format:

Registers to Memory:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	0	0	1	Sz	Effective			Address		
											← Mode →			← Reg. →		
	Register List Bit Mask															

Memory to Registers:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	1	0	0	1	Sz	Effective			Address		
											← Mode →			← Reg. →		
	Register List Bit Mask															

The Sz (size) bit is a 0 for word transfers and a 1 for long transfers. The register list bit mask contains a single bit for each register that can be transferred by a MOVEM instruction (16 registers). If the bit is a 1, then the register is transferred. If the bit is 0, then the register is not transferred.

The register-list word has two possible orientations, one for predecrement mode addressing and one for all other addressing modes. In all cases, the low-order bit of the mask corresponds to the register that is to be transferred first. The two sets of correspondence between registers and bits are as follows:

Pre-decrement (−(An)) addressing mode:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

All other addressing modes:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Example:

```
PC=0000079E USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 000000D0  000000D1  00000000  00000000  00000000  00000000  00000000  00000000
A 000000A0  000000A1  00000000  00000000  00000000  00000000  00000000  0001558C
movem.l D0-D1/A0-A0,-(A7)
-t
PC=000007A2 USP=00015580 SSP=0007BF08 ST=0000=>IM=0
D 000000D0  000000D1  00000000  00000000  00000000  00000000  00000000  00000000
A 000000A0  000000A1  00000000  00000000  00000000  00000000  00000000  00015580
```

This instruction pushes the contents of registers D0, D1, and A0 on the stack. Using the S command, we can look at the stack and see the saved registers.

```
-s15580
00015580 000000D0
00015584 000000D1
00015588 000000A0
-t
PC=000007A4 USP=00015580 SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  000000A1  00000000  00000000  00000000  00000000  00000000  00015580
movem.l (A7)+,D0-D1/A0-A0
```

```
-t
PC=000007A8 USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 000000D0  000000D1  00000000  00000000  00000000  00000000  00000000  00000000
A 000000A0  000000A1  00000000  00000000  00000000  00000000  00000000  0001558C
```

The second half of the example above restores the registers from the stack. Notice that D0, D1, and A0 are valid hexadecimal numbers as well as names for registers.

MOVEP Instruction

The MOVEP (MOVE Peripheral) instruction provides a convenient method for accessing 8-bit peripheral devices connected to a 68000. The 68000 is a 16-bit microprocessor. This means that there are sixteen data lines connecting the processor to memory, as illustrated in Figure 3.6.

Many I/O devices were designed for 8-bit microprocessors, and thus have only eight data lines. These devices can be connected to the 68000 using either the Upper byte data lines or the Lower byte data lines. Addressing a device connected in such a fashion is done by using alternate memory addresses (for example, 1,3,5 for devices connected to the Lower byte lines, or 0,2,4 for devices connected to the Upper byte lines). The MOVEP instruction is facilitates this process.

The MOVEP instruction transfers a word or a longword contained in a data register to or from these alternate memory addresses. The destination memory address is specified using the address register with displacement addressing mode— x(An).

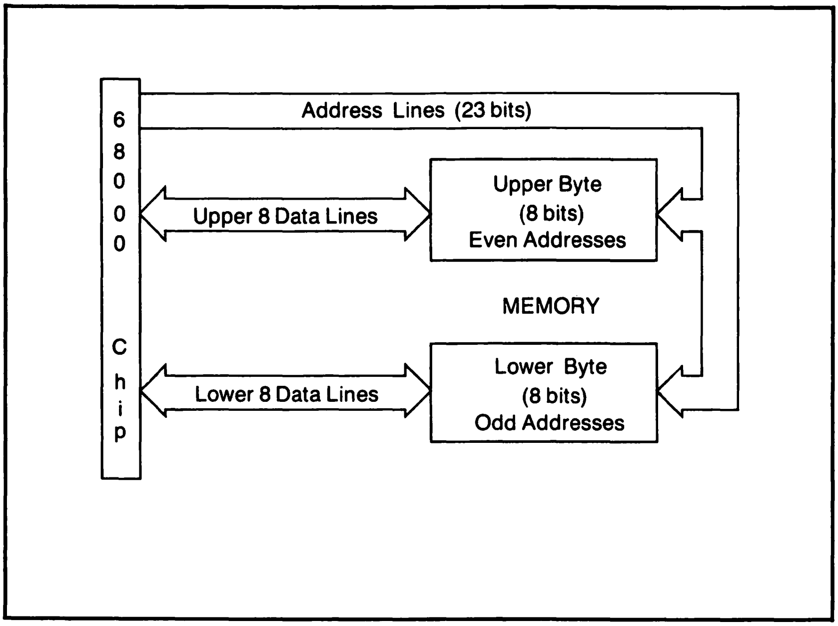


Figure 3.6 - *Sixteen data lines connecting the processor to memory*

The high-order byte of the data register is transferred to or from address x(An), the next byte to or from x+2(An), and so forth. If the original address was odd, then all transfers from the MOVEP will use the Lower byte of the 68000 data bus. Even addresses use the High byte.

Addressing Modes Allowed: x(An) only

Data Sizes: word, long

Condition Codes Affected: None

Assembler Syntax: MOVEP Dn,x(An)
MOVEP x(An),Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	Dn Reg.			1	Dr	Sz	0	0	1	An Reg.		

Displacement

The Dn Register and An Register give the numbers of the data and address registers to be used, respectively. The Dr (direction) bit is 0 for memory to register transfers and 1 for register to memory transfers. The Sz (size) bit is 1 for long data and 0 for words. The displacement field is a 16-bit offset that is added to the address register to form the base memory address at which the transfer begins.

Example:

```
PC=000007B6 USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 01234567  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000964  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
movep.l D0,0(A0)
-t
PC=000007BA USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 01234567  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000964  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
-d964,96c
00000964 01 00 23 00 45 00 67 00 00                         ..#.E.g..
```

In this example, register D0 is stored starting at address 964. Since the starting address is even, the bytes in D0 will go into the high-order bytes of successive words, as illustrated by the d command. The high-order byte of D0 is stored first.

MOVEQ Instruction

The MOVEQ instruction provides a method for loading a small immediate quantity into a data register. The instruction is two bytes in length, and can load any constant in the range −128 to +127 (decimal). All 32 bits of the register are affected. (The corresponding MOVE immediate long instruction takes six bytes.)

Data Size: long

Condition Codes Affected:

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Assembler Syntax: MOVEQ #<data>,Dn

Many assemblers automatically convert a move with the appropriate operands into a MOVEQ.
Machine Code Format:

Bit	15	14	13	12	11 10 9	8	7 6 5 4 3 2 1 0
	0	1	1	1	Register	0	Immediate Data

The Register field identifies the destination data register. The Immediate Data is an 8-bit immediate operand that is sign-extended before loading into the data register.

Example:

```
PC=000007BC USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0001558C
moveq #$80,D0
```

```
-t
PC=000007BE USP=0001558C SSP=0007BF08 ST=0008=>IM=0 NEG
D FFFFFF80  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
```

This example loads register D0 with the constant 80 hex (−128 decimal). The data is sign-extended into a long by the operation.

MULS Instruction

The MULS (MULtiply Signed) instruction multiplies a 16-bit data register operand by a 16-bit effective address operand, leaving the 32-bit result in the data register. The operation assumes two's complement arithmetic.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Size: word

Condition Codes Affected:

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Assembler Syntax: MULS <ea>,Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	Register			1	1	1	Effective Address					
											← Mode →			← Reg. →		

The Register field identifies the data register to be used as the destination.

Example:

```
PC=000007C6 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00001234  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
muls #$10,D0
-t
PC=000007CA USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00012340  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example multiplies the contents of register D0 by the constant 10 hex (16 decimal).

MULU Instruction

The MULU (MULtiply Unsigned) instruction multiplies a 16-bit data register operand by a 16-bit effective address operand, leaving the 32-bit result in the data register. The operation assumes unsigned arithmetic.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Data Size: word

Condition Codes Affected:

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Assembler Syntax: MULU <ea>,Dn

Machine Code Format:

Bit	15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
	1	1	0	0	Register	0	1	1	Effective	Address
									← Mode →	← Reg. →

The Register field identifies the data register to be used as the destination.

Example:

```
PC=000007D2 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00008000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
mulu #$5,D0
-t
PC=000007D6 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00028000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This example multiplies hex 8000 (normally a negative number) by 5. Notice that the result is positive. This is because unsigned arithmetic was used in the MULU instruction.

NBCD Instruction

The NBCD (Negate BCD) instruction forms the negative of a BCD (binary coded decimal) number. The technique used is ten's complement, analogous to the two's complement used with binary numbers. The ten's complement of a number is formed by subtracting the number from all 9s and then adding one. As with binary numbers, the complementing process works only if you have a fixed number of digits.

For example, assume you have a 4-digit BCD system. The ten's complement of 0001 is (9999 − 0001) + 1, or 9999. Adding this number to a positive decimal number is equivalent to subtracting 1.

As with the other 68000 BCD instructions, NBCD is intended for multiprecision BCD arithmetic. The X (eXtend) bit is added to the ten's complement process. This bit provides the borrow necessary for multiprecision arithmetic. The Z-bit is only cleared by this instruction. This allows the Z-bit to accurately reflect a multiprecision result. Normally, a series of NBCD instructions begins with the X-bit clear and the Z-bit set. The size of the operation is restricted to byte size.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Size: byte

Condition Codes Affected:

X Set if a borrow was generated in the subtraction operation. Cleared otherwise.

N Undefined.

Z Cleared if the result was not a zero. Unchanged otherwise.

V Undefined.
C Set if a borrow was generated in the subtraction operation. Cleared otherwise.

Assembler Syntax: NBCD <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	0	0	0	0	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
PC=000007DA USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
nbcd D0
-t
PC=000007DC USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 00000099  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This example takes the ten's complement of 01 to produce 99. (Remember, BCD is the same as hex.)

NEG Instruction

The NEG (NEGate) instruction forms the two's complement of an effective address operand.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

- X Cleared if the result is zero. Set otherwise.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow is generated. Cleared otherwise.
- C Cleared if the result is zero. Set otherwise.

Assembler Syntax: NEG <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	1	0	0	Size		Effective			Address		
											← Mode →			← Reg. →		

The Size field is 00 for byte data, 01 for words, and 10 for longs.

Example:

```
PC=000007E2 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 00000001  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
neg D0
-t
PC=000007E4 USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 0000FFFF  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This example takes the two's complement in the low-order word of register D0. (1 is complemented to become −1.)

NEGX Instruction

The NEGX (Negate with eXtend) instruction provides a method for taking the two's complement of a multiprecision binary number. The X (extended) condition code serves as the carry between multiprecision instructions.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Set if a borrow is generated. Cleared otherwise.
N Set if the result is negative. Cleared otherwise.
Z Cleared if the result is not zero. Unchanged otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

As with other 68000 multiprecision instructions, a group of NEGX instructions should begin with the Z-bit set and the X-bit clear. At the completion of the multiprecision operation, the Z-bit will then correctly indicate whether the entire operand is zero. The usual storage order for multiprecision integers on the 68000 is to place the high-order portion at the lowest address, and the low-order portion at the highest address. ADDs, SUBtracts, and NEGates begin with the low-order word and normally use the pre-decrement addressing mode.

Assembler Syntax: NEGX <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	0	0	Size		Effective			Address		
											← Mode →			← Reg. →		

The Size field is 00 for byte operands, 01 for words, and 10 for longs.

Example:

```
PC=000007EA USP=0001598C SSP=0007BF08 ST=0014=>IM=0 EXT ZER
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
neg.l D0
-t
PC=000007EC USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D FFFFFFFF  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
negx.l D1
-t
PC=000007EE USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D FFFFFFFF  FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example uses data registers D0 and D1 as an 8-byte (64 bit) binary integer. Register D1 is the high-order part of the integer. The example complements 1 to become −1 across 64 bits.

NOP Instruction

The NOP (No OPeration) instruction provides a way for idling the 68000 for one instruction. Nothing is changed, except that the Program Counter is advanced to the next instruction. NOP instructions are often used for inserting small delays, or for providing space in a program for patching purposes.

Addressing Modes Allowed: None

Data Sizes: None

Condition Codes Affected: None

Assembler Syntax: NOP

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	(4E71 hex)

Example:

```
PC=000007F2 USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
nop
-t
PC=000007F4 USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

NOT Instruction

The NOT instruction forms the one's complement of an effective address operand.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An) +**	**− (An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Assembler Syntax: NOT <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	1	1	0	Size		Effective			Address		
											← Mode →			← Reg. →		

The Size field is 00 for byte operations, 01 for words, and 10 for longs.

Example:

```
PC=000007F6 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
not D0
-t
PC=000007F8 USP=0001598C SSP=0007BF08 ST=0018=>IM=0 EXT NEG
D 0000FFFE  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example takes the one's complement of the 16-bit quantity 0001 hex to form FFFE hex.

OR Instruction

The OR instruction performs a bit-wise inclusive binary OR operation between a data register and an effective address operand. There are two forms of this instruction:

1. OR the contents of the effective address operand with a data register, leaving the result in the data register.
2. OR the contents of the effective address operand with a data register, leaving the result in the effective address operand.

Addressing Modes Allowed:

Effective address as source:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

Effective address as destination:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Assembler Syntax: OR <ea>,Dn
OR Dn,<ea>

The ORI instruction is usually used when the source is immediate data and the destination is not a data register. Many assemblers use the ORI instruction when OR is specified with this combination of operands.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	0	Register			Dr	Size		Effective			Address		
											← Mode →			← Reg. →		

The Register field indicates which data register is to be used. The Dr bit is 0 if the data register is the destination, and 1 if the effective address is the destination. Size is 00 for byte operations, 01 for words, and 10 for longs.

Example:

```
PC=00000806 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 11111111  22222222 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
or.l D0,D1
-t
PC=00000808 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 11111111  33333333 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example ORs the contents of registers D0 and D1, leaving the result in D1. Notice that this is the data register destination form of the instruction.

ORI Instruction

The ORI (OR Immediate) performs a bit-wise OR between an immediate operand (always the source) and an effective address operand (always the destination).

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	Yes	Yes

When the status register is specified as the destination, the S-bit in the status register must be set (i.e., the 68000 must be executing in Supervisor state).

Data Sizes: byte, word, long

Condition Codes Affected:

X Not affected.
N Set if the high-order bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

If the status register (SR) or condition code register (CCR) is specified as the destination, the condition code bits are set according to bits 5–0 of the immediate source.

Assembler Syntax: ORI #<data>,<ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Word data: 16 bits	Byte data: 8 bits

Long data: 32 bits, including previous word

The Size field is 00 for byte data, 01 for word data, and 10 for long data.

Example:

```
PC=00000810 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 11111111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
ori.l #$22222222,D0
-t
PC=00000816 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 33333333  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example performs an OR operation on the contents of register D0 with a long constant.

PEA Instruction

The PEA (Push Effective Address) instruction places a computed address on top of the stack. The size of the instruction is restricted to long data.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
No	No	Yes	No	No	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	No	No	No

Data Size: long

Condition Codes Affected: None

Assembler Syntax: PEA <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	0	0	0	1	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
PC=00000816 USP=0001598C SSP=0007BF08 ST=0010=>IM=0 EXT
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
pea $81E
-t
PC=0000081C USP=00015988 SSP=0007BF08 ST=0010=>IM=0 EXT
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00015988
-sl15988
00015988 0000081E .
```

This example pushes the absolute long address 81E. The PUSH is verified by examining memory at the address contained in address register A7 after the instruction executes.

RESET Instruction

The RESET instruction is a privileged operation that causes all external devices to be reset.

Addressing Modes Allowed: None

Data Sizes: unsized

Condition Codes Affected: None

Assembler Syntax: RESET

Machine Code Format:

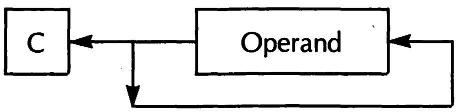
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	(4E70 hex)

ROL Instruction

The ROL (ROtate Left) instruction performs a left rotate on a data register or memory operand. There are three forms of this instruction:

1. Rotate a data register to the left by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Rotate a data register to the left by the number of bits contained in another data register.
3. Rotate a memory word left by one bit only.

The Rotate operation is performed without an auxiliary bit. Bits shifted out of the high-order bit position go to both the carry bit and the low-order bit position.



Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Data size is restricted to word for the in-memory form of the instruction.

Condition Codes Affected:

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- C Set according to the last bit shifted out of the operand. Cleared if the shift count is zero.
- V Always cleared.

Assembler Syntax: ROL #<count>,Dy
ROL Dx,Dy
ROL <ea>

Machine Code Format:

Data Register as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			1	Size		T	1	1	Register		

Memory location as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	1	1	1	1	1	Effective			Address		
											← Mode →			← Reg. →		

The T-field determines the type of the register-destination form of the instruction. If T is 0, then the Immediate field contains the shift count, with 000 binary representing a count of 8. If T is 1, then the register number that contains the shift count is contained in the Immediate field.

Example:

```
PC=00000830 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 11111111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
rol #3,D0
```

```
-t
PC=00000832 USP=0001598C SSP=0007BF08 ST=0008=>IM=0 NEG
D 11118888  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

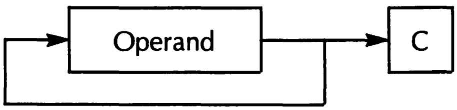
This instruction rotates the lower word of register D0 left by three bits. Notice that the high word of D0 is unaffected.

ROR Instruction

The ROR (ROtate Right) instruction performs a right rotate on a data register or memory operand. There are three forms of this instruction:

1. Rotate a data register to the right by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Rotate a data register to the right by the number of bits contained in another data register.
3. Rotate a memory word right by one bit only.

The Rotate operation is performed without an auxiliary bit. Bits shifted out of the low-order bit position go to both the carry bit and the high-order bit position.



Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Data size is restricted to word for the in-memory form of the instruction.

Condition Codes Affected:

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- C Set according to the last bit shifted out of the operand. Cleared if the shift count is zero.
- V Always cleared.

Assembler Syntax: ROR #<count>,Dy
ROR Dx,Dy
ROR <ea>

Machine Code Format:

Data Register as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			0	Size		T	1	1	Register		

Memory location as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	1	1	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

The T-field determines the type of the register-destination form of the instruction. If T is 0, then the Immediate field contains the shift count, with 000 binary representing a count of 8. If T is 1, the register number that contains the shift count is contained in the Immediate field.

Example:

```
PC=0000083E USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 11111111  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
ror #3,D0
```

```
-t
PC=00000840 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 11112222  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

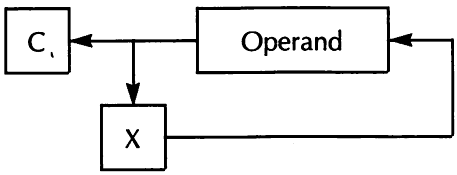
This instruction rotates the low-order word of register D0 right by three bits. The upper word of register D0 is unaffected.

ROXL Instruction

The ROXL (ROtate Left with eXtend) instruction performs a left rotate on a data register or memory operand. There are three forms of this instruction:

1. Rotate a data register to the left by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Rotate a data register to the left by the number of bits contained in another data register.
3. Rotate a memory word left by one bit only.

The Rotate operation is performed using the X-bit as an auxiliary bit. Bits shifted out of the high-order bit position go to both the carry bit and the X-bit. The X-bit is rotated into the low-order bit position.



Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Data size is restricted to word for the in-memory form of the instruction.

Condition Codes Affected:

X	Set according to the last bit shifted out of the operand. Unaffected if the shift count is zero.
N	Set if the most significant bit of the result is set. Cleared otherwise.
Z	Set if the result is zero. Cleared otherwise.
C	Set according to the last bit shifted out of the operand. Set to the value of the X-bit if the shift count is zero.
V	Always cleared.

Assembler Syntax: ROXL #<count>,Dy
ROXL Dx,Dy
ROXL <ea>

Machine Code Format:

Data Register as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			1	Size		T	1	0	Register		

Memory location as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	1	0	1	1	1	Effective Address					
											← Mode →			← Reg. →		

The T-field determines the type of the register-destination form of the instruction. If T is 0, then the Immediate field contains the shift count, with 000 binary representing a count of 8. If T is 1, then the register number that contains the shift count is contained in the Immediate field.

Example:

```
PC=0000084C USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 11111111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
roxl #4,D0
```

```
-t
PC=0000084E USP=0001598C SSP=0007BF08 ST=0011=>IM=0 EXT CRY
D 11111110  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

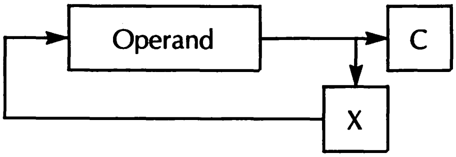
This example rotates the low-order word of register D0 to the left by four bits. The high-order word of register D0 is unaffected. The X-bit is set following the instruction.

ROXR Instruction

The ROXR (ROtate Right with eXtend) instruction performs a right rotate on a data register or memory operand. There are three forms of this instruction:

1. Rotate a data register to the right by a constant contained in the instruction. Shifts from one to eight bits can be accomplished using this form of the instruction.
2. Rotate a data register to the right by the number of bits contained in another data register.
3 Rotate a memory word right by one bit only.

The Rotate operation is performed using the X bit as an auxiliary bit. Bits shifted out of the low-order bit position go to both the carry-bit and the X bit. The X bit is rotated into the high-order bit position.



Addressing Modes Allowed: Memory form only

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.1**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word long
Data size is restricted to word for the in-memory form of the instruction.

Condition Codes Affected:

- X Set according to the last bit shifted out of the operand. Unaffected if the shift count is zero.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- C Set according to the last bit shifted out of the operand. Set to the value of the X-bit if the shift count is zero.
- V Always cleared.

Assembler Syntax: ROXR #<count>,Dy
ROXR Dx,Dy
ROXR <ea>

Machine Code Format:

Data Register as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	Immed.			0	Size		T	1	0	Register		

Memory location as destination:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	0	0	1	0	0	1	1	Effective Address					
											← Mode →			← Reg. →		

The T-field determines the type of the register-destination form of the instruction. If T is 0, then the Immediate field contains the shift count, with 000 binary representing a count of 8. If T is 1, then the register number that contains the shift count is contained in the Immediate field.

Example:

```
PC=0000085A USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 11111111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
roxr #4,D0
```

```
-t
PC=0000085C USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 11112111  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example rotates the low-order word of register D0 to the right by four places. The upper word of register D0 is not affected.

RTE Instruction

The RTE (ReTurn from Exception) is used to load the status register and the program counter (PC) with a single instruction. This type of operation is required when an operating system in supervisor mode passes control to a user program in user mode. The new contents of the status register and PC are popped off the stack. The status register is taken from the first 16-bit word on the stack, and the PC from the next 32-bit long word. The stack pointer is incremented by six bytes.

This is a privileged instruction. The processor must be in supervisor state (i.e., bit 13 of the status register must be set) at the beginning of the RTE instruction. The RTE instruction changes all the bits of the status register, so the processor might be in user mode at the *completion* of the instruction.

Condition Codes Affected:

The condition codes are all loaded from the word at the top of the stack.

Assembler Syntax: RTE

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	(4E73 hex)

Example:

```
PC=00000868 USP=0001598C SSP=0007BF08 ST=2004=>SUP IM=0 ZER
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0007BF08
pea $876
```

The following code pushes a new status register contents (with all the condition code bits set), and a new PC. Since the PC is lower on the stack than the status register, it must be pushed first.

Here is a disassembly of the program:

```
-l868,876
  00000868 pea $876
```

```
0000086E move #$1F,-(A7)
00000872 rte
00000874 moveq #$FF,D0
00000876 nop
```

Executing this program yields the following results:

```
-g,876
PC=00000876 USP=0001598C SSP=0007BF08 ST=001F=>IM=0 EXT NEG ZER OFL CRY
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

Notice that the breakpoint was set on the address that the RTE instruction loads into the PC. (The debugger will not trace RTE instructions.)

This example shows how a supervisor mode program can transfer control to a user mode program. Note how register A7 reflects the supervisor stack pointer (SSP) before the RTE and the user stack pointer (USP) afterward.

RTR Instruction

The RTR (ReTurn and Restore) instruction loads the condition codes and the program counter (PC) from the stack. The condition codes are loaded from the low byte from the word at the top of the stack. The high byte of this word is discarded. The PC is loaded from the long word immediately after the word containing the condition codes. The stack pointer is incremented by six by an RTR instruction.

Condition Codes Affected:
The condition codes are loaded from the first word popped from the stack.

Assembler Syntax: RTR

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1	(4E77 hex)

Example:

```
PC=00000888 USP=00015986 SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00015986
rtr
```

At this point, we are about to execute an RTR instruction. Using the *s* command, we will examine the information on the stack.

```
-sw15986          (Contents of A7)
00015986 001F     (New condition codes)
00015988 0000     (High word of PC)
0001598A 088C .   (Low word of PC)
-t
PC=0000088C USP=0001598C SSP=0007BF08 ST=001F=>IM=0 EXT NEG ZER OFL CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

The stack pointer was incremented by six bytes and the PC and status register now have different contents.

RTS Instruction

The RTS (ReTurn from Subroutine) instruction reverses the action of a BSR (Branch to SubRoutine) or JSR (Jump to SubRoutine) instruction. The PC is loaded from the long word at the top of the stack. This causes execution to resume at the instruction that follows the JSR or BSR instruction.

Condition Codes Affected: None

Assembler Syntax: RTS

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1	(4E75 hex)

Example:

```
PC=0000089A USP=00015988 SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00015988
rts
-sl15988
00015988 0000089E .
-t
PC=0000089E USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This RTS instruction causes a transfer to address 89E. The stack pointer is incremented by four bytes (which is the size of the address that was popped off the stack).

SBCD Instruction

The SBCD (Subtract BCD with extend) instruction subtracts two bytes in BCD format. The destination operand is replaced with the difference (Destination − Source − Extend bit).

Addressing Modes Allowed:

Dn	An	(An)	(An) +	− (An)	x(An)	x(An,xr.s)
Yes	No	No	No	Yes	No	No
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
No	No	No	No	No	No	No

There are two forms of this instruction:

1. Subtract data register from a data register (Dn addressing modes). The low-order bytes of two data registers are subtracted and the result stored in the destination register.
2. Subtract memory to memory. This form of the instruction allows multiple bytes to be subtracted. The only valid addressing mode is −(An). Since the 68000 stores BCD data with the highest byte first, to subtract multibyte quantities, one must start at the highest address and work down. (Hence the use of pre-decrement addressing.) If there is a carry out of the most significant BCD digit in the byte, each instruction sets the X bit. The X bit is then subtracted from the next pair of bytes.

Data Sizes: byte only

Condition Codes Affected:

X Set by a borrow-out of the most significant BCD digit.
N Undefined.
Z Cleared if result is not zero. Unchanged otherwise.

V Undefined.
C Set by a borrow-out of the most significant BCD digit.

The Z bit is cleared if the result is not zero. Not setting the bit when the result of the present byte is zero allows the Z bit to be accurate after a series of SBCD instructions is executed. The Z bit must be set before beginning such a series. (Comparing a register to itself is an easy way to set the Z bit.) The N and V bits are undefined as a result of this instruction.

Assembler Syntax: SBCD Dx,Dy
SBCD −(Ax),−(Ay)

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	0	D. Reg			1	0	0	0	0	F	S. Reg.		

The D. Reg and S. Reg fields specify the destination and source register numbers, respectively. If the F (format) bit is 0, then the registers are data registers. If the F bit is a 1, then the registers are address registers, and the pre-decrement addressing mode is used.

Example:

```
PC=00000510 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000518  0000051A 00000000 00000000 00000000 00000000 00000000 0001598C
sbcd -(A0),-(A1)
-d516,519
00000516 00 01 01 00                                                   ....
-t
PC=00000512 USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000517  00000519 00000000 00000000 00000000 00000000 00000000 0001598C
sbcd -(A0),-(A1)
-d516,519
00000516 00 01 01 99                                                   ....
-t
PC=00000514 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000516  00000518 00000000 00000000 00000000 00000000 00000000 0001598C
-d516,519
00000516 00 01 00 99                                                   ....
```

This example illustrates how to do a multiprecision BCD subtraction operation. A two-byte subtraction operation is executed, subtracting 1 from 100 to obtain 99. The operands are displayed using the *d* command after each step of the process. Notice the action of the borrow.

Scc Instruction

The Scc instruction sets a byte specified by an effective address operand to all ones if a specified condition is true. The byte is cleared if the condition is false. The permissible instructions are:

SCC Set <ea> if the C-bit (carry) is clear.

SCS Set <ea> if the C-bit is set.

SEQ Set<ea> on EQual. The byte is set if the Z- bit is set.

SGE Set <ea> on Greater than or Equal. The byte is set if the N (negative) and V (overflow) bits are either both set or both clear. SGE is used for two's complement binary numbers.

SGT Set <ea> on Greater Than. The byte is set if:

- The N and V bits are both set *and* the Z-bit is clear, or,
- The N, V, and Z-bits are *all* clear.

SHI Set <ea> on HIgher than. The byte is set if the C and Z bits are both clear. SHI is similar to SGT, except it works on unsigned numbers.

SLE Set <ea> on Less than or Equal. The byte is set if:

- The Z-bit is set, or,
- The N-bit is set *and* the V-bit is clear, or,
- The N-bit is clear *and* the V-bit is set.

The SLE instruction is used for two's complement binary numbers.

SLS Set <ea> on Lower or Same.
The byte is set if either the C or Z bits are set. SLS is similar to the SLE instruction, except it works on unsigned numbers.

SLT Set <ea> on Less Than. The byte is set if:

- The N-bit is set and the V-bit is clear, or,
- The N-bit is clear and the V-bit is set.

SMI Set <ea> on MInus. The byte is set if the N-bit is set.

SNE Set <ea> on Not Equal. The byte is set if the Z-bit is clear.

SPL Set <ea> on Plus. The byte is set if the N-bit is clear.

SVC Set <ea> on V Clear. The byte is set if the V-bit is clear, indicating no overflow.

SVS Set <ea> on V Set. The byte is set if the V-bit is set, indicating overflow.

SF Never set <ea>.

ST Always set <ea>.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An)+**	**–(An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Size: byte

Condition Codes Affected: None

Assembler Syntax: Scc <ea>

Machine Code Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Condition				1	1	Effective			Address		
										← Mode →			← Reg. →		

The Condition is a four-bit encoding of the condition code combination. The conditions are as follows:

Condition	Instruction	Condition	Instruction
0000	ST	1000	SVC
0001	SF	1001	SVS
0010	SHI	1010	SPL
0011	SLS	1011	SMI
0100	SCC	1100	SGE
0101	SCS	1101	SLT
0110	SNE	1110	SGT
0111	SEQ	1111	SLE

Example:

```
PC=000008BC USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
cmp.l D0,D1
-t
PC=000008BE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
slt D2
-t
PC=000008C0 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000002 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
sgt D2
-t
PC=000008C2 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000002 000000FF 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example illustrates the use of the Scc instruction with a compare instruction. Since the value in register D1 is greater than the value in register D0, the SLT instruction *did not* set D2, and the SGT instruction *did* set D2. With the Scc instruction, as with the Bcc and DBcc instructions, the operands of a compare instruction must be read in reverse order.

STOP Instruction

The STOP instruction provides a way to simultaneously enable interrupts and to wait for an interrupt to occur. Other processors, notably the PDP-11, had separate instructions for enabling interrupts and for waiting for interrupts. An interrupt between the enabling instruction and the waiting instruction could result in waiting for an interrupt that has already occurred.

The STOP instruction is a privileged instruction and is used only by code that must service interrupts. The Supervisor bit in the status register must be set at the beginning of the instruction. The contents of a 16-bit immediate data field are loaded into the Status Register. Bit 13 (which corresponds to the S-bit in the status register) of the immediate data must be set or a privilege violation exception will occur. See Chapter 7 on Exception Processing for additional information.

Addressing Modes Allowed: None

Data Size: unsized

Condition Codes Affected:
The condition codes are set from bits 5–0 of the immediate operand.

Assembler Syntax: STOP #<data>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0	(4E72 hex)

Immediate Data (16 bits)

SUB Instruction

The SUB (SUBtract binary) instruction subtracts a source operand from a destination operand and stores the result in the destination operand. There are two forms of this instruction:

1. Subtract an effective address operand from a data register.
2. Subtract a data register from an effective address operand.

Addressing Modes Allowed:

All Addressing modes (except SR and CCR) are allowed when the effective address specifies a source operand. When the effective address field is the destination, then the following addressing modes are allowed:

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
No	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Using a data register as a destination must be accomplished using the register destination form of the instruction.

Data Sizes: byte, word, long

Using an address register as the source is valid only for word and long data lengths.

Condition Codes Affected:

- X Set by borrow out of most significant bit. Cleared otherwise.
- N Set if high-order bit of result was 1. Cleared otherwise.
- Z Set if result is zero. Cleared otherwise.
- C Set by borrow-out of most significant bit. Cleared otherwise.
- V Set if operation resulted in overflow condition. Cleared otherwise.

Assembler Syntax: SUB Dn,<ea>
SUB <ea>,Dn

The SUBA instruction is used when the destination is an address register. The SUBI and SUBQ instructions are used when the source is immediate data. Many assemblers will accept the SUB mnemonic for these instructions, and choose the correct instruction based on the operands.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	1	Register			D	Size		Effective			Address		
											← Mode →			← Reg. →		

The Register field gives the data register that must be one of the operands. The D bit is 0 if the Register field is the destination operand, and 1 if the effective address is the destination.

The Size field is 00 for byte, 01 for word, and 10 for long operands.

Example:

```
PC=000008CE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000002  00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
sub.l D1,D0
-t
PC=000008D0 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000001 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This instruction subtracts 1 from 2 to form 1. When both operands are data registers, the data register destination form of the instruction is used.

SUBA Instruction

The SUBA instruction does a binary subtraction operation with an address register destination. In order to allow address computations to be freely intermixed with data operations, this instruction does not affect the condition codes.

The source operand is subtracted from the address register. The result is placed in the address register.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	Yes	Yes	Yes	No	No

All addressing modes are allowed, except SR and CCR. The effective address must be the source operand.

Data Sizes: word, long

The SUBA operation always affects all 32 bits of the destination address register.

Condition Codes Affected: None

Assembler Syntax: SUBA <ea>,An

Many assemblers will generate a SUBA instruction if a SUB instruction is specified with an address register as the destination operand.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	1	Register			S	1	1	Effective Address					
											← Mode →			← Reg. →		

The Register field gives the address register to be used as the destination operand. The S-bit is 1 for long operands and 0 for word operands.

Example:

```
PC=000008D8 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 000008DC  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
suba #$22,A0
-t
PC=000008DC USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 000008BA  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This example subtracts a constant 22 hex from the address contained in address register A0.

SUBI Instruction

The SUBI instruction subtracts an immediate quantity from an effective address operand. The result is left in the effective address operand.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Set on borrow out of high-order bit. Cleared otherwise.

N Set if high-order bit of result is set. Cleared otherwise.

Z Set if result is zero. Cleared otherwise.

C Set on borrow out of high-order bit. Cleared otherwise.

V Set on overflow condition. Cleared otherwise.

Assembler Syntax: SUBI #x,<ea>

Most assemblers automatically choose the SUBI instruction if the source operand of a SUB instruction is immediate.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	1	0	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Word data: 16 bits	Byte data: 8 bits

Long data: 32 bits, including previous word

The Size field is 00 for byte operands, 01 for words, and 10 for longs.

Example:

```
PC=000008E4 USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000100  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
subi.l #$10,D0
-t
PC=000008EA USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 000000F0  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001598C
```

This instruction subtracts the constant 10 hex (16 decimal) from data register D0.

SUBQ Instruction

The SUBQ instruction subtracts a three-bit immediate value from an effective address operand. This allows you to subtract a small number from a register or memory address using a small, fast instruction.

Addressing Modes Allowed:

Dn	An	(An)	(An) +	– (An)	x(An)	x(An,xr.s)
Yes	Yes	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long
When an address register is used as the destination, only word and long sizes are allowed.

Condition Codes Affected:

X Set on borrow out of high-order bit position. Cleared otherwise.
N Set if high-order bit of result is set. Cleared otherwise.
V Set on overflow. Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
C Set on borrow out of high-order bit position. Cleared otherwise.

No condition codes are affected if an address register is used as the destination operand.

Assembler Syntax: SUBQ #<data>,<ea>
#<data> is a constant number in the range 1 to 8.

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	1	Data			1	Size		Effective			Address		
											← Mode →			← Reg. →		

Data is a three-bit immediate field, with 000 representing 8, 001–111 representing 1–7. Size is 00 for byte operations, 01 for word, and 10 for long operations.

Example:

```
PC=000008EE USP=0001598C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
subq.l #$2,D0
-t
PC=000008F0 USP=0001598C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D FFFFFFFF  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001598C
```

This example subtracts 2 from 1 (in data register D0) to form -1.

SUBX Instruction

The SUBX (SUBtract eXtended) instruction executes multiple precision subtraction operations. Integers of any length can be subtracted using the SUB and SUBX instructions. This makes it possible to represent numbers much larger than the 32-bit longword allows.

There are two forms of this instruction:

1. Subtract a data register from a data register.
2. Subtract a memory location from a memory location. The −(An) addressing mode is used for both the source and destination in this form.

In both cases, the difference (Destination − Source − X-bit) is placed in the destination operand.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An)+**	**−(An)**	**x(An)**	**x(An,xr.s)**
Yes	No	No	No	Yes	No	No
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
No	No	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

X Set on borrow out of high-order bit. Cleared otherwise.
N Set if result is negative. Cleared otherwise.
Z Cleared if result is not zero. Unchanged otherwise.
C Set on carry out of high-order bit. Cleared otherwise.
V Set on overflow condition. Cleared otherwise.

The Z bit is not set if the result is zero, but it is cleared if the result is not zero. This property of the instruction allows the Z bit to correctly indicate the result of a multiprecision subtraction operation. The Z bit must be set before subtraction begins, however. (This can be done with a MOVE to CCR, or by comparing a register to itself. The latter instruction is two bytes shorter.)

Assembler Syntax: SUBX Dy,Dx
SUBX −(Ay), −(Ax)

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	1	Reg. Rx			1	Size		0	0	T	Reg. Ry		

The Reg. Rx and Reg. Ry fields contain the destination and source register numbers, respectively. The size field is 00 for byte operations, 01 for word, and 10 for long operations. The T (type) bit is 0 for the data register to data register form of the instruction. The Reg. Rx and Reg. Ry fields identify data registers in this case. The T-bit is 1 for the memory to memory form of the instruction. The Rx and Ry fields identify the address registers used by the pre-decrement addressing mode for this form.

Example:

```
PC=000008FE USP=0001558C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000001  00000000 00000000 00000001  00000000 00000000  00000000 00000000
A 00000000  00000000 00000000 00000000  00000000 00000000  00000000 0001558C
subx.l D3,D1
-t
PC=00000900 USP=0001558C SSP=0007BF08 ST=0019=>IM=0 EXT NEG CRY
D 00000001  FFFFFFFF 00000000 00000001  00000000 00000000  00000000 00000000
A 00000000  00000000 00000000 00000000  00000000 00000000  00000000 0001558C
subx.l D2,D0
-t
PC=00000902 USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 00000000  FFFFFFFF 00000000 00000001  00000000 00000000  00000000 00000000
A 00000000  00000000 00000000 00000000  00000000 00000000  00000000 0001558C
```

This operation subtracts the register pair (D2,D3) from the register pair (D0,D1). The high-order longword of each pair is contained in the even numbered register. The example shows that (1,0) − (0,1) is (0,FFFFFFFF).

SWAP Instruction

The SWAP instruction exchanges the 16-bit words in a data register. Bits 31–16 are exchanged with bits 15–0.

Addressing Modes Allowed: Dn only

Data Size: word

Condition Codes Affected:

X Not affected.
N Set if bit 31 of the result is set. Cleared otherwise.
Z Set if all 32 bits of the register are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Assembler Syntax: SWAP Dn

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	0	0	0	1	0	0	0	Register		

The Register field specifies which data register is to be swapped.

Example:

```
-x
PC=0000090A USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 11112222  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
swap D0
-t
PC=0000090C USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 22221111  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
```

This example swaps the words in data register D0.

TAS Instruction

The TAS (Test And Set) instruction tests a byte specified by an effective address operand. The high-order bit of the byte is set to 1. The N- and Z-bits are set according to the value of the byte *before* the operation. The operation is indivisible, using a read-modify-write memory operation.

The TAS operation provides synchronization when two or more CPU chips have access to the same area of memory. Since TAS is indivisible, a processor can claim a resource and mark it as claimed before another processor can test the memory location. If the operation were not indivisible, two processors could test the flag and set it in such a way that they both assess the resource as free and claim it erroneously. The TAS instruction guarantees that one processor will win and all others lose.

Addressing Modes Allowed:

Dn	An	(An)	(An)+	−(An)	x(An)	x(An,xr.s)
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

A TAS operation on a data register (which is allowed) is of no value for synchronization purposes.

Data Size: byte

Condition Codes Affected:

X Not affected.
N Set if the high-order bit of the operand is set prior to the operation. Cleared otherwise.
Z Set if all bits of the operand are zero prior to the operation. Cleared otherwise.
V Always cleared.
C Always cleared.

Assembler Syntax: TAS <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	1	0	1	1	Effective			Address		
											← Mode →			← Reg. →		

Example:

```
PC=0000090C USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
tas $97C
-s97c
0000097C 00 .
-t
PC=00000912 USP=0001558C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
-s97c
0000097C 80 .
```

This example executes a TAS operation on memory location 97C. The memory location contained zero before the operation and 80 hex afterward. The Z-bit is set, indicating that the operand *was* zero initially.

TRAP Instruction

The TRAP instruction stacks the PC and the status register on the supervisor mode stack. The Processor is switched to supervisor state, and the PC is taken from one of sixteen trap vectors specified by a four-bit quantity in the TRAP instruction.

This instruction is normally used by user mode programs to call supervisor mode programs (such as operating systems). The TRAP instruction provides a method for the user mode program to request an operating system function, such as I/O, without having to know where the operating system is located in memory.

Condition Codes Affected: None

Assembler Syntax: TRAP #<vector>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	0	0	Vector				(4E4x hex)

Vectors used by the trap instruction are located at the following absolute memory locations:

Vector	Address	Vector	Address
0	80	8	A0
1	84	9	A4
2	88	10	A8
3	8C	11	AC
4	90	12	B0
5	94	13	B4
6	98	14	B8
7	9C	15	BC

TRAPV Instruction

The TRAPV instruction tests for overflow. The TRAPV instruction does nothing if the V-bit is clear. If the V-bit is set, the PC and the status register are pushed onto the stack, and a new PC is loaded from absolute location 1C hex. The CPU is switched into Supervisor state. This action is called a TRAPV exception.

The TRAPV instruction is used after computations in which an overflow condition would result in meaningless data. Many high-level languages use this instruction to detect overflow.

Data Size: unsized

Condition Codes Affected: None

Assembler Syntax: TRAPV

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0	(4E76)

TST Instruction

The TST instruction tests an effective address operand for negative or zero. The results are not saved, except that the condition codes are set appropriately.

Addressing Modes Allowed:

Dn	**An**	**(An)**	**(An) +**	**– (An)**	**x(An)**	**x(An,xr.s)**
Yes	No	Yes	Yes	Yes	Yes	Yes
x.w	**x.l**	**x(PC)**	**x(PC,xr.s)**	**#x**	**SR**	**CCR**
Yes	Yes	No	No	No	No	No

Data Sizes: byte, word, long

Condition Codes Affected:

- X Not affected.
- N Set if the high-order bit of the operand is set. Cleared otherwise.
- Z Set if the operand is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Assembler Syntax: TST <ea>

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	1	0	Size		Effective			Address		
											← Mode →			← Reg. →		

Size is 00 to test a byte, 01 for a word, and 10 for a long.

Example:

```
PC=00000918 USP=0001558C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000001  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
tst.l D0
-t
PC=0000091A USP=0001558C SSP=0007BF08 ST=0000=>IM=0
D 00000001  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
tst.l D1
-t
PC=0000091C USP=0001558C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000001  00000000  00000000  00000000  00000000  00000000  00000000  00000000
A 00000000  00000000  00000000  00000000  00000000  00000000  00000000  0001558C
```

This example shows two TST instructions: one on a register that is non-zero and one on a register that is zero.

UNLK Instruction

The UNLK (UNLinK) instruction frees a stack frame that was allocated previously by a LINK instruction. (See the description of the LINK instruction.) The instruction works like this:

The specified address register (normally the frame pointer) is placed in the stack pointer. A long word is then popped off the stack into the address register. This is exactly the opposite of the action of the LINK operation. The UNLK instruction functions properly regardless of stack PUSHes and POPs between the LINK and UNLK instructions.

Data Size: unsized

Condition Codes Affected: None

Assembler Syntax: UNLK An

Machine Code Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	1	1	1	0	0	1	0	1	1	Register		

The Register field is the address register specified as the frame pointer.

Example:

```
PC=0000091E USP=0001558C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
link A0,#$FFF4
-t
PC=00000922 USP=0001557C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00015588  00000000 00000000 00000000 00000000 00000000 00000000 0001557C
-sl15588
00015588 00000000 . (Old A0 contents)
PC=00000924 USP=0001557C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00015588  00000000 00000000 00000000 00000000 00000000 00000000 0001557C
unlk A0
```

```
-t
PC=00000926 USP=0001558C SSP=0007BF08 ST=0004=>IM=0 ZER
D 00000000  00000000 00000000 00000000 00000000 00000000 00000000 00000000
A 00000000  00000000 00000000 00000000 00000000 00000000 00000000 0001558C
```

This example shows a typical pair of LINK and UNLK instructions. The LINK instruction pushes sixteen bytes (four for the address register and twelve specified by the link instruction), which are in turned popped by the UNLK instruction.

SUMMARY

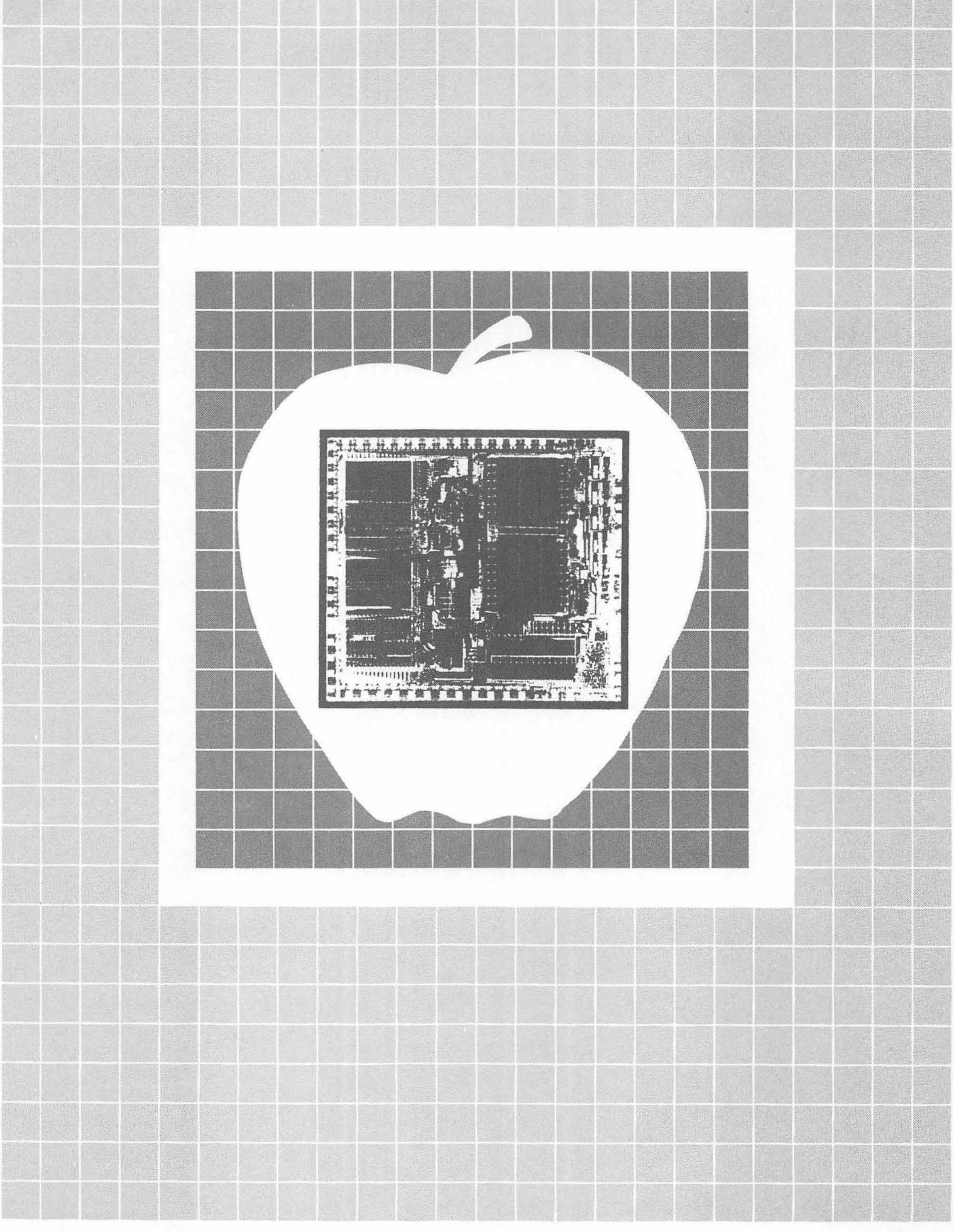
In this chapter we have covered:

1. The 68000 instruction classes
2. Program development mechanics using the Macintosh MDS software
3. The 68000 instruction set

This material is primarily for reference in later chapters. It is important that you at least know how to generate a program on your system before continuing, however.

EXERCISES

1. Use your own computer to run the program shown in Listing 3.1.
2. Learn how to make backup copies of your files on your system.
3. Why do the ADDX and SUBX instructions use the pre-decrement (−(An)) addressing mode while CMPM uses the post-increment ((An)+) addressing mode?
4. Why won't the debugger trace RTE instructions? What other instructions will the debugger fail to trace?



Chapter 4

Simple Macintosh Programs

Nothing is particularly hard if you divide it into small jobs.

—Henry Ford

INTRODUCTION

Now that we've been over the necessary background material, we can start to write Macintosh programs in earnest. Computer programming is the art of combining a small number of simple concepts to produce something both innovative and functional, much as the artist combines oils and canvas to produce a painting. The materials of computer programming are relatively commonplace and simple. What you can create with them is limited only by your own ability and imagination.

To make the process of learning to program a little easier, we have provided standardized software in the form of a macro library, listed in Appendix E. This software makes it possible to learn part of the system without having to learn immediately how Macintosh applications manage windows, files, and other parts of the system. Once you learn how these macros work and how they are used in the programs in this chapter, you can modify them, rewrite them, or create more.

THE HELLO WORLD PROGRAM

Perhaps the simplest computer program is one that displays a short message on the screen. This program was christened the "Hello World" program by Kernighan and Ritchie in *The C Programming Language* (Prentice Hall, 1978). A BASIC version of the program looks like this:

```
10 PRINT "Hello, World"
20 END
```

To write the same program in assembly language on the Macintosh, you need the Macintosh Development System (or an alternative) and three separate files: a source file (Hello.asm), a link file (Hello.link), and an EXEC job file (Hello.job) to assemble and link the program. These files are shown in Figure 4.1.

The Hello.asm File

The file in the window labeled Hello.asm is the assembly language source file. The lines beginning with semicolons are comments and have no effect on the program's execution. All of the lines in this program, with the exception of the INCLUDE directive, are macro calls. The INCLUDE

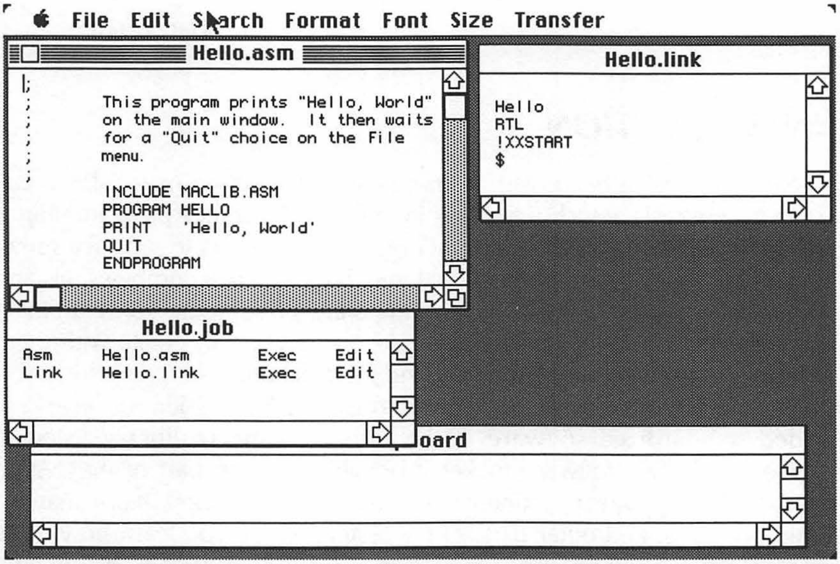


Figure 4.1 – *Hello World files*

directive references a file containing macro definitions, which in turn contain macros for each of the other program statements.

The PROGRAM macro is normally the first line of code in the programs we will be writing for the rest of this book. The PROGRAM macro contains the first instructions that the program must execute. The first operand for this macro appears as the title for the window created when the program is run. The second operand, if present, is either DEBUGON or DEBUGOFF. This operand determines whether an illegal instruction exists in the program just before the first instruction following the PROGRAM macro. (In Hello.asm, this would be just before the PRINT statement.

The PRINT macro functions like a PRINT statement in BASIC, except that the macro version prints only a single string. The string can be specified as an operand (contained in single quotes) or as a label on a DC.B statement that specifies a string to be printed. The PRINT macro expects the address specified as the operand to contain a byte indicating the number of characters to be printed. The string immediately follows this count byte. The assembler generates this byte automatically for strings in single-quote characters.

The QUIT macro causes the program to wait for a Quit selection from the File menu. After Quit has been selected, the program returns to the Finder.

The program must end with an ENDPROGRAM macro call. An END statement may follow ENDPROGRAM if desired.

The Hello.link File

The linker command file for the Hello World program is called Hello.link. This file links two .REL files together: Hello.rel and RTL.REL. Hello.rel is the result of assembling Hello.asm, and RTL.REL is a file which contains code that is the same from one application to the next. The source code for this file is explained in detail in Chapter 8.

The Hello.job File

The Hello.job file contains the commands that assemble and link the Hello World program. A quirk in some early versions of EXEC required that the fields on each line be separated by a single tab character.

Running the Hello World Program

In order to run this program, create a disk containing the files MACLIB-.ASM, RTL.REL, and RTL.RSRC. See Appendix E for instructions on creating these files. Boot from the disk supplied with the Macintosh Development System that contains the assembler, linker, and EXEC programs. Put the disk you just made into the other drive. Start the editor and create the files shown in Figure 4.1 on the new disk.

To build the executable file, start by choosing EXEC from the editor's Transfer menu. Then choose Open Job File from EXEC's File menu. Choose Hello.job from the file window that appears. When the assembler and linker finish execution, you will be in the EXEC program. Choose Open Application from the File menu, and then choose Hello from the file window that appears.

When Hello executes, the screen will look like Figure 4.2. Choose Quit from the File menu to return to the Finder.

All of the programs in this book require a similar build procedure. You can use the files just created and edit them as necessary to make files for subsequent programs.

OUTPUT CONVERSION ROUTINES

Most programs must output numeric quantities to provide the user with computational results. To output numeric quantities, you have to translate the binary numbers into character format. This is normally accomplished

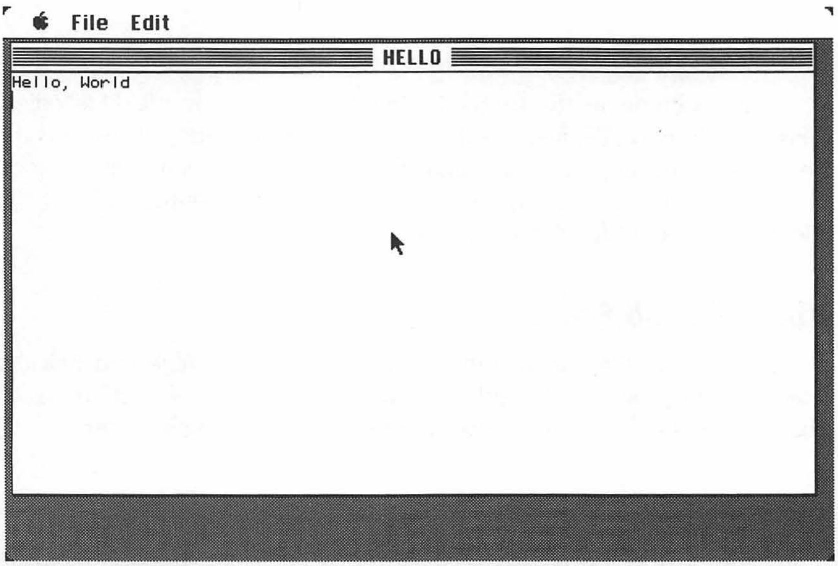


Figure 4.2 *– Hello World execution*

using subroutines called *conversion routines.* We will develop two such routines next: one for hexadecimal conversion and one for decimal conversion.

Converting Binary to Hexadecimal

Perhaps the simplest conversion to write is binary to hexadecimal. Our routine is called BinHex and is shown in Listing 4.1.

This routine converts a binary longword to eight ASCII characters, which are a printable hexadecimal number. The technique employed is to take four bits at a time and index them into a string of printable hex characters, placing the resulting character into the output area.

Lines 10–13 are initialization code. Register D0 contains the number to be converted when the routine is called. Register A0 contains the address of an eight-byte memory area that will receive the converted number. This area is filled from right to left, to minimize shifting. Register D1 contains a loop count. This value is 7 to obtain eight hex digits. (Remember, the DBRA instruction terminates the loop when the count value becomes −1.)

Lines 14–18 form the conversion loop. In lines 14–15, the low-order four

```
0001  ;
0002  ;        BinHex routine.  Converts a longword to ASCII hex.
0003  ;
0004  ;        Enter with:
0005  ;
0006  ;           D0.L =  Binary value
0007  ;           A0.L -> 8-byte output area
0008  ;
0009           XDEF     BinHex
0010  BinHex:  MOVEM.L  D0-D2/A1,-(SP)   ;     Save caller's registers
0011           LEA      HexTab,A1        ;     A1 -> Table of hex digits
0012           ADDA.L   #8,A0            ;     A0 -> 1 past last digit position
0013           MOVE.L   #7,D1            ;     DBRA-Adjusted loop counter
0014  Loop:    MOVE.L   D0,D2            ;     Copy present number
0015           ANDI.L   #15,D2           ;     Isolate low-order 4 bits
0016           MOVE.B   0(A1,D2),-(A0)   ;     Place a byte in output area
0017           LSR.L    #4,D0            ;     Shift away bits just used
0018           DBRA     D1,Loop          ;     Loop until done
0019           MOVEM.L  (SP)+,D0-D2/A1   ;     Restore registers
0020           RTS                       ;     Return to caller
0021           STRING_FORMAT   0         ;     Non-Pascal string format
0022  HexTab:  DC.B     '0123456789ABCDEF'
```

Listing 4.1 *– The BinHex routine* bits of register D0 are copied to register D2. The ANDI instruction guarantees that D2 will contain a value from 0–15. The MOVE.B instruction at line 16 places the proper ASCII value into the buffer. Lines 17–18 position the next hex digit in the low-order four bits of D0 and repeat the loop.

Note the STRING_FORMAT directive at line 21. The Macintosh assembler normally treats characters between single quotes as Pascal language strings. When it comes across single quotes, the assembler inserts a one-byte character count at the beginning of the string. The STRING_FORMAT directive instructs the assembler not to insert this byte for subsequent strings.

Converting Binary to Decimal

Since decimal numbers are used more frequently than hex numbers, the ability to convert decimal numbers is more important than the ability to convert hex numbers. However, it is harder to convert from binary to decimal numbers because you must divide the binary numbers by 10 to determine each new digit. The process is further complicated by the fact that the 68000 has no 32-bit division instruction. Our binary to decimal converter therefore works only on 16-bit numbers. Fortunately, this is usually adequate. The routine is called BinDec and is shown in Listing 4.2.

You call BinDec with the number to be converted in the low-order word of D0, and the address of a six-byte area that will receive the converted number in A0. Lines 11–15 handle negative numbers. If D0 is negative, a minus sign is placed in the buffer and D0 is negated, producing a positive number. Lines 17–18 form the initialization code for the conversion loop. A0 is advanced to point one byte beyond the conversion area, and D1 contains the loop count (adjusted for the DBRA instruction).

The main body of the routine (lines 19–24) works as follows:

1. Extend D0.W to a longword to clear the high word of the register. Divide this by 10.
2. Place the remainder (in the high word of D0) in the buffer. Add the ASCII code for "0" (hex 30) to this byte, yielding a character between 0 and 9.
3. Use the quotient from step 1 as the new number, and repeat until five digits have been processed.

The output of this routine is a little crude. Possible improvements include adding a floating minus sign and suppressing leading zeros. Despite the lack of sophistication, this routine is adequate for the programs in the rest of this chapter.

SUMMING THE FIRST FIVE INTEGERS

We can use the routines just developed to rewrite the program that sums the first five integers, used as the debugger example in Chapter 3. The first improvement we will make is to display the answer on the Macintosh screen with the PRINT macro. We can also use the DBRA instruction to make the program considerably shorter, as shown in Listing 4.3.

Lines 10–15 form the sum of the first five integers in register D0. Lines 19–22 convert this number to hex and decimal at the data areas labeled Hex and Dec respectively. Finally, line 22 prints the answer on the screen, as shown in Figure 4.3.

Note how the count byte for the PRINT macro is calculated in line 26. Remember that the STRING_FORMAT directive instructs the assembler not to insert the count byte in front of quoted strings.

It is important to notice how much easier it is to write this program given the conversion routines developed earlier in this chapter. We were

```
;
;       BinDec routine.  Converts binary to decimal ASCII.
;
;       Enter with:
;
;          D0.W =  Number to convert
;          A0.L -> 6-byte output area
;
        XDEF    BinDec
BinDec: MOVEM.L D0-D1/A0,-(SP)   ;     Preserve registers
        MOVE.B  #' ',D1          ;     Assume positive number
        TST.W   D0               ;     Is number negative?
        BPL     NotNeg           ;     No, use ' ' sign
        MOVE.B  #'-',D1          ;     Number is negative, use '-'
        NEG.W   D0               ;     Force positive
NotNeg: MOVE.B  D1,(A0)+         ;     Put sign in buffer
        ADDA.L  #5,A0            ;     A0 -> 1 beyond last digit
        MOVE.W  #4,D1            ;     DBRA-Adjusted loop count
Loop:   EXT.L   D0               ;     Extend to long for DIV instruction
        DIVS    #10,D0           ;     Divide by 10
        SWAP    D0               ;     Remainder to low-order word
        MOVE.B  D0,-(A0)         ;     Put in buffer (0-9 binary)
        ADD.B   #'0',(A0)        ;     Add ASCII offset (0-9 ASCII)
        SWAP    D0               ;     Put quotient in low word now
        DBRA    D1,Loop          ;     Do 5 digits
        MOVEM.L (SP)+,D0-D1/A0   ;     Restore registers
        RTS                      ;     Return to caller
```

***Listing 4.2** – The BinDec routine*

able to treat each conversion process as two instructions, rather than the 20–30 instructions that the routines actually require. Indeed, we don't even have to know how the routines work, only how to call them.

READING KEYBOARD INPUT

Many programs require user interaction or input to function properly. Examples of this type of software include editor programs, calculator programs, game programs, spreadsheets, and database managers.

Keyboard input on the Macintosh can be accomplished using the INPUT macro, contained in the macro library file mentioned earlier. This macro reads a line of characters from the keyboard and stops reading when the Return key is pressed. The characters are stored at an address

```
;
;        This program computes the sum of the first 5 integers,
;        and prints the result on the Macintosh screen.
;
         XREF    BinHex          ;      Hex conversion routine
         XREF    BinDec          ;      Decimal conversion routine
         Include MACLIB.ASM      ;      Include RTL macros

         PROGRAM Sum5
         CLR.L   D0              ;      Clear accumulator
         MOVE.L  #1,D1           ;      Initialize counter
         MOVE.L  #4,D2           ;      DBRA-Adjusted loop counter
Loop:    ADD.L   D1,D0           ;      Add to sum
         ADD.L   #1,D1           ;      Increment integer
         DBRA    D2,Loop         ;      Loop till done
;
;        Now convert and print the answer
;
         LEA     Hex,A0          ;      A0 -> Hex area
         JSR     BinHex          ;      Do the conversion
         LEA     Dec,A0          ;      A0 -> Decimal area
         JSR     BinDec          ;      Do the conversion
         PRINT   Answer          ;      Print answer on screen
         QUIT                    ;      and quit
         STRING_FORMAT   0       ;      No count bytes
Answer:  DC.B    EndAnswer-*     ;      Compute count byte
         DC.B    'Sum Is: '      ;      First part of string
Hex:     DC.B    'xxxxxxxx (Hex) or ' ; Hex part of string
Dec:     DC.B    'xxxxxx (Decimal)'   ; Decimal part of string
EndAnswer:                       ;      Label for string end
         ENDPROGRAM
```

***Listing 4.3** – The Sum5 program*

specified as the first parameter in the macro call. The second parameter, if present, is the number of characters in the buffer. If this parameter is not present, a default buffer length of 80 characters is assumed. The INPUT macro sets the first byte in the buffer to the number of characters actually read. This count does not include the Return character, nor is the Return character stored in the buffer. An empty line (that is, one consisting only of a Return) causes a count byte of zero to be stored. The count byte is not included in the character count, so the buffer in memory must be one byte larger than the size specified in the macro call, or 81 bytes if the size parameter is omitted.

Use of the INPUT macro is demonstrated in Listing 4.4. This program echoes a line typed by the user back to the screen.

This program introduces two new macro features. The second parameter to the PRINT macro, if not blank, specifies that the cursor not be advanced to the next line after printing. This is useful for prompting the user for input and for building a line from several PRINT calls. The second feature is the EXIT macro. This macro returns to the Finder directly, without waiting for the user to select Quit from a menu.

This program also uses a new assembler directive. The buffer is defined using a DS (Define Storage) directive. This directive causes the assembler to reserve an area of memory. The memory thus reserved has no particular initial value. The Macintosh assembler requires that memory reserved

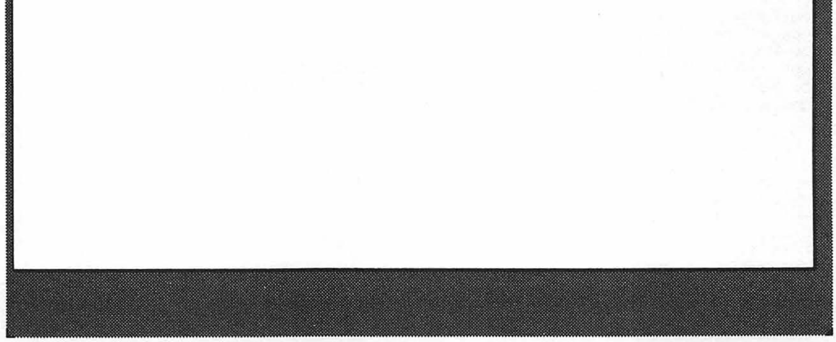


Figure 4.3 *– Sum5 execution*

in this way be addressed using register A5, as in lines 11, 12, and 15. Using the DS directive is advantageous because memory reserved in this way is not stored in the application file on disk. The program thus takes up less disk space and also loads faster. A sample execution of the INPUT program is shown in Figure 4.4.

```
0001  ;
0002  ;      Input.Asm.  This program illustrates the use of the INPUT macro.
0003  ;      The user is prompted for a line of input, and the input is then
0004  ;      echoed to the screen with a PRINT macro.
0005  ;
0006  ;      The program terminates when the user types an empty line.
0007  ;
0008         INCLUDE MACLIB.ASM                ;      Include macro definitions
0009         PROGRAM INPUT
0010  Loop:  PRINT   'Type a line: ',$         ;      Prompt the user
0011         INPUT   Buffer(A5)                ;      Take the input
0012         TST.B   Buffer(A5)                ;      Any characters?
0013         BEQ     NoChars                   ;      No, quit
0014         PRINT   'You Typed:    ',$        ;      Print
0015         PRINT   Buffer(A5)                ;              the input
0016         BRA     Loop                      ;      Repeat till empty line
0017  NoChars:                                 ;      Here to exit
0018         EXIT                              ;      Back to Finder
0019  Buffer: DS.B   81                        ;      Storage for input
0020         ENDPROGRAM
```

***Listing 4.4** – The Input program*

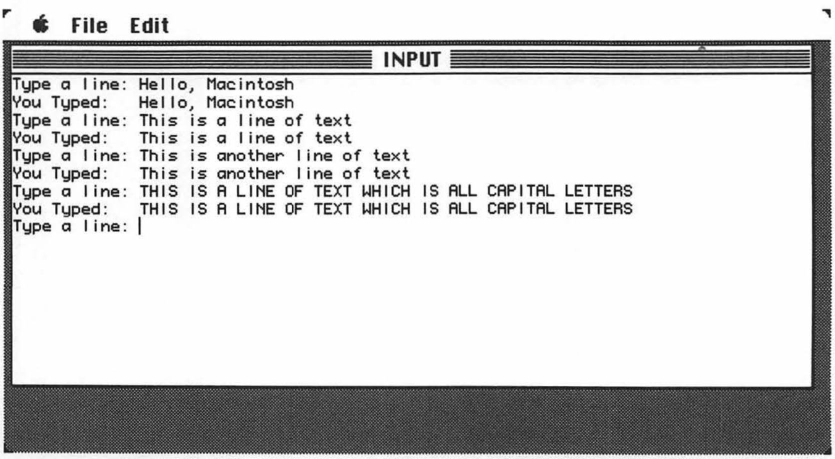


***Figure 4.4** – Input program execution*

Input Conversion

In order to use numeric keyboard input, one must first convert it from ASCII characters to binary. This is the reverse of output conversion.

A routine that converts a decimal ASCII number to binary is shown in Listing 4.5. This routine is named DecBin, and it takes the address of the ASCII string in A0.L. The converted number is returned in D0.L.

DecBin works using a technique called an *accumulator variable* (which is register D0 in this case). The input is processed from left to right (starting with the most significant digit). For each digit processed, the accumulator is multiplied by 10 and the digit added. (This obviates the need to assign a place value to the first digit encountered.) Processing stops when

```
;
;        DecBin routine.  Converts decimal ASCII to longword binary.
;
;        Enter with:
;
;           A0.L -> Input string
;
;        Exit with:
;
;           D0.L =  Converted number
;
;        Conversion stops at the first nondecimal digit.  No
;        overflow or error detection.
;
         XDEF     DecBin
DecBin:  MOVEM.L  D1/A0,-(SP)      ;      Preserve registers
         CLR.L    D0               ;      Clear accumulator
Loop:    CMP.B    #'9',(A0)        ;      Check upper bound
         BHI      NotDec           ;      Not a decimal digit
         CMPI.B   #'0',(A0)        ;      Check lower bound
         BLO      NotDec           ;      Not a decimal digit
         LSL.L    #1,D0            ;      Multiply current value x 2
         MOVE.L   D0,D1            ;      Save this value
         LSL.L    #2,D0            ;      Multiply again x 4
         ADD.L    D1,D0            ;      32-bit multiply x 10
         MOVE.B   (A0)+,D1         ;      Fetch digit from memory
         ANDI.L   #15,D1           ;      Isolate low-order 4 bits
         ADD.L    D1,D0            ;      Add to accumulator
         BRA      Loop             ;      Back for next digit
NotDec:  MOVEM.L  (SP)+,D1/A0      ;      Restore registers
         RTS                       ;      Return to caller
```

***Listing 4.5** – The DecBin routine*

a nondecimal digit is encountered. There is no provision for overflow detection or for the calling program to learn how many digits were processed. This routine also does not handle negative numbers.

Lines 18–21 perform a range check to ensure that the next byte in the buffer does, in fact, contain a decimal digit. Lines 22–25 multiply register D0 by 10. This technique is a holdover from machines that had no hardware multiplication feature. You use left shift operations to calculate $2n$ and $8n$. Adding these two quantities yields $10n$. We use this trick because the 68000 hardware multiply instruction does not work on 32-bit quantities. Lines 26–28 convert the digit to binary and add it to the accumulator.

DECIMAL TO HEX CONVERSION PROGRAM

Now that we have both input and output conversion routines, let's use them to write a program that receives a decimal number as input and produces the hexadecimal equivalent. The program should prompt for input, read the number from the keyboard, and print both the original number and its hex equivalent. Listing 4.6 shows the finished conversion program.

Lines 11–14 of this program prompt the user for input and accept a line from the keyboard. Lines 15–20 perform the conversions from decimal ASCII to binary and then back to decimal and hex ASCII. Line 21 prints the answer on the screen.

This program would have to be much longer if all the subroutines that we developed earlier had destroyed register contents, or had taken their inputs in different registers. The program is very short because the subroutines all expect parameters in the same registers. This allows you to place many parameters in registers at one time.

Figure 4.5 shows a sample execution of the conversion program. Note that an input value of 99999 caused an incorrect decimal value. Because of the limitation of the 68000 DIVS instruction, the BinDec routine (see Listing 4.2) works only for 16-bit numbers. The exercises at the end of the chapter suggest a remedy for this bug.

FILE I/O

Understanding how Macintosh disks work is an important part of learning to write Macintosh programs. Information on a disk is kept in collections of bytes known as *files*. Each file consists of three collections of data:

1. Data concerning the file itself: file type, size, creation date, modification date, and file attributes. This data is called the *file information*.

2. A set of bytes that form the file contents. This information is called the file's *data fork.*

3. A secondary set of bytes that normally contain auxiliary information. This area is known as the file's *resource fork.*

Each file on the disk contains all three of these areas, as shown in Figure 4.6. The file information is always present and always occupies a fixed amount of available storage space. Either or both of the resource and data forks can be empty (contain no bytes).

Operations on Files

The most straightforward way to interact with files is to first select a file in an OPEN or CREATE macro. These macros attach the program to an

```
;
;       DecHex program.  This program converts decimal numbers
;       to hex.  Numbers are input from the keyboard and output
;       to the screen.
;
include MACLIB.ASM
        XREF    BinHex          ;       Hex output conversion
        XREF    BinDec          ;       Decimal output conversion
        XREF    DecBin          ;       Decimal input conversion
        PROGRAM DecHex
Loop:   PRINT   'Enter Decimal Number: ',$ ; Prompt for input
        INPUT   InBuff(A5)      ;       Get a line of input
        TST.B   InBuff(A5)      ;       Check for null line
        BEQ     Done            ;       if null, then quit
        LEA     InBuff+1(A5),A0 ;       A0 -> Digits input
        JSR     DecBin          ;       Convert to binary
        LEA     DecBuf,A0       ;       A0 -> Decimal output area
        JSR     BinDec          ;       Convert to decimal ASCII
        LEA     HexBuf,A0       ;       A0 -> Hex output area
        JSR     BinHex          ;       Convert to hex ASCII
        PRINT   OutLine         ;       Type the answer
        BRA     Loop            ;       Do it again
Done:   EXIT                    ;       Back to Finder
        STRING_FORMAT 0         ;       No count bytes
OutLine DC.B    EndLine-DecBuf  ;       Count byte
DecBuf: DC.B    'XXXXXX Decimal is '
HexBuf: DC.B    'XXXXXXXX Hex'
EndLine:                        ;       Tag for end of line
Inbuff: DS.B    81              ;       Input area
        ENDPROGRAM
```

Listing 4.6 *– The DecHex program*

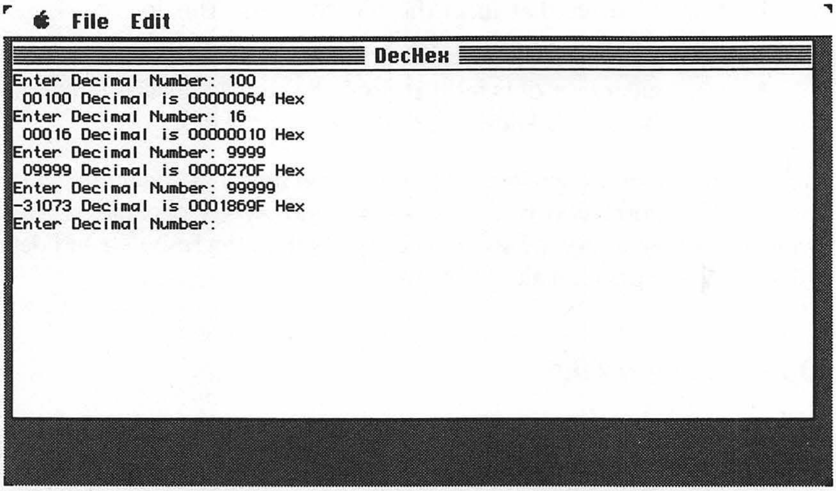


Figure 4.5 *– DecHex execution*

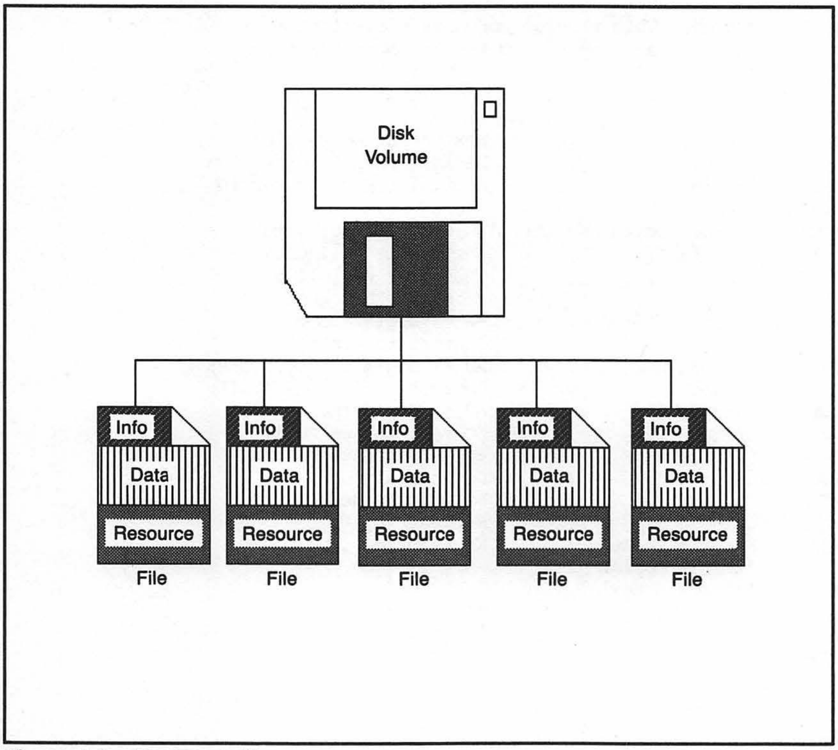


Figure 4.6 *– Macintosh files*

existing or new file, respectively. You can then transfer information from the file into memory with a READ macro, or from memory into the file with a WRITE macro. When you are through with a file, the CLOSE macro disconnects it from your program. The resource and data forks of the file are treated independently. There are separate OPEN macros for each file fork. To access both forks of a file, you must open and process the forks separately.

Normally, files are processed from beginning to end. This is known as *sequential access.* The SEEK macro provides a method for specifying the byte offset within the file at which the next transfer will begin, allowing you to access the information in the file in any order. This capability is known as *random access.*

The GETFINFO and SETFINFO macros allow you to obtain and change a file's information area.

File Information

Associated with each Macintosh file is a data area that contains the *file information.* Figure 4.7 shows the layout of this data area.

The first seven longwords of this area (offsets 0–27) constitute a Macintosh *input/output parameter block.* An I/O parameter block is used for each file operation on the Macintosh. This parameter block is explained in detail in Chapter 8.

Offset 28 contains the *file directory index,* which gives the order in which the file appears in the disk's directory. This field is normally used in advanced applications to reference a file by position rather than by name.

Offset 30 contains the *file attributes,* a set of eight single-bit flags that give the following information on the file's current state:

- If bit 0 is a 1, the file is *locked.* Locked files cannot be altered or deleted.
- Bits 1–6 are unused.
- If bit 7 is a 1, the file is currently open.

Offset 31 is the *file version number.* Different files with the same name must have different version numbers.

The *Finder file type* (offset 32) is a four-byte identifier that the Finder uses to determine the type of file. For example, the Finder file type is APPL for applications and TEXT for textfiles. Files generated by the CREATE macro are textfiles. The *Finder file creator* (offset 36) is a four-byte identifier that indicates the application that will start when the user clicks

Macintosh File Information

Offset	Contents		
0	I/O Link Word		
4	I/O Type	I/O Trap	
8	I/O Command Address		
12	Completion Routine Address		
16	I/O Result Code	File Name Address (High)	
20	File Name Address (Low)	Volume Reference Number	
24	I/O Reference Number	I/O Version	
28	File Directory Index	Attributes	File Version
32	Finder File Type		
36	Finder File Creator (Application Signature)		
40	Finder Flags	Finder Icon Location (High)	
44	Finder Icon Location (Low)	Finder Folder Number	
48	File Number		
52	Data Fork Start Block	Data Fork Logical EOF (High)	
56	Data Fork Logical EOF (Low)	Data Fork Physical EOF (High)	
60	Data Fork Physical EOF (Low)	Resource Fork Start Block	
64	Resource Fork Logical EOF		
68	Resource Fork Physical EOF		
72	Creation Date and Time		
76	Last Modification Date and Time		

***Figure 4.7** – File information layout*

on the file icon. This is normally EDIT (the Macintosh editor) for textfiles.

The *Finder flags* field (offset 40) is a series of 16 one-bit flags that the Finder uses when displaying the file. If bit 14 is a 1, the file is an "invisible" file. Invisible files do not show up in the Finder file display. If bit 13 is a 1, the file has "bundled" resources (explained in Chapter 6). This bit is used by applications that have custom icons. The rest of the word is unused.

The two *Finder icon location* words give the location of the file icon within its folder. The first word gives the vertical distance from the top of the folder, and the second word gives the horizontal distance from the left edge of the folder.

The *Finder folder number* associates the file with a given folder displayed by the Finder. An application has no access to the folder information, except through the folder number. The folder number can take on certain special values:

−3	A value of −3 indicates that the file is in the trash icon. The file will be deleted when an application is launched or Empty Trash is selected from the Finder's Special menu.
−2	A value of −2 indicates that the file is located on the desktop and is not in any folder.
0	A value of zero indicates that the file is in the folder that will be opened if the user clicks on the disk icon.

Other folders have values of 1 or greater.

The *file number* (offset 48) is a unique 32-bit number assigned to each file by the operating system. This number is used by the operating system to distinguish the file from other files on the same disk.

File space is assigned in terms of *allocation blocks.* An allocation block is the minimum quantity of space that can be added to a file. For the original Macintosh without a hard disk, an allocation block is 1024 (1K) bytes. Thus, a file containing only a single byte still occupies 1K on the disk. A file containing 1025 bytes occupies 2048 (2K) bytes on the disk. The *data fork start block* (offset 52) and the *resource fork start block* (offset 62) contain the numbers of the first allocation block in each fork. If a file fork is empty, the start block field will be zero. The *data fork logical EOF* and *resource fork logical EOF* each contain a 32-bit number that is one greater than the number of the last byte in the file fork. Bytes in the file are numbered from zero, so this quantity is also equal to the number of bytes in the fork. The *data fork physical EOF* and *resource fork physical EOF* give the number of bytes actually allocated to each fork of the file.

The *creation date and time* (offset 72) and the *last modification date and time* (offset 76) indicate when the file was created and last modified. These times are recorded in units of seconds starting from 12:00 A.M. on January 1, 1904.

File I/O Macros

The macros contained in MACLIB.ASM provide a convenient way to perform standard file I/O. These macros are designed to work with Macintosh textfiles, although they will work with other file types with a little extra effort. Macros exist for all of the file operations mentioned previously.

Open File Macros

The macro library provides three methods of opening a file:

- The OPEN macro, which opens the data fork of a file when given the file name
- The OPENR macro, which opens the resource fork of a file
- The POPEN macro, which opens a file selected by the user from a menu generated by the Macintosh standard file package

The OPEN and OPENR macros have a common syntax:

```
OPEN[R]    filename,filebuff,volref
```

The *filename* parameter is the address of a Pascal string that gives the name of the file to be opened. This parameter can be a label or a string contained in single quotes. A disk or volume name can be specified as part of the string. The string

```
vol1:file.type
```

specifies a file name of "file.type" on the disk "vol1". File names cannot contain the colon (:) character.

The *filebuff* parameter, if specified, is the address of a 522-byte buffer for the file system. If no buffer is specified, the file system uses a buffer that is shared by the entire disk. Using a buffer that is not shared results in slightly better file I/O speed.

The *volref* parameter is a *volume reference number.* A volume reference number is either a drive number or a negative number, which references an ejected disk. The Macintosh internal drive is drive 1, and the add-on drive is drive 2. If this parameter is omitted and the file name does not contain a volume name, all drives are scanned for the named file.

The POPEN macro has the following syntax:

```
POPEN     types,extent,namebuf,filebuff
```

This macro generates a display like the one shown in Figure 4.8. Files are displayed in the file window, and the user selects one by clicking on the file name and then on the Open button. Double-clicking on the file name also opens that file. The user can also change drives or eject a disk and insert a new one.

The POPEN *types* parameter is the address of a type array of the form shown in Figure 4.9. The first item in the array is a 16-bit count of the number of types that follow. Each type is a four-byte quantity. If this parameter is specified, only those files whose types appear in the array are shown in the selection window. If the *types* parameter is the string DEFAULT, files of all types are displayed.

The *extent* parameter is the address of a Pascal string that must match the file name extension if the file is to appear in the selection window. Use the string DEFAULT if you do not wish to select files in this manner. This feature is normally used by software tools that have standard file types. For instance, the linker requires that the link file name end in .LINK. When specifying the extent parameter, use all capital letters. The software invoked by the macro relies on the comparison string being uppercase. Only in the file name are lowercase characters converted to uppercase for comparison. This means that an extent parameter of .LINK will match both File.link and File.LINK. However, an extent parameter of .link will match neither File.link nor File.LINK.

The *namebuf* parameter is the address of a buffer or memory area that receives the name of the file selected by the user. This area need be only 65

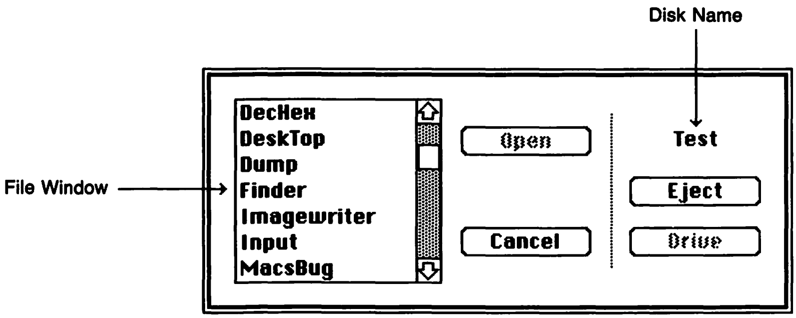


Figure 4.8 – *The POPEN dialog box*

bytes in length to store the names of files that can be displayed by the Finder. The first byte of the buffer is set to the number of characters in the file name string. You can therefore display the buffer with the PRINT macro.

The *filebuff* parameter is the address of a 522-byte buffer discussed in the description of the OPEN and OPENR macros.

All three open macros return a 16-bit identifier in D0 called the *I/O reference number.* This quantity is used by subsequent READ, WRITE, and CLOSE requests to reference the file. If the file cannot be opened for some reason (for example, the named file doesn't exist), D0 will contain an error code. Error codes are all less than zero, so a

```
TST.W    D0
BMI
```

instruction sequence can be used to detect an unsuccessful open attempt.

The POPEN macro opens only the data fork of the file selected by the user. To open the resource fork, use the name returned in the namebuf area as the file name for an OPENR macro.

Offset	Contents
0	Number of entries that follow (N)
2	Type 0 (4 bytes)
6	Type 1 (4 bytes)
	•
	•
	•
$4 \times (N - 1) + 2$	Type N − 1 (4 bytes)

Figure 4.9 *– The POPEN type array*

Create File Macros

There are two macros used to create new files:

- The *CREATE* macro, used when the name of the file is not to be selected by the user
- The *PCREATE* macro, used when the name of the file is to be entered by the user

The CREATE macro installs a new file on the disk and leaves the data fork open for subsequent I/O operations. Any previously existing file with the same name is destroyed. The CREATE macro is used as follows:

```
CREATE    filename,filebuff,volref
```

The *filename* parameter is the address of a Pascal string containing the name of the file to be created. This name can be prefixed by a volume name, like the OPEN and OPENR macros. *Filebuff* is the address of a 522-byte buffer to be used for the file. Use the string DEFAULT to specify no buffering. *Volref* is an optional volume reference number, which is an alternate way to specify the volume on which to create the file.

The PCREATE macro installs a new file whose name is supplied by the user. To call this macro, use this form:

```
PCREATE    prompt,defname,namebuf,filebuff
```

Prompt is the address of a prompt string that appears above the area where the user types the file name. *Defname* is a default name, which the user can select by clicking on the Save choice in the dialog box. *Namebuf* is the optional address of an area of memory that will receive the selected file name. *Filebuff* is the address of a 522-byte area to be used as a buffer for the open file.

The layout of the dialog box for the PCREATE macro is shown in Figure 4.10. The user is given the options of accepting the file name, typing in a new file name, canceling the request, or specifying the disk drive on which the file is to be created.

The I/O reference number for the new file is returned in D0.W for both the CREATE and PCREATE macros. If the file cannot be created, D0.W will be less than zero. This can happen for a variety of reasons, such as using a write-protected disk, having a locked file of the same name already on the disk, or, in the case of the PCREATE macro, a selection of Cancel by the user.

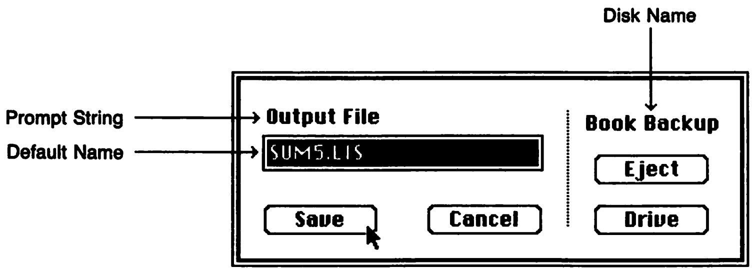


Figure 4.10 – *The PCREATE dialog box*

Any file created with the CREATE or PCREATE macros is initially a text-file, and the default application for the file is EDIT. This can be changed using the GETFINFO and SETFINFO macros described below.

Read File Macros

There are two macros that transfer data from a file into memory. The READ macro transfers a fixed number of bytes, and the READLN macro transfers bytes until it encounters a carriage return. The READ macro is normally used on files that do not contain ASCII text, while the READLN macro is very useful in reading a single line at a time from a file. These macros are used as follows:

```
READ      refnum,buffer,count
READLN    refnum,buffer,count
```

The *refnum* parameter is a file reference number returned by an OPEN, OPENR, POPEN, CREATE, or PCREATE macro. *Buffer* is the memory address where the first byte read from the file is to be placed. *Count* is the number of bytes available in the buffer. Both macros return the number of bytes actually transferred in D0.L.

The READ macro tries to place exactly the requested number of bytes in the buffer. If there are insufficient bytes remaining in the file to do this, READ returns less than the number of bytes requested.

The READLN macro reads bytes until it encounters a carriage-return character (a byte containing hex 0D, or decimal 13). Macintosh textfiles use this character to signal the end of a line. READLN returns the number of characters read in D0.L. Normally, the last character placed in the buffer is the carriage return. If the buffer is too short to accommodate the whole line, however, READLN will put only as much as will fit in the buffer. A subsequent READ or READLN will begin reading the next part of the line.

The WRITE Macro

To transfer data from memory to a file, use the WRITE macro. This macro is used as follows:

```
WRITE    refnum,buffer,count
```

The parameters are analogous to the READ macro parameters. *Refnum* is an I/O reference number, *buffer* is the address of the first byte to be written to the file, and *count* is the number of bytes to be written.

The WRITE macro returns the number of bytes actually written in D0.L. The actual number will always be the same as the *count* parameter, unless an error occurs during the transfer. Errors can occur if you use WRITE on a full disk, a write-protected disk, or a locked file.

The SEEK Macro

Normally, you read or write a file in order from beginning to end. This is called *sequential access* to the file. In some applications, however, it is necessary to access information in a different order. The SEEK macro allows you to specify the byte number within the file (relative to zero) at which the next READ or WRITE macro will begin. The SEEK macro is used as follows:

```
SEEK    refnum,offset,sense
```

Refnum is the reference number for an open file. *Offset* is the 32-bit offset that gives the position in the file. This number can be interpreted in any of three ways, as indicated by the *sense* parameter:

Sense Value	Meaning of Offset
0	From beginning of file
1	From current file position
2	From end of file

If the offset is a negative value, you can seek backwards from the current file position or from the end of the file. If you attempt to seek beyond the end of the file, the file pointer will be set to one byte beyond the end of the file.

The SEEK macro returns the resulting offset from the beginning of the file in D0.L. You can thus obtain the value of the file pointer by using SEEK and specifying zero bytes from the current file position. You can find the file size by using SEEK and specifying zero bytes from the end of the file.

Macros for Accessing File Information

You can obtain and change the file information (described previously) using the GETFINFO and SETFINFO macros. These macros are used as follows:

```
GETFINFO    filename,buffer,volref
SETFINFO    filename,buffer,volref
```

Filename is the address of a string that specifies the name of the file. This string can contain a volume name, as with the OPEN, OPENR, and CREATE macros. *Buffer* is the address of an 80-byte buffer that receives the information shown in Figure 4.7. *Volref* is an optional volume reference (drive) number. Omit this parameter if you want to search all drives for the named file.

FILE I/O EXAMPLE

Having covered the necessary background material, we can now write a sophisticated program. The final section of this chapter presents and discusses a dump utility for the Macintosh—a sample program that performs file I/O. This program produces a textfile containing a hexadecimal listing of the contents of the resource and data forks of a user-selected file. The file information is also formatted and sent to the file.

This sample program is typical of noninteractive programs that deal with files. Such programs normally take an input file and perform some transformation on it to produce an output file. The Macintosh assembler and linker programs are also examples of such programs.

This program is large enough that it must be explained in several different parts. Splitting a program into smaller components is known as *modular programming*. Well-written programs are composed of small and for the most part independent pieces of code called *modules*. The subroutine call mechanism (BSR/JSR and RTS instructions) is normally used as the interface between modules.

A module diagram showing the structure of the Dump program is shown in Figure 4.11. This program is implemented in a typical hierarchical fashion. The subroutine relationships are shown by lines between the boxes. Calling is done in a downward direction only.

The main routine (at the beginning of the program) calls three other routines: OpenFiles, DoFInfo, and DoFork. OpenFiles is a routine that opens the data fork of the input file and creates the output file. DoFInfo formats and writes the file information, and DoFork formats and writes the contents of a file fork. The main routine calls DoFInfo and then calls DoFork once for each fork of the input file.

The DoFInfo routine calls a set of routines that convert binary to ASCII. The Date routine converts a binary date in Macintosh format to ASCII. A routine called TwoDig converts a small binary number to a two-digit decimal ASCII number as part of this process. DumpAsc formats the binary data into printable ASCII, substituting a period character for bytes that have no meaningful ASCII representation. Routines HexL, HexW, and HexB convert a binary longword, word, or byte, respectively, to hex ASCII. The hex routines and the TwoDig routine are left as exercises.

The DoFork routine reads, converts, and writes the contents of a file fork. This routine also calls the hex conversion routines and the DumpAsc routine described above.

The Dump Program Main Routine

Listing 4.7 contains the main routine of the Dump program. Line 20 is the call to the OpenFiles routine. Lines 21–24 echo on the screen the file names selected by the user. OpenFiles leaves the file reference numbers

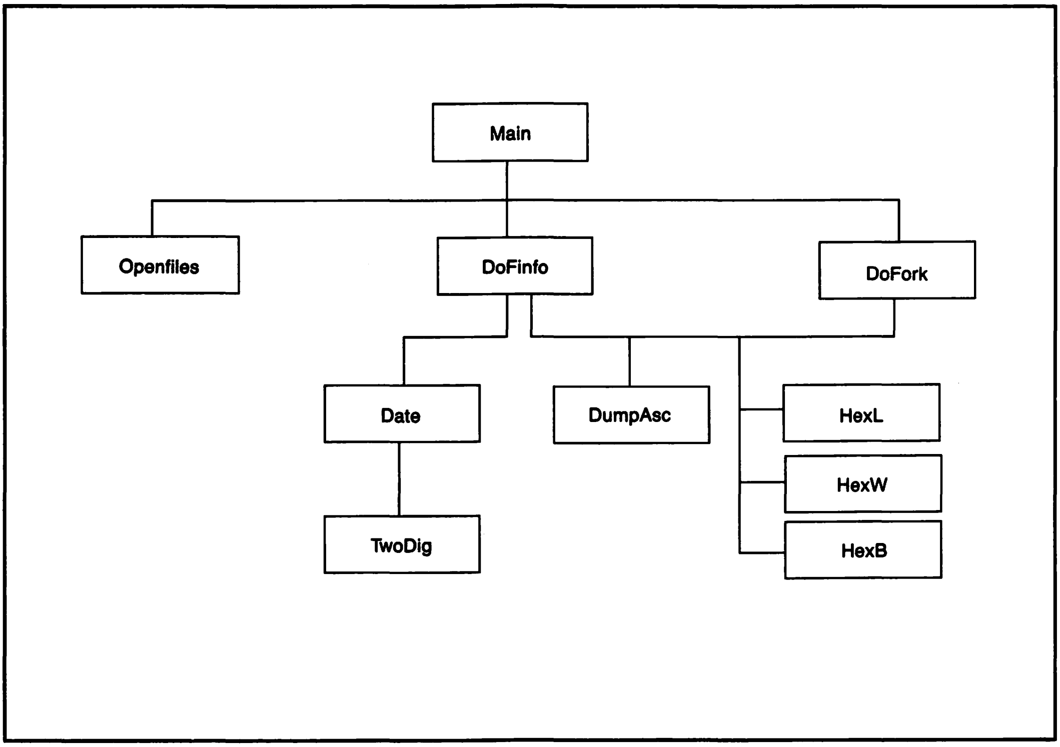


Figure 4.11 *– Structure of the Dump program*

```
;
;       Dump.asm  This program dumps an arbitrary file to an ASCII textfile.
;
;       Version 01.00   18-Feb-85
;
;       This program prompts for a file to be dumped, changes the extension to
;       ".dmp", and prints the contents of the file in hex and ASCII in the
;       output file.
;
;       The dump may be repeated as desired.
;
include MACLIB.ASM
;
BufSize EQU     16                      ;       16 bytes per dump line
Space   EQU     $20                     ;       ASCII value for a blank
;
;
;
        PROGRAM DUMP                    ;       Set up program entry
Start:  BSR     OpenFiles               ;       Start with opening input/output
        PRINT   'Dumping ',*            ;       Echo file names
        PRINT   INameBuf(A5),*          ;       to main window
        PRINT   ' To ',*                ;
        PRINT   ONamebuf(A5)            ;
        WRITE   ORefNum(A5),InfoStr,InfLen;     Write "File Information" on file
        BSR     DoFInfo                 ;       Dump file information first
        TST.L   D0                      ;       Successful?
        BNE     DQuit                   ;       No, quit
        WRITE   ORefNum(A5),DataStr,DatLen;     Write "Data Fork" on file
        BSR     DoFork                  ;       Do the data fork
        TST.L   D0                      ;       Successful?
        BNE     DQuit                   ;       No, quit
        CLOSE   IRefNum(A5)             ;       Close the file
        OPENR   INameBuf(A5),IFBuff(A5) ;       Open resource fork now
        MOVE.W  D0,IRefNum(A5)          ;       Save file number
        BGT     ROpenOK                 ;       If GT, then OK
        PRINT   'Unable to Open Resource Fork: ',*
        PRINT   INameBuf(A5)            ;       Identify file name
        CLOSE   ORefNum(A5)             ;       Close output file
        BRA     Start                   ;       To next file
ROpenOK:                                ;       Here if resource open OK
        WRITE   ORefNum(A5),ResStr,ResLen;      Write resource ID
        BSR     DoFork                  ;       Do the dump
DQuit:  CLOSE   IRefNum(A5)             ;       Close the file
        CLOSE   ORefNum(A5)             ;       and output file
        BRA     Start                   ;       Go get next file name
```

***Listing 4.7** – The main routine for the Dump program* for the input and output files in global variables IRefNum and ORefNum. The names of the files are Pascal strings in INameBuf and ONameBuf.

Lines 25–28 initiate the file information output part of the program. The WRITE macro in line 25 places the heading "File Information:" in the output file. The DoFInfo routine called in line 26 formats and writes the file information. If an output error occurs, register D0.L returns a nonzero value. Lines 27–28 handle this case.

Lines 29–46 complete the dump process. Line 29 writes a heading for the data fork. The call to DoFork in line 30 writes the converted data fork contents. Lines 31–32 check for an error from DoFork. Lines 33–35 close the data fork of the file and open the resource fork. Lines 36–40 handle the unlikely case of a problem with opening the resource fork. Lines 37–38 print a message on the screen identifying the error and the file name. Lines 39–40 close the output file and restart the program. Lines 42–43 dump the resource fork. Lines 44–46 close the input and output files and restart the program.

The OpenFiles Routine

The Dump program OpenFiles routine is shown in Listing 4.8. This routine uses the POPEN and PCREATE macros to open the input and output files, respectively.

Lines 62–64 perform the POPEN for the input file. If the user selects the Cancel option, the reference number returned in D0 will be negative. In this case the program exits to the Finder at line 94.

Lines 65–86 construct the output file name from the input file name. Lines 70–74 find a period (.) character in the input file name. Line 75 appends a period if none is found. Lines 77–81 change whatever appears after the period to the characters DMP. Thus if the input file name is Test.asm, the program produces the file name Test.DMP as output. Lines 82–86 calculate the count byte for the output file name string.

Lines 87–93 complete the OpenFiles routine. The PCREATE macro opens the output file. If this fails, the input file is closed and the OpenFiles procedure begins all over again. If PCREATE is successful, OpenFiles returns to the main routine.

The DoFInfo Routine

Listing 4.9 contains the DoFInfo routine, which formats and writes the file information to the output file. Lines 107–123 of this routine are equates defining the offsets in the buffer shown in Figure 4.7. The DoFInfo routine

```
;
;       OpenFiles routine.  Here to open input file and output file using
;       POPEN and PCREATE macros.
;
;       Entry conditions:
;
;       <none>
;
;       Exit conditions:
;
;               IRefNum   = Input file number
;               ORefNum   = Output file number
;
OpenFiles:
        MOVEM.L D0/A0-A1,-(SP)         ;       Save registers
        POPEN   DEFAULT,DEFAULT,INameBuf(A5),IFbuff(A5); Display all files for the user
        MOVE.W  D0,IRefNum(A5)         ;       Save file number
        BLT     DoExit                 ;       IF > 0, OK
        LEA     INameBuf(A5),A0        ;       A0 -> File name specified by the user
        LEA     DefName+1(A5),A1       ;       A1 -> Place to put prompt name
        CLR.L   D0                     ;       Load count
        MOVE.B  (A0)+,D0               ;       D0 = # bytes in file name
        SUBQ.L  #1,D0                  ;       Adjust for DBRA
FindDot:                               ;       Label for DBRA loop
        MOVE.B  (A0),(A1)+             ;       Move a byte
        CMPI.B  #'.',(A0)+             ;       Was it a period?
        BEQ     FoundDot               ;       Yes, break out
        DBRA    D0,FindDot             ;       Loop till run out of name
        MOVE.B  #'.',(A1)+             ;       Move in a period
FoundDot:                              ;       Here when dot found
        MOVE.L  #2,D0                  ;       DBRA-Adjusted count
        LEA     'DMP'+1,A0             ;       A0 -> Extension
MoveExt:                               ;       Loop label
        MOVE.B  (A0)+,(A1)+            ;       Move a byte
        DBRA    D0,MoveExt             ;       Loop on count
        LEA     DefName(A5),A0         ;       Find address
        SUBA.L  A0,A1                  ;       Subtract to find length
        MOVE.W  A1,D0                  ;       Move in
        SUB.L   #1,D0                  ;               corrected
        MOVE.B  D0,(A0)                ;                       count byte
        PCREATE OPrompt,DefName(A5),ONamebuf(A5),OFBuff(A5);    Create dump file
        MOVE.W  D0,ORefNum(A5)         ;       Save output file number
        BGT     CreatOK                ;       If OK, continue
        CLOSE   IRefNum(A5)            ;       Close input file
        BRA     OpenFiles              ;       Go try next file
CreatOK MOVEM.L (SP)+,D0/A0-A1         ;       Restore registers
        RTS                            ;       Back to main routine
DoExit: EXIT                           ;       Quit routine
```

Listing 4.8 – The OpenFiles routine

```
;
;       DoFInfo routine.  Dumps the file information block.
;
;       Entry conditions:
;
;        (none)
;
;       Exit conditions:
;
;        (none)
;
;       Equates for file info:
DirIndex        EQU     28              ;       Directory index
Attrib          EQU     30              ;       File attributes
Version         EQU     31              ;       File version number
FiFType         EQU     32              ;       Finder file type
FiCreate        EQU     36              ;       Finder file creator
FiFlags         EQU     40              ;       Finder flags word
FiIcon          EQU     42              ;       Finder icon location
FiFolder        EQU     46              ;       Finder folder number
FileNum         EQU     48              ;       File number
DataStart       EQU     52              ;       Data fork start block
DataLEOF        EQU     54              ;       Data fork logical  EOF
DataPEOF        EQU     58              ;       Data fork physical EOF
RsrcStart       EQU     62              ;       Resource fork start block
RsrcLEOF        EQU     64              ;       Resource fork logical EOF
RsrcPEOF        EQU     68              ;       Resource fork physical EOF
CreateDate      EQU     72              ;       Creation date
ModifyDate      EQU     76              ;       Modification date
DoFInfo MOVEM.L D1-D7/A0-A4,-(SP)       ;       Save caller's registers
        GETFINFO INameBuf(A5),InfoBuf(A5);      Get file info for input file
        LEA     InfoBuf(A5),A3          ;       A3 -> File information
        MOVE.W  DirIndex(A3),D0         ;       D0 = Directory index
        LEA     Diridx,A0               ;       A0 -> Output area
        BSR     HexW                    ;       Convert
        MOVE.B  Attrib(A3),D0           ;       D0 = File attribute byte
        LEA     Attr,A0                 ;       A0 -> Output area
        BSR     HexB                    ;       Convert
        MOVE.B  Version(A3),D0          ;       D0 = File version number
        LEA     Vers,A0                 ;       A0 -> Output area
        BSR     HexB                    ;       Convert
        MOVE.L  FiFType(A3),D0          ;       D0 = Finder file type
        LEA     FFType,A0               ;       A0 -> Output area
        BSR     HexL                    ;       Convert
        LEA     FiFType(A3),A1          ;       A1 -> Binary file type
        LEA     FFTAsc,A0               ;       A0 -> Output area
        MOVEQ.L #4,D0                   ;       D0 = Byte count
        BSR     DumpAsc                 ;       Convert
        MOVE.L  FiCreate(A3),D0         ;       D0 = Finder file creator
        LEA     FFCreate,A0             ;       A0 -> Output area
        BSR     Hex                     ;       Convert
```

***Listing 4.9** – The DoFInfo routine*

```
        LEA     FiCreate(A3),A1        ;       A1 -> Binary creator
        LEA     FFCAsc,A0              ;       A0 -> Output area
        MOVEQ.L #4,D0                  ;       D0 = Byte count
        BSR     DumpAsc                ;       Convert
        MOVE.W  FiFlags(A3),D0         ;       D0 = Finder flags word
        LEA     FFlags,A0              ;       A0 -> Output area
        BSR     HexW                   ;       Convert
        MOVE.L  FiIcon(A3),D0          ;       D0 = Finder icon location
        LEA     FFIloc,A0              ;       A0 -> Output area
        BSR     HexL                   ;       Convert
        MOVE.W  FiFolder(A3),D0        ;       D0 = Finder folder number
        LEA     FFNum,A0               ;       A0 -> Output area
        BSR     HexW                   ;       Convert
        MOVE.L  FileNum(A3),D0         ;       D0 = File number
        LEA     OFileNum,A0            ;       A0 -> Output area
        BSR     HexL                   ;       Convert
        MOVE.W  DataStart(A3),D0       ;       D0 = Data fork start block
        LEA     DFStart,A0             ;       A0 -> Output area
        BSR     HexW                   ;       Convert
        MOVE.L  DataLEOF(A3),D0        ;       D0 = Data fork logical EOF
        LEA     DFLEOF,A0              ;       A0 -> Output area
        BSR     HexL                   ;       Convert
        MOVE.L  DataPEOF(A3),D0        ;       D0 = Data fork physical EOF
        LEA     DFPEOF,A0              ;       A0 -> Output area
        BSR     HexL                   ;       Convert
        MOVE.W  RsrcStart(A3),D0       ;       D0 = Resource fork start block
        LEA     RFStart,A0             ;       A0 -> Output area
        BSR     HexW                   ;       Convert
        MOVE.L  RsrcLEOF(A3),D0        ;       D0 = Resource fork logical EOF
        LEA     RFLEOF,A0              ;       A0 -> Output area
        BSR     HexL                   ;       Convert
        MOVE.L  RsrcPEOF(A3),D0        ;       D0 = Resource fork physical EOF
        LEA     RFPEOF,A0              ;       A0 -> Output area
        BSR     HexL                   ;       Convert
        MOVE.L  CreateDate(A3),D0      ;       D0 = Creation date
        LEA     Creation,A0            ;       A0 -> Output area
        BSR     Date                   ;       Convert
        MOVE.L  ModifyDate(A3),D0      ;       D0 = Modification date
        LEA     Modify,A0              ;       A0 -> Output area
        BSR     Date                   ;       Convert
        WRITE   ORefNum(A5),FInfoStr,FinfoLen; Write converted data to file
        CMP.L   FinfoLen,D0            ;       Check for write error
        BNE     WError                 ;       Had an error, indicate it
        CLR.L   D0                     ;       Clear return register
FIRet:  MOVEM.L (SP)+,D1-D7/A0-A4      ;       Unstack registers
        RS                             ;       Back to caller
WError: PRINT   'Write Error on File:',* ;     Print error message
        PRINT   ONameBuf(A5)           ;       Identify file by name
        MOVEQ.L #-1,D0                 ;       Return error code
        BRA     FIRet                  ;       Do return
```

Listing 4.9 *– The DoFInfo routine (continued)* first converts each of the binary values in the buffer to ASCII and then performs a single WRITE operation to place the converted data in the output file. There is a multiline output buffer in the data area that defines the output format. This buffer is contained in lines 453–474 of Listing 4.13.

Line 125 contains the GETFINFO macro invocation, which returns the file information. Note that the third parameter (the volume reference number) has been omitted. This causes the program to search all drives for the input file.

Lines 126–185 are a series of calls to the ASCII conversion routines to format the various fields of the file information buffer. Depending on the field size and contents, either the HexL, HexW, HexB, DumpAsc, or Date routine does the conversion. All of these routines take a binary value, usually in D0, and convert it to an ASCII string whose address is passed in A0.

Line 182 writes the entire file information display to the output file. The buffer is defined (in Listing 4.13) with carriage-return characters at the end of each line, to produce multiline output. Following the WRITE macro, lines 187–191 check for a WRITE error and return zero in D0 if the operation is successful. If it fails, lines 192–195 print an error message. Register D0.L is set to −1 to indicate the error.

The DoFork Routine

Listing 4.10 contains the DoFork routine, which formats and outputs one fork of a file. This routine refers to the input file by reference number only, and it works equally well on the data or resource fork of a file. Register D7 contains the file address throughout this routine. Dumping starts at address 0, and the initialization is performed in line 211.

Lines 213–216 initialize the binary input area to all zeros. When the end of the file is reached, the final READ operation may not completely fill the buffer. Since the output routine always converts 16 bytes, we must initialize the buffer before reading so that there will not be any garbage left from the previous READ at the end of the buffer.

Lines 217–219 give the user a chance to terminate the dumping process. The CHKABORT macro tests to see whether the user has typed Command-Period (indicating a desire to terminate). Register D0 contains a nonzero number if Command-Period was typed at the keyboard. A user abort is treated like an output error: D0.L is set to −1, and DoFork returns to the main routine immediately.

Lines 220–222 read the input file. If the READ macro returns an error in D0, this is treated as the end of the input file fork. In this case DoFork returns zero in D0.L, indicating successful completion.

Lines 223–226 convert the file address to hex in the output buffer. Lines

```
;
;       DoFork routine.  This routine outputs the contents of an open file
;       in hex to another open file.  Called once for data fork and once
;       for resource fork.
;
;       Entry conditions:
;
;               IRefNum = Input file number
;               ORefNum = Output file number
;
;       Exit conditions:
;
;               <none>
;
DoFork: MOVEM.L D1-D7/A0-A4,-(SP)       ;       Save registers
        CLR.L   D7                      ;       File address pointer
READXX:                                 ;
        LEA     Buffer(A5),A3           ;       A3 -> Buffer area
        MOVE.W  #BufSize-1,D0           ;       DBRA-Adjusted count
Clear:  CLR.B   (A3)+                   ;       Clear a byte
        DBRA    D0,Clear                ;       Until done
        CHKABORT                        ;       Check for Apple-Period
        TST.W   D0                      ;       D0 <> 0 if aborted
        BNE     Next                    ;       Yes, quit
        READ    IRefNum(A5),BUFFER(A5),#BufSize
        MOVE.L  D0,BytesRead(A5)        ;       Remember byte count
        BLE     AtEOF                   ;       None, return to caller
        LEA     DumpLine,A0             ;       A0 -> ASCII area
        MOVE.L  D7,D0                   ;       Address to D0
        JSR     HexL                    ;       Convert to ASCII
        ADD.L   #10,A0                  ;       Skip number, and ": "
        LEA     Buffer(A5),A1           ;       A1 -> Input data
        MOVE.W  #3,D2                   ;       DBRA-Adjusted count
Hloop:                                  ;       Conversion loop
        MOVE.L  (A1)+,D0                ;       Fetch next data longword
        BSR     HexL                    ;       Convert to ASCII
        ADD.L   #9,A0                   ;       Bump past number and space
        DBRA    D2,Hloop                ;       Repeat as needed
        LEA     Buffer(A5),A1           ;       A1 -> Input data again
        MOVE.W  #BufSize,D0             ;       D0 = # bytes in buffer
        BSR     DumpAsc                 ;       Convert to dump ASCII
        WRITE   ORefNum(A5),DumpLine,Olen;      Write to output file
        CMP.L   Olen,D0                 ;       Was write successful?
        BNE     Werror                  ;       No, quit
        ADD.L   #BufSize,D7             ;       Bump file address
        CMP.L   #BufSize,BytesRead(A5)  ;       Did last read go OK?
        BEQ     READXX                  ;       Yes, back for more
AtEOF:  CLR.L   D0                      ;       Indicate no error
Next:   MOVEM.L (SP)+,D1-D7/A0-A4       ;       Unsave registers
        RTS                             ;       Return to caller
```

***Listing 4.10** – The DoFork routine*

227–233 convert the 16 bytes read from the file to four hexadecimal longwords in the buffer. Lines 234–236 convert these same 16 bytes to printable ASCII code using the DumpAsc routine.

Line 237 sends the converted output string to the output file. Lines 238–239 check to see that the WRITE macro was successful. A WRITE failure is handled by the code in the DoFInfo routine (lines 192–195).

Lines 240–242 form the end of the loop. The file address is incremented by 16, and the loop repeats if 16 bytes were transferred on the last READ. If less than 16 bytes were transferred, the end of the file was reached on the previous READ. Lines 243–245 return a success indicator to the calling routine.

The DumpAsc Routine

Listing 4.11 contains the DumpAsc routine. This routine moves a specified number of bytes from one location to another, replacing all nonprintable characters with periods. This makes it easy to spot ASCII strings in a dump output.

```
;
;       Dump ASCII conversion routine.  Converts binary to printable ASCII.
;       Unprintable binary characters are converted to periods.
;
;       Enter with:
;
;       A0 -> Output area
;       A1 -> Input data
;       D0 =  Byte count
;
;       Exit with:
;
;       (No change to registers)
;
DumpAsc MOVEM.L D0/A0-A1,-(SP)          ;       Save registers
        SUBQ.W  #1,D0                   ;       Adjust for DBRA
DALoop: CMPI.B  #Space,(A1)+            ;       Printable?
        BGE     NoDot                   ;       Yes, use character
        MOVE.B  #'.',(A0)+              ;       No, use '.'
        BRA     DADBRA                  ;       Do loop end stuff
NoDot:  MOVE.B  -1(A1),(A0)+            ;       Move in actual character
DADBRA: DBRA    D0,DALoop               ;       Repeat until done
        MOVEM.L (SP)+,D0/A0-A1          ;       Restore registers
        RTS                             ;       and return
```

***Listing 4.11** – The DumpAsc routine*

Lines 304–310 copy the source to the destination. If a byte's value is less than the value for a space (20 hex), it is replaced by a period (2E hex). Registers are preserved and restored in lines 303 and 311.

The Date Routine

The Date routine is shown in Listing 4.12. This routine converts a Macintosh-format date in D0.L to an ASCII string at the address contained in A0. Macintosh dates consist of the number of seconds since 12:00 A.M. (midnight) on January 1, 1904. The format of the output string is

```
mm/dd/yy hh:mm:ss
```

which is the usual representation for date and time in the United States. The algorithm used for the conversion is to divide the date by the number of seconds in a year (allowing for leap years) to obtain the year, subtract the number of seconds in each month to find the month, and so on. Lines 328–336 contain equates for the number of seconds in all of the needed units (minutes, hours, months, years). Note that because the number of seconds in a day is greater than 64K, the 68000 hardware divide cannot be used for many of these calculations.

Lines 338–352 calculate the year minus 1900. This is done by repeatedly subtracting the number of seconds in a year until the remaining number is less than a year. Note that the number of seconds in a year changes depending on whether the year is a leap year. This is handled by a loop that first subtracts the number of seconds in a leap year (lines 340–342) and then subtracts the number of seconds in a regular year three times (lines 343–350). Register D1, initially 4, is incremented by 1 after each subtraction. At line 352, D1 contains the correct year number. Note that the branches at lines 341 and 347 are unsigned. This is due to the fact that the number of seconds since 1904 is larger than 31 bits can represent. This means that unsigned arithmetic must be used in calculating the year.

The month is calculated by subtracting entries in a table of months from the remainder in D0.L. There are two tables, one for nonleap years and one for leap years. Lines 353–357 load the correct table address into register A1. Lines 358–365 calculate the month number (1–12) in D2.

The day of the month is calculated by repeatedly subtracting the number of seconds in a day from the remainder in D0.L. When D0 becomes negative, the last number subtracted is added back. Lines 366–373 perform this calculation.

Once the correct day of the month is found, the DIVS instruction can

```
;
;       Macintosh Date conversion routine.  Converts binary date
;       to date/time:  mm/dd/yy hh:mm:ss
;
;       Enter with:
;
;       D0.L = Binary date
;       A0  -> Output area
;
;       Exit with:
;
;       (No change to registers)
;
;       Doesn't work for year 2100 (wrongly considers 2100 a leap year).
;
SecsMin EQU     60                      ;       Seconds/minute
SecsHr  EQU     3600                    ;       Seconds/hour
SecsDay EQU     86400                   ;       Seconds/day
SecsM28 EQU     2419200                 ;       Seconds/28-day month
SecsM29 EQU     2505600                 ;       Seconds/29-day month
SecsM30 EQU     2592000                 ;       Seconds/30-day month
SecsM31 EQU     2678400                 ;       Seconds/31-day month
SecsYr  EQU     31536000                ;       Seconds/(nonleap) year
SecsLYr EQU     31622400                ;       Seconds/leap year
Dte:    MOVEM.L D0-D6/A0-A1,-(SP)       ;       Save registers
        MOVE.L  #4,D1                   ;       D1 to be year
LYearLoop:                              ;       Here to do years
        CMP.L   #SecsLYr,D0             ;       (Starts with 1904, a leap year)
        BLO     YearFound               ;       Year is now correct
        SUB.L   #SecsLYr,D0             ;       Subtract a year
        ADDQ.L  #1,D1                   ;       Bump year counter
        MOVE.W  #2,D3                   ;       Count 3 more nonleap years
YearLoop:                               ;       Loop of nonleap years
        CMP.L   #SecsYr,D0              ;       Another year left?
        BLO     YearFound               ;       No, do the rest
        SUB.L   #SecsYr,D0              ;       Subtract a year
        ADDQ.L  #1,D1                   ;       Bump year counter
        DBRA    D3,YearLoop             ;       Repeat for nonleap years
        BRA     LYearLoop               ;       Repeat till expired
YearFound:                              ;       Here when year found
        LEA     MonthTab,A1             ;       A1 -> Table of secs/month
        MOVE.W  D1,D3                   ;       Copy year to D3
        ANDI.W  #3,D3                   ;       Check for divisible by 4
        BNE     NotLeap                 ;       Not a leap year
        LEA     LMonthTab,A1            ;       A1 -> Secs/month for leap year
NotLeap MOVEQ.L #1,D2                   ;       D2 to be month
MonthLoop:                              ;       Now calculate month number
        CMP.L   (A1),D0                 ;       Enough seconds left?
        BLT     MonthFound              ;       No, month correct
        SUB.L   (A1)+,D0                ;       Subtract off month
        ADDQ.L  #1,D2                   ;       Bump counter
        BRA     MonthLoop               ;       Back for more
```

Listing 4.12 *– The Date routine*

```
MonthFound:                                ;
        MOVEQ.L #1,D3                      ;       D3 will accumulate days
DayLoop:                                   ;
        SUB.L   #SecsDay,D0                ;       Subtract # seconds/day
        BLT     DayFound                   ;       Found day, quit
        ADDQ.L  #1,D3                      ;       Increment D3
        BRA     DayLoop                    ;       Loop till found
DayFound:                                  ;       Here when day found
        ADD.L   #SecsDay,D0                ;       Fix what we broke
        DIVS    #SecsHr,D0                 ;       Can use 16-bit divide now
        MOVE.W  D0,D4                      ;       D4 = Hours
        SWAP    D0                         ;       Remainder back to D0
        EXT.L   D0                         ;       Clear high word
        DIVS    #SecsMin,D0                ;       Divide to get mins & secs
        MOVE.W  D0,D5                      ;       D5 = Minutes
        SWAP    D0                         ;       D0.W = Seconds
        MOVE.W  D0,D6                      ;       Preserve in D6
;
;       We have computed all fields in binary.  Now convert them to ASCII
;
        MOVE.W  D2,D0                      ;       Month to D0
        BSR     TwoDig                     ;       Convert
        MOVE.B  #'/',(A0)+                 ;       Drop in separator
        MOVE.W  D3,D0                      ;       Days to D0
        BSR     TwoDig                     ;       Convert
        MOVE.B  #'/',(A0)+                 ;       Separator
        MOVE.W  D1,D0                      ;       Year to D0
        BSR     TwoDig                     ;       Convert
        MOVE.B  #' ',(A0)+                 ;       Separate date from time
        MOVE.W  D4,D0                      ;       Hours to D0
        BSR     TwoDig                     ;       Convert
        MOVE.B  #':',(A0)+                 ;       Separator
        MOVE.W  D5,D0                      ;       Minutes to D0
        BSR     TwoDig                     ;       Convert
        MOVE.B  #':',(A0)+                 ;       Separator
        MOVE.W  D6,D0                      ;       Seconds to D0
        BSR     TwoDig                     ;       Convert
        MOVEM.L (SP)+,D0-D6/A0-A1          ;       Restore registers
        RTS                                ;       Return
;
;       "Thirty days hath September, April, June, and November."
;
;               January Feb.    March   April   May     June
;               July    August  Septem. October Novem.  December
Monthtab  DC.L  SecsM31,SecsM28,SecsM31,SecsM30,SecsM31,SecsM30
          DC.L  SecsM31,SecsM31,SecsM30,SecsM31,SecsM30,SecsM31
LMonthtab DC.L  SecsM31,SecsM29,SecsM31,SecsM30,SecsM31,SecsM30
          DC.L  SecsM31,SecsM31,SecsM30,SecsM31,SecsM30,SecsM31
```

***Listing 4.12** – The Date routine (continued)* be used to calculate the time of day. Lines 373–381 calculate the correct time of day in D4 through D6.

Lines 385–401 convert the binary quantities calculated above to ASCII. The TwoDig routine is used for numeric conversion. This routine converts a byte to two decimal digits.

Lines 404–412 contain the data used by this routine, the two tables of the number of seconds in each month of the year.

The Dump Program Data Area

Listing 4.13 contains the data locations used by the Dump program. Lines 442–447 define ASCII strings used as headers in the output file. Lines 448–449 define the line output by routine DoFork, with fields for the file address, contents in hex, and contents in ASCII. Lines 453–474 contain the file information buffer, with labels added for ease of conversion. Line 479 contains the output file prompt used by the PCREATE macro in OpenFiles. Lines 484–488 define the lengths of the buffers used by WRITE macros. Note that the lengths must be 32 bits for use in a WRITE call.

Lines 492–501 define data that have no initial value. This is normally done for buffers or variables that need not be initialized, such as the file names returned by POPEN and PCREATE.

```
0435  ;
0436  ;
0437  ;        Dump program data area.
0438  ;
0439  ;        Strings that are written to the .DMP file:
0440  ;
0441           STRING_FORMAT 0
0442  InfoStr:  DC.B  13,'File Information:',13
0443  InfEnd:
0444  ResStr:   DC.B  13,'Resource Fork:',13
0445  ResEnd:
0446  DataStr:  DC.B  13,'Data Fork:     ',13
0447  DataEnd:
0448  DumpLine: DC.B  'xxxxxxxx: XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX ................',13
0449  EndDump:
0450  ;
0451  ;        ASCII output area for file information
0452  ;
0453  FInfoStr:
0454            DC.B  'Directory Index:      '
0455  Diridx:   DC.B  'hhhh',13,'Attributes:           '
```

Listing 4.13 – *The Dump program data area (continued)*

```
Attr:      DC.B  'hh',13,'Version:              '
Vers:      DC.B  'hh',13,'Finder File Type:     '
FFType:    DC.B  'hhhhhhhh '
FFTAsc:    DC.B  '....',13,'Finder File Creator:  '
FFCreate:  DC.B  'hhhhhhhh '
FFCAsc:    DC.B  '....',13,'Finder Flags:         '
FFlags:    DC.B  'hhhh',13,'Finder Icon Location: '
FFILoc:    DC.B  'hhhhhhhh',13,'Finder Folder Number: '
FFNum:     DC.B  'hhhh',13,'File Number:          '
OFileNum:  DC.B  'hhhhhhhh',13,'Data     Fork Start:  '
DFStart:   DC.B  'hhhh Logical EOF: '
DFLEOF:    DC.B  'hhhhhhhh Physical EOF: '
DFPEOF:    DC.B  'hhhhhhhh',13,'Resource Fork Start:  '
RFStart:   DC.B  'hhhh Logical EOF: '
RFLEOF:    DC.B  'hhhhhhhh Physical EOF: '
RFPEOF:    DC.B  'hhhhhhhh',13,'Creation Date:        '
Creation:  DC.B  'xx/xx/xx hh:mm:ss',13,'Modification Date:    '
Modify:    DC.B  'xx/xx/xx hh:mm:ss',13
EndFinfo:
;
;          Pascal strings
;
           STRING_FORMAT 3                ;       Inform assembler
OPrompt:   DC.B  'Output File'
           .ALIGN 2
;
;          Other initialized data
;
InfLen:    DC.L  InfEnd-InfoStr           ;       Length of file info ID
ResLen:    DC.L  ResEnd-ResStr            ;       Length of resource fork ID
DatLen:    DC.L  DataEnd-DataStr          ;       Length of data fork ID
Olen:      DC.L  EndDump-DumpLine         ;       Length of output line
FinfoLen:  DC.L  EndFinfo-FinfoStr        ;       Length of file info output
;
;          Uninitialized data
;
Buffer:    DS.W  8                        ;       Data buffer for input file
BytesRead  DS.L  1                        ;       Number of bytes read
ORefNum:   DS.W  1                        ;       Output file number
IRefNum:   DS.W  1                        ;       Input file number
IFbuff:    DS.B  522                      ;       Input file buffer
OFbuff:    DS.B  522                      ;       Output file buffer
InfoBuf:   DS.B  80                       ;       File information buffer
INameBuf:  DS.B  64                       ;       Input file name
ONameBuf:  DS.B  64                       ;       Output file name
DefName:   DS.B  64                       ;       Constructed default name
           ENDPROGRAM
```

Listing 4.13 *– The Dump program data area (continued)*

Sample Output

Listing 4.14 shows the output of the Dump program for an input file that contains the single line

```
Hello, World
```

The line of text is contained in the file's data fork. Note that the editor uses the resource fork to store information about the file.

```
File Information:
Directory Index:      0000
Attributes:           80
Version:              00
Finder File Type:     54455854 TEXT
Finder File Creator:  45444954 EDIT
Finder Flags:         0100
Finder Icon Location: 000300EB
Finder Folder Number: 0000
File Number:          00000025
Data     Fork Start:  00DC Logical EOF: 0000000D Physical EOF: 00000400
Resource Fork Start:  00DB Logical EOF: 000001EA Physical EOF: 00000400
Creation Date:        02/23/85 08:22:02
Modification Date:    02/23/85 08:22:03

Data Fork:
00000000: 48656C6C 6F2C2057 6F726C64 0D000000 Hello, World....

Resource Fork:
00000000: 00000100 00000116 00000016 000000D4 ................
00000010: 61726444 69736B3A 69736B3A 3AFFFFFF ardDisk:isk::...
00000020: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000030: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000040: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000050: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000060: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000070: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000080: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000090: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
000000A0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
000000B0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
000000C0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
000000D0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
000000E0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
000000F0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF ................
00000100: 0000000A 0009064D 6F6E6163 6F780000 .......Monacox..
00000110: 00040006 00080000 01000000 01160000 ................
00000120: 00160000 00D40000 C09C007A 0000001C ...........z....
00000130: 00460001 45464E54 00000012 45544142 .F..EFNT....ETAB
00000140: 0000001E 03EB0000 00000000 0001622A ..............b*
```

***Listing 4.14** – The Hello.DMP file*

```
00000150: 03EC0047 0000000E 00016222 464F424A ...G......b"FOBJ
00000160: 00FE06EC 00003FBA 00003FA6 00003FD4 ......?...?...?.
00000170: 00003FEC 00003FC2 00003FC8 00003FE0 ..?...?...?...?.
00000180: 00003FE6 004010CA 00003FCE 004001A2 ..?..@....?..@..
00000190: 004001A2 004001A2 004001A2 004001A2 .@...@...@...@..
000001A0: 00400146 4F424A00 FE06EC00 003FBA00 .@.FOBJ......?..
000001B0: 003FA600 003FD400 003FEC00 003FC200 .?...?...?...?..
000001C0: 003FC800 003FE000 003FE600 4010CA00 .?...?...?..@...
000001D0: 003FCE00 4001A200 4001A200 4001A200 .?..@...@...@...
000001E0: 4001A200 4001A200 40010000 00000000 @...@...@.......
```

***Listing 4.14** – The Hello.DMP file (continued)*

SUMMARY

In this chapter, we have developed the background input/output material for writing sophisticated programs: keyboard input, terminal-style output, file operations, and numeric conversion. This basic tool kit provides us with the building blocks necessary to construct larger programs on the Macintosh.

EXERCISES

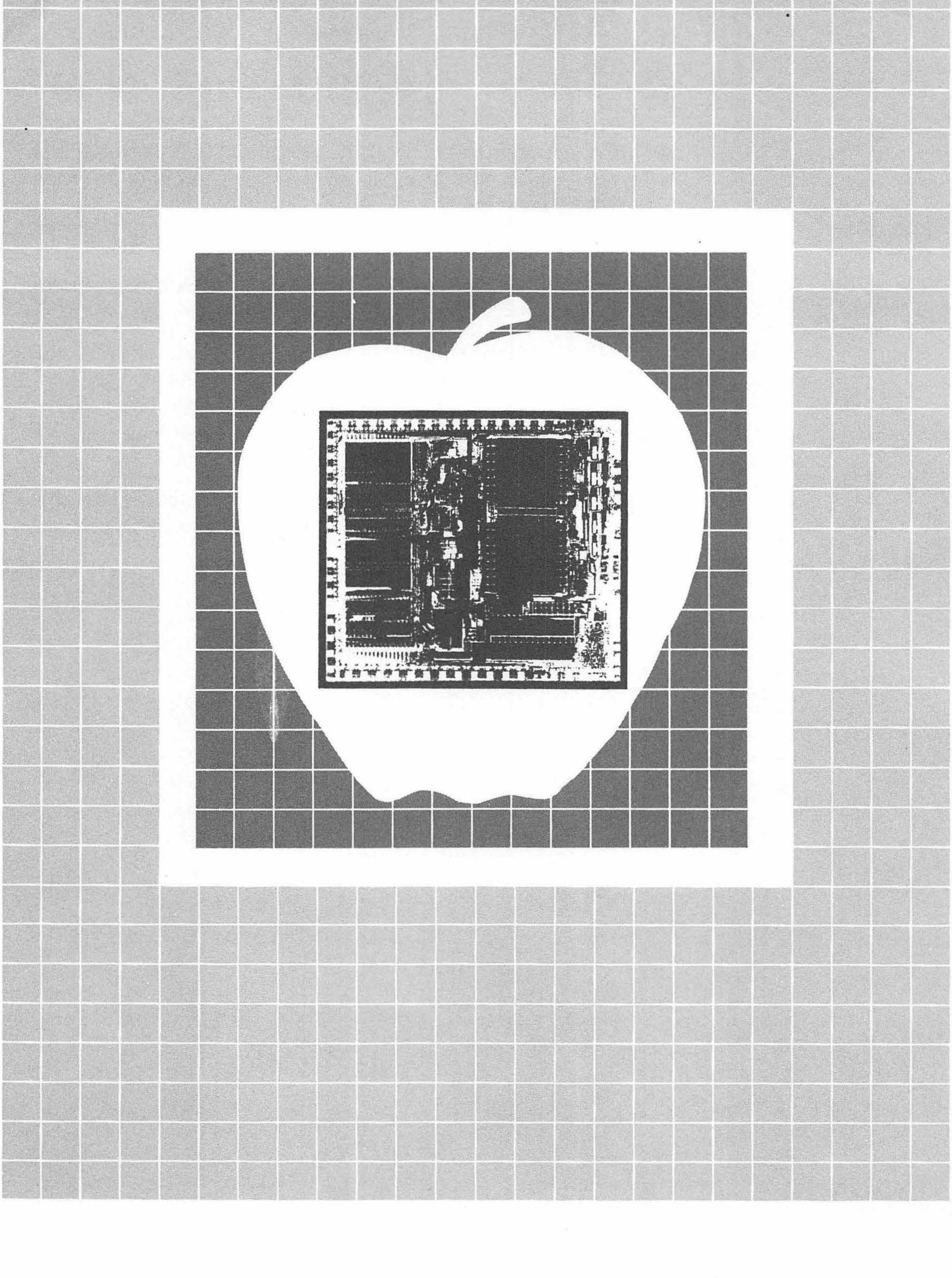
1. A division instruction can be simulated as a loop that subtracts until the dividend is reduced to a number less than the divisor. Write a subroutine called "ldiv" that divides a 32-bit number by another 32-bit number. Use the following calling convention:

   Enter with:
   D0.L = dividend
   D1.L = divisor
   Exit with:
   D0.L = quotient
   D1.L = remainder

   What is the major disadvantage of this scheme?

2. Use the "ldiv" subroutine developed in Exercise 1 to create a new routine that successfully converts 32-bit binary numbers to decimal ASCII. Use a 10-byte output area. Can you think of a better way to do this function?

3. Modify the DecHex program (Listing 4.6) to use the "ldiv" conversion routine developed for Exercise 2.
4. Write a routine that converts hexadecimal ASCII to binary in a manner similar to Listing 4.5. Allow both upper- and lowercase ASCII for the digits A–F.
5. Write a program that converts hex numbers to decimal.
6. Modify the DecBin routine (Listing 4.5) to accept leading spaces and negative numbers.
7. Write the HexB, HexW, HexL, and TwoDig routines for the Dump program. These routines take binary input in D0 and convert it to ASCII at an address in A0. Routine TwoDig is required to leave A0 at the next byte beyond the converted data. The hex routines must return A0 unchanged.
8. Write a program that numbers the lines in an assembly language source file. Use four-digit line numbers. Use POPEN and PCREATE to open the input and output files.



Chapter 5

Advanced Concepts

Difficulties mastered are opportunities won.

—Winston Churchill

INTRODUCTION

This chapter begins by introducing topics essential to writing advanced applications. These include data storage techniques, sorting, and searching. It goes on to cover the use of some advanced features of the assembler: conditional assembly and macros. It also contains information on combining high-level languages with assembly language. Finally, it covers the structure of Macintosh Toolbox and operating system calls.

DATA ORGANIZATION

One of the most important considerations in writing a large program is how to organize the data both inside and outside the program. You can organize data in the form of records, arrays, linked lists, trees, queues, and dequeues. We will look at each of these types in detail.

Records

One of the simplest forms of data structure is called a *record* (or *structure*). A record is a set of contiguous memory locations that contains related data. For instance, a program that organizes data into a telephone directory would probably have a record with fields for name, address, and telephone number.

Records can exist either in primary memory or on disk. There is an almost infinite variety of record formats. Two of the most popular ones are fixed-length records and variable-length records.

Fixed-Length Records

Fixed-length records (abbreviated FLR) have the same number of bytes in each record. This has distinct advantages when accessing records randomly. When the records in a file contain differing amounts of information, however, fixed-length records waste storage space.

Fixed-length records are a natural outgrowth of the days when punched cards were used to run programs and enter data into a computer. Cards had a fixed number of columns, usually 80 or 96. Each column held one byte of information. When programming machines that used punched cards, it was common to place each field of the record in a particular card column. When the card was read into memory, the fields would always be offset the same distance from the beginning of the card image in memory.

The FORTRAN language is an example of a holdover from this technique. FORTRAN statements begin in what would be column 7 of the card, which is very difficult to do on a terminal! If a FORTRAN statement overflows from one line to the next, you have to put a nonblank character in column 6. Fortunately, most systems no longer use cards.

Many present-day storage techniques rely on fixed-length techniques. This is especially true for records contained in primary memory. The simplicity and speed of the FLR technique often outweigh considerations of storage inefficiency.

Variable-Length Records

Variable-length records (abbreviated VLR) allow for a different number of bytes in each record. This technique avoids the problem of wasted space inherent in fixed-length records, but it makes random access slower and more difficult. ASCII textfiles often use one of the VLR techniques. Here are some of the more popular methods:

1. Records are prefixed with a count field that gives the number of bytes in the record. The Macintosh uses this technique for text strings in memory. There are minor variations in the way computers use this technique: the number of bytes dedicated to the

count field varies, the count field may or may not include the number of bytes in the count field, and the count field may count units other than bytes. Most implementations of BASIC on microcomputers use the count technique for string variables. This technique is also used to store textfiles on many operating systems for mainframes and minicomputers.

2. Records are terminated with some special character or character sequence that cannot occur within the record itself. For example, Macintosh textfiles terminate lines of text with a carriage-return character (decimal 13). The C language uses a null byte (decimal 0) to terminate strings in memory.

3. The beginning of a record is marked with a unique sequence. This technique is often used in work involving communications, where faulty transmission may distort, add, or delete bytes within a record. A unique sequence of bytes helps resynchronize the receiver and transmitter.

Hybrid Techniques

A number of techniques have been devised that combine the desirable features of fixed- and variable-length techniques. These "hybrid" techniques usually allow reasonable random access with reasonable storage efficiency.

For records that are processed sequentially, a fixed portion of the record is often used in combination with a variable portion of the record. The fixed portion of the record contains an indication of how big the variable portion is. This technique is useful for applications that must represent variable-length tables.

Another common technique is to split the fixed and variable portions of the records and store the fixed portions together and the variable portions separately. The fixed portion contains the address of the variable portion. In this way, the capability of fixed-length records to enhance random access is combined with the storage efficiency of variable-length records.

Describing Records in Assembly Language

There are some common techniques for manipulating records in assembly language that are advantageous on the 68000 chip. If a record is less than 32K (as most are), the address-register-indirect-with-displacement addressing mode can be used to access the individual fields. This is particularly advantageous when more than one record must be handled at one time.

In coding references to records, it is good practice to use equated names for the different fields in the record. This allows you to go back later and change the size and order of the fields in the record without changing all the references to those fields.

STORAGE ALLOCATION

The way in which records are arranged in memory is often critical to a program's performance. There are two techniques commonly used to allocate memory:

1. Allocating the records contiguously; that is, one following the other. A collection of records arranged in this way is called an *array.*
2. Allocating the records noncontiguously, with each record containing the address of the next record in logical order. This is called a *linked list.*

Arrays

An array is composed of records arranged contiguously in memory. When used with fixed-length records, the array technique makes random access extremely easy. To access record *n*, this is how you calculate the address:

$$\text{address} = (n - 1) \times (\text{record size}) + \text{the starting address of the array}$$

Note that this formula assumes a starting value of 1. Multidimensional arrays require a more complicated formula that depends on the order in which the dimensions are stored.

This technique is used in many programs for matrix calculations, tables, and other data that must be accessed randomly. The same technique can be used for accessing fixed-length records in a disk file. This is sometimes called *direct access.*

The disadvantage of arrays comes when you are adding or deleting items. The rapid retrieval of information often depends on the information occurring in some particular order. Adding or deleting items in an ordered array requires moving all items below the insertion or deletion point. For example, consider the array of numbers displayed in Figure 5.1.

Let's say that we need to insert the number 101 into this array. In order to make room for a new entry between 100 and 103, we must move the lower three elements down. When manipulating large tables, this process can require a lot of time.

Linked Lists

Another technique for allocating storage to a group of records is to use a linked list. With a linked list, each element of the list contains a way of finding the next entry. This is usually accomplished by inserting the address of the next element in each item of the list. There are several variations on the linked-list technique.

Linked lists can be either *linear* or *circular.* Examples of both types are shown in Figure 5.2. Both types of linked lists start at some known point called the *list head.* This is usually a memory location that contains the address of the first element in the list. The linear list terminates with some special value in the link portion of the record (usually zero). This value indicates that there are no more records in the list. The circular list is linked in a circle, with the last element pointing to either the first element in the list or the list head.

Linked lists are extremely flexible for inserting and deleting items. To insert an item, you modify two links: the link in the item before the item to be inserted and the link in the item to be inserted. To delete an item, you need to modify only the link that points to the item to be deleted. The cost of this flexibility is the additional processing time it requires to access a random element of a linked list. To access element k in a linked list, you must access the previous $k - 1$ elements.

Address	Contents
n	100
n+1	103
n+2	105
n+3	107

***Figure 5.1** – An array of numbers*

One advantage of a circular list is that you can tell whether one of the links in the list is corrupt. If you cannot reach the list head within a reasonable number of tries, something is amiss. This reliability is purchased at the cost of extra processing time.

A second variation of a linked list is to use two pointers in each record of the list: one to the next record and one to the previous record. This technique is called a *doubly-linked list.* To insert or delete items from a linked list, you need to know the address of the previous element in the list. The advantage of having a pointer to the previous element is that it speeds insertion and deletion of random elements in the list.

Yet another variation of the linked-list technique is to maintain a pointer to the last element in the list. Such a pointer is known as a *tail pointer.* This is useful when elements are added to the end of a list. To facilitate removal from the end of a list, you would need to use a doubly-linked list with a tail pointer or, alternatively, a circular doubly-linked list.

It is poor programming practice to use a linked list on a disk or other form of external memory. The reason for this is that in order to add or delete items from a list, you need to modify *two* items in the list. If the

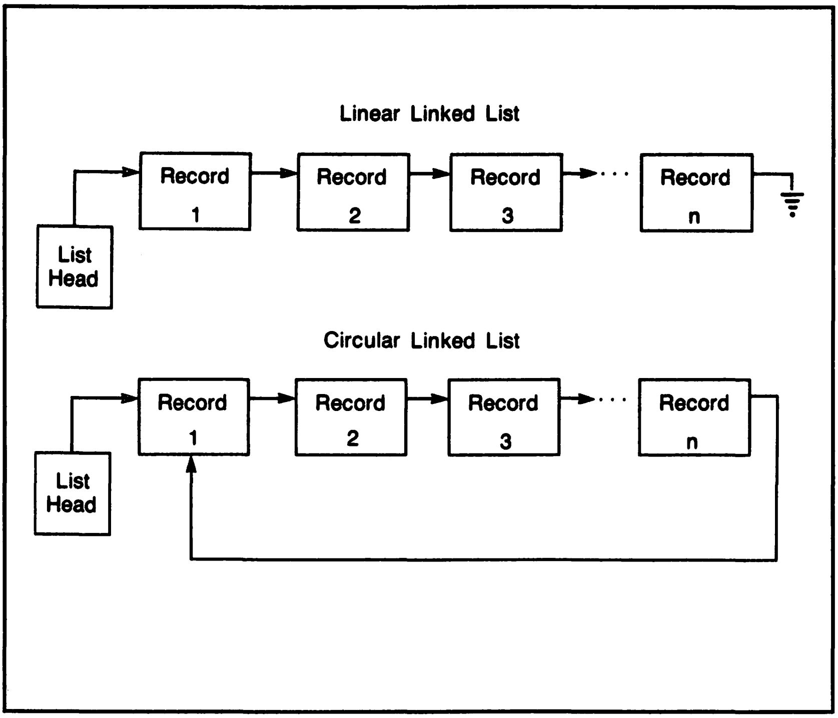


***Figure 5.2** – Linear and circular linked lists*

machine crashes between these updates (and this does happen), the linked-list structure is no longer consistent. This can lead to situations where a disk block appears in two files or cannot be used at all. This was a problem in many early versions of UNIX.

DATA STRUCTURES

There are several logical data structures that can be superimposed on top of arrays or linked lists, such as stacks, queues, and trees. We will now explore these structures in more detail.

Stacks

You are already familiar with the concept of a stack (the 68000 has a stack implemented in hardware). A stack is a data structure in which the last item added is the first item removed. This is also called a *Last In First Out* (LIFO) arrangement.

Stacks can be implemented either as arrays or linked lists. Implementation of a stack as an array requires a separate variable that defines the top of the stack. The 68000 hardware stack pointer (register A7) is an example of such a variable. Implementation of a stack as a linked list requires adding and removing elements of the list only at the beginning of the list.

There are two types of error conditions that you will probably encounter when using stacks:

1. You can run out of room for new stack entries. This condition is known as a *stack overflow*. This can happen, for example, when a program gets caught in an infinite loop that pushes items on the stack.
2. You can pop more entries off the stack than were pushed on the stack. This condition is known as a *stack underflow*.

Queues

A *queue* is a list of items in which the first item added is the first item removed. This is also known as a *First In First Out* (FIFO) or a *First Come First Served* (FCFS) arrangement. FIFO arrangements can be observed in any environment where people wait in lines for service. The first person to arrive is the first person served.

You can implement a queue either as an array or as a linked list. Efficient implementation as an array often uses a *circular* (or *ring*) *buffer.* This data structure consists of two pointers: an insertion pointer and a removal pointer.

Items are added using an insertion pointer and removed using a removal pointer. When the pointers are equal, the buffer is empty. If adding an element causes the pointers to become equal, the buffer is full. Notice that this means you can't use one of the locations in the buffer (that is, if the pointers are equal, you can't tell if the buffer is empty or full). You can eliminate this problem by keeping a count of the number of items in the buffer.

Figure 5.3 shows an example of a circular buffer. Items are added and removed from left to right. Whenever a pointer runs off the right end of the buffer, it is moved back to the left end. Thus, the pointers move in a circular fashion. The shaded areas represent unused elements in the buffer.

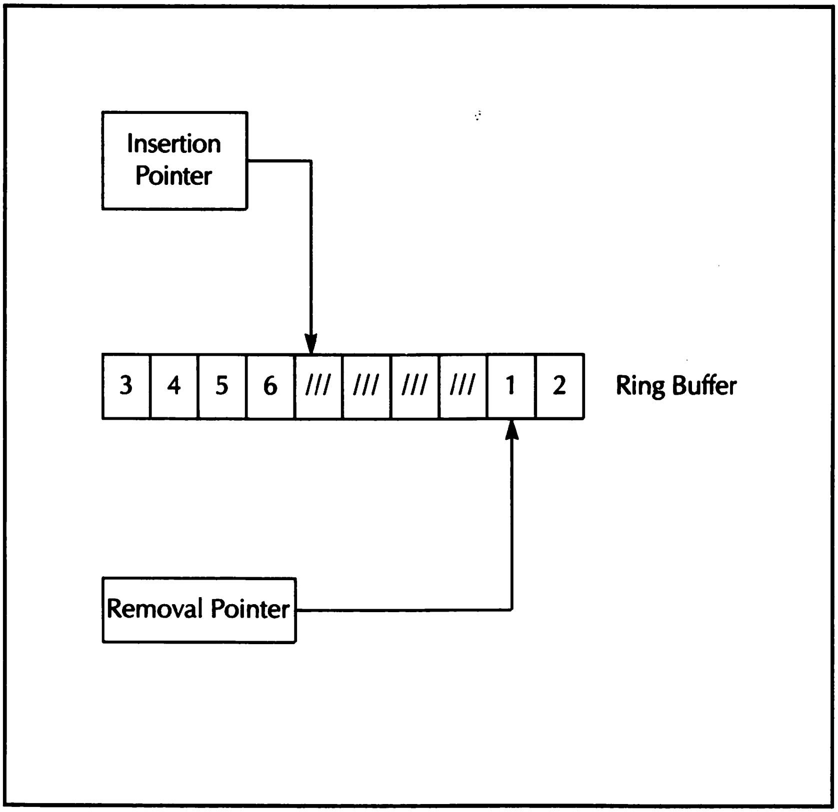


***Figure* 5.3** – *A circular buffer*

Implementing a queue with a linked list is best done using a tail pointer. This makes both insertion and deletion operations quite simple.

Several modifications to the basic queue technique are also useful in many applications. For example, a queue can be based on some order other than chronological. The most common example of this technique is the notion of a *priority order* list of jobs in a larger computer system. A linked list organized according to priorities is often used in an operating system to determine who gets what resources.

A special form of queue, called a *double-ended queue* or *dequeue* (pronounced "deck"), allows insertion and removal of elements at both ends. Altered forms of this technique provide for insertion at both ends but removal at only one, or removal at both ends and insertion at only one. These are called *output-restricted* and *input-restricted* dequeues, respectively.

Trees

A *tree* is a data structure in which each item can point to more than one item. A tree begins with a single element, called the *root*. The root points to other *nodes*, which in turn point to still other nodes, and so on. (An element in a tree is often called a node.) Figure 5.4 shows an example of a tree structure. The elements of the tree are shown as numbered boxes.

Borrowing terminology from genealogy, element 2 is termed a *child* of the root, as are elements 3 and 4. The root is said to be the *parent* of these elements. Elements that are children of the same node are termed *siblings*. Nodes 5 and 6 in Figure 5.4 are siblings. A node that has no children is called a *leaf* of the tree. Any given node and all its descendants make up a *subtree*. For example, nodes 2, 5, and 6 form a subtree of the root.

A tree is useful for describing something that can be defined in terms of itself. This is called *recursion*. For example, on the UNIX operating system, disk devices such as floppies have a directory of files on the disk. It is also possible for one of these files to be another directory. This directory can contain other directories, which can contain other directories, and so on.

A tree is an ideal way to represent this concept. There is a single directory, called the *root directory*, which corresponds to the root of the tree. Nodes in the tree are either directories or ordinary files. Ordinary files are always leaves of the tree, as are empty directories.

Modifying the definition of a tree so that each node has at most two children gives us a special kind of tree called a *binary tree*. The children of a binary tree are called the *left and right descendants*, and the distinction between the two is meaningful. A node with a single right descendant is different from the same node with a single left descendant. Binary trees are used extensively in many areas of computer science. One of the best examples is an *expression tree,* which is found in many high-level language compilers and interpreters.

The handling of arithmetic expressions in assignment statements provides an excellent example of an expression tree. For example, the statement

```
x = (a + b)/2
```

would cause the compiler to generate instructions to first add a and b, divide this sum by 2, and then place the result in x. A compiler would represent this expression in a tree as shown in Figure 5.5.

Note how the assignment for the equal sign operator (=) is treated as if it were an arithmetic operation. Since most arithmetic operations involve

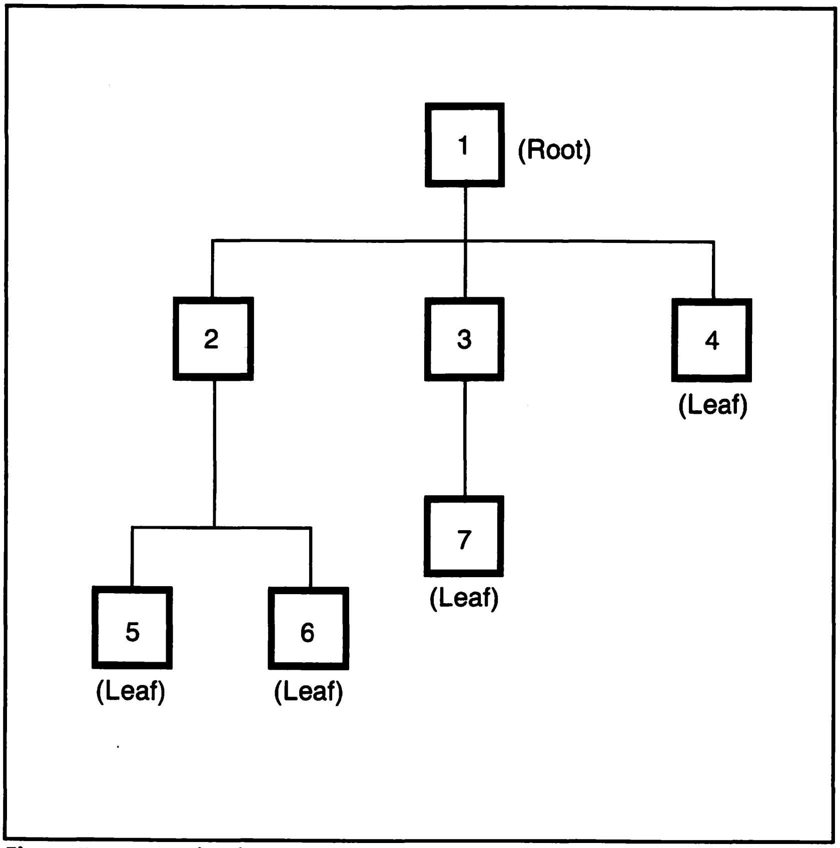


Figure 5.4 *– Example of a tree structure*

an operator and two operands, the binary tree is a convenient way of representing expressions.

In evaluating this expression, you evaluate the subtree on the left first, and then the subtree on the right. In order to evaluate the equal sign operator, you must first evaluate the division operator (/). To evaluate the division operator, you must evaluate the plus sign operator (+). This is called *traversing* the tree. Notice that traversing the binary tree produces the same evaluation order that you would use in evaluating the expression by hand.

ADVANCED PROGRAMMING CONCEPTS

There are three areas of programming that deserve more in-depth study:

1. *Sorting.* The process of taking randomly ordered data and arranging them in a specified order.
2. *Searching.* The process of retrieving a specified piece of information from a large set of data.
3. *Recursion.* The ability to define a particular function or set of functions in terms of itself.

Sorting

Sorting is the process of ordering a randomly ordered set of records. Sorting has received considerable attention in programming literature because it is so easy to do poorly. We will discuss some of the simple techniques used to sort records. Our discussion assumes that the data to be sorted are in memory rather than on disk.

Insertion Sort

An *insertion sort* is generally performed as data are being input to a program. To perform such a sort, take the items one at a time and put each item into a separate memory array in sorted order. When you have placed all the records in the array, you have sorted data. Due to the large number of insertions required, this technique is particularly suited to a linked-list structure in memory.

Interchange Sort

An *interchange sort* is usually performed on data that is arranged contiguously in memory. The simplest form of interchange sort involves taking the top element and comparing it to each element in turn, exchanging where items are out of order. When all the elements have been compared, the top element is certain to be correct. Each element is then compared to all the elements below it. This type of sort requires $n - 1$ passes through the data, where n is the number of records. This means that n^2 comparisons are required. The sort time rises dramatically as the number of records increases.

Bubble Sort

A more efficient type of sort is the *bubble sort,* which compares successive pairs of elements throughout an array and then swaps elements that

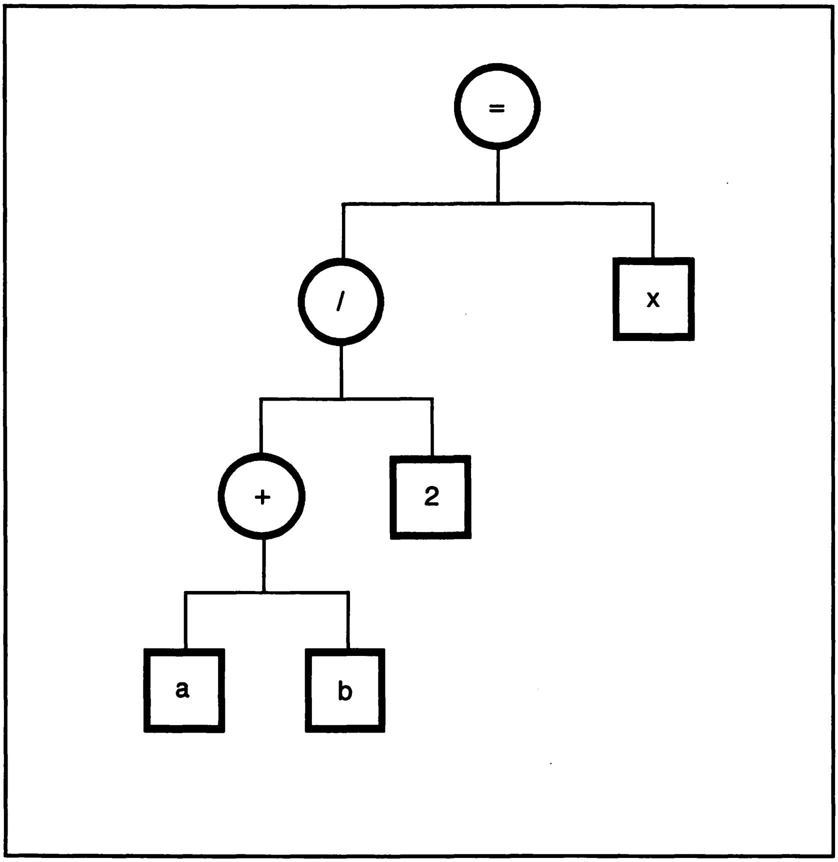


Figure 5.5 *– An expression tree*

are out of order. When a pass is made through the data without exchanging any items, the sort is complete. This method takes advantage of data that may already be in a partially correct order.

Listing 5.1 shows a simple bubble sort program that sorts an array of memory words at the label "list". Lines 5 through 17 constitute a single pass through the data. Register D0.W is used as a flag to indicate whether any exchanges have taken place on the current pass. Lines 7 through 15 perform one pass on the data, with the comparison taking place at lines 7 through 9 and the exchange at lines 10 through 12.

Searching

Searching is the process of finding a particular record in a large collection. For large amounts of data or frequent searches, the search algorithm can be extremely important.

Sequential Searches

The simplest way to find an entry in a table is to start at the beginning of the table and look at each entry until you find the desired one. If you run

```
 1: ;
 2: ;      Sample bubble sort.  Sorts words at "List".  Uses
 3: ;      flag technique.
 4: ;
 5: Bubble LEA     List,A0       ;     A0 -> Data
 6:        LEA     EndBuf,A1     ;     A1 -> End of data
 7:        CLR.W   D0            ;     D0 is "changed" flag
 8: BLoop  MOVE.W  (A0),D1       ;     Load for compare
 9:        CMP.W   2(A0),D1      ;     Need to
10:        BLE     NoSwap        ;            swap?
11:        MOVE.W  2(A0),(A0)    ;     Yes,
12:        MOVE.W  D1,2(A0)      ;           do swap
13:        MOVE.W  #1,D0         ;     Set flag
14: NoSwap ADDQ.L  #2,A0         ;     Bump A0 to next word
15:        CMP.L   A1,A0         ;     End of buffer yet?
16:        BLO     BLOOP         ;     No, continue
17:        TST.W   D0            ;     Is flag set?
18:        BNE     Bubble        ;     Yes, another pass required
19:        RTS                   ;     If not, we're done
20: List   DC.W    6,5,3,4,1,2,0 ;     Unsorted data
21: EndBuf EQU     *-2           ;     Address of last item
```

Listing 5.1 *– Bubble sort routine* off the end of the table, you know that the item you want is not in the table. The average number of comparisons using this technique is one-half the number of entries in the table (assuming that all the entries are accessed an equal number of times). Significant improvements in search times can be made if the data is not accessed in an evenly distributed manner. Simply placing the most commonly accessed data at the beginning of the table can make an amazing difference in performance.

The advantage of the sequential technique is that it does not require the data to be in any particular order. Most of the faster search techniques impose some ordering criteria on the data. A sequential search may well be the best method to use, especially if the effort to sort the data outweighs the effort expended in the search. The decision is based on the relative frequency of sorts to searches. For rapidly changing data that is searched infrequently, a sequential search is probably the best technique.

Binary Search

For data that are in sorted order, a *binary search* technique can be used to substantially reduce search time. A binary search works like this:

1. Given a table of *n* elements in sorted order, establish two pointers to the first and last entries in the table.
2. Compute the element that is halfway between the two pointers. (We'll call this element *H*.) Compare this element to the desired element. If the elements are equal, you have found the desired data. If the element is less than the desired value, move the bottom pointer to element *H*. If element *H* is greater than the desired element, move the top pointer. Reverse these conditions if the table is in descending order instead of ascending order.
3. If the two pointers are equal or adjacent, the item is not in the table. Otherwise, repeat step 2.

Figure 5.6 shows a sample search sequence. *T* is the top pointer, *B* is the bottom pointer, and *H* is the halfway pointer. In this example, the desired entry is found in three tries, as opposed to five for a sequential search. The desired entry is 135.

The size of the table being considered is reduced by a factor of two for each iteration of the search. For large tables, this is a substantial savings over sequential-search techniques. This is true only if the effort to maintain a sorted table is less than the effort saved by the binary search technique. Effort is measured in terms of program execution time and programming time. The number of times a program is used determines the wisdom of spending a lot of time putting in features that save execution time.

Hashing

Another technique for reducing search times is called *hashing.* Hashing imposes a different kind of structure on the table to be searched. A transformation called a *hash function* is performed on the data to be looked up in the table. The hash function yields a number indicating a position in the table. This number is called a *hash code.* Entries are placed in the table at the locations dictated by their hash codes. A simple way to generate a hash code for alphanumeric (string) data is to add each of the characters in the string and take the resulting sum as the hash code.

When two or more different table entries have the same hash code, a situation known as a *hash collision* results. The number of hash collisions is an indication of the quality of the hash function. A good hash function will produce few collisions. A hash collision is usually detected when you need to add an entry to the table and you find that an old entry has the same hash code as the new entry. One solution to this problem is to place the new entry in the next free slot in the table. Then do a sequential search starting at the position indicated by the hash code.

Another technique for dealing with hash collisions is to use a large number of linked-list heads. The hash code determines the list in which a given element belongs. Then the lists are searched sequentially. If the hash

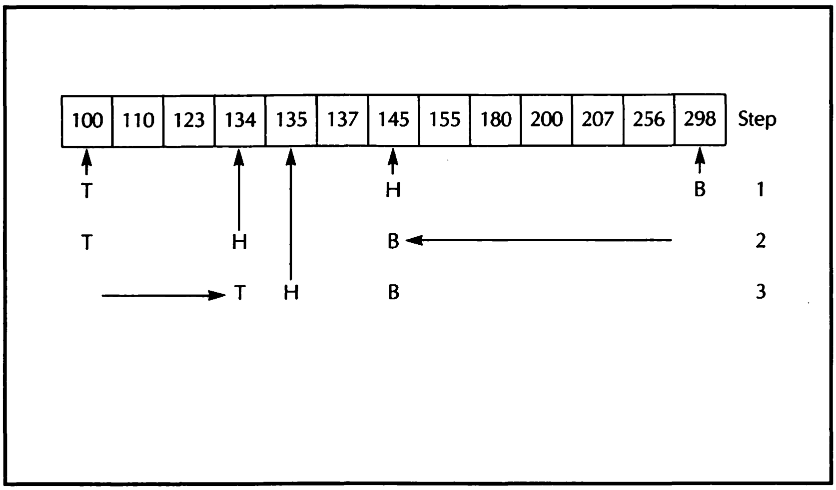


Figure 5.6 – *Example of a binary search*

function yields a reasonably even distribution of elements across the lists, the savings over a sequential search is the number of list heads divided by two. Thus if you used 50 list heads, you would expect to see an improvement of 25 to 1 over a sequential search.

Recursion

Recursion is the ability to define a function in terms of itself. For example, the factorial function can be defined recursively. A factorial multiplies a number by all the integers less than the desired number. Thus, five factorial (written 5!) is 5 × 4 × 3 × 2 × 1, or 120. Zero factorial is defined to be 1. The factorial function for a number *n* can be defined recursively:

1. If *n* is 0 or 1, *n* factorial is 1.
2. Otherwise, *n* factorial is *n* times *n* − 1 factorial.

Listing 5.2 shows a factorial routine that uses this definition to compute factorials.

Lines 8 through 11 handle the case where *n* is 0 or 1. Lines 12 through 16 handle the case where *n* is greater than 1. The fact function is called

```
 1: ;
 2: ;       Recursive factorial routine
 3: ;
 4: ;               Enter with number in D0.W
 5: ;               Exit  with answer in D0.W
 6: ;
 7:         XDEF    Fact            ;       Make linkable
 8: Fact    CMP.W   #1,D0           ;       Easy?
 9:         BGT     DoFact          ;       No, do recursion
10:         MOVE.W  #1,D0           ;       0! or 1! is 1
11:         RTS                     ;       Done
12: DoFact  MOVE.W  D0,-(SP)        ;       Save present value
13:         SUB.W   #1,D0           ;       Decrement
14:         BSR     Fact            ;       Take (n-1)!
15:         MULS    (SP)+,D0        ;       n! = n * (n-1)!
16:         RTS                     ;       Done
```

***Listing 5.2** – Recursive factorial routine* with $n - 1$ as the parameter. Note that when n reaches 1, the recursion will stop.

In general, recursive routines cannot modify registers or static variables without saving and restoring them on the stack. Languages such as C and Pascal, which support recursion, put all variables local to a procedure on the stack. You can do the same in assembly language using the LINK and UNLK instructions to allocate the stack space and the address-register-indirect-with-displacement addressing mode to access the data.

ADVANCED ASSEMBLY LANGUAGE CONCEPTS

There are three additional techniques that are commonly employed in writing large programs:

1. Macros
2. Conditional assembly
3. Interfacing assembly and high-level languages

We will now explore these techniques in the Macintosh environment.

Macros

A *macro* is a programming shorthand technique. Macros consist of a set of assembly language source statements. These statements can be placed in a program by coding a single line. This technique reduces the size of a source program, although the amount of memory the program takes will often be greater when using macros.

Using a macro in a program requires two operations. First, the assembler must be told which instructions are to be placed in the program when a macro instruction is coded. The method by which a new macro is introduced into a program is called a *macro definition.* Later, when you want to use the macro, you code a *macro call.*

Macros on the Macintosh are identified by a string called the *macro name.* You must specify the macro name when you define the macro. Then you can issue a macro call by coding the macro name in the instruction part of a source line.

As part of the macro definition, you can specify parts of the macro that will be replaced by strings specified on the macro call. These strings are called *macro parameters* or *macro operands.* They are similar in nature to operands used with instructions.

Macintosh Macro Definitions

The Macintosh assembler supports two types of macros. These are called *Macintosh-style* and *Lisa-style* macros. The names stem from the history of developing Macintosh applications. When the Macintosh first came out, you had to use a Lisa computer (now called a *Macintosh XL*) to write Macintosh programs. The development software was an addition to the Lisa Pascal compiler package. This package was called the *Lisa Pascal Workshop*, and it included an assembler. Macros accepted by this assembler are the Lisa-style macros. The Macintosh-style macros are unique to the Macintosh assembler.

A Lisa-style macro is defined with a .MACRO directive. This directive takes a single operand, the macro name. Following the .MACRO directive, you code the instructions you want placed in the program when the macro is called. These instructions form the *macro body.* You signal the end of a macro definition with a .ENDM directive. The strings %1 through %9 indicate that a macro parameter is to be substituted in the macro body. %1 corresponds to the first parameter, %2 to the second, and so on.

For example, consider a macro that exchanges the contents of two words in memory. The macro will be named EXGM and will have two operands that are the locations to be exchanged. The macro can be written as follows:

```
.MACRO     EXGM
MOVE.W     %1,-(SP)      ; Save first operand
MOVE.W     %2,%1         ; Replace first with second
MOVE.W     (SP)+,%2      ; Replace second with first
.ENDM
```

The macro call

```
EXGM       A(A5),4(A0)
```

would generate

```
MOVE.W     A(A5),-(SP)
MOVE.W     4(A0),A(A5)
MOVE.W     (SP)+,4(A0)
```

Note that use of pre-decrement or post-increment addressing modes will cause an error. Since each operand is referenced twice in the macro body, any increment or decrement would be performed twice.

A Macintosh-style macro is defined somewhat differently. The macro begins with a MACRO directive (note the missing "."). The MACRO keyword is followed by the macro name, a list of parameter names, and an equal sign (=). The end of the macro is denoted by a vertical bar (|). Within the body of the macro, the parameter names from the MACRO directive are enclosed in braces ({}). For example, the EXGM macro above can be written as a Macintosh macro in the following way:

```
MACRO      EXGM src,dest =
MOVE.W     {src},-(SP)        ; Save first operand
MOVE.W     {dest},{src}       ; Replace first with second
MOVE.W     (SP)+,{dest}       ; Replace second with first
|
```

None of the parameters in a Macintosh-style macro can be omitted. It is possible to invoke a Lisa-style macro with fewer parameters than the macro requires, but the assembler gives you an error if you attempt this with a Macintosh-style macro.

A Macintosh-style macro can occupy only part of a line; Lisa-style macros cannot. For instance, the macro

```
MACRO      ONA5 label = {label}(A5)|
```

can be used in an operand field like this:

```
MOVE.W     ONA5 ABC,A0
```

This will generate the instruction

```
MOVE.W     ABC(A5),A0
```

A second feature found in Macintosh-style macros, but not Lisa-style macros, is the use of only part of a parameter. The notation

```
{arg|n:m}
```

denotes using *m* characters of the macro parameter *arg,* starting with character number *n*. Characters are numbered from zero. For instance, to take only the first three characters of the parameter "blivot", you would use the notation

```
{blivot|0:3}
```

Conditional Assembly

Very large assembly language programs usually require tailoring for specific configurations. Most of this customization involves limited changes to parts of the program. A technique called *conditional assembly* allows you to select one of two sections of code depending on the value of an expression. This allows you to maintain one version of the program source, which can be configured at assembly time to yield many different custom versions of the program. This process is used rather extensively in the RTL code described in Chapter 8.

The Macintosh assembler IF, ELSE, and ENDIF directives perform conditional assembly. These directives are used as follows:

```
IF     condition
(lines assembled if condition is true)
ELSE
(lines assembled if condition is false)
ENDIF
```

The ELSE part of the conditional assembly is optional. The *condition* tests the relationship of two expressions. The expressions can be either numeric or strings.

Numeric Conditionals

A numeric condition requires that both expressions be constants or the difference between two labels. A special form of the numeric condition checks to see if a single expression is equal to zero. The statement

```
IF     expression
```

is true if the expression is nonzero and false otherwise. This feature can be used to conditionally assemble code for a feature as follows:

```
FEATURE     EQU  0          ; to disable; 1 to enable
            .
            .
            .
            IF    FEATURE   ; Conditional code for feature
            .
            . (Code if feature is enabled)
            .
            ELSE            ; Optional else part
            .
            . (Code if feature is disabled)
            .
            ENDIF
```

Normal numeric conditionals compare the value of two expressions.

The syntax is

```
IF    expr1 comp expr2
```

where *expr1* and *expr2* are two expressions. *Comp* is a comparison operator, which determines how the expressions are to be compared, as shown in the following list:

Comparison Operator	Meaning
=	True if *expr1* is equal to *expr2*
<>	True if *expr1* is not equal to *expr2*
<	True if *expr1* is less than *expr2*
<=	True if *expr1* is less than or equal to *expr2*
>	True if *expr1* is greater than *expr2*
>=	True if *expr1* is greater than or equal to *expr2*

String Conditionals

String conditionals are more limited than numeric conditionals. A string conditional can compare only two strings for equality or inequality using the "=" and "<>" comparison operators.

String conditionals are extremely useful in defining macros. You can use a string conditional to test whether a parameter is present on a Lisa-style macro as follows:

```
.MACRO    CTEST
IF        '%1' <> ''     ; Parameter 1 is present
MOVE      %1,D1          ; Place it in D1
ELSE                     ; Parameter 1 not present
CLR.W     D1             ; Use zero
.ENDIF                   ; End conditional
.
.
.
.ENDM
```

This macro places the first parameter, if present, in D1.W. If the macro was invoked with no parameters, a zero is placed in D1.W.

Conditionals inside macros can also be used to avoid redundant instructions. Often a macro parameter is placed in a register using a MOVE instruction. If the macro is called with the correct register name as the parameter, you can avoid generating the MOVE instruction. For example, if the first parameter is to be placed in D1.W, you can use the following code:

```
MACRO     CTEST1 d1parm =
IF        '{d1parm}' <> 'D1'   ; Already in D1?
MOVE.W    {d1parm},D1          ; No, move it there
ENDIF                          ; Parameter now in D1.W
  .
  .
  .
  |
```

You can combine the above examples to produce a macro that will load a default value and avoid redundant MOVE instructions by placing one conditional inside the other. This is left as an exercise.

Interfacing to High-Level Languages

One of the most common uses for assembly language is to add functionality or speed to a program written in a high-level language. Because this is a such a common technique, there are very few high-level language compilers that do not allow it. You can take advantage of this capability in your assembly language programs by calling language run-time support routines. These routines typically include I/O support, conversion, and other useful capabilities. It is often possible to call these routines directly from assembly language.

The technique for interfacing assembly language to a high-level language is different for each compiler. Some compilers insist on having a main program written in a high-level language. Others allow a main program to be in assembly language, but require certain initialization procedures in the main program. Still others have no such restrictions. The proper techniques are usually documented (although not always well) in one of the manuals associated with the high-level language you are using.

In interfacing to a high-level language, you need to answer the following questions:

1. Does the language require a high-level main program? Certain languages, such as Microsoft BASIC, do.
2. If the language allows the main program to be in assembly language, does the main program have to do anything to make the run-time library work?

3. How does one call a high-level procedure from assembly language? Is the sequence any different for calling the run-time modules directly?

4. What registers are preserved by the run-time modules and the routines written in the high-level language? What registers are destroyed? What registers must be preserved by an assembly language subroutine?

5. Do you have to do anything special to exit to the operating system? Many languages automatically close all open files upon exiting. If you are coding a main program in assembly language, you may have to code a call to the language exit or stop routine.

6. What techniques are available for accessing global data areas from assembly language? Do you have to pass all data as parameters to the assembly language procedure?

We will now explore mixing assembly language with the following Macintosh high-level languages:

- Microsoft BASIC Version 2.0
- Consulair C Version 1.0
- Lisa Pascal Version 3.9

Sieve of Eratosthenes Program

We will write the same program in each of these languages: a program called the *Sieve of Eratosthenes.* This program computes a set of integers called *prime numbers.* A prime number has no factors other than itself and one. The first five prime numbers are 1, 2, 3, 5, and 7. Note that even numbers (other than 2) can never be prime, since they are by definition divisible by 2.

The sieve program uses a large array of flag variables representing the odd integers starting at 3. The first element of the array must be a prime number. The algorithm works by removing multiples of the prime numbers, starting with the first element. After all the multiples are removed, the numbers remaining are prime.

This program is used extensively in benchmarking compiler products, as it reflects compiler efficiency in looping, array manipulation, and simple arithmetic. By placing parts of the program in assembly language, you can save a substantial amount of execution time.

Listing 5.3 contains the sieve program written in Microsoft BASIC. This version of the program measures the time required to find the first 1899 primes. An array of integers called "flags" represents the odd integers from 3 to 16385. Each member of the array contains a 1 if the corresponding integer is a prime number, and a 0 otherwise.

Line 3 reads the TIMER variable, which contains the time since the system was booted, in seconds.

Lines 4 and 20 cause the entire program to be repeated 10 times. This ensures that the execution time is large with respect to the timer interval, minimizing errors due to timer granularity. Microsoft BASIC reports time in units of whole seconds. The maximum error in measuring an interval is one second. You should therefore measure intervals of over 100 seconds. This makes the timer error less than one percent.

Line 5 initializes a counter variable to zero. This variable will be incremented each time a prime number is found.

Lines 6–8 set all members of the flag array to 1 to indicate prime. The program finds prime numbers by eliminating multiples of primes. The first element of the array must be a prime number, and the program initially assumes that all the numbers are prime.

```
 1: DEFINT A-Z                              'All integer variables in this program
 2: DIM flags(8191)                         'Declare flags array
 3: A# = TIMER                              'Fetch start time
 4: FOR j = 1 TO 10                         '10 iterations
 5:   count = 0                             'No primes found yet
 6:   FOR i = 1 TO 8191                     'Set all
 7:     flags(i) = 1                        '       flags to TRUE
 8:   NEXT i                                '-----------------------------------
 9:   FOR i = 1 TO 8191                     'Start elimination loop
10:     IF flags(i) = 0  THEN GOTO NotPrime 'Skip nonprime numbers
11:       prime = i + i + 1                 'Transform array index to prime number
12:       k = i + prime                     'Calculate first multiple to eliminate
13:       WHILE k <= 8191                   'Eliminate all multiples of prime
14:         flags(k) = 0                    'Zero out current flag word
15:         k = k + prime                   'Calculate next multiple
16:       WEND                              'Repeat loop till done
17:       count = count + 1                 'Indicate another prime found
18:  NotPrime:                              'Here if number not prime
19:   NEXT i                                '-----------------------------------
20: NEXT j                                  'Repeat entire program 10 times
21: B# = TIMER                              'Fetch end time
22: PRINT count;" Primes.  Time = ";B#-A#   'Print prime count and time it took
```

Listing 5.3 *– Sieve of Eratosthenes in Microsoft BASIC*

Lines 9–19 form a loop that eliminates multiples of prime numbers. If the number under consideration is not prime (its flag variable is zero), the IF statement at line 10 skips the elimination process. Line 11 calculates the prime number corresponding to the array index i. The first such number (for i=1) is 1 + 1 + 1, or 3. Line 12 calculates the array index that corresponds to the first multiple of the prime number. Note that since only the odd numbers are represented by the array, this is the first odd multiple of the prime number. Lines 13–16 eliminate the multiples from the flag array. Line 17 increments the count of prime numbers found.

Line 21 reads the TIMER variable at the end of the process. Line 22 prints the number of prime numbers found and the time required.

Two areas of this program consume a lot of time. The first is the loop that sets all the flag variables to 1 (lines 6–8). The second is the elimination loop (lines 9–19). The examples that follow illustrate how to replace these sections of code with assembly language. You can save a considerable amount of time by doing this, as shown in the following table:

Language	Unoptimized	Optimized	Ratio
BASIC	308.7	0.259	1191.9:1
C	0.741	0.241	3.1:1
Pascal	0.643	0.242	2.7:1

All times are in seconds per iteration. Some of the programs required a large number of iterations to obtain accurate timings.

BASIC fares poorly in this comparison because it is an *interpreted* rather than a *compiled* language. A language interpreter is a program that operates on the program source. This means that each statement in the program is retranslated each time it is executed. A compiler, on the other hand, translates the program into machine instructions. This is several hundred times faster, as shown by the timings above.

The exact savings you can achieve by recoding part of a program depends on the algorithm, the language, and the compiler being used. The sieve program is a rigorous test of a compiler. Recoding parts of a more ordinary program might not yield quite the savings indicated above. A ratio of 2:1 might be a bit more realistic.

Microsoft BASIC Version 2.0

Microsoft BASIC programs can call assembly language subroutines with the CALL statement. The subroutine must reside in a special disk file known as a *machine language library*. There is no provision for an assembly language routine to call a part of the BASIC program. You can, however, call some of the BASIC run-time support routines from assembly language.

The BASIC program that calls an assembly language routine must have a LIBRARY statement which identifies the library file. The format of this statement is

```
LIBRARY    filename
```

where *filename* is the name of the library file. This statement must execute before any calls to subroutines in the library. You can have up to five libraries open at any given time. The LIBRARY CLOSE statement allows you to free up the memory associated with a library.

You invoke the routine from BASIC using a CALL statement. For instance, to call a routine named STUFF with two integer parameters, you can write

```
CALL STUFF(A%,B%)
```

or

```
STUFF A%,B%
```

The CALL reserved word is assumed in the second form. Due to an anomaly in BASIC, the syntax

```
STUFF(A%,B%)
```

(using parentheses without the CALL reserved word) is not legal.

BASIC treats the routine names as variables. These variables contain the address of the routine in memory. An integer variable is not large enough to store this information. If your program contains a DEFINT statement, which would cause the routine name to be classified as an integer, you must use either the "!" (single-precision floating point) or "#" (double-precision floating point) characters after the name in the CALL statement. For example, the call to routine STUFF in the above example would take the following form:

```
DEFINT A-Z
CALL STUFF#(A,B)
```

or

```
STUFF# A,B
```

Note that the two integer parameters no longer need the "%" character at the end, because of the DEFINT statement.

Several conventions apply to the use of registers in the subroutine. Upon entry to each subroutine, A4 contains the address of a longword that is unique to the entire library. If the routines in the library need a global data area, you must allocate it with a dynamic memory call (described in Chapter 6), and store the handle in the longword whose address is in A4. Before returning to BASIC, set D0 to a nonzero value if the routine will be called again, and zero otherwise. A return value of zero allows BASIC to reclaim the memory for the routine if needed. Only registers A5 and A7 need be preserved by the subroutine.

The library must have a routine named LIBinit, which BASIC calls while processing the LIBRARY statement. This routine receives the address of a version number record in A0 on entry. The format of a version number record is shown in Figure 5.7.

The first two fields of this record are the BASIC version number. Version numbers are written as a number with a decimal fraction. For instance, Microsoft BASIC version 2.00 has a version number of 2 and a revision number of 0. The first word of the version number record would contain 2 in this case, and the second word would contain 0.

Offset 4 in the version number record indicates the math package in use. A value of 0 in this word indicates decimal (BCD) arithmetic, and a value of 1 indicates the IEEE format mentioned in Chapter 2.

Offset 6 is set by the LIBinit routine to indicate compatibility with the version of BASIC in use. Set this field to −1 if the library is incompatible

Offset	Contents	Label
0	BASIC Version Number	BasicVersion
2	BASIC Revision Number	BasicRevision
4	BASIC Math Pack Number	Math
6	Library Return Value	Result
10	Reserved (8 bytes)	Reserved

Figure 5.7 *– BASIC version record*

with BASIC, and 0 otherwise. BASIC will terminate the program with an Illegal Function Call error if the library indicates incompatibility.

Offsets 10–17 are reserved for future expansion, and are set to zero by BASIC version 2.00.

BASIC library routines receive parameters by calling routines supplied by BASIC. There are five such routines that are generally useful:

Routine Name	Purpose
GetNextLibArg	Returns address and type of next parameter
IntegerArg	Converts parameter to 16-bit integer
LongArg	Converts parameter to 32-bit integer
BasicErr	Reports error in library routine
CheckBreak	Checks to see if user aborted program

You must call the GetNextLibArg routine for each parameter. This code returns the parameter type in D0 and the parameter address in A2. The following table lists the parameter types:

Value	Data Type
0	End of parameter list
1	Null (omitted) parameter
2	String variable
3	Integer (16-bit) variable
4	Single-precision floating point number
5	Double-precision floating point number

In this book, we will use only integer variables. Documentation on the other variable types is available from Microsoft in a document entitled *Microsoft BASIC for the Macintosh—Building Machine Language Libraries.*

The IntegerArg and LongArg routines convert a parameter to a 16- or 32-bit integer, respectively. The converted value comes back in register D3. Normally, you call GetNextLibArg and then IntegerArg or LongArg immediately.

The BasicErr routine reports an error to the user. Error codes are listed in the back of the Microsoft BASIC manual. Place the error code desired in D2.W. This routine does not return. Instead, program execution terminates with the error, and BASIC reverts to immediate mode.

The CheckBreak routine allows you to test whether the user typed Command-Period. This routine is similar to the CHKABORT macro discussed in Chapter 4. There are no parameters. The Z bit in the condition code register is cleared if the user tried to abort the program. The usual calling sequence for this routine is as follows:

```
        JSR                CheckBreak(A5)  ; See if abort desired
        BEQ                @1              ; EQ => Z set; no abort
        (Code for abort)                   ;
@1:                                        ; Here if no abort
```

Note that BASIC checks for abort quite frequently, so you need call this routine only if your assembly language routine takes a lot of time to complete.

Listing 5.4 shows the BASIC program modified to call the two assembly language routines. Note that the number of iterations had to be increased, because the BASIC TIMER variable reports time only in whole seconds.

There are two assembly language routines: one called SetFlags, which initializes the flags array, and one called Sieve, which performs the actual algorithm. SetFlags has two parameters: the first element of the flags array and the number of elements. Sieve has an additional parameter, the count variable. Sieve returns the number of primes found in this variable.

The equates necessary for the machine language routines appear in Listing 5.5. This listing contains only the definitions for the routines defined above (lines 7–11) and for the version number record (lines 15–18).

Listing 5.6 contains the LIBinit routine for the optimized sieve program. This code assumes compatibility with BASIC versions 2.00 and higher. Line 5 clears the result field, and lines 6–7 check the version number. Line 8 handles the case where the version number is less than 2.00. Line 9 clears D0.L to indicate that the LIBinit routine can be purged from memory.

Listing 5.7 contains the SetFlags routine for use with BASIC. SetFlags has

```
DEFINT A-Z                               'All integer variables in this program
DIM flags(8191)                          'Declare flags array
A# = TIMER                               'Fetch start time
LIBRARY "BSieve.RSRC"                    'Declare library file
count = 0                                'Must initialize for library to work
FOR j = 1 TO 1000                        'Repeat 1000 times for accurate timing
  SetFlags# flags(1),8191                'Set all flag variables
  Sieve# flags(1),8191,count             'Perform 1 iteration of the sieve
 NEXT j                                  'End of repeat loop
B# = TIMER                               'Fetch end time
PRINT count;" Primes.  Time = ";B#-A#    'Print count and time it took
```

***Listing 5.4** – The optimized BASIC sieve program* two parameters. The first parameter is the first element of the flags array. Line 5 fetches the address of this parameter. Lines 6–7 check the parameter type. Note that BASIC does not provide any indication that the parameter is an array element. This makes it impossible to check types completely. Lines 9–10 fetch the value of the second parameter, which is the number of elements in the array. Line 11 places the address of the

```
 1: ;
 2: ;       BASIC.D.  Include file for use with BASIC libraries.
 3: ;
 4: ;       Routines available to the machine language code via JSR with
 5: ;       offset from A5:
 6: ;
 7: GetNextLibArg   EQU     $2A     ;       Fetch next parameter
 8: IntegerArg      EQU     $32     ;       Convert parameter to integer value
 9: LongArg         EQU     $3A     ;       Convert parameter to long (32 bit)
10: BasicError      EQU     $42     ;       BASIC error routine
11: CheckBreak      EQU     $A2     ;       Check for "stop" by user
12: ;
13: ;       Version number record offsets:
14: ;
15: BasicVersion    EQU     $0      ;       BASIC version number
16: BasicRevision   EQU     $2      ;       BASIC revision number
17: BasicMath       EQU     $4      ;       Math pack (0=>BCD; 1=>IEEE)
18: BasicResult     EQU     $6      ;       Result (0=>ok; 1=>incompatible)
```

Listing 5.5 *– BASIC equate file*

```
 1: ;
 2: ;       LIBinit routine for sieve program
 3: ;
 4:         INCLUDE BASIC.D         ;       Define BASIC offsets
 5:         CLR.W   BasicResult(A0) ;       Assume compatible
 6:         CMP.W   #2,BasicVersion(A0);    See if version 2 or higher
 7:         BGE     @1              ;       It's OK
 8:         MOVE.W  #-1,BasicResult(A0);    Indicate no good
 9: @1:     CLR.L   D0              ;       Indicate OK to purge LIBinit
10:         RTS                     ;       Done
```

Listing 5.6 *– LIBinit routine*

array into register A0. Lines 11–14 set all bytes in the array to 1. Since the array is an array of integers, only half the space is used. Line 15 sets D0 to −1, indicating that the routine is not to be purged, since it will be called ten times. Lines 18–20 illustrate how to call the BasicErr routine.

The actual prime number (sieve) algorithm appears in Listing 5.8. This code uses an array of bytes, rather than an array of words, to simplify the calculations. Since the BASIC program does not reference any values in the flags array, we use only the first half of the storage.

Lines 5–11 place the array address in register A0 and the number of elements in D1. The integer variables in Listing 5.3 are placed in data registers as follows:

BASIC Variable	**Data Register**
count	D2.W
i	D3.W
prime	D5.W
k	D6.W

D4.W contains the array size for comparison purposes. D1 is used as a

```
 1: ;
 2: ;       SetFlags routine for optimized BASIC sieve program
 3: ;
 4:          INCLUDE BASIC.D        ;      Include BASIC defs.
 5: SetFlags JSR    GetNextLibArg(A5);     Fetch next parameter
 6:          CMP.W  #$3,D0          ;      Integer variable?
 7:          BNE    @2              ;      No, bail out
 8:          MOVE.L A2,-(SP)        ;      Save address on stack
 9:          JSR    GetNextLibArg(A5);     Fetch next parameter
10:          JSR    IntegerArg(A5)  ;      Convert parameter
11:          MOVE.L (SP)+,A0        ;      A0 -> Array
12:          SUB.W  #1,D3           ;      DBRA-Adjust count
13: @1:      MOVE.B #1,(A0)+        ;      Set flag byte
14:          DBRA   D3,@1           ;      Loop till done
15:          MOVE.W #-1,D0          ;      Not purgeable
16:          RTS                    ;      Return
17: @2:                             ;      Here if parameter mismatch
18:          MOVE.W #5,D2           ;      "Illegal Function Call"
19:          JSR    BasicError(A5)  ;      Bail out
20:          RTS                    ;      Quit
```

Listing 5.7 *– Assembly language SetFlags routine for Microsoft BASIC*

```
;
;       Sieve routine.  Does the main part of the sieve algorithm
;
        INCLUDE BASIC.D           ;       Include BASIC defs.
Sieve   JSR     GetNextLibArg(A5);       Fetch next parameter
        CMP.W   #$3,D0            ;       Integer variable?
        BNE     @5                ;       No, bail out
        MOVE.L  A2,-(SP)          ;       Save address on stack
        JSR     GetNextLibArg(A5);       Fetch next parameter
        JSR     IntegerArg(A5)    ;       Convert parameter
        MOVE.W  D3,D1             ;       Place count in D1
        MOVE.L  (SP)+,A0          ;       A0 -> Array
        MOVE.W  D1,D4             ;       Replicate count
        SUBQ.W  #1,D1             ;       DBRA-Adjust D1
        CLR.L   D2                ;       D2 =  count
        CLR.W   D3                ;       D3 =  i
@1:     TST.B   0(A0,D3.W)        ;       if(flags[i])
        BEQ     @4                ;       Not prime, skip this one
        MOVE.W  D3,D5             ;       D5 = "prime" variable
        ADD.W   D3,D5             ;       prime = i + i
        ADDQ.W  #3,D5             ;                     + 3;
        MOVE.W  D3,D6             ;       D6 = "k"
@2:     ADD.W   D5,D6             ;       Add in "prime"
        CMP.W   D4,D6             ;       while (k <= SIZE)
        BGT     @3                ;       Not <=, do next iteration
        CLR.B   0(A0,D6.W)        ;       flags[k] = 0;
        BRA     @2                ;       Loop again
@3:     ADDQ.W  #1,D2             ;       Increment count variable
@4:     ADDQ.W  #1,D3             ;       Increment i
        DBRA    D1,@1             ;       Loop till done
        JSR     GetNextLibArg(A5);       Get last parameter
        CMP.W   #3,D0             ;       Integer variable?
        BNE     @5                ;       No, quit
        MOVE.W  D2,(A2)           ;       Set return value
        MOVE.W  #-1,D0            ;       Mark not purgeable
        RTS                       ;       Return
@5:                               ;       Here if parameter mismatch
        MOVE.W  #5,D2             ;       "Illegal Function Call"
        JSR     BasicError(A5)    ;       Bail out
```

Listing 5.8 – *Assembly language Sieve routine for Microsoft BASIC* loop counter, allowing the use of a DBRA loop. The reason for placing all the variables in registers instead of in memory is that register operations are smaller and take less time than memory operations on the 68000. Optimizing code on the 68000 generally involves minimizing the number of times the processor must access memory. Shortening the time required by the instructions executed with the highest frequency produces the biggest time savings. The WHILE loop in the sieve algorithm (lines 13–16 of Listing 5.4) contains the most frequently executed instructions. This is the most important section of code to optimize. The loop for elimination of multiples is the second most important section.

BASIC has some important restrictions on generating the library file:

1. Each routine must be assembled separately.
2. Routines cannot have data defined with DS directives.
3. You must use the linker and a program called RMAKER to place each routine in the segment whose name matches the name used in the BASIC program.
4. The starting address is the first location in each routine.

Listing 5.9 contains the linker command file for the BASIC sieve library. The /Output directive causes the linker to place the load module in a file named Bsieve. The "<" characters between the file names cause each routine to be placed in a different *segment*. A Macintosh segment is a part of a program that can be loaded into memory as a separate entity.

Following the link step, you must run a program called RMAKER to produce the library file. RMAKER is described in detail in Chapter 6. The RMAKER input file for the BASIC sieve library appears in Listing 5.10.

```
/Output Bsieve
LIBinit.Rel
<
Bsieve.Rel
<
BSetFlags.Rel
$
```

***Listing 5.9** – BASIC sieve linker command file*

The first two lines of this file specify the output file (BSieve.RSRC) and the file type (BLIB). Lines 3–6 specify a code resource named LIBinit, which is the first segment in the linker output file. Lines 8–11 cause the next segment to be named Sieve, and lines 13–16 name the last segment SetFlags. The segment order in the RMAKER file must match the order in the linker file. The segments must be numbered consecutively from 1.

Consulair C

The sieve program in C is shown in Listing 5.11. This program is a literal translation of the BASIC version in Listing 5.3. C differs from BASIC in that array elements are numbered from zero instead of from one. This changes the dimension of the flags array, as well as the limits on the for loops.

The timing technique is also different. Location 16A (hex) in memory contains a longword that is incremented every 1/60th of a second. Lines 13–14 and 37 fetch the value of this word before and after the loop. The call to the printf routine in lines 38–40 outputs the difference in the two. The ticks are converted to seconds by a rather complicated formula in lines 39–40. Line 39 calculates the number of whole seconds, and line 40 calculates fractional seconds.

Consulair C is perhaps the easiest language to interface to MDS assembly language. You can call assembly language routines from C or C routines from assembly language. Since the C language uses subroutine calls for I/O, you can call any of the C library routines as well.

```
 1: BSieve.RSRC
 2: BLIB
 3: TYPE CODE=GNRL
 4: LIBinit,1
 5: .R
 6: Bsieve CODE 1
 7:
 8: TYPE CODE=GNRL
 9: Sieve,2
10: .R
11: Bsieve CODE 2
12:
13: TYPE CODE=GNRL
14: SetFlags,3
15: .R
16: Bsieve CODE 3
```

***Listing 5.10** – BASIC sieve RMAKER input file*

```
/*
 *      Sieve Program for Consulair C
 */                                         /********************************/
#define ITER    10                          /* Number of iterations         */
#define SIZE    8190                        /* Completion count             */
#define TIMELOC 0x16A                       /* Location of ticks global var.*/
main()                                      /*                              */
{                                           /*------------------------------*/
        short   i,j,k,count,prime;          /* Integer variables            */
        char    flags[SIZE+1];              /* Flag array                   */
        long    *timeptr,start,end;         /* Time variables               */
                                            /*------------------------------*/
        timeptr = TIMELOC;                  /* Build a pointer to time locn */
        start   = *timeptr;                 /* Fetch start time             */
        for(j=0;j<ITER;j++)                 /* Loop through 10 iterations   */
        {                                   /*                              */
          count = 0;                        /* Clear out count of primes    */
          for(i=0;i<SIZE;i++)               /* Initialize                   */
            flags[i] = 1;                   /*              flag array      */
                                            /*------------------------------*/
          for(i=0;i<=SIZE;i++)              /* Loop to eliminate multiples  */
          {                                 /*                              */
            if(flags[i])                    /* Prime?                       */
            {                               /* Here if yes                  */
              prime = i + i + 3;            /* Compute actual prime number  */
              k = i + prime;                /* Compute first multiple       */
              while(k<=SIZE)                /* Now eliminate multiples      */
              {                             /*                              */
                flags[k] = 0;               /* Turn off flag                */
                k += prime;                 /* Compute next array index     */
              }                             /* ----- while(k<=SIZE) -----   */
              count++;                      /* Another prime found          */
            }                               /* -----   if(flags[i])   ----- */
          }                                 /* --- for(i=0;i<=SIZE;i++) --- */
                                            /*------------------------------*/
        }                                   /* --- for(j=0;j<=ITER;j++) --- */
        end = *timeptr;                     /* Fetch ending time            */
        printf("%ld.%02ld Seconds ",        /* Report                       */
          (end-start)/60L,                  /*         time                 */
          (((end-start)%60L)*100L)/60L);    /*              taken           */
        printf("%d Primes found\n",count);  /* Report number found          */
}                                           /********************************/
```

***Listing 5.11** – Consulair C sieve program*

Version 1.0 of Consulair C has five data types:

C Data Type	Length
char	8 bits
short	16 bits
int	32 bits
long	32 bits
pointer	32 bits

Later versions of this compiler should also support floating point.

The first seven parameters to a subroutine are passed in registers D0–D6. Subsequent parameters are pushed on the stack in reverse order. Parameters less than 32 bits are extended to 32 bits before calling the subroutine.

For instance, consider the following section of C code:

```
int   i1,i2,i3,i4,i5,i6,i9;
char  c5,c6,c7,c8;
xyzzy(i1,i2,i3,i4,i5,c7,c8,i9);
```

This generates the following assembly language code:

```
MOVE.L    i9,-(SP)     ; Push last parameter
MOVE.B    c8,D0        ; Push
EXT.W     D0           ;      parameter
EXT.L     D0           ;                number
MOVE.L    D0,-(SP)     ;                      8
MOVE.B    c7,D6        ; Parameter 7
EXT.W     D6           ;            goes in
EXT.L     D6           ;                    D6
MOVE.L    i6,D5        ; Parameter 6 to D5
MOVE.L    i5,D4        ; Parameter 5 to D4
MOVE.L    i4,D3        ; Parameter 4 to D3
MOVE.L    i3,D2        ; Parameter 3 to D2
MOVE.L    i2,D1        ; Parameter 2 to D1
MOVE.L    i1,D0        ; Parameter 1 to D0
JSR       xyzzy        ; Call subroutine
ADD.L     #8,SP        ; Pop two stack parameters
```

Values returned by a C function appear in either D0 or A0. D0 is the return register for all *char, short, int,* or *long* data. A0 is used for pointers of any kind. The subroutine can modify the contents of all registers except A5–A7. Note that the calling routine removes the parameters from the stack following the JSR instruction.

The advantage of the Consulair scheme is that most subroutines have seven or fewer parameters. This means that most subroutines can receive parameters in registers, rather than on the stack. Assembly language routines can use the parameters in the registers directly.

Listing 5.12 contains the C portion of the optimized sieve program. The assembly language code is split into the same two functions as in the BASIC version of the sieve. The number of primes is returned as the result of the function rather than as a separate parameter, however.

Listing 5.13 shows the assembly language routine for the optimized C sieve program. This is the same code as in Listings 5.7 and 5.8, except that the parameters are now passed in registers. This makes the code considerably less complex.

Lisa Pascal

The original development system for Macintosh software is something that ran on the Lisa (Macintosh XL) computer. This system is called the Lisa Pascal Workshop. It was originally marketed for writing applications for

```
/*
 *      Sieve Program for Consulair C
 */                                     /********************************/
#define ITER    10                      /* Number of iterations         */
#define SIZE    8190                    /* Completion count             */
#define TIMELOC 0x16A                   /* Location of ticks global var.*/
main()                                  /*                              */
{                                       /*------------------------------*/
        int     j,count;                /* Integer variables            */
        char    flags[SIZE+1];          /* Flag array                   */
        long    *timeptr,start,end;     /* Time variables               */
                                        /*------------------------------*/
        timeptr = TIMELOC;              /* Build a pointer to time locn */
        start   = *timeptr;             /* Fetch start time             */
        for(j=0;j<ITER;j++)             /* Loop through 10 iterations   */
        {                               /*                              */
          setflags(flags,SIZE);         /* Set the flags                */
          count = sieve(flags,SIZE);    /* Do the sieve algorithm       */
        }                               /* --- for(j=0;j<=ITER;j++) --- */
        end = *timeptr;                 /* Fetch ending time            */
        printf("%ld.%02ld Seconds ",    /* Report                       */
          (end-start)/60L,              /*        time                  */
          (((end-start)%60L)*100L)/60L);/*            taken             */
        printf("%d Primes found\n",count); /* Report number found       */
}                                       /********************************/
```

***Listing 5.12** – Optimized C sieve program* the Lisa Office System, the predecessor to Macintosh. The subroutine calling sequence for this compiler is also used by many of the Macintosh Toolbox routines.

Listing 5.14 contains the Pascal version of the sieve program. Lines 1–4 specify compiler options for generating Macintosh code. Lines 11–15 and 26–33 initialize the Macintosh Toolbox. The other languages perform this initialization automatically. Listing 8.2 (in Chapter 8) contains the assembly language equivalent to this code. This program also uses zero-relative array subscripts.

```
;
;       Assembler part of optimized C sieve program
;
         XDEF   setflags        ;       Set all flag bytes
setflags MOVE.L D0,A0           ;       A0 -> Array
         SUB.W  #1,D1           ;       DBRA-Adjust count
@1:      MOVE.B #1,(A0)+        ;       Set flag byte
         DBRA   D1,@1           ;       Loop till done
         RTS                    ;       Return
;
;       Sieve routine.  Does the main part of the sieve algorithm
;
         XDEF   sieve           ;       Make accessible
sieve    MOVE.L D0,A0           ;       A0 -> Flags array
         MOVE.W D1,D4           ;       Replicate count
         SUBQ.W #1,D1           ;       DBRA-Adjust D1
         CLR.L  D2              ;       D2 =  count
         CLR.W  D3              ;       D3 =  i
@1:      TST.B  0(A0,D3.W)      ;       if(flags[i])
         BEQ    @4              ;       Not prime, skip this one
         MOVE.W D3,D5           ;       D5 = "prime" variable
         ADD.W  D3,D5           ;       prime = i + i
         ADDQ.W #3,D5           ;                     + 3;
         MOVE.W D3,D6           ;       D6 = "k"
@2:      ADD.W  D5,D6           ;       Add in "prime"
         CMP.W  D4,D6           ;       while (k <= SIZE)
         BGT    @3              ;       Not <=, do next iteration
         CLR.B  0(A0,D6.W)      ;       flags[k] = 0;
         BRA    @2              ;       Loop again
@3:      ADDQ.W #1,D2           ;       Increment count variable
@4:      ADDQ.W #1,D3           ;       Increment i
         DBRA   D1,@1           ;       Loop till done
         MOVE.L D2,D0           ;       Set return value
         RTS                    ;       Return
```

***Listing 5.13** – Assembly routine for C sieve program*

```
{$M+}
{$X-}
{$R-}
{$D-}
(***********************************************************************)
(*                                                                     *)
(*   Lisa Pascal version of the sieve of Eratosthenes                  *)
(*                                                                     *)
(***********************************************************************)
PROGRAM   sieve;                                 (*                      *)
USES {$U-}                                       (* Standard definitions *)
      {$U OBJ/MemTypes}  MemTypes,               (* Memory type defns    *)
      {$U OBJ/QuickDraw} QuickDraw,              (* QuickDraw stuff      *)
      {$U OBJ/OSIntf}    OSIntf,                 (* O/S interface        *)
      {$U OBJ/ToolIntf}  ToolIntf;               (* Toolbox interface    *)
CONST                                            (* Constant section     *)
      ASIZE = 8190;                              (* Array size           *)
      ITER = 10;                                 (* Number of iterations *)
      TIMELOC = 362;                             (* 0x16A in decimal     *)
VAR                                              (*----------------------*)
      i,j,k,count,prime : integer;               (* 16-bit temporaries   *)
      flags             : array[0..ASIZE] of char; (* Flags array        *)
      start,stop        : longint;               (* Time variables       *)
      timeptr           : ^longint;              (* Points to timer long *)
BEGIN                                            (*----------------------*)
      InitGraf(@thePort);                        (* Initialize QuickDraw *)
      InitFonts;                                 (* Initialize Font Mgr  *)
      InitWindows;                               (* Initialize Window Mgr *)
      FlushEvents(everyEvent,0);                 (* Clear Event Queue    *)
      InitMenus;                                 (* Initialize Menu Mgr  *)
      TEInit;                                    (* Initialize Text Edit *)
      InitDialogs(NIL);                          (* Initialize Dialogs   *)
      InitCursor;                                (* Set cursor to arrow  *)
                                                 (*----------------------*)
      timeptr := pointer(TIMELOC);               (* -> Tick word         *)
      start   := timeptr^;                       (* Fetch starting value *)
      FOR j := 1 TO ITER DO                      (* Iteration loop       *)
      BEGIN                                      (*                      *)
        count := 0;                              (* Clear prime count vbl *)
        FOR i := 0 TO ASIZE DO                   (* Initialize           *)
          flags[i] := '1';                       (*           flag array*)
        FOR i := 0 TO ASIZE DO                   (* Start elimination loop*)
        BEGIN                                    (*++++++++++++++++++++++*)
          IF flags[i] = '1'  THEN                (* if number is prime   *)
          BEGIN                                  (*======================*)
            prime := i + i + 3;                  (* Calculate prime value *)
            k := i + prime;                      (* Calculate 1st multiple*)
            WHILE k <= ASIZE DO                  (* Eliminate multiples  *)
```

Listing 5.14 – Pascal sieve program

The optimized Pascal program appears in Listing 5.15. Lines 28–32 declare the parameters and function return types as dictated by Pascal. The only other change is the use of a Pascal TYPE declaration for the array in lines 21 and 24.

Pascal subroutines receive parameters and return function values on the stack. VAR parameters (those that can be modified by the subroutine) are passed by address. The value of non-VAR parameters that are less than four bytes is pushed on the stack. Any parameter that is longer than four bytes is passed by address. Lisa Pascal has the following simple data types:

Name	Length	Description
char	1	A single 8-bit ASCII character
boolean	1	A byte that is zero or nonzero
integer	2	16-bit signed integer
longint	4	32-bit signed integer
real	4	32-bit floating point
(pointer)	4	32-bit address

The syntax for a pointer variable is simply ^data, as shown in line 26 of Listing 5.15.

Pascal invokes a subroutine using the following steps:

1. Decrement the stack pointer to make room for the function return value, if any.

```
            BEGIN                                        (*                       *)
              flags[k] := '0';                           (* Eliminate current     *)
              k := k + prime;                            (* Calculate next        *)
            END;                                         (*-----------------------*)
            count := count + 1;                          (* One more prime done   *)
          END;                                           (*=======================*)
        END;                                             (*+++++++++++++++++++++++*)
      END;                                               (* FOR J := 1 TO ITER    *)
      stop := timeptr^;                                  (* Fetch stop time       *)
      writeln(count,' primes ',stop-start,' ticks');(* Output time            *)
  END.                                                   (* Main program          *)
                                                         (*************************)
```

Listing 5.14 – *Pascal sieve program (continued)*

```
{$M+}
{$X-}
{$R-}
{$D-}
(**************************************************************************)
(*                                                                        *)
(*   Lisa Pascal version of the sieve of Eratosthenes (optimized)         *)
(*                                                                        *)
(**************************************************************************)
PROGRAM   xsieve;                                (*                      *)
USES {$U-}                                       (* Standard definitions *)
     {$U OBJ/MemTypes}  MemTypes,                (* Memory type defns    *)
     {$U OBJ/QuickDraw} QuickDraw,               (* QuickDraw stuff      *)
     {$U OBJ/OSIntf}    OSIntf,                  (* O/S interface        *)
     {$U OBJ/ToolIntf}  ToolIntf;                (* Toolbox interface    *)
CONST                                            (* Constant section     *)
     ASIZE = 8190;                               (* Array size           *)
     ITER = 10;                                  (* Number of iterations *)
     TIMELOC = 362;                              (* 0x16A in decimal     *)
TYPE                                             (* Type section         *)
     flagtype = array[0..ASIZE] of char;         (* Flag array type      *)
VAR                                              (*----------------------*)
     i,j,k,count,prime : integer;                (* 16-bit temporaries   *)
     flags             : flagtype;               (* Flags array          *)
     start,stop        : longint;                (* Time variables       *)
     timeptr           : ^longint;               (* Points to timer long *)
                                                 (*----------------------*)
PROCEDURE setflags(var flag  : flagtype;         (* Declare              *)
                   xcount: integer); external;   (*        setflags proc*)
FUNCTION  sieve   (var flag  : flagtype;         (* Declare              *)
                   xcount: integer) : integer;   (*         sieve funct. *)
                   external;                     (*----------------------*)
BEGIN                                            (* Main program         *)
     InitGraf(@thePort);                         (* Initialize QuickDraw *)
     InitFonts;                                  (* Initialize Font Mgr  *)
     InitWindows;                                (* Initialize Window Mgr *)
     FlushEvents(everyEvent,0);                  (* Clear Event Queue    *)
     InitMenus;                                  (* Initialize Menu Mgr  *)
     TEInit;                                     (* Initialize Text Edit *)
     InitDialogs(NIL);                           (* Initialize Dialogs   *)
     InitCursor;                                 (* Set cursor to arrow  *)
                                                 (*----------------------*)
     timeptr := pointer(TIMELOC);                (* -> Tick word         *)
     start   := timeptr^;                        (* Fetch starting value *)
     FOR j := 1 TO ITER DO                       (* Iteration loop       *)
     BEGIN                                       (*                      *)
       setflags(flags,ASIZE);                    (* Set all flag bits    *)
       count := sieve(flags,ASIZE);              (* Do the sieve algorithm*)
     END;                                        (* FOR J := 1 TO ITER   *)
     stop := timeptr^;                           (* Fetch stop time      *)
     writeln(count,' primes ',stop-start,' ticks');(* Output time        *)
END.                                             (* Main program         *)
                                                 (************************)
```

Listing 5.15 – *Optimized Pascal sieve program*

2. Push the parameters from left to right. Parameters that require an odd number of bytes are padded to keep the stack pointer even. Note that this is the opposite order from C.
3. Call the subroutine with a JSR instruction.
4. Upon return from the subroutine, pop the return value, if any, from the stack.

Note that the caller pushes all values on the stack and the called routine pops off the parameters. The caller pops the function return value, if any. This is also different from C. The subroutine must preserve all registers except D0–D2 and A0–A1. There is no way for an assembly routine to access global data. The address must be passed as a parameter.

Listing 5.16 shows the sieve assembly language routines properly interfaced to Lisa Pascal. Note that the Lisa assembler directives are slightly different from the MDS assembler. The .PROC and .END statements are mandatory.

Macintosh System Calls

Application programs execute routines in the Macintosh ROM by means of system traps. Instructions that have the value 1010 in the upper four bits cause a 68000 *exception*. Exceptions are explained fully in Chapter 7. The Macintosh designers used the low-order 12 bits of the instruction to define an entire set of calls into the ROM. These calls require only 16 bits of application program memory, while a BSR or JSR would require 32 or 48 bits.

There are two formats of operating system calls, named *trap words* in the Macintosh documentation. Figure 5.8 shows the layout of these words.

The two formats are distinguished by the value of bit 11. When bit 11 is zero, the trap is an operating system call. Parameters to operating system routines are normally passed in registers. Bit 8 in an operating system call, if zero, indicates that a return value will be placed in A0 by the routine invoked by the trap word. The _NewHandle function, used in Chapter 6, is an example of such a routine. Bits 10–9 are flag bits whose meaning depends on the routine being called.

When bit 11 of the trap word is a 1, the trap is a Toolbox call. Toolbox calls normally use the Lisa Pascal calling conventions described previously, except that the JSR instruction is replaced by the trap word. The Lisa Pascal compiler can generate calls to the Toolbox routines directly by using

```
;       Assembler part of optimized PASCAL sieve program
;
         .PROC   SieveAsm        ;       Procedure statement
         .DEF    setflags        ;       Declare global
setflags MOVE.L  (SP)+,A1        ;       A1 = return address
         MOVE.W  (SP)+,D1        ;       D1 = array size
         SUB.W   #1,D1           ;       DBRA-Adjust count
         MOVE.L  (SP)+,A0        ;       A0 -> Array
@1:      MOVE.B  #1,(A0)+        ;       Set flag byte
         DBRA    D1,@1           ;       Loop till done
         JMP     (A1)            ;       Return
;
;       Sieve routine.  Does the main part of the sieve algorithm
;
        .DEF     sieve           ;       Make accessible
sieve    MOVE.L  (SP)+,A1        ;       A1 =  Return address
         MOVE.W  (SP)+,D1        ;       D1 =  Array size
         MOVE.L  (SP)+,A0        ;       A0 -> Flags array
         MOVEM.W D3-D6,-(SP)     ;       Save registers
         MOVE.W  D1,D4           ;       Replicate count
         SUBQ.W  #1,D1           ;       DBRA-Adjust D1
         CLR.W   D2              ;       D2 =  count
         CLR.W   D3              ;       D3 =  i
@1:      TST.B   0(A0,D3.W)      ;       if(flags[i])
         BEQ     @4              ;       Not prime, skip this one
         MOVE.W  D3,D5           ;       D5 = "prime" variable
         ADD.W   D3,D5           ;       prime = i + i
         ADDQ.W  #3,D5           ;                     + 3;
         MOVE.W  D3,D6           ;       D6 = "k"
@2:      ADD.W   D5,D6           ;       Add in "prime"
         CMP.W   D4,D6           ;       while (k <= SIZE)
         BGT     @3              ;       Not <=, do next iteration
         CLR.B   0(A0,D6.W)      ;       flags[k] = 0;
         BRA     @2              ;       Loop again
@3:      ADDQ.W  #1,D2           ;       Increment count variable
@4:      ADDQ.W  #1,D3           ;       Increment i
         DBRA    D1,@1           ;       Loop till done
         MOVEM.W (SP)+,D3-D6     ;       Restore registers
         MOVE.W  D2,(SP)         ;       Set return value
         JMP     (A1)            ;       Return
         .END                    ;       Mandatory END statement
```

***Listing 5.16** – Assembler routine for Pascal sieve program*

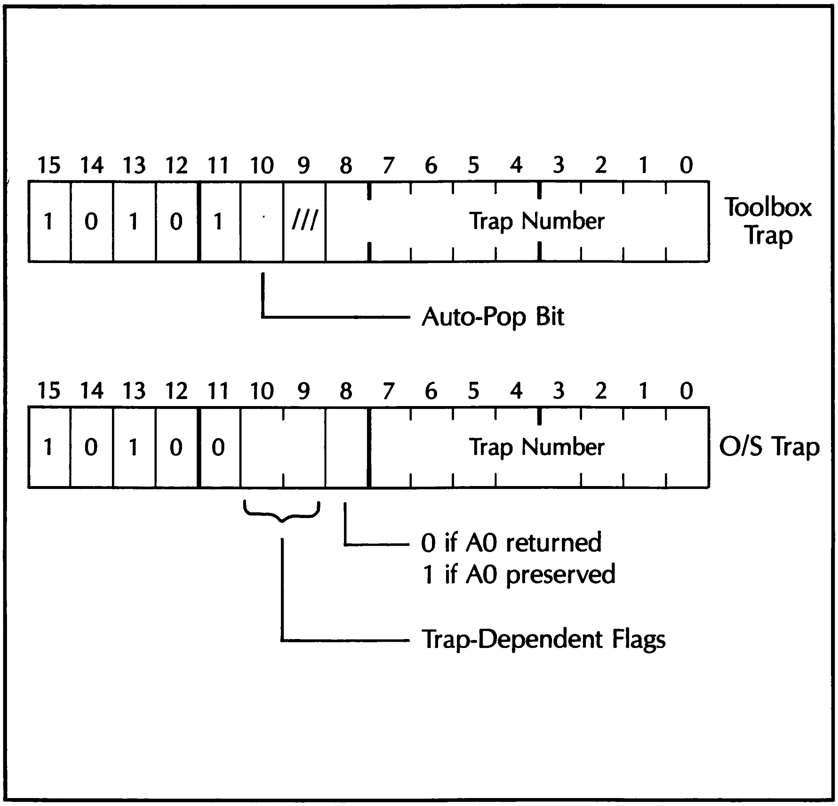


***Figure 5.8** – Macintosh trap words*

the INLINE directive to specify the trap number. This minimizes overhead in calling Toolbox routines from Pascal.

Bit 10 of a Toolbox trap is the *auto-pop* bit. This is designed for compilers that do not support the INLINE feature. Such compilers must generate a JSR to a trap word. By setting bit 10 of the trap word, you can avoid the overhead of the extra RTS instruction. The Toolbox will return directly to the instruction following the JSR if this bit is set.

SUMMARY

In this chapter, we dealt with a number of important topics:

- The concept of records and the various types of records: fixed-length, variable-length, and hybrid

- How records are arranged in memory: arrays and linked lists
- Data structures: stacks, queues, and trees
- Simple methods of sorting data: insertion sorts and interchange sorts
- Sequential, binary, and hash techniques of searching through a table
- The concept of recursive programming
- The use of macros and conditional assembly
- Mixing assembly and high-level languages
- Macintosh system calls

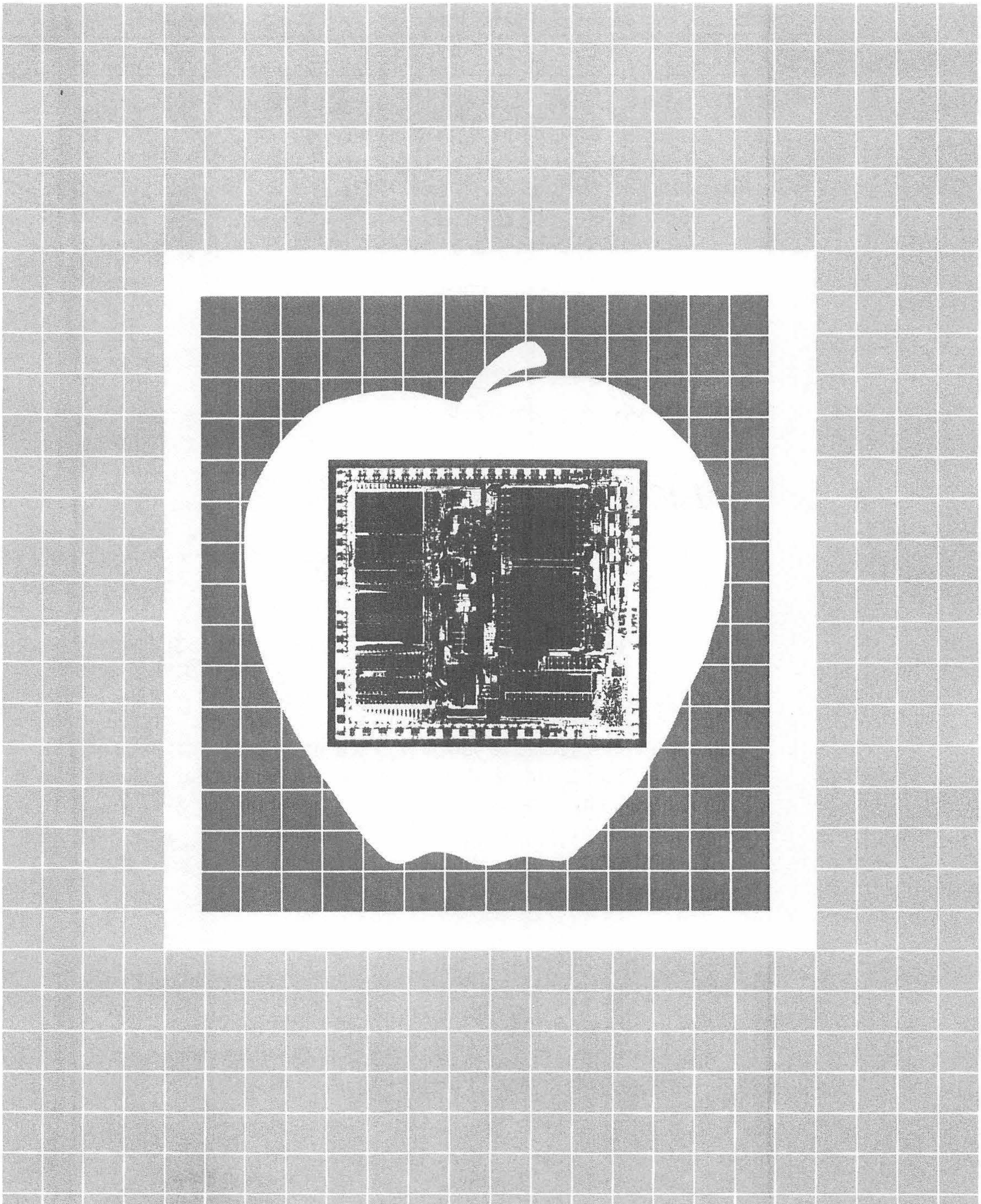
EXERCISES

1. The Fibonacci series of numbers, denoted as F(n), is recursively defined as follows:

   a. F(0) = 0, and F(1) = 1.
   b. For all other numbers, F(n) = F(n − 1) + F(n − 2). (For example, F(2) = F(1) + F(0) = 1; F(3) = F(2) + F(1) = 2, and so forth.)

   Write a recursive function called fib that returns, in D0.W, the Fibonacci number that corresponds to the number originally contained in D0.W.

2. Modify the program in Exercise 1 to be callable from the high-level languages discussed in this chapter.



Chapter 6

The Macintosh Programming Environment

He who rides a tiger cannot dismount.

—Chinese proverb

OVERVIEW

In the previous chapters, we have explored programming the Macintosh from a traditional computer science standpoint. In this chapter, we will explore elements of programming that are unique to the Macintosh. This includes several topics:

1. The QuickDraw graphics system
2. Memory layout of a Macintosh program
3. Use of the Macintosh resource capability
4. Windows, dialogs, and alerts
5. Menu operations
6. The Macintosh event system

The last part of the chapter is an explanation of a sample Macintosh application, called MacDoodle, which illustrates these principles.

Macintosh-style user interaction is based on a technique known as the "desktop metaphor," in which the computer screen (background) is analogous to a desktop containing sheets of paper (the windows), and other objects (the icons). Figure 6.1 illustrates these various areas of the desktop.

The Desktop

At the top of the screen is an area called the *menu bar.* The words appearing in this area are labels for panels called *menus.* Each menu contains a list of items that the user can choose with the mouse. The gray area of the screen is known as the *desktop* or *desk area.* Inside this area, large display areas called *windows* appear.

Each window is a separate display. Windows are like sheets of paper on the desktop and can overlap. Clicking the mouse button in a window that is behind another window causes the first window to be placed on top. This operation is akin to shuffling two pieces of paper.

The window that is on top of all the others is designated as the *active window.* This is the window titled "Book Programs" in Figure 6.1. The active window is identified by the lines on each side of the title. The area in which the title appears is called the *title bar,* and an active window's title bar is said to be *highlighted.*

The title bar of an active window may also contain an area called the *close box.* Clicking the mouse inside this box causes the window to go away. When you close a window in this way, it either disappears completely or appears in a compacted image known as an *icon,* depending on the application displaying the window.

Icons are used by the Finder to represent folders (groups of files), documents (files), and applications (programs). The trash can is really a special type of folder, as are disk drives. Files are displayed in folders according to the Finder folder number in the file information area of the file, as described in Chapter 4. Recall that each Macintosh file has a four-byte

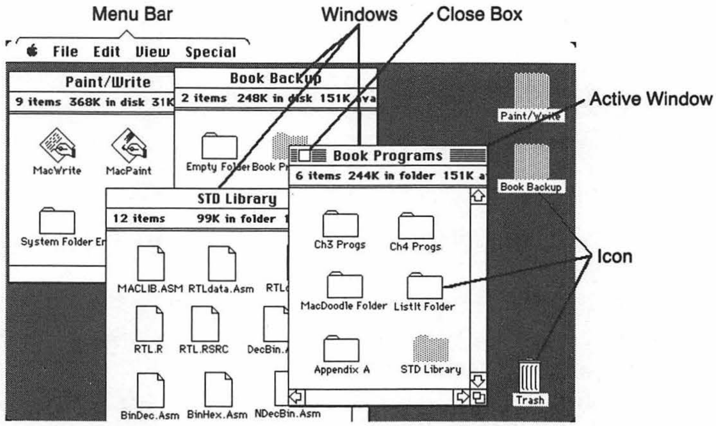


Figure 6.1 – *The Macintosh desktop*

field that identifies an application associated with the file, as explained in Chapter 4. This field is called the application's *signature.* When you open a document file in the Finder, the Finder starts the application whose signature matches the signature in the document file. The application receives the name of the document file as a parameter.

A document's icon is also determined by its application signature. The Finder constructs a file called "Desktop," which contains the icons for all the applications present on a given disk. The application itself usually has an icon different from the documents processed by that application.

QuickDraw

Output to the screen and other graphics devices is normally performed using the Macintosh QuickDraw package. QuickDraw is a substantially complete graphics software package. Our discussion only covers those points necessary for understanding the sample application at the end of this chapter. A more complete discussion can be found in *Inside Macintosh.*

The Macintosh screen is a big rectangle made up of bits. Each bit corresponds to a dot or *pixel* on the screen. There are 175,104 pixels (21,888 bytes) on the Macintosh screen, arranged in a rectangle 342 pixels high and 512 pixels wide. The pixels are stored such that all the pixels in a byte are on the same horizontal line. Bit 7 of each byte is the left-most pixel displayed. A zero bit causes a white pixel to appear on the screen, and a one bit appears as a black pixel. The screen is a 64 × 342 byte array, arranged as shown in Figure 6.2.

QuickDraw Data Structures

QuickDraw uses several specialized arrangements of data in memory. These arrangements are called *data structures.* You must understand these data structures in order to use the QuickDraw system calls.

QuickDraw expands the Macintosh screen organization to an arbitrary grid called the *coordinate plane.* This grid is a two-dimensional array of points. Each point has a vertical and a horizontal address. A point specification is written as (v,h), where v and h are the vertical and horizontal coordinates of the point. Coordinates are signed 16-bit numbers, with a range of −32768 to +32767, as shown in Figure 6.3.

A point is stored in memory as two consecutive 16-bit numbers. You can load a point into a 32-bit register with a MOVE.L instruction. The vertical coordinate resides in the high word of the register in this case.

A common data structure in QuickDraw applications is the *rectangle.* A rectangle consists of two points, the top left and the bottom right. A rectangle data structure is shown in Figure 6.4. The in-memory structure is simply four 16-bit integers. This corresponds to two-point data structures.

An important consideration in using QuickDraw rectangles is that the pixels lie between the grid lines, and the grid lines are assumed to have zero width. A rectangle thus divides the coordinate plane into two sets of points: those inside and those outside the rectangle. In addition, there are more *points* associated with the rectangle than *pixels.* In the rectangle in Figure 6.4, there are 200 (20 × 10) pixels, but there are 231 (21 × 11) points.

The QuickDraw coordinate plane is mapped onto the physical screen by a data structure known as a *bitmap.* A bitmap data structure contains a memory address, called the *base address,* which is the address of the first byte in the bitmap. The width of the bitmap is given by an integer called *rowBytes,* which contains the number of bytes in a single horizontal row

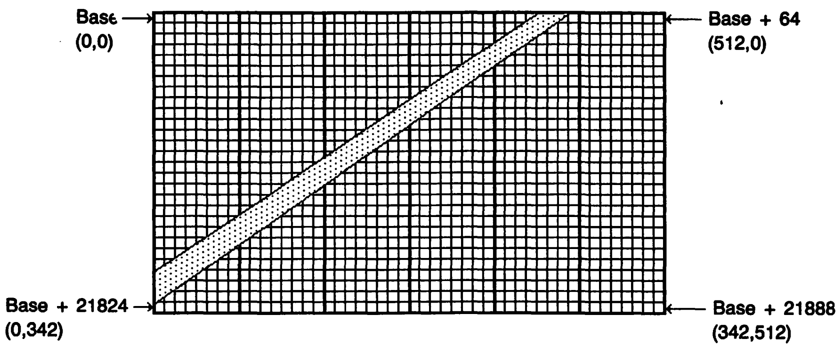


Figure 6.2 *– Macintosh screen organization*

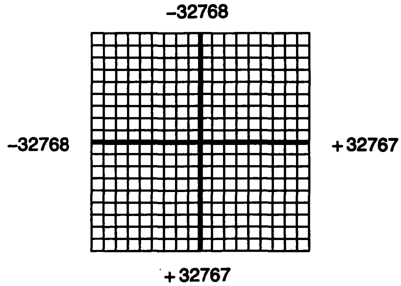


Figure 6.3 *– The QuickDraw coordinate plane*

of the bitmap. Finally, there is a rectangle, called the *bounds rectangle,* which gives the number of bits in a row and the number of rows. Figure 6.5 shows the relationship between a bitmap data structure and an actual array of pixels.

The high bit of the byte at the base address usually corresponds to the pixel at (0,0), although this need not be the case. The bounds rectangle

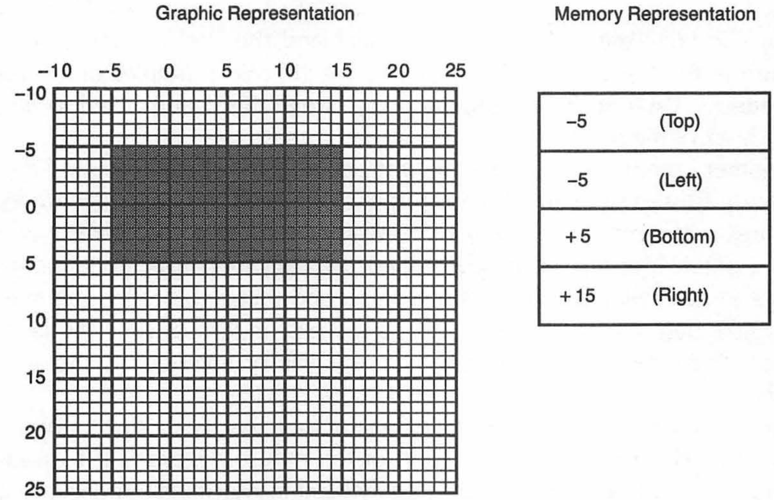


Figure 6.4 *– QuickDraw rectangle structure*

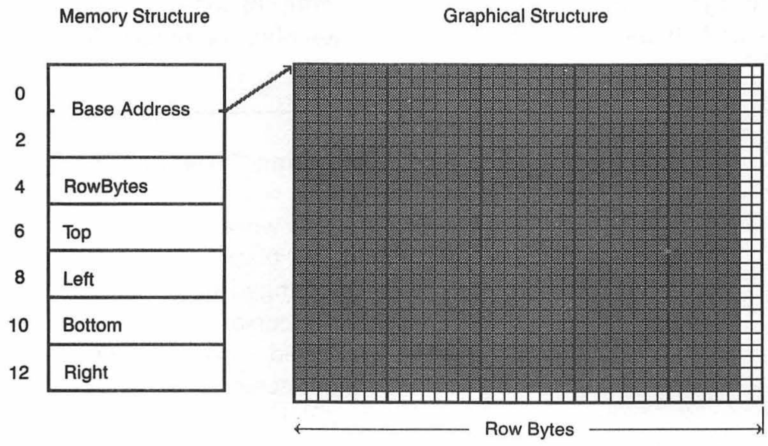


Figure 6.5 *– The QuickDraw bitmap structure*

can be used to impose any arbitrary coordinate system on the bitmap. The top-right point of the bounds rectangle always corresponds to the high bit of the byte at the base address. Another requirement is that the width of the bitmap as indicated by rowBytes be greater than or equal to the width of the bounds rectangle.

Filling an area of the screen (such as the gray background) requires a data structure known as a *pattern*. A pattern is a 64-bit image, organized as an 8 × 8-bit rectangle. A pattern therefore has a row width of one byte. The Edit Pattern option in MacPaint and the desktop background feature in the Control Panel desk accessory are two examples of the use of patterns. Each of these allows you to change the pattern used to fill a large area of the screen.

Another common QuickDraw data structure is the *cursor*. A cursor data structure is used to define the appearance of the screen image controlled by moving the mouse. The cursor consists of two 16 × 16-bit rectangles and a point. The first rectangle, called the *data rectangle*, contains the image to be drawn on the screen. The second rectangle, called the *mask rectangle*, determines what happens to the bits underneath the displayed cursor. These bits can be ignored, painted over, or inverted, as shown in Table 6.1.

The point contained in a cursor data structure specifies a single point in the cursor that indicates the location of the mouse. Figure 6.6 illustrates the structure of some cursors commonly used in Macintosh applications.

A more versatile structure than the rectangle is the QuickDraw *region*. A region can describe any arbitrary area on a QuickDraw bitmap. The data structure for a region consists of a word that indicates the total region size in bytes, a rectangle that completely contains the region on the bitmap, and an area of variable length that describes the region. The format

Data Bit	Mask Bit	Resulting Pixel on the Screen
0	1	Always white
1	1	Always black
0	0	Unchanged from image under cursor
1	0	Inverted from image under cursor

Table 6.1 – *Cursor and Mask Bit Values*

of this area is not documented by Apple. A region can change size during a graphics operation on the region and, as a result, its location in memory may change. For this reason, the normal way of referring to a region is via a pointer to a pointer to the region. This doubly-indirected pointer is called a *handle.* Handles are described in the "Dynamic Memory Allocation" section in this chapter. None of the applications in this book makes extensive use of the region data structure.

The most important QuickDraw data structure is the *GrafPort.* A GrafPort imposes a coordinate system on a bitmap. The bitmap coordinate system is often called the *global* coordinate system, while the GrafPort coordinate system is termed the *local* coordinate system. More than one GrafPort may use the same bitmap structure. Each window on the screen, for instance, has a separate GrafPort.

A GrafPort contains a great deal of drawing state information. The *pen* is used for all drawing operations inside the GrafPort. It is rectangular in shape and writes a pattern onto the GrafPort. Pen width, height, and pattern are all controlled by the current application.

QuickDraw Copy Modes

The pen may affect the GrafPort in eight different ways or modes, called *transfer modes.* These modes specify the Boolean (bitwise) operation to be used in transferring data from the pen to the GrafPort. Table 6.2 lists the eight transfer modes.

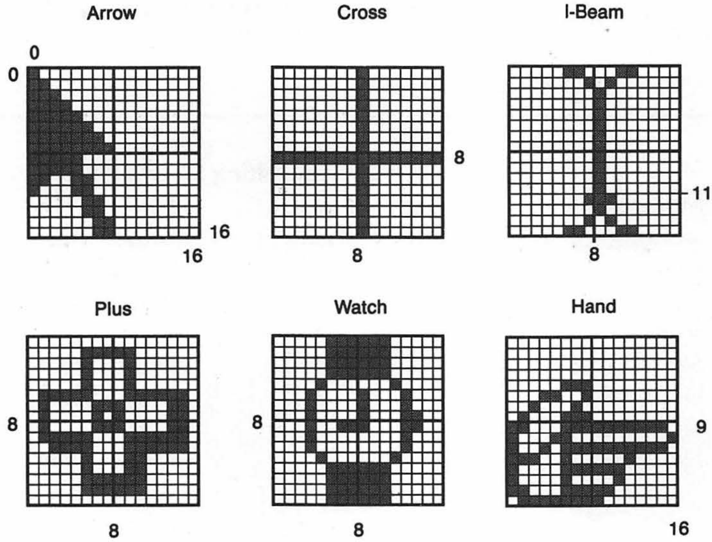


Figure 6.6 *– Common Macintosh cursors*

The Copy, Or, and Xor copy modes are analogous to the MOVE, OR, and EOR 68000 instructions, respectively, except that they work on an arbitrary number of bits. The Bic copy mode is an AND-NOT instruction (an AND with the one's complement of the source). The NOT forms of the transfer modes are formed by taking the one's complement of the source and then performing a standard Copy, Or, Xor, or Bic transfer.

The Xor transfer mode is particularly useful. If you draw an object on the screen with the Xor transfer mode and then redraw it, the screen will be left in the same state as before you drew the object the first time. This is how MacPaint draws lines, circles, and boxes with the mouse. Drawing an object with the mouse requires that the object change size as the mouse moves. MacPaint draws the object once in Xor mode. When the mouse moves, MacPaint redraws the object in the old location. This erases the object from the screen. MacPaint then redraws the object at the new location. This causes the object to change size on the screen as the mouse moves. The sample application later in this chapter also uses this technique.

Basic QuickDraw Procedures

In this section, we will explore some simple QuickDraw procedure calls. These calls all conform to the Lisa Pascal calling sequences given in the last chapter. This is by no means a complete list of QuickDraw functions, and you should consult *Inside Macintosh* for additional details.

The first two functions are used to get and set QuickDraw's idea of the current GrafPort. You can have as many GrafPorts as you have memory

	Resulting Screen Pixel	
Mode Name	**Pen Pixel Black**	**Pen Pixel White**
srcCopy	Always black	Always white
srcOr	Always black	Unchanged
srcXor	Inverted	Unchanged
srcBic	Always white	Unchanged
NotsrcCopy	Always white	Always black
NotsrcOr	Unchanged	Always black
NotsrcXor	Unchanged	Inverted
NotsrcBic	Unchanged	Always white

Table 6.2 *– QuickDraw Transfer Modes*

for, but you can draw in only one of them at a time. The following code illustrates how to obtain the current GrafPort address and restore it at a later time, using two functions called _GetPort and _SetPort.

```
         PEA       Port(A5)          ; Push address of return value
         _GetPort                    ; Issue QD call
         .
         .
         .
         MOVE.L    Port(A5),-(SP)    ; Push port address (value)
         _SetPort                    ; Restore previous port
         .
         .
         .
Port     DS.L      1                 ; Port address storage location
```

One of the most useful features of QuickDraw is the ability to specify an area of the GrafPort outside which no drawing can occur. This area is called the *clipping region,* and can be an arbitrary collection of areas inside the GrafPort. This is especially useful in dealing with multiple overlapping windows, for instance. A simplified call that sets the clipping region to a rectangle is all that is required for our sample application.

```
         PEA       CRect(A5)         ; Push address of clip rectangle
         _ClipRect                   ; Set clip region to rectangle
         .
         .
         .
CRect    DS.W 4                      ; Top-left bottom-right storage
```

The _ClipRect procedure is called ClipRect in Pascal.

QuickDraw offers a number of functions that affect the pen within a given GrafPort. Two of these functions, _GetPenState and _SetPenState, save and restore all the information associated with a QuickDraw pen. Figure 6.7 shows the format of this data.

The pen location is a point that defines the pen's position within the GrafPort. The pen size is a point that gives the width of the line drawn when the pen is moved. The pen transfer mode is one of the transfer modes described previously. The pen pattern fills the rectangle created by moving the pen. Figure 6.8 shows the pen operation.

The dark line at the top indicates the pen path, specified by the application that moves the pen. The pen location is always the top-left corner of the rectangle defined by the pen size.

The _GetPenState and _SetPenState functions are called as follows:

```
          PEA        PenState(A5)     ; Push address of pen area
          _GetPenState                ; Save pen
          .
          .
          .
          PEA        PenState(A5)     ; Push address of pen area
          _SetPenState                ; Restore pen
          .
          .
          .
PenState  DS.W       9                ; Pen state storage area
```

Use these functions when you need to change the pen temporarily and then restore it to what it was previously.

QuickDraw also offers functions that get and set individual pen fields. They are listed in Table 6.3.

These functions are called as follows:

```
          PEA        PenLoc(A5)        ; Push return area address
          _GetPen                      ; Returns a point
          MOVE.L     PenLoc(A5),-(SP)  ; Push new pen location
          _MoveTo                      ; Moves pen with no drawing
          MOVE.L     PenSize(A5),-(SP); Push new pen size
          _PenSize                     ; Set pen to new size
          MOVE.W     XMode(A5),-(SP)   ; Push transfer mode
          _PenMode                     ; Set transfer mode
```

0	Pen location (v)
2	Pen location (h)
4	Pen size (v)
6	Pen size (h)
8	Transfer mode
10	Pen pattern (8 bytes)

Figure 6.7 *– The pen state data structure*

```
          PEA      Pattrn(A5),-(SP)  ;  Push pattern address
          _PenPat                    ;  Set new pen pattern
          .
          .
          .
PenLoc    DS.W     1                 ;  Vertical coordinate
          DS.W     1                 ;  Horizontal coordinate
PenSize   DS.W     1                 ;  Pen height
          DS.W     1                 ;  Pen width
XMode     DS.W     1                 ;  Pen transfer mode
Pattern   DS.B     8                 ;  Pen pattern
```

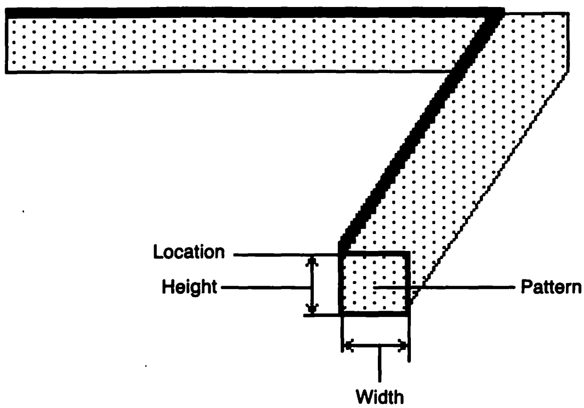


Figure 6.8 – *The graphics pen operation*

Function Name	**Purpose**
_GetPen	Returns the pen location
_MoveTo	Sets the pen location
_PenSize	Sets the pen size
_PenMode	Sets the pen transfer mode
_PenPat	Sets the pen pattern

Table 6.3 – *QuickDraw Pen Functions*

Note the use of the MOVE.L instruction to move a point. The normal Pascal calling conventions for PenSize and MoveTo require separate 16-bit parameters for the two components of a point. In assembly language, you can save space and time using a single MOVE.L instead.

QuickDraw Drawing Procedures

QuickDraw can draw a number of different objects. In this section, we will explore the three objects used in the MacDoodle program: lines, rectangles, and ovals. This is an extremely tiny subset of the functions offered by QuickDraw, but it is adequate for our purposes.

You draw lines with the _LineTo call. This draws a line from the current pen position to a point you supply. Call it like this:

```
        MOVE.L   Point(A5),-(SP)  ;  Push end point
        _LineTo                   ;  Draw the line
        .
        .
        .
Point:  DS.W     2                ;  Desired end point
```

To draw a rectangle, use the _FrameRect call. This draws a box on the screen. The area of the screen inside the box is not affected. Thus, items behind the rectangle you draw will show through. Other functions can be used to erase the inside of the rectangle, invert all the bits in the rectangle, or fill the inside of the rectangle with a pattern.

For each function that works on a rectangle, there is an analogous function that works on an oval. Thus, a _FrameOval call draws an oval on the screen. Table 6.4 gives the QuickDraw rectangle and oval function calls. These functions are called as follows:

```
PEA        Rect(A5)   ;  Push rectangle address
_FrameRect            ;  Draw a rectangle
PEA        Rect(A5)   ;  Push rectangle address
_FrameOval            ;  Draw an oval
PEA        Rect(A5)   ;  Push rectangle address
_PaintRect            ;  Fill with pen pattern
PEA        Rect(A5)   ;  Push rectangle address
_PaintOval            ;  Fill with pen pattern
PEA        Rect(A5)   ;  Push rectangle address
_EraseRect            ;  Erase (background fill)
PEA        Rect(A5)   ;  Push rectangle address
_EraseOval            ;  Erase (background fill)
PEA        Rect(A5)   ;  Push rectangle address
_InvertRect           ;  Invert rectangle
```

```
            PEA         Rect(A5)   ; Push rectangle address
            _InvertOval            ; Invert oval
      ;
      ;     _FillRect and _FillOval have an extra parameter
      ;
            PEA         Rect(A5)   ; Push rectangle address
            PEA         Patt(A5)   ; Push pattern address
            _FillRect              ; Fill the rectangle
            PEA         Rect(A5)   ; Push rectangle address
            PEA         Patt(A5)   ; Push pattern address
            _FillOval              ; Fill the oval
            .
            .
            .
Rect        DS.W        4          ; Top-left bottom-right
Patt        DS.B        8          ; Eight-byte pattern
```

_PaintRect uses the pen's current transfer mode to do the filling. _FillRect and _EraseRect always use copy mode.

QuickDraw Text Procedures

QuickDraw also handles text. Together with a related part of the system called the *Font Manager,* QuickDraw can place text of any font, size, and characteristic on the screen. The MacDoodle application draws text on the screen, using a subset of QuickDraw's text routines. We will discuss only the routines needed for MacDoodle.

Text drawing, like any other drawing, is done with the pen. Figure 6.9 shows three characters from the Chicago font. The *base line* is the vertical position of the pen when drawing text. The pen starts drawing a character at the *character origin.* The maximum height of a character is given by the

Function Name (Rectangle)	(Oval)	**Function Purpose**
_FrameRect	_FrameOval	Draws a "hollow" figure on the screen
_PaintRect	_PaintOval	Fills figure with the pen pattern
_EraseRect	_EraseOval	Fills figure with the background pattern
_FillRect	_FillOval	Fills figure with a specified pattern
_InvertRect	_InvertOval	Inverts (complements) bits inside figure

Table 6.4 *- QuickDraw Rectangle and Oval Functions*

ascent line, and the maximum depth of the character by the *descent line.* The *leading* line shows the amount of space between lines of single-spaced text. This line would be the ascent line of text on the next line. Note that the lowercase j character has one pixel to the left of its character origin. This is a technique called *kerning,* whereby a character (usually a lowercase descender) is placed closer than normal to the preceding character.

There are three types of text routines: one specifies how the text is to be drawn, one actually draws the text, and one provides information about the text. Table 6.5 gives a summary of the text routines used in this chapter.

The _GetFName and _RealFont calls are documented in the Font Manager chapter of *Inside Macintosh;* the others are in the QuickDraw chapter. _GetFName is called FontName in the Pascal documentation.

Fonts are stored as *resources,* usually in the System file on the bootstrap disk. Resources are discussed later in this chapter. You identify a particular font by a 16-bit number called a *resource ID*. All of the calls in Table 6.5 that deal with fonts use the resource ID of the font. Therefore, you don't have to know the actual structure of a font. The font routines are called as follows:

```
        MOVE.W    FontID(A5),-(SP)  ; Push font ID
        PEA       Buffer(A5)        ; Push buffer address
        _GetFName                   ; Fetch the font name

        CLR.W     -(SP)             ; Make room for return
        MOVE.W    FontID(A5),-(SP)  ; Push font ID
        MOVE.W    Size(A5),-(SP)    ; And size in points
        _RealFont                   ; See if font exists
        TST.W     (SP)+             ; Test return word
        BNE       WasReal           ; NE means yes

        MOVE.W    FontID(A5),-(SP)  ; Push font ID
        _TextFont                   ; Set font

FontID  DS.W      1                 ; Font ID storage
Size    DS.W      1                 ; Text size in points
Buffer  DS.B      256               ; Font name storage
```

The _GetFName call returns the name associated with the font ID in the buffer specified in the second parameter. This name is in Pascal string format. A zero is returned in the first byte of the buffer if the font does not exist.

If a font exists, there are certain sizes of the font that look better than other sizes. Each font and size combination is stored separately in the resource file. If the specified size does not exist for a font, it is created by scaling an existing size. The results can be coarse and unattractive. You can test to see whether a font/size combination is available by using the _RealFont call illustrated above.

The _TextFont call sets the font to be used in subsequent QuickDraw text drawing. You can use the _GetFName call to see if the font exists (in any size).

The _TextFace, _TextMode, and _TextSize calls specify the text attributes, transfer mode, and point size, respectively. Call these routines as follows:

```
        MOVE.W   Attr(A5),-(SP)   ; Push attribute word
        _TextFace                 ; Set text attributes

        MOVE.W   Size(A5),-(SP)   ; Push text size
        _TextSize                 ; Set text size

        MOVE.W   Xfer(A5),-(SP)   ; Push transfer mode
        _TextMode                 ; Set transfer mode

Attr    DS.W     1                ; Text attributes
Size    DS.W     1                ; Text size in points
Xfer    DS.W     1                ; Text transfer mode
```

The attributes word used by the _TextFace call is a set of independent bits, with one bit per attribute. The possible attributes are shown in Figure 6.10. You can specify any desired combination of text attributes by adding the values together.

Routine	Purpose
_GetFName	Gets name of a font
_RealFont	Determines whether a font exists
_TextFont	Sets text font for subsequent characters
_TextFace	Sets character style (bold, italic, etc.)
_TextMode	Sets text transfer mode (srcCopy, srcOr, etc.)
_TextSize	Sets text size
_DrawChar	Draws a single character
_DrawString	Draws a set of characters from a Pascal string
_CharWidth	Determines the width of a character

Table 6.5 *- QuickDraw Text Routines*

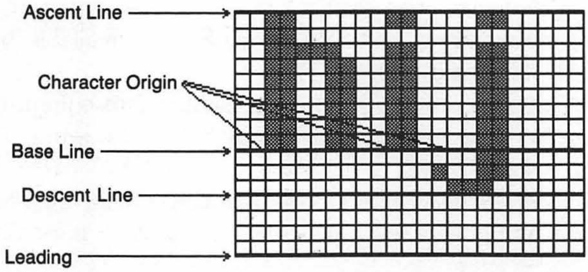


Figure 6.9 *– Sample characters in Chicago font*

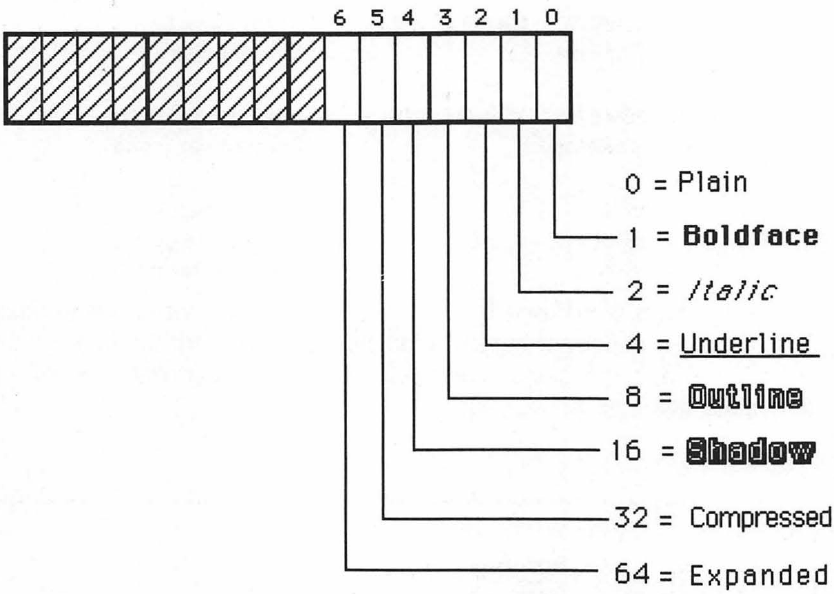


Figure 6.10 *– The available text attributes*

Text size is measured in *points*. A point is a typesetter's unit of length, roughly equivalent to 1/72 inch. Macintosh text sizes are slightly different from traditional text sizes, due to the resolution of the Macintosh screen and printer.

The text transfer modes are identical to the pen transfer modes discussed previously. The two modes work independently, however.

The _DrawChar, _DrawString, and _CharWidth calls are used to actually draw text. These routines are called as follows:

```
MOVE.W    Char(A5),-(SP)   ;  Push character
_DrawChar                  ;  Draw it
```

```
        PEA       String(A5)        ; Push string address
        _DrawString                 ; Draw string

        CLR.W     -(SP)             ; Room for return
        MOVE.W    Char(A5),-(SP)    ; Push character
        _CharWidth                  ; Calculate width
        MOVE.W    (SP)+,D0          ; D0 = width in pixels

Char    DS.W      1                 ; Single character
String  DS.B      256               ; Pascal string
```

The character parameter to _DrawChar and _CharWidth must be a word with the character in the low-order byte.

Miscellaneous QuickDraw Routines

QuickDraw offers many other useful routines for manipulating graphics objects. In our sample program, we will use five of them:

1. The *CopyBits* routine, which copies a rectangle from one bitmap structure to another
2. The *PtInRect* routine, which determines if a point is within a given rectangle
3. The *EqualRect* routine, which determines whether two rectangles are equal
4. The *PackBits* routine, which compresses bitmap data into a much smaller space
5. The *UnPackBits* routine, which expands the compressed data back into bitmap form

The CopyBits routine is probably the most complicated. This routine copies a portion of a bitmap to another bitmap. The MacDoodle program uses this function to save the contents of the screen in a memory buffer. This routine is called as follows:

```
PEA       SBitMap(A5)       ; → Source bitmap
PEA       DBitMap(A5)       ; → Destination bitmap
PEA       SrcRect(A5)       ; → Source rectangle
PEA       DstRect(A5)       ; → Destination rectangle
MOVE.W    TrnMode(A5),-(SP) ; = Transfer mode
PEA       MaskRgn(A5)       ; → Mask region
```

```
         _CopyBits                      ; Do the copy
         .
         .
         .
SBitMap  DS.W    7                      ; Source bitmap structure
DBitMap  DS.W    7                      ; Destination bitmap
SrcRect  DS.W    4                      ; Source rectangle
DstRect  DS.W    4                      ; Destination rectangle
TrnMode  DS.W    1                      ; Transfer mode
MaskRgn  DS.W    ?                      ; Mask region
```

The first parameter is the address of the bitmap data structure that contains the source data. The second parameter is the address of the destination bitmap. The third and fourth parameters are rectangles that describe the area of the source bitmap that will be copied and the area of the destination bitmap that will receive the information. These rectangles need not be of equal size. CopyBits will scale them if necessary. The fifth parameter indicates the transfer mode to be used. The final parameter is the address of a region that masks the source rectangle, permitting the transfer of arbitrary areas. If this address is zero, the entire rectangle will be transferred.

The PtInRect and EqualRect procedures are called as follows:

```
         CLR.W    -(SP)             ; Room for result
         MOVE.L   Point(A5),-(SP)   ; Push point
         PEA      Rect1(A5)         ; Push rectangle address
         _PtInRect                  ; Do the call
         TST.W    (SP)+             ; Check result
         BNE      InRect            ; NE means yes

         CLR.W    -(SP)             ; Room for result
         PEA      Rect1             ; → First rectangle
         PEA      Rect2             ; → Second rectangle
         _EqualRect                 ; Do the call
         TST.W    (SP)+             ; Check result
         BNE      Equal             ; NE means yes

Point    DS.W     2                 ; Point parameter
Rect1    DS.W     4                 ; Rectangle #1
Rect2    DS.W     4                 ; Rectangle #2
```

The rectangles and point must be in the same coordinate system for these calls to function properly.

Graphics data typically contain many bytes that have the same value. A technique called *run-length encoding* is often used to reduce the size of such data in memory or on disk. The Macintosh ROM provides two procedures for converting between bitmap and run-length encoded data. These routines are PackBits and UnPackBits and are called as follows:

```
            PEA       SrcAddr(A5)          ; →→ Source data
            PEA       DstAddr(A5)          ; →→ Destination data
            MOVE.W    ByteCnt(A5),-(SP)    ; # bytes to pack
            _PackBits                      ; Pack it

            PEA       SrcAddr(A5)          ; →→ Source data
            PEA       DstAddr(A5)          ; →→ Destination data
            MOVE.W    ByteCnt(A5),-(SP)    ; # bytes to unpack
            _UnPackBits                    ; Unpack it

SrcAddr     DS.L      1                    ; → Source data
DstAddr     DS.L      1                    ; → Destination data
ByteCnt     DS.W      1                    ; = Number of unpacked bytes
```

The source and destination addresses are updated to the first byte unused by the system call. The byte-count parameter always refers to the number of bytes in the unpacked (bitmap) data. The normal method of converting a bitmap is to pack or unpack each row individually.

Macintosh Memory Management

A second major area of concern to Macintosh application writers is that of memory management. At this writing there are four memory configurations for the Macintosh:

- The 128K Macintosh
- The 512K Macintosh
- The 512K Macintosh XL (Lisa)
- The 1024K Macintosh XL (Lisa)

Macintosh Memory Layout

Figure 6.11 shows the memory layout for a 512K Macintosh. The sizes of each of the items shown are in decimal, while the addresses are hexadecimal. A 128K Macintosh has a 16.5K system heap instead of a 46K system heap, and two of the address bits are not present. All the addresses in Figure 6.11 of the form "07xxxx" are "01xxxx" on a 128K Macintosh. The first 2816 bytes contain the 68000 interrupt vectors and system global variables. Interrupt vectors are described in Chapter 7.

The Macintosh has two dynamic memory areas: the system heap and the application heap. The system heap is a fixed-size area, but the application heap is allowed to grow toward higher addresses. The stack is placed at the upper end of the application heap and grows toward lower addresses.

The system heap contains memory that remains relatively constant. It is used for data structures that must remain in place across applications, such as information about disk drives, device drivers, and other system-related information.

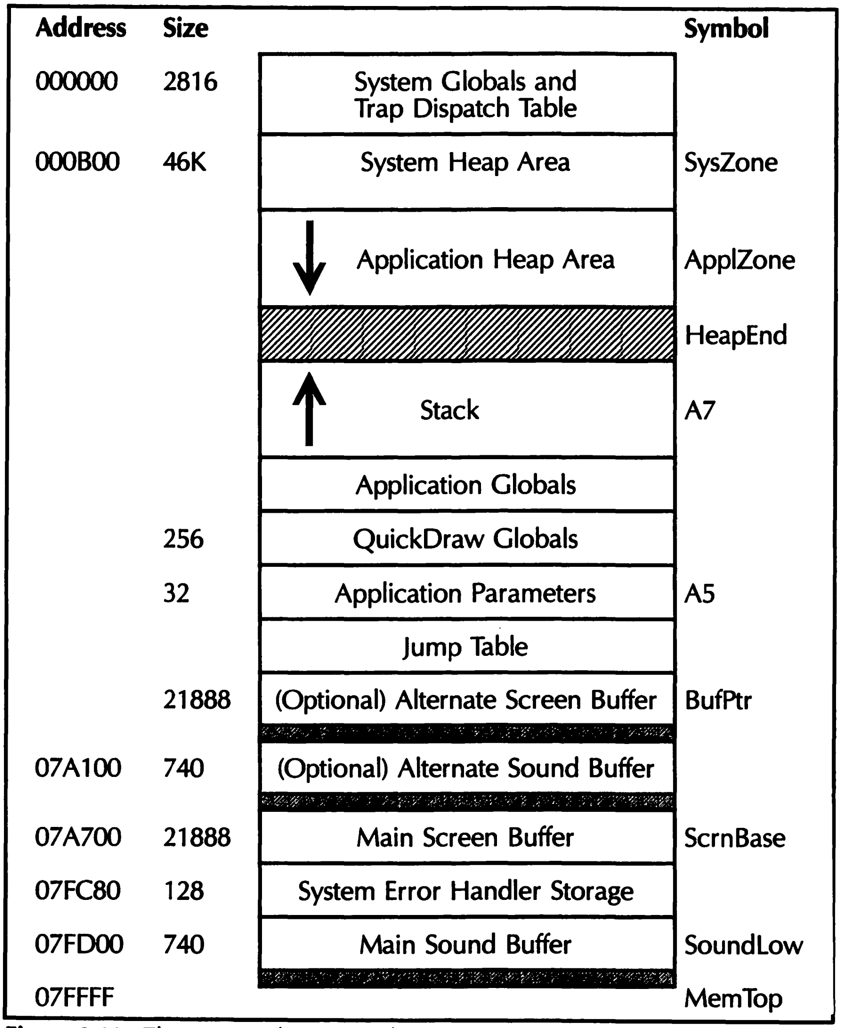


***Figure 6.11** – The Macintosh memory layout*

The application heap is reinitialized each time an application program runs. Most memory that is allocated dynamically comes from the application heap, such as the memory for the application code and data. The application can also allocate memory dynamically from this area. Many system calls allocate memory from the heap as well.

The application's uninitialized global variables are placed at the high-address end of the application heap, along with the parameter area passed from the Finder and the jump table used for overlays. Register A5 is used to address all of these areas. Negative offsets from A5 contain data locations defined by DS directives in the assembler. Locations −1(A5) to −256(A5) are used by QuickDraw and are automatically reserved by the linker. The application is required to voluntarily relinquish control of these locations, as shown in Listing 8.2 (Chapter 8).

Positive offsets from A5 contain both parameters passed to the application from the Finder and the jump table used for loading overlays. The application parameters are described later in this chapter. None of the programs in this book use the jump table.

The alternate screen buffer and alternate sound buffer are used by specialized applications. They provide a way to maintain two copies of the screen and sound output. Most applications do not require this feature, and these buffers are normally not present.

The main screen buffer contains the memory used for the Macintosh screen display. Storing data in this area directly affects the appearance of the screen. QuickDraw uses a bitmap data structure (which contains the screen address in the base address field as seen in Figure 6.5) for drawing on the screen.

The system error handler storage is used by the component that displays the "Bomb" template on the screen when an application fails and the system must be rebooted.

The main sound buffer is used to hold data waiting to be used in sound generation. This area is also used by the disk software to control speed changes.

Macintosh XL Memory Layout

Figure 6.12 shows the memory layout of a Macintosh XL (formerly known as a "Lisa running MacWorks"). The Lisa hardware supports either 512K or 1024K (1M) of memory. The memory layout is very similar to a Macintosh, except that there is no alternate screen or sound buffer. In addition, the screen is a different size (364 × 720), necessitating a 32K screen buffer. The hardware interface area contains code that simulates the Macintosh hardware and ROM on the Lisa.

The addresses shown are for a 1024K (1M) Lisa. Simply subtract 80000 hex to obtain the corresponding addresses on a 512K Lisa. Addresses of the form Fxxxxx become 7xxxxx, and addresses of the form Cxxxxx become 4xxxxx.

Dynamic Memory Allocation

Macintosh software makes extensive use of dynamically allocated memory. QuickDraw's use of variable-length data structures makes it imperative that an area of memory be able to grow when necessary. Therefore, dynamically allocated blocks of memory must be moved to accommodate growth.

The Apple solution to this rather knotty problem is to use one pointer for each dynamically allocated memory area. This pointer is called a *master pointer*. Code that refers to the dynamic region uses a pointer, called a

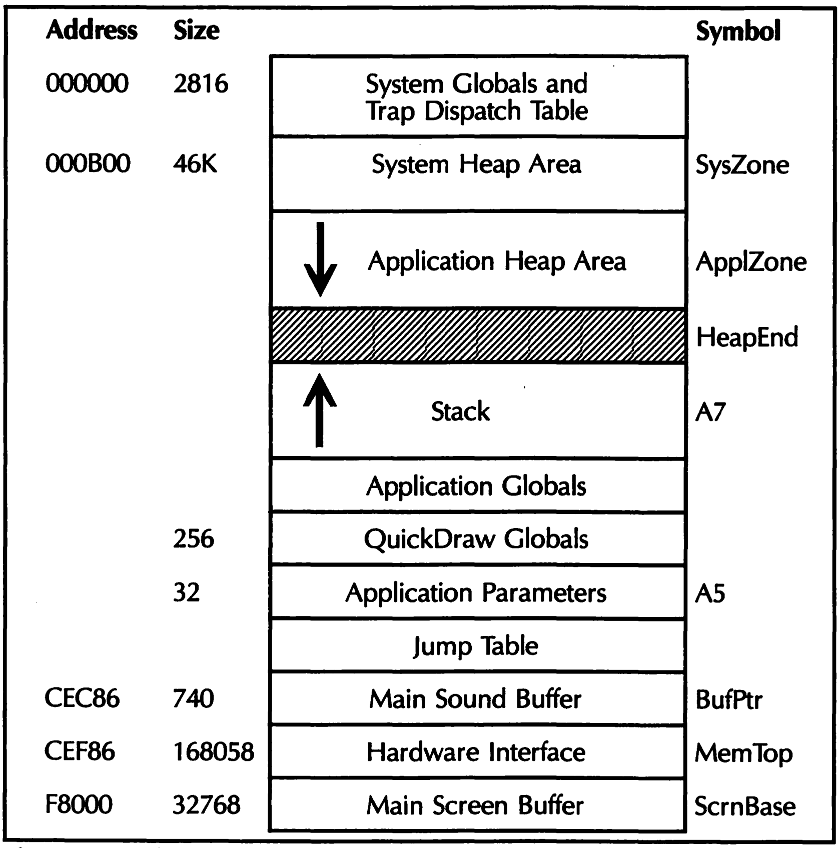


***Figure 6.12** – The Macintosh XL (Lisa) memory layout*

handle, which contains the address of the master pointer. The relationship between master pointers and handles is shown in Figure 6.13. Each separately allocated memory area has a master pointer.

Applications code that references a dynamic area uses a technique called *double indirection,* making it necessary to use two MOVE.L instructions where, ordinarily, only one would suffice:

```
          MOVE.L    Handle(A5),A0    ; Fetch the handle contents
          MOVE.L    (A0),A0          ; Convert to address handle
Handle    DS.L      1                ; Handle storage
```

The second MOVE.L would not be required if the area were not dynamically allocated.

There is a big problem with this technique, particularly for assembly language applications. Because the dynamic memory scheme is used by most of the operating system, practically any system call you make can move your dynamically allocated memory. If you make a system call with the address of a dynamic area in an address register, forget to reload the address register from the handle, and subsequently use the address register, it might no longer point to the dynamic area. It might instead point to some other unrelated dynamic area. Modifying the dynamic area can cause unpredictable program behavior.

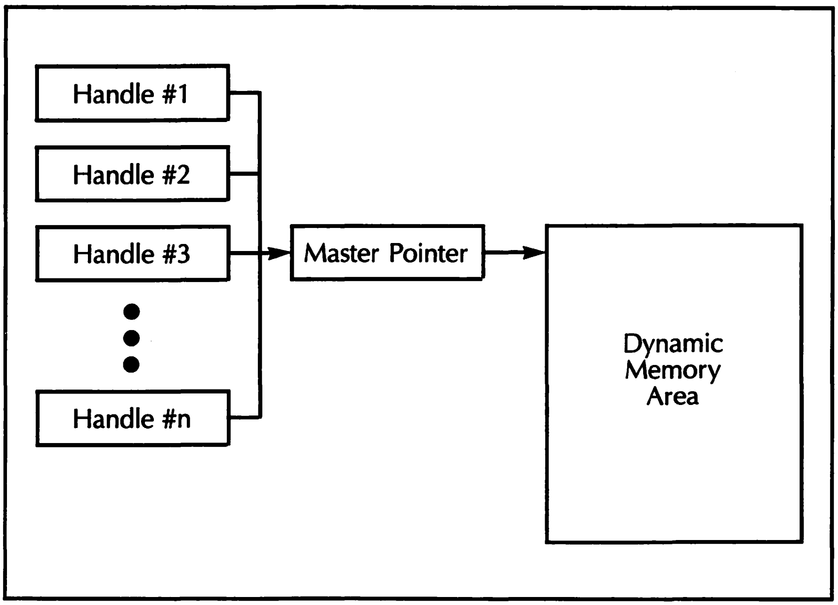


Figure 6.13 – *Dynamic memory pointers*

This scenario is called the *dangling pointer problem*. It is particularly bad in assembly language because you must code extra instructions to dereference the handle. It is very easy to forget. Worse, a program with this problem may behave normally for a long time. The problem occurs only when a lot of memory is used. Thus, your program may work fine until you try to run it with a desk accessory or with some other system feature that uses more memory, such as a laser printer, network connection, or hard disk.

You can prevent the system from moving a block of dynamic memory by *locking* the memory, using the _HLock system call described below. You can then freely use the address of the dynamically allocated area across system calls. However, this is not a general solution to the dangling pointer problem. In programs that use dynamic memory extensively, locking can cause the system to run out of memory due to a condition known as *fragmentation*. Locked blocks limit the flexibility the system usually has in allocating memory. There might be enough memory to satisfy a request for more memory, but, due to the presence of locked areas, the system might not be able to get a single piece large enough.

Dynamic Memory System Calls

Most dynamic memory allocation can be done using only four system calls:

1. The _NewHandle function, which allocates a new block
2. The _HLock function, which prevents the system from moving a block
3. The _HUnLock function, which allows the system to move a block
4. The _DisposHandle function, which frees a block

These routines are called as follows:

```
MOVE.L    Size(A5),D0        ; D0 = # bytes needed
_NewHandle                   ; Try to allocate it
TST.W     D0                 ; Success?
BMI       Error              ; No, couldn't get it
MOVE.L    A0,Handle(A5)      ; Save the handle

MOVE.L    Handle(A5),A0      ; Handle to A0
_HLock                       ; Lock it
```

```
                MOVE.L   Handle(A5),A0    ; Handle to A0
                _HUnLock                  ; Unlock it

                MOVE.L   Handle(A5),A0    ; Handle to A0
                _DisposHandle             ; Free dynamic area

        Handle  DS.L     1                ; Handle storage
```

All the routines return a completion code in D0. A TST.W D0/BMI sequence tests for errors on any of the above calls. The _NewHandle call requires a size in D0.L, and returns the handle in A0. A0 will be set to zero if the memory is not available.

Resources

A concept related to Macintosh memory management is the *resource* feature. Most applications use large numbers of small data structures that contain only constant values. The Macintosh resource feature provides the application with a means of fetching a small (or large) data structure from disk on demand. Use of this feature accomplishes two ends:

1. You can write programs such that all user-visible displays can be modified without touching the program source code. This is very useful in writing programs that must be used in many different countries. You can translate all of the program's messages to a different language quite easily.
2. You can save memory since only the items that are needed are loaded from disk into memory.

Macintosh resources are kept in the resource fork of a disk file. Resources can be associated with a document file, an application file, or the system file.

Resource Types

Six bytes are required to identify a resource. Four of these bytes, called the *resource type,* specify the general class of the resource. Resource types are generally four ASCII letters. For instance, a menu template has a resource type of MENU (four ASCII characters). The remaining two bytes make up a binary number that identifies a specific instance of the resource class. A resource can also have a name, but this feature is not normally used.

Table 6.6 contains some of the more common resource types. This is not a complete list. However, all programs in this book require only this short list of resource IDs. The STR resource ID and all other three-character resource IDs are padded with a trailing space to make four characters.

You can also define resources other than the Macintosh predefined resource types. Any four-byte value can be used. This is generally not required for normal applications.

RMAKER

You can create a resource file for an application using the RMAKER program in the Macintosh Development System. With the run-time library described in Chapter 8, you can either place the resources in the same file as your program or use a separate resource file. It is better to use a separate resource file while developing your program, as this reduces the time required to make a change. When the program is finished, you can then combine the program and resource files so that only one file must be copied to move the application to another disk.

RMAKER uses a textfile to define the resources to be created in the output file. The first nonblank, noncomment line of the file is the name of the output file. Placing an exclamation point in front of the file name causes

Resource Type	**Use**
ALRT	Alert template
BNDL	"Bundle" resource
CNTL	Control (e.g., scroll bar) definition
CODE	Application program code segment
CURS	Mouse cursor definition
DITL	Dialog item list
DLOG	Dialog template
DRVR	Device driver or desk accessory code
FREF	File reference
ICN#	List of icons
ICON	Single icon
MENU	Menu definition
STR	String
STR#	List of strings
WIND	Window template

Table 6.6 – *Common Macintosh Resource Types*

RMAKER to append resources to an existing file. Thus, for a separate resource file, the first line of the file might be

```
program.RSRC
```

To add the resources to the end of an existing program, use

```
!program
```

"Program" is the name of the file produced by the linker. The run-time library appends the string .RSRC to the name specified in the PROGRAM macro for the application-specific resource file. The file RTL.RSRC is used if the custom resource file does not exist.

For instance, if you specify the PROGRAM macro as

```
PROGRAM    Blivot
```

the run-time library will try to open Blivot.RSRC as the resource file. If Blivot.RSRC does not exist, the run-time library will open RTL.RSRC instead.

The remaining lines in the file describe resources to be placed in the output file. Perhaps the simplest way to do this is to include an existing resource file in the textfile. The RMAKER INCLUDE statement performs this function. The syntax of this statement is

```
INCLUDE    filename
```

The RMAKER TYPE statement defines a new resource for inclusion in the output file. A TYPE statement has the following form:

```
    TYPE rtyp
[name1],ID1
(Definition data)
[name2],ID2
(Definition data)
     .
     .
     .
```

"Rtyp" is the four-character resource type from Table 6.10. A single TYPE statement can be used to define multiple instances of the same resource type. Each resource definition consists of an optional name and a resource ID (a decimal number). One or more type-specific lines follow. For example, a string resource (type STR) requires only the string as definition data. You can define string resources as follows:

```
    TYPE STR
String1,1
This is a named string
    ,2
This is an unnamed string
    ,3
Third string
LastString,34
Bench Warmer
```

This example defines four strings, with resource IDs 1–3 and 34. The first and last strings have names.

Other resource types are explained in the description of the sample application later in this chapter.

Resource IDs

Resource IDs are 16-bit integers. Different types of resources can have the same ID. A second definition of a resource ID that has already been defined will render the first definition inaccessible. Resource files are searched in reverse order from which they were opened. Thus the current document is searched first, followed by the application file, followed by the system file. This allows you to override application resources in a document file, or to override system resources with either the application file or a document file.

Certain conventions apply to resource ID values. All negative values are used for special purposes by the system. Values 0–127 are also used by the system. Only values 128–32767 are guaranteed to be available for whatever purpose you wish.

Resource Attributes

Each resource is associated with a word, called the *resource attribute,* which contains additional resource characteristics. Figure 6.14 illustrates the format of this word.

Bit 1 of this word indicates whether an in-memory resource has been altered and needs to be rewritten to the resource file. Bit 2 of this word indicates whether the resource is to be loaded into memory when the resource file is first opened. Preloading resources, especially those in an application file, can appreciably improve application performance. Bit 3 is set if the resource is protected. Protected resources cannot be modified on disk except to change the attributes. Bit 4 is set to prevent the resource from being purged or moved after loading. Bit 5 is set to allow the resource to be purged during a memory allocation operation. Bit 6 is set to cause the resource to be loaded into the system heap.

You can specify resource attributes in the RMAKER input file. Simply enclose the decimal value in parentheses following the resource ID. For example, to preload a string resource, you would specify the following:

```
    TYPE STR
    ,1(4)
This string is preloaded
```

Windows

Another important Macintosh feature is the window concept. Information presented to the computer user usually appears in a window on the screen. There are three types of windows used in the programs in this book:

1. Document windows
2. Alerts
3. Dialogs

Document windows represent data selected or input by the user, while alerts and dialogs are used for temporary displays (such as error messages).

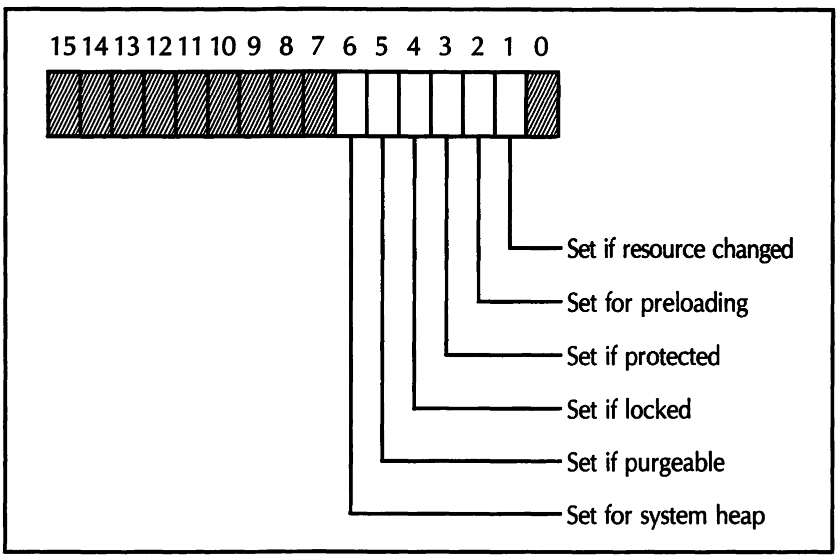


Figure 6.14 *– The resource attribute word*

Document Windows

The windows used by most applications are called *document windows.* Figure 6.15 shows the structure of a standard Macintosh document window. Most document windows have a title bar, close box, and three areas called *controls:* the *vertical scroll bar,* the *horizontal scroll bar,* and the *grow box.* The rest of the window is called the *content region.*

The content region is the area of the window where information is displayed. Each window contains a QuickDraw GrafPort, whose origin is the upper-left corner of the content region. Drawing in the window's GrafPort causes information to appear on the screen in the content region.

QuickDraw includes the controls in the content region. Normal drawing does not affect the controls because of the QuickDraw _ClipRect call. The run-time library uses this call to prevent the application from drawing in the control area of the window.

The horizontal and vertical scroll bars allow the user to change the area of the document displayed in the window. When the user clicks the mouse in the arrow region of one of the scroll bars, the visible part of the document moves in the indicated direction by a small amount. For text windows, this is usually one line of text. When the user clicks the mouse between the arrow and the thumb, the screen scrolls by a "page" (the size of the window).

The user can also elect to move the thumb along the scroll bar. The screen then displays the area of the document that is proportional to the thumb's new position in the scroll bar. For example, moving the thumb to the top of the vertical scroll bar displays the beginning of the document. Moving the thumb to the end of the vertical scroll bar displays the end of the document. Moving the thumb to the midpoint of the scroll bar displays the midpoint of the document, and so on.

The run-time library provides several macros that display the correct

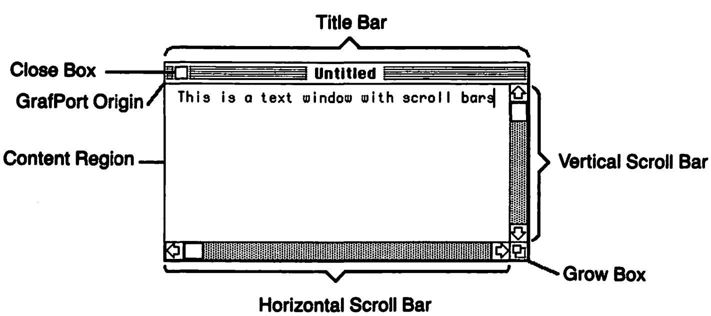


Figure 6.15 *– A Macintosh document window*

position of the thumb following a scrolling operation:

SETVMIN	Sets the vertical low value
SETVMAX	Sets the vertical maximum value
SETVVAL	Sets the vertical current value
SETHMIN	Sets the horizontal low value
SETHMAX	Sets the horizontal maximum value
SETHVAL	Sets the horizontal current value

Each of these macros takes a single 16-bit parameter, which is the value to be set. Normally, you should use the SETVMIN, SETVMAX, SETHMIN, and SETHMAX macros at initialization time. This sets the values for the endpoints of the scroll bar. When a scrolling event occurs, call SETVVAL or SETHVAL to position the thumb correctly within these two endpoints.

The window *grow box* allows the user to change the size of the active window.

Alerts

Error conditions in a Macintosh application usually result in the execution of a run-time library ALERT macro, which creates a special window called an *alert box* to appear on the screen. The alert box may have one or more buttons or other input mechanisms. The program continues when the user clicks the mouse on one of the buttons.

There are four types of alerts, as shown in Figure 6.16. The only difference is the icon in the upper-left corner of the alert box. These four alerts are generated by the ALERT, CAUTIONALERT, NOTEALERT, and STOPALERT macros. These macros take the resource ID of an alert template as the only parameter. The alert template is a resource that defines the appearance of the alert box. Following the macro, D0.W contains the number of the button selected by the user. A CAUTIONALERT (with the question mark) almost always has more than one button, since the user must make some kind of a selection between alternatives.

Associated with each alert template is a 16-bit number called the *stages word*. This word is a series of four-bit fields that define actions on the first, second, third, and subsequent occurrences of the alert. These are the actions the stages word defines:

1. Whether or not the alert box is drawn
2. What kind of sound is emitted
3. Which button is the default (if the user hits Return)

The format of the stages word is shown in Figure 6.17.

The low-order nibble of the stages word defines the action on the first occurrence of the alert. Subsequent nibbles define actions on subsequent occurrences of the alert. The high-order nibble defines the action on occurrence number 4 and later.

For example, if you wanted the default button to be the first button defined, the alert box to be always drawn, and the sound to increment from 0 beeps to 3 beeps, the alert stages word would be 7654 hex.

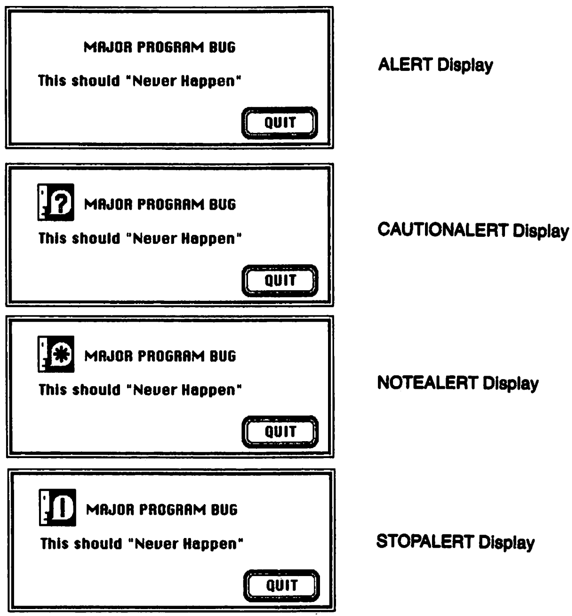


Figure 6.16 *– Macintosh alerts*

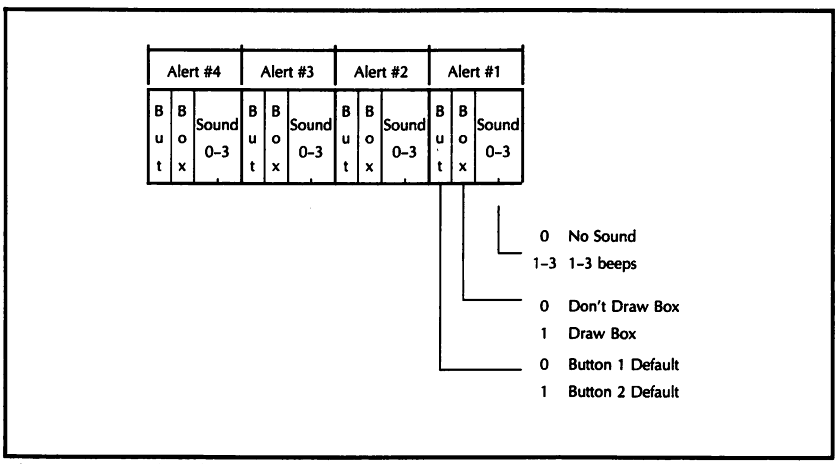


Figure 6.17 *– The alert stages word*

Dialogs

Another mechanism employed when the user must make a choice is a *dialog.* A dialog is similar to an alert, except that no icon is automatically drawn and no stages word defines the action. The DIALOG macro has a single parameter, the resource ID of a dialog template. The dialog template defines a special window, called the *dialog box.* The DIALOG macro causes the dialog box to be displayed on top of existing windows, and it returns the item number selected by the user in D0.W.

Menus

Another method of interacting with the user is the Macintosh *menu* facility. This feature allows the user to select from a number of options regarding a particular topic. For instance, the File menu normally has options for creating a new file, opening an existing file, saving changes to a file, and printing a file. This menu usually contains the Quit choice, to allow the user to exit back to the Finder.

The normal technique for using menus in a program is to define the menus in the program's resource file. A menu selection is indicated by the menu number and the selection number within that menu. Menus are numbered by their order in the menu bar, starting with menu number 1. Selections within a menu are numbered from top to bottom, with selection number 1 at the top.

Figure 6.18 shows a menu selection in progress. The Apple menu is menu number 1. The selection being made is the last selection in the menu, selection number 9. The gray line between About MacDoodle and Scrapbook counts as a choice, although it cannot be selected.

The run-time library provides for three operations on menus:

1. Enabling and disabling a menu selection
2. Checking and unchecking a menu selection
3. Setting the type style of a menu selection

Figure 6.19 shows the effects of these functions.

Enabling and Disabling a Menu Selection

The ENABLE and DISABLE macros allow and disallow selection of an item within a menu. Items that have been disabled are dimmed, as shown in Figure 6.19. These macros have the following calling sequence:

```
ENABLE      menu,item
DISABLE     menu,item
```

"Menu" is the menu number, and "item" is the item number. Both are 16-bit quantities.

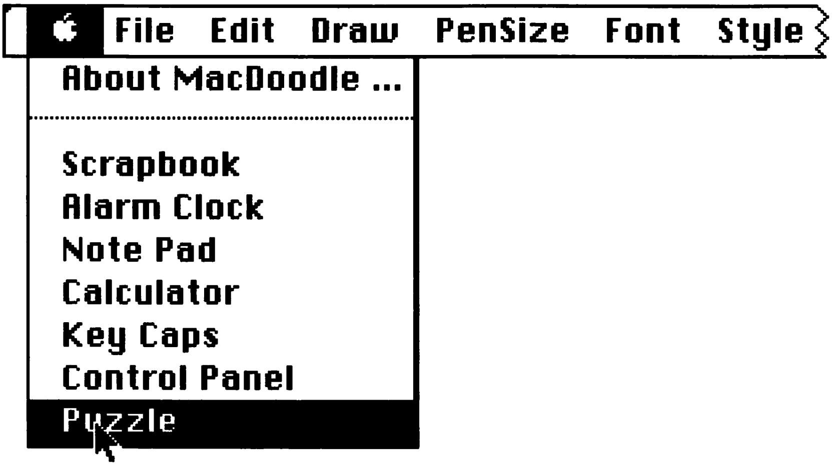


***Figure 6.18** – Menu selection*

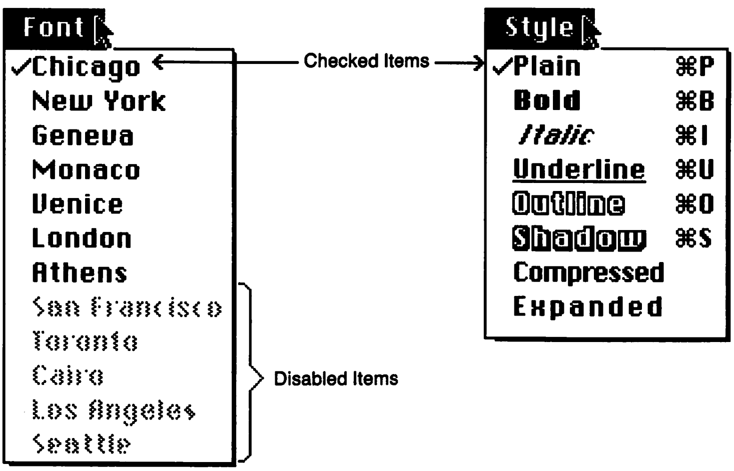


***Figure 6.19** – Menu item attributes*

Checking and Unchecking a Menu Selection

The CHECK and UNCHECK macros cause the menu item to be preceded by a check mark or a space, respectively. This visually confirms the menu item selection. When selecting a menu item causes subsequent actions to be interpreted differently, you should use the CHECK feature.

These macros have the following calling sequence:

```
CHECK      menu,item
UNCHECK    menu,item
```

"Menu" and "item" are the menu number and item number, respectively. Both are 16-bit quantities.

Setting Menu Item Style

The STYLE macro sets the text style associated with a menu item. This macro has the following calling sequence:

```
STYLE      menu,item,style
```

"Menu" and "item" are the menu and item numbers. "Style" is a 16-bit quantity described in Figure 6.10.

Events

Typical Macintosh applications are *event-driven.* These programs wait for an external event, process it, and wait for the next event. The Macintosh Toolbox defines 16 basic event types. The run-time library handles a few of the Macintosh events directly and partially processes others. This allows the application code to be small because a lot of event-handling code is in the run-time library.

The GETEVENT Macro

The GETEVENT macro is the center of the event code in an application that uses the run-time library. This macro has one optional parameter, the resource ID of whatever cursor is to be used when the mouse is inside the application window. If the parameter is omitted, the I-beam (text) cursor is used. The system provides five standard cursors, which you can use without any definition:

Code	Resulting Cursor
0	Arrow
1	I-beam (text)

Code	Resulting Cursor
2	Cross
3	Plus
4	Watch

These cursors are shown in Figure 6.6. In addition, you can define a CURS resource in the application resource file for use with the GETEVENT macro. This will be described in detail in the sample application.

The macro returns two registers: D0.L and D1.L. The low word of D0 is a code indicating the type of event. D1 and the high word of D0 are additional parameters for some of the event types. Table 6.7 shows the return values from the GETEVENT macro.

Normal applications ignore null (0), mouse-up (2), key-up (4), disk-inserted (7), abort (9), network (10), driver (11), and the application (12–15) events. Events with codes greater than 15 are run-time library extensions to the Macintosh events.

Mouse Down

A mouse-down event is generated when the mouse button is pressed and the mouse is inside the content region of the application window. All other uses of the mouse, such as menu selection, window dragging, scrolling, and window sizing are at least partially handled by the run-time library. All of the extra events defined by the library are partially processed mouse-down events.

A mouse-down event reports the mouse position as a point in D1.L. The vertical coordinate is in the high word, and the horizontal coordinate is in the low word. The point is in the local coordinate system of the application window, so you can use it directly in QuickDraw calls.

Key Down and Repeated Key Down

A key-down event returns the character code in the low byte of D1. The high-order word of D0 contains a word that gives the status of the modifier keys on the keyboard. The bits of this word are arranged as follows:

Bit	Meaning If Set
0–7	(Not used)
8	Command key down
9	Shift key down
10	Caps Lock key down
11	Option key down

When the user holds a key down, the key begins to repeat after a certain amount of time. Settings on the Control Panel desk accessory control the time before repeating begins and the rate of repetition. Repeated keys are identical to regular keystrokes, except that they have different event codes.

Window Update

A window-update event indicates that some portion of the window needs to be redrawn. The run-time library takes care of everything except the window content area. The normal way of handling this type of event is to redraw the entire window. Register D1.L contains the address of the window to be updated.

Update events are generated any time window overlapping changes. Dialog and alert boxes generate an update event once the user selects an

Code D0 Low	D0 High	D1 High	D1 Low	Event
0	–	–	–	No event is ready
1	–	Mouse V	Mouse H	Mouse down (D1 = point)
2	–	–	–	Mouse up
3	Modifier	–	Character	Key down
4	–	–	–	Key up
5	Modifier	–	Character	(Repeated) Key down
6	–	Window Address		Update-window event
7	–	–	Drive #	Disk-inserted event
8	–	Window Address		Activate event
9	–	–	–	Abort event
10	–	–	–	Network event
11	–	–	–	Driver event
12–15	–	–	–	Application event #1–#4
16*	–	Menu #	Item #	Menu selection
17*	–	–	–	Close-window event
18*	–	Height	Width	Resize window event
19*	SType	Control ID		Plain scroll event
20*	Value	Control ID		Vertical thumb scroll
21*	Value	Control ID		Horizontal thumb scroll

* Run-time library extension

Table 6.7 – *The GETEVENT macro return values*

item and the box disappears. A desk accessory can also cause an update event. A menu selection does not generate an update event, however.

Window Activate

A mouse click inside an inactive window results in an activate event for the inactive window and a deactivate event for the active window. A 1 in bit 16 of D0 activates a window and a 0 deactivates a window. D1.L contains the address of the affected window. The run-time library handles the normal actions (such as highlighting the title bar) for activate events, so most applications can ignore them. When a window is activated, it need not be redrawn. An update event will exist to do this (if necessary).

Menu Selection

Menu selection causes an event to be generated by the run-time library. The menu number is in the high word of D1, and the selection number in the low word. Menu selection using both the mouse and keyboard equivalents results in a menu-selection event.

A menu-selection event is a run-time library extension to the Macintosh event system. The library processes mouse and keyboard events, filtering out and reporting menu selections separately. This technique is used for all events with codes 16 or greater.

Close Window

Another extension to the Macintosh event system is the close-window event. This event results when the user clicks the mouse in the close box of the application window. The application usually reacts to this kind of event by simply closing the file displayed in the active window. Some Macintosh applications, such as MacPaint and MacWrite, close the window and also require the user to perform an Open or New operation before any further data is displayed.

Window Sizes

The run-time library automatically handles user resizing of the application window. When the user clicks the mouse in the window's grow box, run-time code performs the operations necessary to change the window size. The code then generates a resize event. Register D1 contains the new size of the window's content region. Your application should redraw the content region based on the new window size.

Scrolling

If the run-time library is configured to operate with window size and scrolling controls, it can generate three events associated with the scroll bars:

1. Line/page scroll (19)
2. Vertical thumb scroll (20)
3. Horizontal thumb scroll (21)

These three events are extensions to the standard Macintosh event set. The line/page-scroll event covers all scrolling done using the arrows and page scroll regions of both scroll bars. The two thumb-scroll events cover the movement of the thumb in a single scroll bar.

Line/Page Scroll Figure 6.20 shows the layout of a standard scroll bar. (The vertical scroll bar is exactly equivalent.)

Clicking the mouse in the line or page scroll region is reported through a line/page-scroll event. D1.L contains the control handle for the scroll bar. The high word of D0 contains a code that indicates the type of scrolling selected by the user:

Scroll Code	Meaning
0	Line scroll up
1	Line scroll down
2	Page scroll up
3	Page scroll down

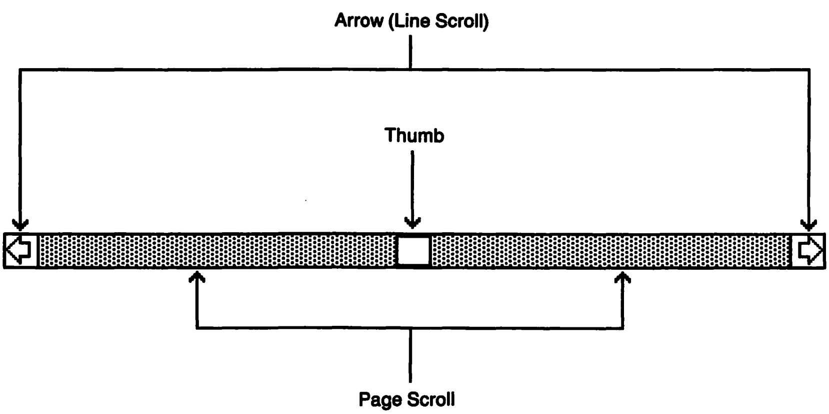


Figure 6.20 – *Scroll bar regions*

Scroll Code	Meaning
4	Line scroll left
5	Line scroll right
6	Page scroll left
7	Page scroll right

The application is responsible for scrolling the document in the window by the proper amount in the proper direction. Also, the application must set the thumb value using the SETVVAL or SETHVAL macros.

Macintosh applications typically repeat the above scrolling actions until the mouse button is released. Using the run-time library, you can accomplish this easily with the TRACK macro. This is the calling sequence:

```
TRACK     controlid,scrollproc
```

"Controlid" is the control handle from the event (as returned by events 19–20), and "scrollproc" is the address of a procedure that performs a single instance of the desired scrolling. This procedure is called repetitively as long as the mouse button remains down.

A Pascal calling sequence (such as described in Chapter 5) is used to call the scrolling procedure. Figure 6.21 shows the structure of the stack on entry to the scrolling procedure.

Your scrolling procedure must pop the parameters off the stack before returning to the caller. The scroll bar handle is a copy of the contents of D1 when the event was reported by the GETEVENT macro. The "part code" indicates which part of the scroll bar is affected. If this quantity is zero, the mouse button is still down, but the mouse has been moved outside of the original scroll bar region. Your scroll procedure should simply return if this is the case. Nonzero part codes include the following:

Part Code	Scroll Bar Area
20	Up (left) arrow
21	Down (right) arrow
22	Page up (left) region
23	Page down (right) region

Return Address	0(SP)
Part Code	4(SP)
Scroll Bar Handle	6(SP)

Figure 6.21 – *Scrolling procedure parameters*

If you store the original code from the scrolling event, you will not need the parameters to the scrolling procedure. Listing 6.28 (later in the chapter) illustrates the use of a scrolling procedure.

Thumb Scrolling A thumb-scroll event indicates that the user has manipulated the thumb in a scroll bar. The event code in D0.W indicates which scroll bar has been changed. The high word of D0 contains the final value of the thumb position. You should set the minimum and maximum values for the scroll bar during initialization and after resizing the window. The value reported in D0 will be in the range defined by the minimum and maximum values. You need only redisplay the appropriate part of the document in the window to respond to a thumb-scroll event. A judicious choice of minimum and maximum values will facilitate finding the right portion of the document to display.

MACINTOSH PROGRAMMING RESTRICTIONS

The structure of the Macintosh systems software imposes some limitations on your flexibility in writing 68000 assembly language code. These are the most serious constraints:

- The requirement for position-independent code
- Size limitations on code and data segments
- Heap compaction requirements

Position-Independent Code

Many 68000 systems perform an operation known as *load-time relocation* when loading a program into memory. The linker on those systems prepares a section of the program file called a *relocation dictionary.* The information contained in the relocation dictionary identifies each memory address within the program. The linker assumes an address called the link address for the first location in the program. If the load address is different from the link address, the loader uses the relocation dictionary to adjust all the addresses in the program.

The Macintosh does not do this. Macintosh programs must not contain absolute addresses. This means that you cannot use absolute addressing for labels within your program or data. You can use absolute addresses for locations outside your program, such as the low address area used for system globals. Code that meets these requirements is called *position independent,* and can be loaded anywhere in memory with no relocation.

Program references must, therefore, use either the PC-relative or address-register-relative addressing modes in instructions. The assembler automatically generates the PC-relative mode for instructions that reference labels which are not on a DS directive. Unfortunately, the PC-relative addressing modes are not legal for the destination operand of an instruction. This means that you cannot modify a data item defined with a DC directive using a simple instruction reference. You must use an address register reference similar to the one in Listing 3.1. Data that require modification should therefore be defined with a DS directive. This means that each reference to these data must specify an address register with displacement mode using A5. (The assembler could and should do this for you. It does not.) If you forget to specify this mode on a source reference, the assembler will cheerfully generate an instruction that references incorrect data. The assembler will report such an error on a destination operand, however. If the data require an initial value, you must add code to explicitly initialize it.

Address constants present another problem that stems from the Macintosh loader. On systems with relocation, you can specify a DC.L directive with a label as an operand. This generates a longword containing the absolute address of the label, which is then relocated by the loader. On the Macintosh, you must construct the address by defining the constant as the difference between two labels and performing the appropriate relocation by adding the effective address of one label to get to the other label.

Size Limitations

A problem related to the lack of relocation is the size limitation on program code and data segments. This stems from the use of 16-bit addressing modes. The 68000 has no modes allowing 32-bit address displacements (other than for absolute addresses). The best that can be done is a 16-bit signed offset (+/−32K) relative to some base address in an address register or the program counter (PC). Program code and initialized data cannot be larger than 32K (32,768) bytes. You can have more than 32K of program space by using the *segment loader* (described in *Inside Macintosh*) to load code in 32K chunks.

The uninitialized (A5-relative) data area is also limited to 32K by the 16-bit displacement on the address-register-relative addressing mode. Larger data areas must be dynamically allocated with memory management calls.

Heap Compaction Problems

Of course, dynamically allocated arrays are subject to the dangling pointer problems presented previously. This implies that handles, not addresses, of dynamic areas must be used. Using handles can be cumbersome, particularly in linked-list manipulation.

SAMPLE APPLICATION

The remainder of this chapter describes the code for a sample application called MacDoodle. MacDoodle is a "poor man's MacPaint." This sample application illustrates the very basic concepts behind a paint program, including simple geometric shapes, text, and printing. It is not nearly as sophisticated as MacPaint, but it is small enough to be readily understood.

MacDoodle Menu Operation

The MacDoodle program can be started by opening the application file or by opening or printing a MacDoodle document from the Finder. MacDoodle documents are compatible with all versions of MacPaint; MacDoodle can read a MacPaint document and MacPaint can read a MacDoodle document. The program that last saved a particular document will be invoked when the document is opened from the Finder.

MacDoodle has a single window and seven menus. The window is a standard Macintosh document window, with scroll bars and a size box. Figure 6.18 shows the MacDoodle menu bar. The first menu is the Apple menu, which contains a standard "About . . ." choice and the desk accessories. The other menu selections are File, Edit, Draw, PenSize, Font, and Style. They are described below.

File

Figure 6.22 shows the File menu selections. The New choice clears the window and the internal file buffer. Open reads an existing file into memory, using the POPEN macro to display the file names. Save saves the file using its existing name. Save As allows the user to specify a new name for the file using the PCREATE macro dialog. Save behaves like Save As if the file has not been created. Print prints the document, and Quit returns the user to the Finder.

Edit

The Edit menu in Figure 6.23 has the standard Undo, Cut, Copy, and Paste choices. Only the Undo choice is supported by MacDoodle. The others are used by desk accessories invoked while MacDoodle is running. These programs rely on the presence and position of these items.

Draw

The Draw menu is shown in Figure 6.24. The option you choose from this menu determines what effect dragging the mouse in the window content area will have.

Selecting Lines lets the user draw straight lines from the point where the mouse button is pressed to the point where it is released. Boxes and Ovals draw rectangles and ellipses, respectively, while the mouse button is down. Freehand acts like the pencil selection in MacPaint—while the mouse button is down, each movement of the mouse is plotted as a point on the screen. Text allows the user to enter a line of text. All the text fonts and attributes are supported by MacDoodle. The Eraser selection performs like freehand drawing, except that the area under the mouse form is erased rather than plotted. Erase Screen erases the area of the document displayed in the window.

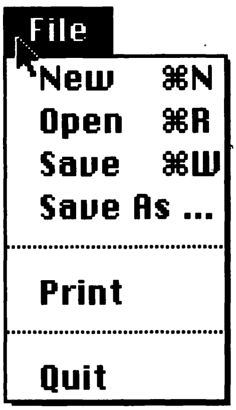


***Figure* 6.22** – *The MacDoodle File menu*

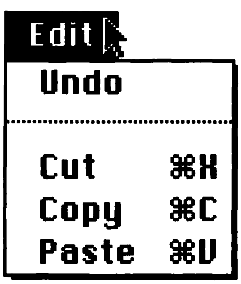


***Figure* 6.23** – *The MacDoodle Edit menu*

PenSize

The PenSize menu (shown in Figure 6.25) controls the size of the graphics pen. The pen shape is always square, and the pen pattern is always solid black.

Pen sizes range from a 1 × 1 to a 16 × 16-pixel square. The PenSize menu affects the thickness of the objects drawn by Lines, Boxes, Ovals, Freehand, and Eraser selections on the Draw menu. The mouse cursor for the Lines and Freehand selections is a solid black square the same size as the pen size. The cursor for the Eraser is a hollow square that corresponds to the pen size. The cursor for Boxes and Ovals is always a cross.

Font

The Font menu is shown in Figure 6.26. Fonts not present in the System file are disabled automatically.

Style

The Style menu, shown in Figure 6.27, allows the user to select text attributes and scaling. The user can select any combination of attributes, with two exceptions. Selecting Plain text disables all of the other attributes, and the Compressed and Expanded options are mutually exclusive. Real font sizes are indicated by changing the proper menu selections. The 12 point selection in Figure 6.27 illustrates this technique. This indicator changes each time the user selects a new font from the Font menu.

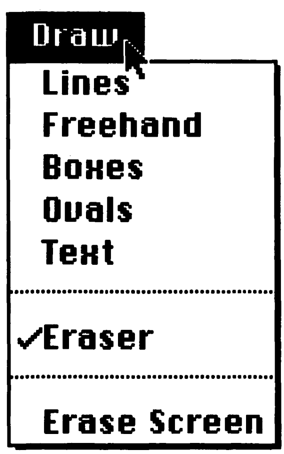


Figure 6.24 *– The MacDoodle Draw menu*

Structure of MacDoodle

The MacDoodle code is divided into three parts:

1. The main program containing the event loop
2. Routines associated with menu processing
3. Routines associated with drawing

These routines communicate by means of shared global variables and subroutine calls. The picture being drawn resides in dynamically allocated buffers. The main routine consists of a loop that calls GETEVENT and then calls the appropriate routine in each of the other modules to process the event. The entire application is based on responses to external events. A complete source listing is available in Appendix E.

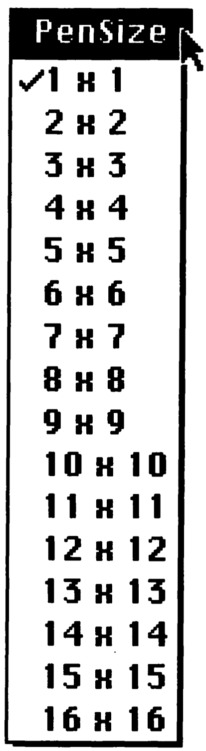


Figure 6.25 *– The MacDoodle PenSize menu*

Buffers

The MacDoodle buffering structure is a key element in understanding the way this program works. There are three buffers: a file buffer, a screen buffer, and an undo buffer. Each of these is dynamically allocated and can therefore be moved as a result of a system call. Figure 6.28 shows the relationship of these buffers.

A MacPaint document covers an 8.5 × 11 sheet of paper, or 576 × 720 pixels. Only part of such a document can be displayed on the screen at one time. MacPaint documents are stored on disk in packed form, using the _PackBits routine described previously. A document is packed one row at a time (576 pixels or 72 bytes). There is also a 512-byte header at the beginning of the file.

The file buffer contains the information that is not displayed on the screen. It consists of two separate areas, one at the beginning of the file and one at the end. These areas are stored in packed format to reduce memory usage. Information between these two regions is stored in the screen buffer.

The screen buffer consists of a bitmap image of the portion of the document currently displayed on the screen. Because the information is packed a row at a time, the screen buffer contains complete rows. The number of rows in the screen buffer depends on the size of the window.

Figure 6.26 – *The MacDoodle Font menu*

The undo buffer contains a packed version of the previous data in the screen buffer. When an Undo operation is selected, the contents of this buffer replace the contents of the screen buffer. MacDoodle packs a copy of the screen buffer data into the undo buffer before each drawing operation. This effectively allows the user to back up one operation.

Another buffer contains the MacPaint file header. A file created with MacDoodle will have 512 bytes of zero in the header. MacPaint uses this area to store the patterns for the document. MacDoodle preserves the header when opening an existing file.

Global Variables

State information, such as current menu selections, is kept in global variables. These variables are generally set by the menu routines when the

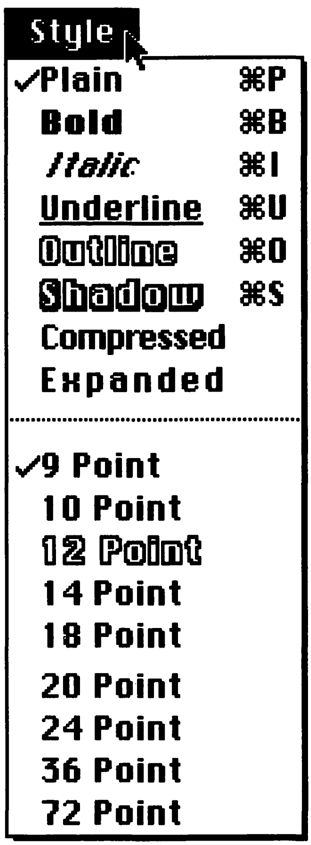


Figure 6.27 *– The MacDoodle Style menu*

user selects options, then referred to by the drawing routines as the user performs drawing actions.

Information that can be encoded in a single bit is contained in the Flags byte. There are macros that clear a flag, set a flag, and test a flag. These macros are called CLRFLAG, SETFLAG, and TSTFLAG, respectively. They each have a single parameter, the name of the flag. The possible flag names are listed below:

Bit	Name	Interpretation
0	F_File	Set when the buffer has a valid file name
1	F_Change	Set when the buffer has been changed but the changes have not been saved in a file
2	F_Text	Set during text entry

In addition to the Flags byte, there are a number of global variables that are used by all modules in the program. They are listed below:

Name	Data Type	Contents
Cursor	Word	Resource ID of current cursor
DrwType	Word	Current Draw menu selection number
FBEnd	Word	Bytes in last part of file buffer
FBStart	Word	Bytes in first part of file buffer
FBufSiz	Long	Size of file buffer
FileBuf	Long	File buffer handle
Fname	String	File name in buffer
FontNum	Word	Current font resource ID
FontSel	Word	Current Font menu selection number

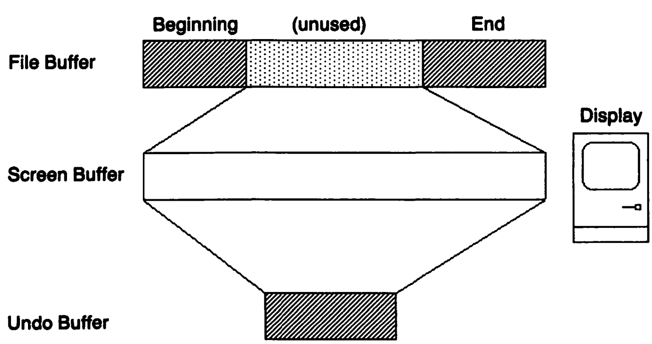


Figure 6.28 *– The MacDoodle buffer structure*

Name	Data Type	Contents
PenSize	Word	Current PenSize menu selection number
PenType	Word	1 for eraser, 2 for drawing
SPoint	Point	Point for window origin
SRow	Word	First row number in screen buffer
SSize	Point	Window content region size
TextFlg	Word	Current text attribute word
TextSel	Word	Current Text Size menu selection number
TextSiz	Word	Current text size in points
VolRef	Word	Volume reference number for file
WPointer	Long	Window address

Macros and global variables are defined in the part of the program that is not listed in this chapter. See Appendix E for these definitions.

Event Code

Listing 6.1 contains the MacDoodle main routine. This part of the program executes first and controls the operation of the rest of the program.

Lines 5–6 save the window address and size from the PROGRAM macro in global variables. Line 7 clears the Flags variable. Lines 8–9 call the initialization routines for the draw and menu code. Lines 14–18 form the event loop for MacDoodle. The return code from the GETEVENT macro serves as an index into a table of JMP instructions. The JSR at line 17 transfers control to the proper JMP instruction, which transfers control to the proper routine. The jump table occupies lines 22–44. The codes returned by GETEVENT appear in the comment field of each entry in the table.

The cursor global variable is the resource ID or the cursor displayed in the document window. The cursor is determined by the Draw and PenSize menus.

Menu Code

The menu module of the MacDoodle program contains code that modifies the global data area to allow changes in the drawing mode. This code consists of an initialization routine, a dispatch routine, and code for each menu selection.

Initialization

The first part of the menu initialization code allocates the dynamic buffers. Listing 6.2 contains this code.

```
        Program MacDoodle,DEBUGOFF,nMenu
;
;       Perform initialization code
;
        MOVE.L  A0,WPointer(A5) ;       Save window address
        MOVE.L  D0,SSize(A5)    ;        "    "    size
        CLR.B   Flags(A5)       ;       Clear Flags byte
        JSR     DrawInit        ;       Call initialization
        JSR     MenuInit        ;                               routines
;
;       This is the top of the event loop.  Get an event from the run-time
;       library routine, and call the proper routine to process it.
;
EVLoop: GETEVENT Cursor(A5)     ;       Get next event
        LSL.W   #2,D0           ;       Multiply by 4
        LEA     JmpTab,A0       ;       A0 -> Jump array
        JSR     0(A0,D0)        ;       Jump to event handler
        BRA     EVLoop          ;       Get next event
;
;       Event Jump Table
;
JmpTab:                         ;       Table of jumps to routine handlers
        JMP     NullEvt         ; 0     No event, get next
        JMP     Mouse           ; 1     Mouse down in window
        JMP     NullEvt         ; 2     Mouse up, ignore
        JMP     KeyDown         ; 3     Key down
        JMP     NullEvt         ; 4     Key up, ignore
        JMP     KeyDown         ; 5     Autokey (same as key down)
        JMP     UpDate          ; 6     Update event
        JMP     NullEvt         ; 7     Disk event, ignore
        JMP     NullEvt         ; 8     Activate event, ignore
        JMP     NullEvt         ; 9     Abort event, ignore
        JMP     NullEvt         ;10     Network event, ignore
        JMP     NullEvt         ;11     Driver event, ignore
        JMP     NullEvt         ;12     Application event #1
        JMP     NullEvt         ;13     Application event #2
        JMP     NullEvt         ;14     Application event #3
        JMP     NullEvt         ;15     Application event #4
        JMP     MenuEvt         ;16     Menu choice
        JMP     CloseEvt        ;17     Close window event
        JMP     ReSize          ;18     Resize window event
        JMP     DoScroll        ;19     Scroll event
        JMP     VScroll         ;20     Scroll V with thumb
        JMP     HScroll         ;21     Scroll H with thumb
NullEvt RTS                     ;       Null event handler
        ENDPROGRAM
```

Listing 6.1 *– MacDoodle event code*

Lines 1–5 illustrate how to define Macintosh trap numbers. On a 512K Macintosh, you would normally use the MacTraps.D Include file supplied with the Macintosh Development System to define all the trap numbers. Defining the trap numbers individually allows you to conserve memory space during assembly. Large files will not assemble on a 128K Macintosh unless you do this.

Lines 7–12 allocate the file buffer. Buffer sizes are done with equates in this program. A fancier version of MacDoodle might allocate the buffers when a file is opened (or the New selection is chosen). This code is duplicated for the undo buffer, the screen buffer, and the header buffer in subsequent lines.

The next part of the initialization code checks to see which fonts exist in the System file. The code uses the _RealFont system trap to perform this function, as shown in Listing 6.3.

There is a table of font resource IDs at the label "Fonts". The table is terminated by a word containing −1. The loop in lines 3–13 calls

```
 1:          .TRAP    _NewHandle       $A122 ; NewHandle trap word
 2:          .TRAP    _HLock           $A029 ; Lock dynamic area
 3:          .TRAP    _HUnLock         $A02A ; Unlock dynamic area
 4:          .TRAP    _GetFName        $A8FF ; Get font name trap
 5:          .TRAP    _RealFont        $A902 ; Real font trap
 6: MenuInit
 7:          MOVE.L   #FileSize,D0       ;       File buffer size
 8:          MOVE.L   D0,FBufSiz(A5)     ;       Remember for later
 9:          _NewHandle                  ;       Get buffer
10:          MOVE.L   A0,FileBuf(A5)     ;       Store handle
11:          TST.L    FileBuf(A5)        ;       Really got it?
12:          BEQ      MemError           ;       No, complain
13:          MOVE.L   #UnDoSize,D0       ;       Allocate
14:          MOVE.L   D0,UBufSiz(A5)     ;
15:          _NewHandle                  ;                 undo
16:          MOVE.L   A0,UnDoBuf(A5)     ;                        buffer
17:          TST.L    UnDoBuf(A5)        ;       Got it?
18:          BEQ      MemError           ;       No
19:          MOVE.L   #ScrSize,D0        ;       Get
20:          MOVE.L   D0,SBufSiz(A5)     ;                screen
21:          _NewHandle                  ;                        buffer
22:          MOVE.L   A0,ScrBuf(A5)      ;       Set buffer handle
23:          TST.L    ScrBuf(A5)         ;       Got it?
24:          BEQ      MemError           ;       No
25:          MOVE.L   #HdrSize,D0        ;       Get
26:          MOVE.L   D0,HBufLen(A5)     ;                header
27:          _NewHandle                  ;                        buffer
28:          MOVE.L   A0,HdrBuf(A5)      ;       Set buffer handle
29:          TST.L    HdrBuf(A5)         ;       Got it?
30:          BEQ      MemError           ;       No
```

***Listing 6.2** – The MacDoodle dynamic buffer initialization*

_GetFName for each font ID in the table. If no name exists for the font ID, the font is not in the System file. The corresponding menu selection is disabled at line 11 in this case. Note that the table and the menu definition must agree. If you change either of these, you must also change the other.

The final initialization code sets up default values for the menu settings. This code is shown in Listing 6.4.

Line 1 initializes the file name to the null string. Lines 2–6 set the default values for the Draw menu. Lines 7–8 set the text style for the Compressed and Expanded selections in the Style menu. Text style for the other menu

```
 1:          LEA      Fonts,A3        ;       A3 -> Font table
 2:          MOVEQ.L  #1,D3           ;       D3 counts menu items
 3: RFontloop                         ;       Top of loop
 4:          MOVE.W   (A3)+,D0        ;       Fetch font ID
 5:          BLT      RFDone          ;       LT => End of list
 6:          MOVE.W   D0,-(SP)        ;       Push font ID
 7:          PEA      FName(A5)       ;       -> Return name space
 8:          _GetFName                ;       Get the font name
 9:          TST.B    FName(A5)       ;       Any bytes in string??
10:          BNE      WasFont         ;       Yes, font must exist
11:          DISABLE  #mFont,D3       ;       Disable the entry
12: WasFont  ADDQ.L   #1,D3           ;       Bump entry number
13:          BRA      RfontLoop       ;       Loop till fonts exhausted
14: RFDone:                           ;       Here when all done
```

Listing 6.3 *– The MacDoodle Font menu initialization*

```
 1:         CLR.B   FName(A5)          ;      Initialize file name to null
 2:         MOVE.W  #2,DrwType(A5)     ;      Set DrawType variable
 3:         MOVE.W  #2,PenType(A5)     ;      Set pen type for cursor
 4:         MOVE.W  #1,PenSize(A5)     ;      Set pen size
 5:         CHECK   #mDraw,#2          ;      Check Freehand
 6:         CHECK   #mPen,#1           ;      Check 1 x 1 pen size
 7:         STYLE   #mStyle,#7,#mCmpress;     Change Compressed style
 8:         STYLE   #mStyle,#8,#mExpand;      Change Expanded style
 9:         BSR     Plain              ;      Make text plain
10:         MOVE.W  #SizeBase,D1       ;      Set size to minimum
11:         MOVE.W  D1,TextSel(A5)     ;      Set up so UNCHECK won't bomb
12:         BSR     TSize              ;      Go set size
13:         MOVE.W  #1,D1              ;      Check first font entry
14:         MOVE.W  D1,FontNum(A5)     ;      Set up so UNCHECK won't bomb
15:         BSR     SetFont            ;      Set it
16:         MOVE.W  #1,D1              ;      Go init cursor
17:         BSR     SetPen             ;      Choose single-pixel pen
18:         BSR     New                ;      Clear out junk
```

Listing 6.4 *– The final menu initialization* items is set in the resource file. Lines 10–12 set the text size part of the Style menu. Lines 13–15 initialize the Font menu to the first selection. Lines 16–17 select the 1 × 1 entry in the PenSize menu. Line 18 initializes the buffer by calling the New File menu selection routine.

Application Parameters

After initialization, MacDoodle checks for application parameters. When you select one or more documents in the Finder, and then choose Open or Print from the File menu, the Finder passes the file names to the application by means of an area of memory called the application parameter area. The format of this area is shown in Figure 6.29.

Offset 16 from register A5 contains a handle to the application parameter area. The action code word contains a 0 if the files are to be opened, and a 1 for printing. The file count word contains the number of file descriptions that follow. This word is zero if the user opens the application instead of a document. The file descriptions that follow contain the volume reference number, file type, version number, and file name. Note that the file name is a variable-length Pascal string. File descriptions always begin on a word boundary. There might be an unused byte at the end of a file name to allow the next volume reference number to begin on a word boundary.

Listing 6.5 contains the code that processes the application parameters for MacDoodle. Lines 1–2 compute the address of the parameter area. Lines 3–5 check the file count and do nothing if there are no file descriptions present. Lines 6–11 move the file name from the parameter area into the FName global variable. Line 12 copies the volume reference number to another global variable. Lines 13–17 open the file, checking for errors. At line 18, code from the normal open routine is called as a subroutine. This code displays the file in the window, and sets the window title to the file name. Line 19 checks the file flag. If this flag was set, the file was processed properly. Lines 21–27 decide whether to start processing events (if the action code is open) or to print the file and exit (if the action code indicates print). Lines 28–30 list the error routine for memory allocation failure.

Dispatching

Now that the menu initialization is done, let's see how menu events are dispatched. MacDoodle handles menu events using a common section of code. This code takes the menu number and selection number, indexes them into a two-dimensional array of routine addresses, and jumps to a routine that handles the event. There is an entry in the table that corresponds to each possible menu/selection combination. Listing 6.6 shows the menu dispatcher.

Lines 1–2 change a window close event to a New menu selection. Lines 10–12 close any open text entry before processing the event. Lines 13–17 place the menu item number in D2 and the menu number in D3. This code also makes these numbers zero-relative, for use in indexing.

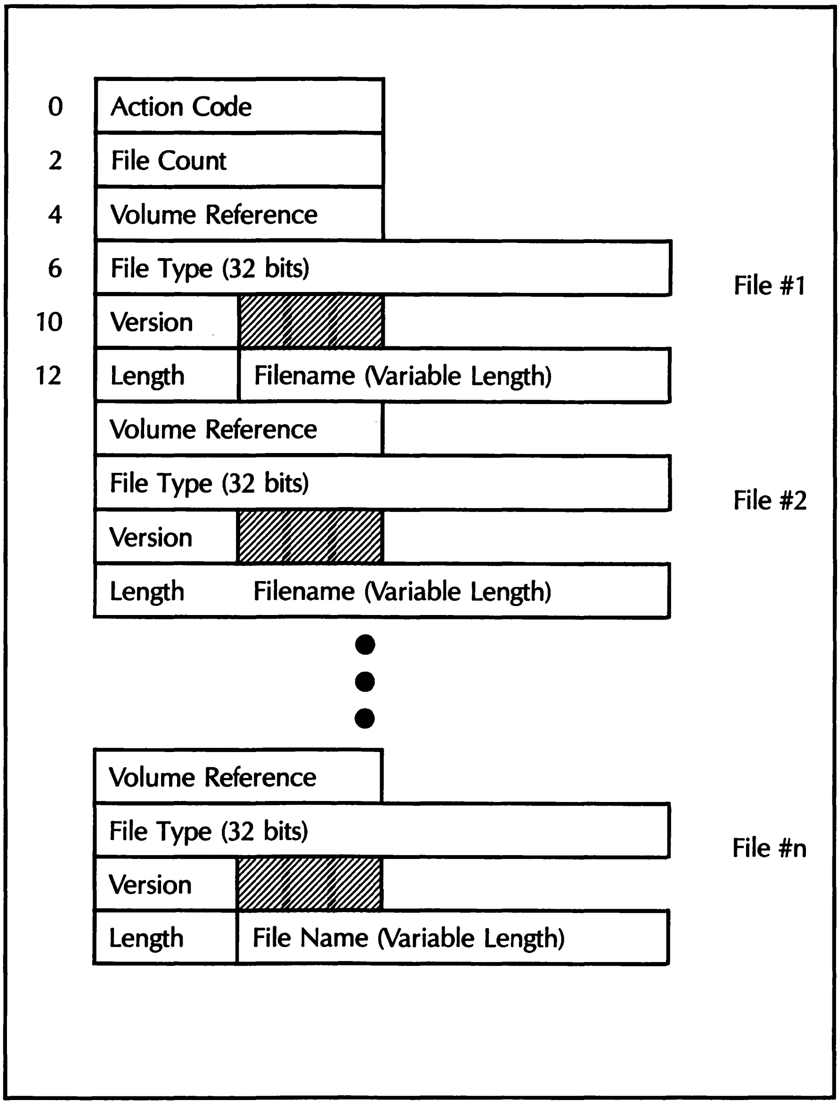


Figure 6.29 *– Application parameters*

Remember that the run-time library handles menu number 1. This is the reason why line 16 subtracts two from the menu number. Lines 19–21 compute the table index, and lines 22–23 jump to the right routine. Note that the menu item number (still 1-relative) remains in D1.W as a result of the SWAP instruction at line 18. Many of the menu routines need this quantity for the CHECK or STYLE macros. Lines 24–25 show the code which handles menu items that cannot be selected unless there is an error in the program.

Lines 29–50 define the menu routine address table. The JTAB macro generates a word for each operand, containing the difference between the operand and the label JBASE. The code generated looks like this:

```
DC.W     %1-JBASE          ;  First operand
DC.W     %2-JBASE          ;  Second operand
(etc.)
```

```
 1:         MOVE.L  ParmHdl(A5),A4  ;      A4 ->
 2:         MOVE.L  (A4),A4         ;              Parameter area
 3:         TST.W   PCount(A4)      ;      Any files there?
 4:         BNE     @1              ;      Yes, go process
 5:         RTS                     ;      Done, back to event handler
 6: @1      LEA     PFName(A4),A0   ;      A0 -> Finder-supplied name
 7:         LEA     FName(A5),A1    ;      A1 -> File name area
 8:         CLR.W   D0              ;      D0 =
 9:         MOVE.B  (A0),D0         ;              Count byte
10: @2      MOVE.B  (A0)+,(A1)+     ;      Move to
11:         DBRA    D0,@2           ;              expected area
12:         MOVE.W  PVref(A4),VolRef(A5);  Save volume reference #
13:         OPEN    FName(A5),DEFAULT,PVRef(A4); Open file
14:         MOVE.W  D0,D2           ;      Save file number
15:         BPL     @4              ;      If OK
16:         StopAlert #OErr         ;      tell user we couldn't do it
17: @3      EXIT                    ;      then quit
18: @4      BSR     OpenOK          ;      Complete the open
19:         TstFlag F_File          ;      Did it go ok?
20:         BEQ     @3              ;      EQ => No, alert already done
21:         MOVE.L  ParmHdl(A5),A4  ;      Reload
22:         MOVE.L  (A4),A4         ;              parameter handle
23:         TST.W   PMsg(A4)        ;      Open or print
24:         BEQ     @5              ;      EQ => Just open
25:         BSR     xPrint          ;      Else, print file
26:         EXIT                    ;      and quit
27: @5      RTS                     ;      File now open
28: MemError                        ;      Here if out of memory
29:         StopAlert #MemErr       ;      Quit
30:         EXIT                    ;
```

Listing 6.5 *– Application parameter handling*

```
CloseEvt:                              ;       Called from the event loop
          MOVE.L   #$20001,D1          ;       Fake out menu handler
;
;         Menu Event Handler.  Entered with constant 10 (hex) in D0.W,
;         and the menu number in D1.W.  The high word of D1 has the selection
;         number.  The menu number is in the range 2-n and the choice number
;         is in the range 1-n. (RTL handles menu #1, the Apple menu.)
;
MenuEvt:                               ;       Here for a menu choice
          TstFlag F_Text               ;       Text open?
          BEQ      @1                  ;       No, continue
          JSR      CloseText           ;       Yes, close first
@1:       MOVE.W   D1,D2               ;       Copy menu choice to D2
          SUBQ.W   #1,D2               ;       Force choice zero relative
          SWAP     D1                  ;       Menu number to D1.W
          SUBQ.W   #2,D1               ;       Force zero relative
          MOVE.W   D1,D3               ;       Copy to D3
          SWAP     D1                  ;       Choice back to D1.W
          MULU     #nMenu-1,D2         ;       Multiply D2 by # menus
          ADD.W    D2,D3               ;       Add in menu choice
          LSL.W    #1,D3               ;       Multiply by entry size (word)
          MOVE.W   JBASE(D3.W),D3      ;       Fetch offset word
          JMP      JBASE(D3.W)         ;       Go to routine
Void:     StopAlert #ProgramErr        ;       Here if a real problem
          EXIT                         ;       Fade away
;
;         Define menu jump table
;
JBASE:                                 ;       Offset for jump
;
;                  File    Edit    Draw    Pensize  Font      Style
;                  ----    ----    ----    -------  ------    -------
          JTAB     New,    Undo,   Lines,  SetPen,  SetFont,  Plain    ; 1
          JTAB     xOpen,  Void,   FreeH,  SetPen,  SetFont,  TStyle   ; 2
          JTAB     Save,   Cut,    Boxes,  SetPen,  SetFont,  TStyle   ; 3
          JTAB     SaveAs, Copy,   Ovals,  SetPen,  SetFont,  TStyle   ; 4
          JTAB     Void,   Paste,  Text,   SetPen,  SetFont,  TStyle   ; 5
          JTAB     xPrint, Void,   Void,   SetPen,  SetFont,  TStyle   ; 6
          JTAB     Void,   Void,   Eraser, SetPen,  SetFont,  TCmpress ; 7
          JTAB     xQuit,  Void,   Void,   SetPen,  SetFont,  TExpand  ; 8
          JTAB     Void,   Void,   EraseA, SetPen,  SetFont,  Void     ; 9
          JTAB     Void,   Void,   Void,   SetPen,  SetFont,  TSize    ;10
          JTAB     Void,   Void,   Void,   SetPen,  SetFont,  TSize    ;11
          JTAB     Void,   Void,   Void,   SetPen,  SetFont,  TSize    ;12
          JTAB     Void,   Void,   Void,   SetPen,  Void,     TSize    ;13
          JTAB     Void,   Void,   Void,   SetPen,  Void,     TSize    ;14
          JTAB     Void,   Void,   Void,   SetPen,  Void,     TSize    ;15
          JTAB     Void,   Void,   Void,   SetPen,  Void,     TSize    ;16
          JTAB     Void,   Void,   Void,   Void,    Void,     TSize    ;17
          JTAB     Void,   Void,   Void,   Void,    Void,     TSize    ;18
```

Listing 6.6 *– The menu dispatch routine*

This technique is different from the one used in the event code in Listing 6.1, which uses JMP instructions. Each entry in the menu table takes only a single word, while in the event code two words are required. The words must contain the difference between two addresses because the Macintosh does not perform program relocation, as discussed previously. Either technique is valid only for labels that lie within 32K of the base address.

The rest of the menu code consists of routines called by the menu dispatcher. Each of these routines handles a particular menu selection.

New File Listing 6.7 shows the code that handles a New selection from the File menu.

Line 2 calls the ChkSave routine. This routine displays the "Save Changes Before Closing?" dialog box if there are unsaved changes in the buffer. If the user selects Yes, the routine saves the buffer in a file. If the buffer has no file name, the user is asked to enter one.

Lines 3–8 initialize the file header buffer to all zeros. Lines 9–18 calculate the address at which to begin storing the null pattern in the file buffer. An entire row of zeros in packed form compresses to two bytes. Lines 19–21 store this null pattern in the last 720 words of the file buffer. Since a document has 720 rows, this is equivalent to reading in a completely

```
 1: New:                               ;      
 2:        BSR      ChkSave            ;      Check for save first
 3:        MOVE.L   HdrBuf(A5),A0      ;      A0 ->
 4:        MOVE.L   (A0),A0            ;              Header buffer
 5:        MOVE.L   HBufLen(A5),D0     ;      D0 = Byte count
 6:        SUBQ.L   #1,D0              ;      Adjust for DBRA
 7: @1:    CLR.B    (A0)+              ;      Clear out
 8:        DBRA     D0,@1              ;                   header buffer
 9:        MOVE.L   FileBuf(A5),A4     ;      A4 ->
10:        MOVE.L   (A4),A4            ;              File buffer
11:        MOVE.L   #FLength*2,D3      ;      Buffer length
12:        MOVE.L   D3,D4              ;      Copy to D4
13:        MOVE.W   D3,D0              ;      and to D0
14:        LSR.W    #1,D0              ;      Adjust D0 for word count
15:        SUBQ.W   #1,D0              ;      and DBRA
16:        SUB.L    FBufSiz(A5),D4     ;      Compute
17:        NEG.L    D4                 ;              buffer
18:        LEA      0(A4,D4.L),A4      ;                      offset
19:        MOVE.L   A4,A0              ;      Copy to A0
20: @2:    MOVE.W   #NullPat,(A0)+     ;      Fill buffer with
21:        DBRA     D0,@2              ;      Compressed zero pattern
22:        ClrFlag  F_File             ;      No file name for buffer now
23:        WTitle   '(No File)'        ;      Set the window title
24:        BRA      DoUpdate           ;      Otherwise, it's just like a finished read
```

Listing 6.7 *– The New File routine* blank drawing. Line 22 clears the file flag bit, indicating that the window has no associated disk file. Line 23 sets the window title to "(No File)". Line 24 merges with the code that reads a file into memory.

Open File Listing 6.8 contains the code that opens an existing document for MacDoodle to use.

The routine is called xOpen because the assembler cannot distinguish between an Open label and the OPEN macro. Line 1 calls the ChkSave

```
 1: xOpen                             ;
 2:          BSR      ChkSave         ;      Check for unsaved changes
 3:          POPEN    Ptype,DEFAULT,Fname(A5),DEFAULT;      Open new file
 4:          MOVE.W   D0,D2           ;      Save file reference number
 5:          BPL      OpenOK          ;      If > 0, then OK
 6:          CMP.W    #Cancelled,D0   ;      Cancelled?
 7:          BNE      @1              ;      No
 8:          RTS                      ;      Yes, just return
 9: @1       MOVE.L   #OErr,D0        ;      Indicate Open error
10:          BRA      Bail            ;      Bail out
11: OpenOK   SCursor  #Watch          ;      Indicate patience required
12:          MOVE.W   D1,VolRef(A5)   ;      Save volume reference number
13:          GETFINFO Fname(A5),InfoBuf(A5),VolRef(A5); Get file info
14:          LEA      InfoBuf(A5),A0  ;      A0 -> File information
15:          MOVE.L   DFSize(A0),D4   ;      Copy file size to D4
16:          SUB.L    HBufLen(A5),D4  ;      Subtract file header size
17:          MOVE.L   HdrBuf(A5),A0   ;      A0 = Buffer handle
18:          MOVE.L   (A0),A4         ;      Dereference
19:          MOVE.L   HBufLen(A5),D3  ;      Byte count to D3
20:          BSR      IssueRd         ;      Go read it
21:          MOVE.L   D4,D3           ;      Main byte count to D3 now
22:          CMP.L    FBufSiz(A5),D3  ;      Enough room in buffer?
23:          BLT      BufOK           ;      Yes
24:          MOVE.L #MemErr,D0        ;      No, notify user
25:          BRA      OBail           ;      Bail out
26: BufOK                             ;
27:          MOVE.L   FileBuf(A5),A0  ;      A0 -> Buffer pointer
28:          MOVE.L   (A0),A4         ;      A4 -> Buffer
29:          SUB.L    FBufSiz(A5),D4  ;      Calculate
30:          NEG.L    D4              ;             offset to bottom
31:          LEA      0(A4,D4.L),A4   ;      A4 -> File offset in buffer
32:          BSR      IssueRd         ;      Do the read
33:          CLOSE    D2              ;      Close the open file
34:          WTitle   FName(A5)       ;      Set the window title
35:          SetFlag  F_File          ;      Now have a file in the buffer
36: DoUpdate:                         ;      Now get ready to update screen
37:          CLR.W    FBStart(A5)     ;      No bytes in buffer beginning
38:          CLR.W    SRow(A5)        ;      Screen buffer is row #0
39:          MOVE.W   D3,FBEnd(A5)    ;      Set up proper # of bytes
40:          JSR      UnBuf           ;      Convert to screen image
41:          ClrFlag  F_Change        ;      No changes now
42:          MOVE.L   SSize(A5),D1    ;      Set Size variable
43:          JMP      ReSize          ;      Reinit screen
```

Listing 6.8 – *The Open File routine*

routine to ensure that the user has a chance to save any changes made to the buffer before reading in the new file. Lines 3–10 present the Open dialog and check for Open errors. If the user cancelled the Open (lines 6–8), control reverts back to the event code in Listing 6.1. Any other error (lines 9–10) causes an alert box to be displayed. When the user clicks OK, control reverts back to the event code.

At line 11, the file is open. Line 11 changes the mouse cursor to the watch symbol. Lines 13–16 compute the size of the data portion of the file. Lines 17–20 read the header file, using a subroutine that both locks the buffer and performs the read. Lines 21–25 check to see that the file will fit in the file buffer. Lines 27–32 place the entire file at the high-address end of the file buffer. Line 33 closes the open file. Line 34 changes the window title to the name of the file selected by the user. Line 35 sets the file flag. Subsequent Save menu choices will overwrite this file.

At line 36, the file is completely contained in the end portion of the file buffer. The code in Listing 6.7 branches to the label DoUpdate. Code following this label unpacks the first part of the file into the screen buffer and updates the display. Lines 37–38 indicate that the first part of the file is to be displayed. The beginning part of the file buffer contains no data, and row number zero is the first row in the screen buffer. Lines 39–40 call the UnBuf routine to split the file buffer into three pieces: the beginning part of the file buffer, the screen buffer, and the ending part of the file buffer. Line 41 clears the change flag, so that closing the window will not display the Save Changes dialog. Lines 42–43 call the ReSize routine, which is normally used to redisplay the window after it has been resized. In this case, it simply displays the data in the screen buffer.

Listing 6.9 shows the IssueRd routine, which reads data into a dynamic buffer. This code locks the buffer before use. Lines 2–3 lock the buffer, line 4

```
 1: IssueRd                             ;
 2:          MOVE.L  A0,-(SP)            ;     Save handle
 3:          _HLock                      ;     Lock it
 4:          READ    D2,(A4),D3          ;     Read desired data
 5:          MOVE.L  (SP)+,A0            ;     Restore handle address
 6:          MOVE.L  D0,-(SP)            ;     Save returned byte count
 7:          _HUnLock                    ;     Unlock handle
 8:          CMP.L   (SP)+,D3            ;     Read OK?
 9:          BNE     @1                  ;     No
10:          RTS                         ;     Yes, just return
11: @1       ADDQ.L  #4,SP               ;     Pop off return address
12:          MOVE.L  #ReadErr,D0         ;     Set error code
13:          BRA     OBail               ;     Bail out
```

***Listing 6.9** – IssueRd routine*

performs the read, and lines 5–7 unlock the buffer. Lines 8–13 check for a successful read and issue an error alert on read failure.

Save and Save As The two menu selections that save the buffer contents in a disk file appear in Listing 6.10.

Code that destroys the buffer contents calls ChkSave first. This routine determines whether there are unsaved changes to the buffer (lines 2–3) and issues a dialog box if there are (line 4). Lines 5–6 determine if the item selected by the user was the OK button.

```
 1: ChkSave:                        ;       
 2:         TstFlag F_Change        ;       Any changes?
 3:         BEQ     NoSave          ;       No, just return
 4:         Dialog  #Saveit         ;       Do dialog
 5:         CMP.W   #DoSave,D0      ;       Did user ask to save?
 6:         BNE     NoSave          ;       No, just quit
 7: Save    TstFlag F_File          ;       File in buffer?
 8:         BEQ     SaveAs          ;       No, need file name from user
 9:         CREATE  Fname(A5),DEFAULT,VolRef(A5); Create the file
10:         MOVE.W  D0,D2           ;       Save file ref number
11:         BMI     CFileErr        ;       If < 0, couldn't create
12: DoWrite SetFlag F_File          ;       Now have a file in buffer
13:         SCursor #Watch          ;       Set the cursor to the watch
14:         MOVE.L  HdrBuf(A5),A0   ;       Fetch buffer handle
15:         MOVE.L  (A0),A4         ;       Fetch address
16:         MOVE.L  HBufLen(A5),D3  ;       D3 = Desired byte count
17:         BSR     IssueWr         ;       Go issue write
18: ;
19: ;       Now write out the packed data for the file
20: ;
21: HWriteOK                        ;       Here to write buffer
22:         JSR     RePack          ;       Pack to single buffer
23:         MOVE.L  FileBuf(A5),A0  ;       A0 -> Handle
24:         MOVE.L  (A0),A4         ;       A4 -> Buffer
25:         CLR.L   D3              ;       Clear D3 high word
26:         MOVE.W  FBEnd(A5),D3    ;       D3.L =  Ending byte count
27:         MOVE.L  D3,D4           ;       Copy buffer length
28:         SUB.L   FBufSiz(A5),D4  ;       Calculate
29:         NEG.L   D4              ;                   offset to bottom
30:         LEA     0(A4,D4.L),A4   ;       A4 -> File offset in buffer
31:         BSR     IssueWr         ;       Do the write
32:         CLOSE   D2              ;       Close the file
33:         JSR     UnBuf           ;       Undo the repack
34:         ClrFlag F_Change        ;       Indicate file saved
35:         GETFINFO Fname(A5),InfoBuf(A5),VolRef(A5); Get file info
36:         LEA     InfoBuf(A5),A0  ;       A0 -> Buffer
37:         MOVE.L  #MPType,FFType(A0);     Set file type
38:         MOVE.L  #DAppl,FFAppl(A0);      Set file application
39:         SETFINFO Fname(A5),InfoBuf(A5),VolRef(A5); Update disk
40: NoSave  RTS                     ;       Done
```

Listing 6.10 *– The ChkSave routine*

ChkSave falls through to the Save menu selection at line 7. This code determines if the buffer has an associated file name in lines 7–8. If not, the SaveAs procedure is used instead. The buffer gets a file name as the result of an Open or Save As menu selection. Opening or printing a file from the Finder is essentially the same as selecting Open from the File menu. If the buffer has a file name, Lines 9–11 create a new copy of the file. Any previously existing copy of the file will be overwritten. Line 12 sets the file flag, and line 13 changes the mouse cursor to the watch symbol. Note that the next call to GETEVENT changes the cursor back to whatever it was before. Lines 14–17 set up the registers for a call to the IssueWr routine to write the file header.

Line 22 calls the RePack routine, which packs the entire file into the end section of the file buffer. Lines 23–30 compute the address and length of this data. Line 31 calls the IssueWr routine to write the packed data to the file. Line 32 closes the file. Line 33 calls the UnBuf routine to set up the three buffer areas. Line 34 clears the change flag, indicating that there are now no unsaved changes in the buffer. Lines 35–39 change the file type and application signature to PNTG and DOOD, respectively. The file type is the same as the one used by MacPaint. The DOOD signature causes MacDoodle to be activated when the user double clicks on the document icon.

The IssueWr routine appears in Listing 6.11. This routine writes data to disk from a dynamic buffer and is similar to the IssueRd routine (Listing 6.9).

The SaveAs code is shown in Listing 6.12. Line 1 invokes the PCREATE macro to obtain the file name from the user and create the file. Lines 2–5 set up the registers and branch back to the label DoWrite in Listing 6.10. Lines 6–12 make up the common error routine for all errors in the File menu code. There are two labels: Bail and OBail. Bail is called to change the mouse cursor to the arrow and display the alert box. OBail closes the open file first.

```
IssueWr MOVE.L  A0,-(SP)        ;       Save the handle
        _HLock                  ;       Lock down the buffer
        WRITE   D2,(A4),D3      ;       Do request
        MOVE.L  (SP)+,A0        ;       Restore handle address
        MOVE.L  D0,-(SP)        ;       and keep byte count
        _HUnLock                ;       Unlock it
        CMP.L   (SP)+,D3        ;       All written?
        BNE     @1              ;       No, bail out
        RTS                     ;       Return to caller
@1      ADDQ.L  #4,SP           ;       Pop return address
        MOVE.L  #Werr,D0        ;       Tell the user we didn't
        BRA     OBail           ;       Close file and exit
```

***Listing 6.11** – The IssueWr routine*

Print Selecting Print from the File menu causes the code in Listing 6.13 to run. Line 1 sets the mouse cursor to the watch symbol. Lines 3 and 8 open and close the printer driver, using routines contained in the PrLink.Rel file supplied with the Macintosh MDS system. The PrDraw routine actually does the printing. This routine is part of the drawing code and will be explained later.

Quit Listing 6.14 shows the code executed when Quit is chosen from the File menu. Line 1 checks for unsaved changes to the file buffer. Line 2 exits back to the Finder.

```
SaveAs  PCREATE 'Save As',Fname(A5),Fname(A5),DEFAULT
        MOVE.L  D0,D2           ;       Save file ref number
        BLT     CFileErr        ;       Error, quit
        MOVE.W  D1,VolRef(A5)   ;       Save volume reference number
        BRA     DoWrite         ;       Now go write the file
CFileErr                        ;       Can't create file
        MOVE.L  #Cerr,D0        ;       Set error code
        BRA     Bail            ;       Quit
OBail   CLOSE     D2            ;       Open bail out routine
Bail    SCursor   #Arrow        ;       Set the cursor to the arrow
        StopAlert D0            ;       Do the alert
        RTS                     ;       Return to caller
```

***Listing 6.12** – The SaveAs routine*

```
1: xPrint                       ;       Here when print selected
2:      SCursor #Watch          ;       Set cursor to watch
3:      JSR     PrDrvrOpen      ;       Open the printer driver
4: ;
5: ;    Call print draw routine (in MacDoodleDraw.Asm)
6: ;
7:      JSR     PrDraw          ;       Do print drawing
8:      JSR     PrDrvrClose     ;       Close the printer driver
9:      RTS                     ;       Return to caller
```

***Listing 6.13** – The xPrint routine*

```
1: xQuit:  BSR     ChkSave      ;       Check on unsaved changes
2:         EXIT                 ;       Back to the Finder
```

***Listing 6.14** – The xQuit routine*

Edit Listing 6.15 shows the code for the Edit menu. The Undo function is the only one supported by MacDoodle.

Lines 2–9 unpack the undo buffer to the screen buffer. Line 10 redraws the content area of the window. The UnPack and ReDraw routines are in the MacDoodleDraw file and will be explained later. Lines 11–14 cause the Cut, Copy, and Paste choices to be ignored.

Draw Listing 6.16 contains code for the Draw menu choices. The code beginning at line 7 handles everything except the Erase Screen choice. Lines 7–9 change the menu check mark to the new selection. Memory location DrwType contains the last selection, for use by the UNCHECK macro in line 7.

The mouse cursor for drawing Lines, Freehand, and Eraser depends both on the Draw menu selection and the PenSize menu selection. The cursor is a solid rectangle for the Lines and Freehand choices, and a hollow rectangle for the Eraser choice. The global variable PenType contains either a 2 for a solid rectangle or a 1 for a hollow rectangle. The PenSize menu setting determines the size of the rectangle. Lines 10–13 set up the PenType variable. Lines 14–17 set up the cursor variable. The CurTbl contains cursor resource IDs for each selection in the Draw menu. A zero means that the cursor ID must be calculated by routine SetCursor (in Listing 6.17 below).

The Erase Screen selection executes the code in lines 36–37. Routine EScreen in MacDoodleDraw.asm performs the erase function.

PenSize Listing 6.17 contains the code that handles a PenSize menu selection. Lines 1–2 delete the check mark from the previous selection. Line 3 puts a check next to the new selection. Line 4 stores the new selection for use when the menu is changed again. Lines 5–9 use the cursor

```
 1: Undo
 2:         MOVE.L  UndoBuf(A5),A0   ;       A0 ->
 3:         MOVE.L  (A0),A0          ;               Undo buffer
 4:         MOVE.L  ScrBuf(A5),A1    ;       A1 ->
 5:         MOVE.L  (A1),A1          ;               Screen buffer start
 6:         MOVE.W  SSize(A5),D0     ;       D0 = Number of rows to unpack
 7:         MOVE.L  A1,A6            ;       Calculate
 8:         ADDA.L  SBufSiz(A5),A6   ;               limit
 9:         BSR     UnPack           ;       Unpack data to screen buffer
10:         JMP     ReDraw           ;       Now redraw screen
11: Cut
12: Copy
13: Paste
14:         RTS                      ;       Return to caller
```

Listing 6.15 *– Edit menu routines* table from Listing 6.16 to see if the cursor needs adjusting; it will if the Draw menu is set to Lines, Freehand, or Eraser.

Lines 10–17 perform any necessary cursor adjustment. The cursor resource ID for the variable-size cursors is a byte consisting of a 1 or a 2 in the high nibble, and the PenSize menu selection minus 1 in the low nibble. Thus, the cursor for drawing 8 × 8 lines would be 17 hex. The cursor for erasing 12 × 12 rectangles would be 2B hex. These cursors reside in the resource file.

Font Listing 6.18 shows the code that supports Font menu selections. Lines 2–3 move the check mark to the desired font. Line 4 stores the

```
 1: Lines
 2: FreeH
 3: Boxes
 4: Ovals
 5: Text
 6: Eraser
 7:         UNCHECK #mDraw,DrwType(A5);      Uncheck old item
 8:         CHECK   #mDraw,D1         ;      Check new item
 9:         MOVE.W  D1,DrwType(A5)    ;      Save item number
10:         SUBQ.W  #1,D1             ;      Convert to zero-relative
11:         MOVE.B  PenTbl(D1),D0     ;      Fetch pen byte
12:         EXT.W   D0                ;      Out to word
13:         MOVE.W  D0,PenType(A5)    ;      Set pen type
14:         MOVE.B  CurTbl(D1),D0     ;      Fetch cursor
15:         BEQ     SetCursor         ;      None, compute it
16:         MOVE.W  D0,Cursor(A5)     ;      Set cursor
17:         RTS                       ;      and return
18: PenTbl  DC.B    2                 ;      Lines
19:         DC.B    2                 ;      Freehand
20:         DC.B    0                 ;      Boxes
21:         DC.B    0                 ;      Ovals
22:         DC.B    0                 ;      Text
23:         DC.B    0                 ;      (Unused)
24:         DC.B    1                 ;      Eraser
25: CurTbl  DC.B    0                 ;      Lines
26:         DC.B    0                 ;      Freehand
27:         DC.B    Cross             ;      Boxes
28:         DC.B    Cross             ;      Ovals
29:         DC.B    IBeam             ;      Text
30:         DC.B    0                 ;      (Unused)
31:         DC.B    0                 ;      Eraser
32: ;
33: ;       Erase Screen routine.  Calls routine EScreen in
34: ;       MacDoodleDraw to erase window area in document.
35: ;
36: EraseA
37:         BSR     EScreen           ;      In draw code
38:         RTS                       ;      Back to event code
```

Listing 6.16 *– The Draw menu routines*

```
SetPen  MOVE.W  Pensize(A5),D2   ;      Fetch old item
        UNCHECK #mPen,D2         ;      Uncheck old item
        CHECK   #mPen,D1         ;      Check new item
        MOVE.W  D1,PenSize(A5)   ;      Store pen size
        MOVE.W  DrwType(A5),D0   ;      Fetch draw type
        SUBQ.W  #1,D0            ;      Convert to zero-relative
        MOVE.B  CurTbl(D0),D0    ;      Need new cursor?
        BEQ     SetCursor        ;      EQ => Yes, do it
        RTS                      ;      Otherwise, return
SetCursor                        ;      Here to compute cursor
        MOVE.W  PenType(A5),D0   ;      Fetch pen type
        LSL     #4,D0            ;      Shift left 4
        MOVE.W  PenSize(A5),D3   ;      D3 =
        SUBQ.W  #1,D3            ;              Adjusted pen size
        ADD.W   D3,D0            ;      Cursor resource ID
        MOVE.W  D0,Cursor(A5)    ;      Set cursor
        RTS                      ;      Back for more
```

Listing 6.17 *– The PenSize menu routine*

```
SetFont:                          ;     
        UNCHECK #mFont,FontSel(A5);     Uncheck old selection
        CHECK   #mFont,D1         ;     Check   new selection
        MOVE.W  D1,FontSel(A5)    ;     Remember selection #
        SUBQ.W  #1,D1             ;     Convert to font number
        ADD.W   D1,D1             ;     Double for word index
        MOVE.W  Fonts(D1),FontNum(A5);  Set new font number
        LEA     Points,A3         ;     A3 -> Table of point sizes
        MOVE.W  #SizeBase,D4      ;     D4 = 1st size selection
FindPoint                         ;     Loop to find point size
        CLR.W   -(SP)             ;     Room for result
        MOVE.W  FontNum(A5),-(SP);      Push font num
        MOVE.W  (A3),-(SP)        ;     and size
         _RealFont                ;     Is it
        TST     (SP)+             ;             real or is it...?
        BNE     @1                ;     NE => Real, continue
        STYLE   #mStyle,D4,#0     ;     Clear style word
        BRA     @2                ;     Branch to end of loop code
@1      STYLE   #mStyle,D4,#mOutline;   Set outline mode
@2      ADDQ.W  #1,D4             ;     Bump menu selection
        TST     (A3)+             ;     Bump pointer
        BNE     FindPoint         ;     NE => Keep looking
        RTS                       ;     If none real, leave alone
Fonts:  DC.W    Chicago,NewYork,Geneva,Monaco,Venice,London,Athens
        DC.W    SanFran,Toronto,Cairo,LosAng,Seattle,-1
```

Listing 6.18 *– Font menu routines* selection number for use in the next selection. Lines 5–7 convert the selection number to a font resource ID. Lines 10–22 change the type style of all the entries in the type size (lower) portion of the Style menu. Sizes for the selected font that do not require scaling have the outline attribute set. Sizes that require scaling are left as plain text. Lines 24–25 consist of a table of the font resource IDs in the same order as the menu. The list is terminated by a −1. This list must be close to the instruction in line 7, which uses the PC-relative-with-index addressing mode. The table must be within +126 to −128 bytes of the second word of that instruction, because the addressing mode has a one-byte displacement field.

Style The Text Style menu code appears in Listing 6.19. The Plain selection code starts at line 1. This code clears the TextFlg global variable, which contains the currently selected text attributes. For plain text, no bits are set. Lines 2–8 clear all check marks in the upper half of the menu. The ExpSel symbol is equated to the Expanded selection, which is the last attribute in the menu. Line 9 checks the Plain selection.

```
 1: Plain:  CLR.W    TextFlg(A5)     ;       Clear all attribute bits
 2:         MOVE.W   #mStyle,D2      ;       D2 = Menu number
 3:         MOVE.W   #BoldSel,D3     ;       D3 = First style selection
 4: PLoop:                           ;       Loop till done
 5:         UNCHECK  D2,D3           ;       Uncheck a bit
 6:         ADDQ.W   #1,D3           ;       Bump D3
 7:         CMP.W    #ExpSel,D3      ;       Last one?
 8:         BLE      PLoop           ;       No, continue
 9:         CHECK    #mStyle,#PlainSel;      Now check Plain
10:         RTS                      ;       and return
11: ;
12: ;       TStyle routine.  Add a style bit to TextFlg word.
13: ;
14: TStyle: UNCHECK  #mStyle,#PlainSel;      Uncheck plain
15:         CHECK    #mStyle,D1      ;       Check whatever
16:         SUB.W    #BoldSel,D1     ;       Subtract base
17:         BSET.B   D1,TextFlg+1(A5);       Set the bit in memory
18:         RTS                      ;       Done
19: ;
20: ;       Compress and Expand are different.  Treat them as
21: ;       mutually exclusive.
22: ;
23: TCmpress                         ;       Compressed selected
24:         BCLR.B   #Expand,TextFlg+1(A5);  Clear expanded bit
25:         UNCHECK  #mStyle,#ExpSel ;       Uncheck it
26:         BRA      TStyle          ;       Otherwise, it's the same
27: TExpand                          ;       Expanded selected
28:         BCLR.B   #Cmpress,TextFlg+1(A5); Clear Compressed bit
29:         UNCHECK  #mStyle,#CompSel;       Uncheck it
30:         BRA      TStyle          ;       Go select it
```

Listing 6.19 **–** *The Text Style menu code*

All attribute selections except Plain, Compressed, and Expanded jump to line 14. Line 14 eliminates the check next to the plain selection, and line 15 puts a check by the desired attribute. Lines 16–17 set the desired attribute bit in the TextFlg variable.

The Compressed and Expanded selections go to lines 23 and 27, respectively. The code for Compressed turns off Expanded and vice versa. After turning off the incompatible option, the routines branch to the TStyle label.

The lower part of the Style menu selects text size. Listing 6.20 contains this code.

Lines 1–2 move the check mark to the new selection. Line 3 stores the item number for use in the next selection. Lines 4–6 stores the point size equivalent to the item number in the global variable TextSiz. Lines 8–9 form a table of the point sizes in the same order as the menu selections. The table is terminated by a zero entry.

Draw Code

Code associated with drawing handles all functions associated with output to the screen and printer. This code also deals with data conversion, scrolling, and text entry.

Initialization

Listing 6.21 contains the DrawInit routine, which initializes graphics-related information. Lines 2–3 set the minimum value for the vertical and horizontal scroll bars. Line 4 initializes the SPoint variable. This point is the origin in the screen buffer for display in the window.

```
1: TSize:  UNCHECK #mStyle,TextSel(A5);    Uncheck previous
2:         CHECK   #mStyle,D1      ;       Check current
3:         MOVE.W  D1,TextSel(A5)  ;       Remember for next time
4:         SUB.W   #SizeBase,D1    ;       Subtract size base
5:         ADD.W   D1,D1           ;       Double for word index
6:         MOVE.W  Points(D1),TextSiz(A5); Remember size in points also
7:         RTS                     ;       Done
8: Points  DC.W    9,10,12,14,18,20,24,36,72; Point sizes
9:         DC.W    0               ;       End of table flag
```

Listing 6.20 *– The Text Size menu routine*

ReSize

The Open code in Listing 6.8 ends with a jump to the ReSize routine. Since the code in Listing 6.8 executes at initialization time, the ReSize routine is part of the startup process. This code appears in Listing 6.22.

The ReSize routine expects the new screen size in D1.L on entry. Line 1 saves the old screen size in D2.L. Lines 2–4 calculate the new horizontal maximum and lines 5–9 calculate the vertical maximum. In each case, the maximum value is the width of the document in pixels minus the screen size in pixels.

Lines 10–13 determine if the new size is the same as the old size. If so, the BEQ instruction at line 13 skips the Pack and UnPack operations (which are relatively slow). If the size is different, the RePack and UnBuf routines save the old screen data and restore the screen buffer to the new size. Lines 18–19 refresh the window if the size does not change.

```
DrawInit                              ;       Here to initialize
         SetVMin #0                   ;       Set vertical min to zero
         SETHMin #0                   ;       Set horizontal min to zero
         CLR.L   SPoint(A5)           ;       Init point to (0,0)
         RTS                          ;       Quit
```

Listing 6.21 *– The DrawInit routine*

```
 1: ReSize  MOVE.L  SSize(A5),D2      ;     Fetch old size
 2:         MOVE.W  #(SWidth*8)-1,D3;       Number of pixels in HScroll
 3:         SUB.W   D1,D3             ;     Subtract screen width
 4:         SetHMax D3                ;     Set new max
 5:         SWAP    D1                ;     Vertical to low word
 6:         SWAP    D2                ;       "      "  "   "
 7:         MOVE.W  #FLength-1,D3     ;     D3 = # pixels in VScroll
 8:         SUB.W   D1,D3             ;     Subtract screen length
 9:         SetVMax D3                ;     Set new max
10:         MOVE.W  D1,D4             ;     Save swapped version
11:         SWAP    D1                ;     Restore to original
12:         CMP.W   D2,D4             ;     Compare new to old
13:         BEQ     @1                ;     Same size, just redraw screen
14:         BSR     RePack            ;     Pack screen to one buffer
15:         MOVE.L  D1,SSize(A5)      ;     Set size value
16:         BSR     UnBuf             ;     Unpack screen buffer
17:         RTS                       ;     Done
18: @1      MOVE.L  D1,SSize(A5)      ;     Save new size
19:         BRA     ReDraw            ;     Go do redraw
```

Listing 6.22 *– The ReSize routine*

UpDate and ReDraw

MacDoodle responds to update events and other changes to the window by redrawing the entire window contents. The UpDate routine responds to an update event by calling ReDraw. Code for these routines appears in Listing 6.23.

Updating a window requires issuing a _BeginUpDate system call before the update, and an _EndUpDate call afterwards. Both system calls require the window address as a parameter. The UpDate routine calls ReDraw bracketed by a _BeginUpDate/EndUpDate pair in lines 2–6.

The ReDraw routine works by transferring the screen buffer to the window with the _CopyBits system call. Line 9 calls a routine that sets up the data areas for _CopyBits. Lines 10–17 invoke _CopyBits to transfer the screen buffer to the window GrafPort. Lines 18–19 unlock the buffer (which was locked by CopySetup).

The SaveScreen routine performs the opposite function and appears in Listing 6.24. This routine is the same as ReDraw, except for the reversal of the source and destination operands in lines 5–9.

The CopySetup routine constructs the parameters for the _Copybits call. This code appears in Listing 6.25.

Lines 5–8 construct a rectangle for the window (on the screen) in a temporary variable called TmpRect. This rectangle starts at zero and ends at the screen size. Variable SSize contains the screen size. Lines 14–20

```
 1: UpDate:                             ;
 2:          MOVE.L  WPointer(A5),-(SP);    Push window pointer
 3:          _BeginUpdate               ;    Signal update begin
 4:          BSR     ReDraw             ;    Refresh screen
 5:          MOVE.L  WPointer(A5),-(SP);    Push window pointer
 6:          _EndUpdate                 ;    Signal update end
 7:          RTS                        ;    Back to event handler
 8: ReDraw   PUSHALL                    ;    Save registers
 9:          BSR     CopySetup          ;    Set up for CopyBits
10:          MOVE.L  PortAdr(A5),A0     ;    A0 -> Window's GrafPort
11:          PEA     SBitMap(A5)        ;    Source bitmap address
12:          PEA     PortBits(A0)       ;    Destination bitmap address
13:          PEA     SrcRect(A5)        ;    Source rectangle address
14:          PEA     TmpRect(A5)        ;    Destination rectangle address
15:          MOVE.W  #SrcCopy,-(SP)     ;    Blatant copy mode
16:          PEA     0                  ;    No mask region
17:          _CopyBits                  ;    Redraw entire window
18:          MOVE.L  ScrBuf(A5),A0      ;    A0 = Screen buffer handle
19:          _HUnLock                   ;    Unlock it
20:          POPALL                     ;    Restore registers
21:          RTS                        ;    Return to caller
```

Listing 6.23 *– The UpDate and ReDraw routines*

```
SaveScreen                      ;
        PUSHALL                 ;       Save caller's registers
        BSR     CopySetup       ;       Set up for CopyBits
        MOVE.L  PortAdr(A5),A0  ;       A0 -> Window's GrafPort
        PEA     PortBits(A0)    ;       Source bitmap address
        PEA     SBitMap(A5)     ;       Destination bitmap address
        PEA     TmpRect(A5)     ;       Source rectangle address
        PEA     SrcRect(A5)     ;       Destination rectangle address
        MOVE.W  #SrcCopy,-(SP)  ;       Blatant copy mode
        PEA     0               ;       No mask region
        _CopyBits               ;       Redraw entire window
        MOVE.L  ScrBuf(A5),A0   ;       A0 = Screen buffer handle
        _HUnLock                ;       Unlock it
        POPALL                  ;       Restore registers
        RTS                     ;       Return to caller
```

Listing 6.24 *– The SaveScreen routine*

```
CopySetup                       ;
;
;       Compute Rectangle for screen
;
        MOVE.L  SSize(A5),D0    ;       Pick up screen size
        MOVE.L  SPoint(A5),D1   ;       And screen point
        CLR.L   TmpRect(A5)     ;       Window rectangle
        MOVE.L  D0,TmpRect+4(A5);               is (0,SSize)
;
;       (A bit harder) Compute rectangle for buffer.  The origin is
;       "SFPoint(A5)", and the limit is SFPoint(A5) + SSize(A5).  (And you
;       must add the two halves separately.)
;
        MOVE.L  D1,SrcRect(A5)  ;       Buffer rectangle
        ADD.W   D0,D1           ;       is (SPoint,SPoint+SSize)
        SWAP    D1              ;       Now add high
        SWAP    D0              ;                       order
        ADD.W   D0,D1           ;                               words
        SWAP    D1              ;       Back to correct order
        MOVE.L  D1,SrcRect+4(A5);       Store in rectangle structure
;
;       Now formulate the buffer bitmap structure.
;
        MOVE.L  ScrBuf(A5),A0   ;       A0 = Screen buffer handle
        _HLock                  ;       Lock it
        MOVE.L  (A0),A3         ;       A3 -> Screen buffer
        MOVE.L  A3,SBitMap(A5)  ;       Set bitmap address
        MOVE.W  #SWidth,SRBytes(A5);    Set rowBytes for screen buffer
        CLR.L   SBounds(A5)     ;       Origin is (0,0)
        MOVE.W  #SLength,Sbounds+4(A5);         Limit is (SLength,
        MOVE.W  #SWidth*8,SBounds+6(A5);                SWidth*8)
        PEA     PortAdr(A5)     ;       Location for return value
        _GetPort                ;       Fetch the port address
        RTS                     ;       Return
```

Listing 6.25 *– The CopySetup routine* construct a similar rectangle for the screen buffer. The rectangle is stored in a memory area called SrcRect. Computing this rectangle is a bit harder. The origin is contained in variable SFPoint, and the end is the sum of SFPoint and SSize. Lines 24–25 lock the screen buffer in memory. Lines 26–31 construct a bitmap data structure for the screen buffer. Lines 32–33 fetch the window GrafPort address using a _GetPort call.

EScreen (Erase Screen)

Listing 6.26 contains the EScreen routine, which erases the screen. Areas of the document outside the screen region are unaffected.

Lines 2 and 9 save and restore the registers on the stack. The PackU routine called at line 3 saves the screen buffer contents in the undo buffer. The data is saved in packed format. Choosing Undo from the Edit menu redisplays this information. Lines 4–7 erase the window on the screen, and line 8 calls the SaveScreen routine to copy the new screen to the screen buffer.

PrDraw (Print Document)

Listing 6.27 contains the PrDraw routine, which prints the entire document. The menu code in Listing 6.13 calls this routine after opening the printer driver.

Lines 2–6 reset the printer before starting to print. The PrCtlCall routine is contained in the PrLink.rel file supplied with the Macintosh Development System. Line 7 copies the current screen position within the document to register D7. Register D6 contains the screen position currently being printed. Line 8 initializes D6 to zero.

Lines 9–11 position the screen at the next position to be printed. Routine VScroll is the thumb positioning routine. To move to a place within the document, you load the row number into D1 and call VScroll. On

```
 1: EScreen                          ;      Called from menu code
 2:         PUSHALL                  ;      Save registers
 3:         JSR     PackU            ;      Save screen in undo buffer
 4:         CLR.L   TmpRect(A5)      ;      Clear origin
 5:         MOVE.L  SSize(A5),TmpRect+4(A5); Set screen size
 6:         PEA     TmpRect(A5)      ;      Push rectangle address
 7:         _EraseRect               ;      Erase it
 8:         BSR     SaveScreen       ;      Now update screen buffer
 9:         POPALL                   ;      Pop registers
10:         RTS                      ;      Return to menu code
```

***Listing 6.26** – The EScreen routine*

```
PrDraw  PUSHALL                    ;       Save the registers
        MOVE.W  #PrDCtl,-(SP)      ;       Printer device control code
        MOVE.L  #PrReset,-(SP)     ;       Code for printer reset
        CLR.L   -(SP)              ;       Two null
        CLR.L   -(SP)              ;                 parameters
        JSR     PrCtlCall          ;       Do a reset before starting to print
        MOVE.W  SRow(A5),D7        ;       Remember current screen position
        CLR.L   D6                 ;       D6 counts rows
@1      MOVE.L  D6,D0              ;       Set D0 for scroll operation
        SWAP    D0                 ;       to high word for VScroll
        BSR     VScroll            ;       Do the scroll
        MOVE.W  D6,D5              ;       Copy row number
        SUB.W   Srow(A5),D5        ;       Normalize to displayed screen
        MOVE.W  D5,TmpRect(A5)     ;       Set
        CLR.W   TmpRect+2(A5)      ;           up
        MOVE.W  SSize(A5),TmpRect+4(A5);          destination
        MOVE.W  #SWidth*8,TmpRect+6(A5);                         rectangle
        MOVE.L  ScrBuf(A5),A0      ;       Fetch
        _Hlock                     ;              screen
        MOVE.L  (A0),A3            ;                      address (locked)
        MOVE.L  A3,SBitMap(A5)     ;       Store in bitmap structure
        MOVE.W  #SWidth,SRBytes(A5);       Set row bytes
        CLR.W   SBounds(A5)        ;       Origin is
        CLR.W   SBounds+2(A5)      ;                         (0,0)
        MOVE.W  SSize(A5),SBounds+4(A5); Destination
        SUB.W   D5,SBounds+4(A5);                      is
        MOVE.W  #SWidth*8,SBounds+6(A5);                  (SSize-D5.W,576)
        MOVE.W  #PrBits,-(SP)      ;       Specify bitmap print
        PEA     SBitMap(A5)        ;       Push source bitmap address
        PEA     TmpRect(A5)        ;       Destination rectangle address
        MOVE.L  #PrPaint,-(SP)     ;       Specify square resolution
        JSR     PrCtlCall
        MOVE.L  ScrBuf(A5),A0      ;       Unlock
        _HUnLock                   ;              screen buffer
        ADD.W   SSize(A5),D6       ;       Bump row number
        CMP.L   #Flength,D6        ;       See if done
        BLT     @1                 ;       If less, not done
        MOVE.W  D7,D0              ;       Restore
        SWAP    D0                 ;              vertical
        JSR     VScroll            ;                        position
        MOVE.W  #PrDCtl,-(SP)      ;       Device control code
        MOVE.L  #PrPage,-(SP)      ;       Indicate end of page
        CLR.L   -(SP)              ;       Two
        CLR.L   -(SP)              ;              nulls
        JSR     PrCtlCall          ;       Eject page
        POPALL                     ;       Page is printed
        RTS                        ;       Done
```

Listing 6.27 *– The PrDraw routine* return from VScroll, the screen buffer, display, and thumb position in the scroll bar reflect the new document position.

Lines 12–17 set up a rectangle for the printer. This rectangle is the same height as the window and 576 pixels wide. Lines 19–20 lock the screen buffer in memory. Lines 20–27 set up a bitmap structure for the screen buffer. Lines 29–32 call the PrCtlCall routine to transfer the bitmap to the printer.

This bitmap printing technique is used only for preconstructed images, such as MacPaint documents or screen dumps. The normal technique is to open a special GrafPort that references the printer. QuickDraw operations on this GrafPort cause output to go to the printer. This technique produces better images on devices that do not match the Macintosh screen resolution, such as a LaserWriter. The Macintosh Development System includes a program that illustrates this method of printing.

Lines 33–34 unlock the screen buffer. Lines 35–37 form the end of the printing loop. When the loop completes, lines 38–40 restore the screen's vertical position to the value saved in line 7. Lines 41–45 call the PrCtlCall routine to eject the page.

Line and Page Scrolling

Listing 6.28 contains the routines that handle line and page scrolling events.

Routine DoScroll receives control from the event code in Listing 6.1 for a scrolling event. This routine saves the scrolling type value in variable ScrType at line 3. A TRACK macro causes routine XScroll to execute repeatedly until the mouse button is released.

Parameters to the XScroll routine are shown in Figure 6.21. Lines 8–11 pop the part code and control handle off the stack, leaving only the return address. If the part code is zero, the routine simply returns to the caller at line 14.

Line 16 places the original scroll type word (from the scroll event) in D2.W. The next section of code places the number of pixels to scroll in D3.W, the page size in D4.W, and the starting position in D5.W. Lines 17–18 assume a vertical line scroll. Lines 20–23 reload the registers if the desired scroll is horizontal. Lines 24–26 replace the contents of D3 with the correct number of pixels for a page scroll. Lines 27–29 negate D3 if the scrolling direction is up or right. Line 30 converts the number of pixels to scroll in D3 to an absolute pixel number. This quantity indicates the pixel to be displayed in the upper-left corner of the window (the GrafPort origin). Lines 31–35 simulate a thumb scroll event with this absolute value.

Thumb Scrolling

All scrolling in MacDoodle is handled by the two routines in Listing 6.29. These routines are invoked directly by thumb scroll events and indirectly by the code in Listing 6.28.

The VScroll routine handles a vertical thumb scroll. The pixel number of the document to appear at the top of the window is in the high word of D0 on entry. Lines 3–6 clear D0 if it contains a negative value. Lines 7–11 change any values that are too large to the maximum permissible value. This range checking is necessary because the line scroll routines can ask for values that are out of range. For example, scrolling up past the top of the document generates a negative value for the pixel to display.

Lines 11–15 perform the scrolling operation. The RePack routine called at line 12 converts the entire document to packed form at the end of the

```
DoScroll                          ;
         SWAP     D0              ;      Type to low word
         MOVE.W   D0,ScrType(A5)  ;      Remember type
         TRACK    D1,XScroll      ;      Do the scroll
         RTS                      ;      Return

XScroll                           ;
         MOVE.L   (SP)+,A0        ;      Fetch return address
         MOVE.W   (SP)+,D1        ;      and part code
         MOVE.L   (SP)+,D0        ;      and control handle
         MOVE.L   A0,-(SP)        ;      Restack return address
         TST.W    D1              ;      Still in right spot?
         BNE      @1              ;      Yes, continue
         RTS                      ;      No, quit now
@1       PUSHALL                  ;      Save registers
         MOVE.W   ScrType(A5),D2  ;      Fetch scroll type
         MOVE.W   #LScroll,D3     ;      Move pixel count to D3
         MOVE.W   SSize(A5),D4    ;      D4 = Vertical page size
         MOVE.W   SRow(A5),D5     ;      D5 = Vertical position of (0,0)
         BTST     #2,D2           ;      Vertical or horizontal?
         BEQ      @2              ;      EQ => Vertical
         MOVE.W   SSize+2(A5),D4  ;      Correct page size
         MOVE.W   SPoint+2(A5),D5 ;      D5 = Horizontal position
@2       BTST     #1,D2           ;      Page or line scroll?
         BEQ      @3              ;      EQ => Line
         MOVE.W   D4,D3           ;      Correct pixel count for page
@3       BTST     #0,D2           ;      Left (up) or right (down)?
         BNE      @4              ;      EQ => Right (down)
         NEG.W    D3              ;      Negate # pixels
@4       ADD.W    D5,D3           ;      Make absolute
         MOVE.W   D3,D0           ;      Move to
         SWAP     D0              ;              D0 high word
         BTST     #2,D2           ;      Vertical or horizontal?
         BNE      xHScroll        ;      NE => Horizontal
         BRA      xVScroll        ;      Else, go do vertical
```

***Listing 6.28** – The line and page scrolling routines* file buffer. Line 13 sets the new value for the first row of the document to be displayed. Line 14 sets the new value of the scroll bar thumb. The UnBuf routine called at line 15 splits the single buffer into three: the top area of the file buffer, the screen buffer, and the bottom area of the file buffer. The UnBuf routine calls ReDraw to display the correct area of the document in the window.

Horizontal scrolling is simpler. Lines 23–31 perform the proper range checking, correcting out of range values. Line 32 sets the new thumb value, and line 33 sets the new display point in the SPoint variable. The branch to ReDraw at line 35 updates the screen. Since the screen buffer contains complete horizontal rows, none of the buffers is modified by horizontal scrolling.

```
 1: VScroll PUSHALL                  ;       Save registers
 2: xVScroll                         ;       Here to skip PUSHALL
 3:         SWAP    D0               ;       Value to low half
 4:         TST.W   D0               ;       Value positive?
 5:         BGE     @1               ;       Yes, it's OK
 6:         CLR.W   D0               ;       No, force it
 7: @1      MOVE.W  #FLength-1,D1    ;       D1 =
 8:         SUB.W   SSize(A5),D1     ;               Maximum size
 9:         CMP.W   D0,D1            ;       Check for > max
10:         BGE     @2               ;       OK, leave it
11:         MOVE.W  D1,D0            ;       Too big, force to max
12: @2      BSR     RePack           ;       Force back to single image
13:         MOVE.W  D0,SRow(A5)      ;       Set row number
14:         SetVVal D0               ;       Move the thumb
15:         BSR     UnBuf            ;       Unpack the proper amount of screen buffer
16:         POPALL                   ;       Restore registers
17:         RTS                      ;       Return to caller
18: ;
19: ;       Equivalent routine for horizontal scroll
20: ;
21: HScroll PUSHALL                  ;       Save registers
22: xHScroll                         ;       Here to skip PUSHALL
23:         SWAP    D0               ;       Value to low-order word
24:         TST.W   D0               ;       Value OK?
25:         BGE     @1               ;       Yes, must be > 0
26:         CLR.W   D0               ;       No, make minimum
27: @1      MOVE.W  #SWidth*8,D1     ;       Fetch screen width
28:         SUB.W   SSize+2(A5),D1   ;       Calculate largest value
29:         CMP.W   D0,D1            ;       Check it
30:         BGE     @2               ;       GE means value in range
31:         MOVE.W  D1,D0            ;       Else make maximum value
32: @2      SetHVal D0               ;       Set the value
33:         MOVE.W  D0,SPoint+2(A5)  ;       Set for redraw
34:         POPALL                   ;       Pop registers
35:         BRA     ReDraw           ;       Refresh screen
```

Listing 6.29 *– The thumb scroll routines*

Mouse Down

Listing 6.30 contains the code that handles mouse-down events. Action taken by this code depends on the setting of the DrwType global variable, which contains the last Draw menu selection number.

Lines 2–4 close out any open text entry. Line 5 saves the mouse point in a temporary variable. Line 6 saves the screen contents in the undo buffer. Lines 7–8 save the graphics pen state, and lines 9–12 set the proper pen size. Lines 13–14 move the pen to the mouse point. Lines 15–19 call the proper routine from the table at line 32. Routines in this table correspond to the selections in the Draw menu. Note that the Erase Screen selection is handled by the menu code in Listing 6.16. Lines 20–21 restore the pen state. Line 22 sets the change indicator flag.

Lines 23–25 save the updated screen in the screen buffer if the Draw menu selection was other than Text. Text entry starts with a mouse-down

```
 1: Mouse:                             ;
 2:         TstFlag F_Text             ;         Text open?
 3:         BEQ     @1                 ;         No, continue
 4:         JSR     CloseText          ;         Yes, close first
 5: @1      MOVE.L  D1,Point1(A5)      ;         Save original point
 6:         BSR     PackU              ;         Save screen in undo buffer
 7:         PEA     OldPen(A5)         ;         Save
 8:         _GetPenState               ;                 pen state
 9:         MOVE.W  PenSize(A5),D1     ;         Fetch pen size
10:         MOVE.W  D1,-(SP)           ;         Push
11:         MOVE.W  D1,-(SP)           ;                 2 copies
12:         _PenSize                   ;         Set new pen size
13:         MOVE.L  Point1(A5),-(SP);            Push mouse point
14:         _MoveTo                    ;         Move pen to mouse point
15:         MOVE.W  DrwType(A5),D0     ;         Fetch draw routine index
16:         SUBQ.W  #1,D0              ;         Force zero relative
17:         LSL.W   #1,D0              ;         Multipy x 2 for word index
18:         MOVE.W  JBASE(D0.W),D0     ;         Fetch offset
19:         JSR     JBASE(D0.W)        ;         Call proper routine
20:         PEA     OldPen(A5)         ;         Now
21:         _SetPenState               ;                 restore pen
22:         SetFlag F_Change           ;         Mark a change to file
23:         CMPI.W  #5,DrwType(A5)     ;         Just start text?
24:         BEQ     @2                 ;         Yes, don't save screen
25:         JSR     SaveScreen         ;         Save screen in screen buffer
26: @2      RTS                        ;         Back for next event
27: Void:   StopAlert #ProgramErr      ;         Fatal error!
28:         EXIT                       ;         Back to Finder
29: ;
30: ;       Jump table for draw routines
31: ;
32: JBASE   JTAB    Lines,  FreeH,  Boxes,  Ovals
33:         JTAB    Text,   Void,   Eraser, Void
```

Listing 6.30 *– The Mouse routine* event and terminates with the first event that is not a key-down event. The screen buffer is not updated until the text entry terminates.

Lines

Listing 6.31 contains the code that draws straight lines on the screen. Lines 2–3 set the pen mode to XOR. Lines 4–5 move the original mouse point to D3.L and D4.L. Lines 6–9 fetch the current mouse position and compare it to the last position. If the mouse position has changed, lines 10–12 erase the previous line and draw a new line from the original mouse point to the current mouse point. Lines 13–16 check to see if the mouse button is still down. If so, the routine loops back to continue tracking the mouse position. If not, the code at lines 17–19 redraws the line in copy mode.

The subroutine in lines 25–30 draws a line from the point in D4.L to the point in D3.L. Lines 26–27 move the pen to the original mouse point. Lines 28–29 draw a line to the current mouse point.

```
 1: Lines:                                 ;        Line handler
 2:          MOVE.W  #PatXor,-(SP)         ;        Set pen
 3:          _PenMode                      ;                to XOR for dragging
 4:          MOVE.L  Pointl(A5),D3         ;        Save mouse point
 5:          MOVE.L  D3,D4                 ;        Need original point
 6: @1:      PEA     Pointl(A5)            ;        Temp point
 7:          _GetMouse                     ;        Fetch mouse in local coords
 8:          CMP.L   Pointl(A5),D3         ;        Check against old
 9:          BEQ     @2                    ;        Equal, don't update
10:          BSR     DrawLine              ;        Cancel old line
11:          MOVE.L  Pointl(A5),D3         ;        Save new point
12:          BSR     DrawLine              ;        Draw new line
13: @2:      CLR.W   -(SP)                 ;        Room for result
14:          _StillDown                    ;        Mouse still down?
15:          TST.B   (SP)+                 ;        If NE,
16:          BNE     @1                    ;                yes, still down
17:          MOVE.W  #PatCopy,-(SP)        ;        Change pen
18:          _PenMode                      ;                    to copy mode
19:          BSR     DrawLine              ;        Draw final line
20:          RTS                           ;        Return to caller
21: ;
22: ;        DrawLine routine.  Draws a line from point in D4.L
23: ;        to point in D3.L.
24: ;
25: DrawLine                               ;        Draw a line
26:          MOVE.L  D4,-(SP)              ;        Reposition
27:          _Moveto                       ;                    pen
28:          MOVE.L  D3,-(SP)              ;        Cancel old line
29:          _Lineto                       ;        thusly
30:          RTS                           ;        Return to caller
```

Listing 6.31 – The Line Drawing routine

Eraser and Freehand

Listing 6.32 contains the code that handles eraser and freehand drawing. Erasing is the same as freehand drawing, except that the pen mode is patBic instead of patOr. You can think of the eraser as freehand drawing with white pixels instead of black pixels.

Lines 2–3 set the pen mode to patBic for erasing. Line 6 saves the previous mouse point in D3.L. Lines 7–8 fetch the current mouse point. Lines 9–10 compare the previous and current mouse points to see if the mouse has moved. If so, lines 11–13 draw a line from the previous mouse point to the current mouse point. Lines 14–17 check to see if the mouse button is still down. If so, the routine continues to track the mouse. Line 18 returns to the caller when the mouse button is released.

Boxes and Ovals

Listing 6.33 contains the code that draws boxes and ovals. Register D7 is used to choose between boxes and ovals throughout this code. Line 2 clears D7 for boxes, and line 4 sets D7 to −1 for ovals.

This routine uses three rectangles: two for comparison purposes and a temporary rectangle for drawing. These rectangles are called NewRect, OldRect, and TmpRect. Lines 7–9 load the rectangle addresses into A3, A2, and A4. Lines 10–11 initialize NewRect and OldRect to empty rectangles at the original mouse point. Lines 12–13 set the pen mode to patXor.

Lines 14–34 form a loop that tracks the mouse and redraws the figure each time the mouse moves. Lines 15–18 test to see if the mouse button is

```
 1: Eraser:                              ;
 2:          MOVE.W  #patBic,-(SP)       ;      Push desired mode
 3:          _PenMode                    ;      Set the pen mode
 4: ;
 5: FreeH:                               ;      Sketch handler
 6:          MOVE.L  Pointl(A5),D3       ;      Save mouse point
 7: @1:      PEA     Pointl(A5)          ;      Temp point
 8:          _GetMouse                   ;      Fetch mouse in local coords
 9:          CMP.L   Pointl(A5),D3       ;      Check against old
10:          BEQ     @2                  ;      Equal, don't update
11:          MOVE.L  Pointl(A5),D3       ;      Save new point
12:          MOVE.L  D3,-(SP)            ;      Push point
13:          _Lineto                     ;      Draw a line to new point
14: @2:      CLR.W   -(SP)               ;      Room for result
15:          _StillDown                  ;      Mouse still down?
16:          TST.B   (SP)+               ;      If NE,
17:          BNE     @1                  ;           yes, still down
18:          RTS                         ;      Return to caller
```

Listing 6.32 – *The Eraser and FreeH routines*

```
Boxes
        CLR.L    D7                ;    D7 is flag
        BRA      DoRect            ;    Now draw object
Ovals   MOVEQ.L  #-1,D7            ;    Set all bits of D7
DoRect  MOVE.L   Point1(A5),D1     ;    Fetch mouse point
        MOVE.L   D1,D2             ;    Copy mouse point
        LEA      NewRect(A5),A3    ;    A3 -> New rectangle
        LEA      OldRect(A5),A2    ;    A2 -> Old rectangle
        LEA      TmpRect(A5),A4    ;    A4 -> Temporary rectangle
        MOVEM.L  D1/D2,(A2)        ;    Set old rect
        MOVEM.L  D1/D2,(A3)        ;    and new rect to mouse point
        MOVE.W   #patXor,-(SP)     ;    Set pen
        _PenMode                   ;             to Xor mode
RectLoop                           ;    Here while mouse down
        CLR.W    -(SP)             ;    Room for result
        _StillDown                 ;    See if
        TST.B    (SP)+             ;             still
        BEQ      DoFinal           ;                       down (EQ => No)
        PEA      BotRight(A3)      ;    Get
        _GetMouse                  ;             mouse coords
        CLR.W    -(SP)             ;    Space for result
        MOVEM.L  A2/A3,-(SP)       ;    Push A2 and A3
        _EqualRect                 ;    Has mouse
        TST.B    (SP)+             ;                       moved?
        BNE      RectLoop          ;    No, continue polling
        MOVE.L   (A2),(A4)         ;    Set temp rect
        MOVE.L   4(A2),4(A4)       ;                       to old rect
        JSR      DoDraw            ;    XOR to restore screen contents
        MOVE.L   (A3),(A4)         ;    Set temp rect
        MOVE.L   4(A3),4(A4)       ;                       to new rect
        JSR      DoDraw            ;    Draw new object
        MOVE.L   (A3),(A2)         ;    Copy new rect
        MOVE.L   4(A3),4(A2)       ;                       to old rect
        BRA      RectLoop          ;    Repeat till mouse up
DoFinal                            ;    Here to make final
        MOVE.W   #patCopy,-(SP)    ;    Set pen to copy mode
        _PenMode                   ;    thusly
;
DoDraw:                            ;    FrameRect(Oval) at (A4)
        MOVE.W   Bottom(A4),D0     ;    Fetch bottom word
        CMP.W    Top(A4),D0        ;    Compare with top
        BGE      @1                ;    If GE, don't swap
        MOVE.W   Top(A4),Bottom(A4);    Swap
        MOVE.W   D0,Top(A4)        ;             v coordinates
@1:                                ;    Here to compare h coordinates
        MOVE.W   Right(A4),D0      ;    Fetch right coord
        CMP.W    Left(A4),D0       ;    Compare with left
        BGE      @2                ;    If GE, don't swap
        MOVE.W   Left(A4),Right(A4);    Swap
        MOVE.W   D0,Left(A4)       ;             h coordinates
@2:     MOVE.L   A4,-(SP)          ;    Push rectangle address
        TST.W    D7                ;    Boxes?
        BNE      @3                ;    IF NE, then doing ovals
        _FrameRect                 ;    Draw rectangle
        RTS                        ;    Return
@3:                                ;    Here if ovals
        _FrameOval                 ;    Do the oval instead
        RTS                        ;    Return
```

***Listing 6.33** – The Boxes and Ovals routines* still down. If not, the routine draws the final figure using the patCopy pen mode. Lines 19–20 fetch the mouse point into the bottom-right point of NewRect. Lines 21–25 compare the resulting rectangle to OldRect. If the two are equal, the branch at line 25 restarts the loop. Otherwise, lines 26–28 copy OldRect to the scratch area pointed to by A4, and routine DoDraw erases the old figure. Then lines 29–31 draw the new figure. Lines 32–34 copy OldRect to NewRect and repeat the loop.

Lines 36–37 set the pen mode to patCopy. The code then drops into routine DoDraw to draw the figure a last time in copy mode. Lines 40–50 adjust the temporary rectangle so that the top coordinate is less than the bottom and the left coordinate is less than the right. Lines 51–57 issue either a _FrameRect or a _FrameOval call to draw the figure on the screen.

Text Initialization

Listing 6.34 contains the code executed when a mouse-down event occurs with Text selected.

Lines 2–9 set the pen mode, font, text attributes, and text size. The mode is patXor to allow a backspace character to erase the last character typed. Lines 10–12 initialize a buffer that will contain characters typed by the user. Line 13 sets the flag indicating that text entry is in progress. Each routine that processes an event other than a keystroke will call the CloseText routine to close out the text entry. This structure is necessary if the program is to have a single event loop. Alternatively, the text code could use a separate event loop and branch back to the main event loop when the text entry is closed. The variable Point2 contains the point where the next character will be drawn. Line 14 copies the mouse point to this location. Line 15 branches to the routine that draws the cursor.

```
 1: Text                                 ;
 2:         MOVE.W  #patXor,-(SP)        ;     Set text
 3:         _TextMode                    ;               mode to XOR
 4:         MOVE.W  FontNum(A5),-(SP);         Set
 5:         _TextFont                    ;              proper font
 6:         MOVE.W  TextFlg(A5),-(SP);         Set text
 7:         _TextFace                    ;                attributes
 8:         MOVE.W  TextSiz(A5),-(SP);         Set text
 9:         _TextSize                    ;                size
10:         CLR.W   BuffCnt(A5)          ;     Clear text buffer count
11:         LEA     TBuffer(A5),A0       ;     A0 -> Buffer
12:         MOVE.L  A0,BuffPtr(A5)       ;     Save it
13:         SetFlag F_Text               ;     Remember we're doing text
14:         MOVE.L  Point1(A5),Point2(A5);     Set pen location
15:         BRA     DrawCurs             ;     Draw cursor
```

Listing 6.34 *– The text initialization routine*

Key Down

The code that handles key-down and autokey events appears in Listing 6.35. Lines 2–5 reject keystroke events before the mouse down event has selected a starting point. Line 7 sets the buffer change flag. Lines 8–11 single out backspace (decimal 8) and carriage return (decimal 13) characters for special processing. Any other character is placed in the buffer and on the screen. Lines 12–15 place the character in the temporary buffer. Line 16 erases the previous cursor by drawing it again. Lines 17–20 draw the character, and lines 21–22 get the point at which to draw the next character. Line 23 branches to the routine that draws the cursor. This draws the cursor at the location of the next character.

Lines 24–43 handle backspacing. Lines 25–26 reject a backspace character if there are no characters in the buffer. Line 27 erases the previous cursor. Lines 28–29 decrement the count of characters in the buffer. Lines 30–32 fetch the last character from the buffer and decrement the address at which to store the next character. Lines 33–36 call the _CharWidth system trap to get the width of the character being erased. Lines 37–39 move the pen to the beginning of this character. Lines 40–41 erase the character, and line 42 redraws the cursor at the location of the erased character.

CloseText

Listing 6.36 contains the code that finishes text entry. Line 4 erases the cursor character. Lines 5–6 set up the count byte for the text buffer. The label TBufCnt appears directly in front of the text buffer. Setting this byte to the number of characters in the buffer yields a Pascal-format string. Lines 7–8 change the text mode to patCopy, and lines 9–12 redraw the buffer contents in copy mode using the _DrawString call. Note that scaled fonts may be redrawn slightly differently with _DrawString than with successive _DrawChar calls. The text may appear slightly longer on the screen after it is redrawn. Line 13 clears the text entry flag. Line 14 transfers the window on the screen to the screen buffer. Choosing Undo will cause the entire line of text to disappear.

DrawCurs

Code for drawing the cursor appears in Listing 6.37. The cursor used in this program is a simple underline character, not the flashing cursor used by most Macintosh programs. The underline character places no timing requirements on the program event code, making the code smaller and easier to understand.

Lines 2–3 position the pen to the origin of the cursor character. Lines 4–5 actually draw the cursor.

UnPack and Pack

The code that translates compressed data into a form suitable for display appears in Listing 6.38. This routine expects its input parameters in the following registers:

Register	**Parameter**
A0.L	Address of data in packed format
A1.L	Address of buffer to receive unpacked data

```
KeyDown                                 ;
         TstFlag F_Text                 ;       Doing text?
         BNE     Drawtext               ;       Yes, continue
         StopAlert #NotText             ;       No, give error
         RTS                            ;       and return
DrawText                                ;       Here to draw a character
         SetFlag F_Change               ;       Changed the buffer
         CMP.B   #BS,D1                 ;       A backspace?
         BEQ     IsBS                   ;       Yes, special
         CMP.B   #CR,D1                 ;       Carriage return?
         BEQ     IsCR                   ;       Yes, also special
         MOVE.L  BuffPtr(A5),A0         ;       Fetch buffer pointer
         ADD.W   #1,BuffCnt(A5)         ;       Increment character count
         MOVE.B  D1,(A0)+               ;       Save character
         MOVE.L  A0,BuffPtr(A5)         ;       Remember address
         BSR     DrawCurs               ;       Erase cursor
         MOVE.L  Point2(A5),-(SP);      ;       Reset
         _MoveTo                        ;               pen location
         MOVE.W  D1,-(SP)               ;       Draw
         _DrawChar                      ;               character
         PEA     Point2(A5)             ;       Get
         _GetPen                        ;               pen location
         BRA     DrawCurs               ;       Draw the cursor
IsBS:                                   ;       Here if a backspace
         MOVE.W  BuffCnt(A5),D1         ;       Any characters in buffer?
         BEQ     CantBS                 ;       No, can't backspace
         BSR     DrawCurs               ;       Draw (erase) cursor
         SUBQ.W  #1,D1                  ;       Subtract 1 from count
         MOVE.W  D1,BuffCnt(A5)         ;       Store updated count
         MOVE.L  BuffPtr(A5),A0         ;       A0 -> Free area
         MOVE.B  -(A0),D3               ;       Character to D3
         MOVE.L  A0,BuffPtr(A5)         ;       Store pointer
         CLR.W   -(SP)                  ;       Room for result
         MOVE.W  D3,-(SP)               ;       Find out how
         _CharWidth                     ;             big the character is
         MOVE.W  (SP)+,D4               ;       Result to D4
         SUB.W   D4,Point2+2(A5)        ;       Decrement v coord by that much
         MOVE.L  Point2(A5),-(SP);      ;       Back up
         _MoveTo                        ;               cursor
         MOVE.W  D3,-(SP)               ;       Redraw (erase)
         _DrawChar                      ;                       character
         BSR     DrawCurs               ;       Draw cursor
CantBS   RTS                            ;       Done
```

***Listing 6.35** – The KeyDown routine*

```
IsCR:                                ;     Also carriage return
CloseText:                           ;
        MOVEM.L D0-D7/A0-A6,-(SP);         Save registers
        BSR     DrawCurs             ;     Erase cursor character
        MOVE.W  BuffCnt(A5),D3       ;     Get character count
        MOVE.B  D3,TBufCnt(A5)       ;     Set byte
        MOVE.W  #patCopy,-(SP)       ;     Set text
        _TextMode                    ;             to copy mode
        MOVE.L  Point1(A5),-(SP);          Go back
        _MoveTo                      ;             to start position
        PEA     TBufCnt(A5)          ;     Push string address
        _DrawString                  ;     Redraw in copy mode
        ClrFlag F_Text               ;     No longer in text mode
        BSR     SaveScreen           ;     Save screen in buffer now
        MOVEM.L (SP)+,D0-D7/A0-A6;         Restore registers
        RTS                          ;     Done
```

Listing 6.36 *– The CloseText routine*

```
DrawCurs:                            ;
        MOVE.L  Point2(A5),-(SP);          Reset
        _MoveTo                      ;             pen location
        MOVE.W  #CursChar,-(SP)      ;     Delete
        _DrawChar                    ;             cursor
        RTS                          ;     Return to caller
```

Listing 6.37 *– The DrawCurs routine*

```
UnPack  MOVEM.L D0-D3/A2/A6,-(SP);         Stack registers
        SUBQ.W  #1,D0                ;     Adjust # rows for DBRA
        SUBA.L  #SWidth,A6           ;     Adjust limit register
        MOVE.W  D0,D3                ;     Place in nonvolatile register
        MOVE.L  A0,Addr1(A5)         ;     Store
        MOVE.L  A1,Addr2(A5)         ;             addresses
@1      CMPA.L  Addr2(A5),A6         ;     Check limit
        BLS     PMemErr              ;     Too high, quit
        PEA     Addr1(A5)            ;     Push source address
        PEA     Addr2(A5)            ;     and destination
        MOVE.W  #SWidth,-(SP)        ;     Push byte count
        _UnPackBits                  ;     Do the pack
        DBRA    D3,@1                ;     Loop till done
        MOVE.L  Addr1(A5),A0         ;     Restore
        MOVE.L  Addr2(A5),A1         ;     addresses
        MOVEM.L (SP)+,D0-D3/A2/A6;         Restore registers
        RTS                          ;     Back to caller
PMemErr StopAlert #MemErr            ;     Out of memory
        EXIT                         ;     So quit
```

Listing 6.38 *– The UnPack routine*

Register	Parameter
D0.W	Number of 72-byte rows to unpack
A6.L	First address beyond the unpacked buffer

On return from either Pack or UnPack, registers A0.L and A1.L are set to the first bytes in the source and destination areas not used in the conversion process.

Line 2 adjusts D0 for the DBRA instruction, and line 3 adjusts A6 down by the number of bytes in an unpacked row. Register D3 contains the actual loop count, because D0 is modified by the _UnPackBits system call. Lines 5 and 6 set up the two pointer variables required by _UnPackBits. Lines 7–8 ensure that no part of the unpacked data will land in memory outside the destination buffer. Lines 9–12 issue the _UnPackBits call for a single row of data (72 bytes). The DBRA instruction at line 13 repeats lines 7–13 until all rows have been processed. Lines 14–15 set up the return values in A0 and A1. Lines 18–19 are the error routine invoked when a memory bounds error occurs.

The routine to pack screen data is very similar; it appears in Listing 6.39. The Pack and UnPack routines differ only in the system call used and the fact that the limit value subtracted from A6 in line 3 is one greater in the Pack routine. The maximum number of bytes that can be generated by the _PackBits system call at line 12 is 73. The UnPack bits limit is 72.

```
 1: Pack    MOVEM.L D0-D3/A2/A6,-(SP);      Stack registers
 2:         SUBQ.W  #1,D0          ;        Adjust # rows for DBRA
 3:         SUBA.L  #SWidth+1,A6   ;        Adjust limit
 4:         MOVE.W  D0,D3          ;        Place in nonvolatile register
 5:         MOVE.L  A0,Addr1(A5)   ;        Store
 6:         MOVE.L  A1,Addr2(A5)   ;                addresses
 7: @1      CMPA.L  Addr2(A5),A6   ;        Check limit
 8:         BLS     PMemErr        ;        Too high, quit
 9:         PEA     Addr1(A5)      ;        Push source address
10:         PEA     Addr2(A5)      ;        and destination
11:         MOVE.W  #SWidth,-(SP)  ;        Push byte count
12:         _PackBits              ;        Do the pack
13:         DBRA    D3,@1          ;        Loop till done
14:         MOVE.L  Addr1(A5),A0   ;        Restore
15:         MOVE.L  Addr2(A5),A1   ;        addresses
16:         MOVEM.L (SP)+,D0-D3/A2/A6;      Restore registers
17:         RTS                    ;        Back to caller
18: Addr1   DS.L    1              ;        Temp address
19: Addr2   DS.L    1              ;        Another "
```

***Listing 6.39** – The Pack routine*

PackU

The PackU routine shown in Listing 6.40 packs the screen buffer contents into the undo buffer. This routine is normally called just before modifying the screen.

Lines 2–4 lock the undo buffer in memory. Lines 5–6 place the address of the undo buffer in A1 and A4. Lines 7–10 lock the screen buffer and place its address in A0. The vertical coordinate of the screen size is the number of rows to pack. This is placed in D0 to call the Pack routine in Listing 6.39. Lines 12–13 calculate the limit value for Pack and place it in A6. Line 14 contains the call to Pack. Lines 15–16 place the number of bytes packed in the top longword on the stack. When the registers are popped off the stack just before returning, this quantity is placed in D0.L. Lines 17–20 unlock the two buffers. Lines 21–22 restore the registers and return to the calling routine.

RePack

The RePack routine concentrates the entire document into the high end of the file buffer. This operation is required when the document is to be written to a file and when the display is scrolled.

The routine consists of three steps:

1. Pack the screen buffer to the undo buffer.
2. Move this buffer to the right place in the file buffer.
3. Move the starting part of the file buffer down to be contiguous with the rest of the data.

Figure 6.30 illustrates these steps.

The RePack routine appears in Listing 6.41. Line 2 accomplishes step 1. Lines 3–6 place the undo buffer address in A0 and the file buffer address in A1. Lines 8–13 calculate the address in the file buffer for the data in the undo buffer. Line 15 adjusts D0 (returned by PackU) for a DBRA loop count. Lines 16–19 check for an overlap between the data in the undo buffer and the beginning part of the file buffer.

Lines 20–21 move the undo buffer into the file buffer. This completes step 2 of the procedure.

Lines 22–26 compute the new buffer length. This is the total number of bytes in the buffer after consolidation. Lines 26–33 move the top section of the buffer down, completing the procedure. Note that the top section must be moved starting with the highest address. If the free area between the top and bottom areas of the buffer is smaller than the data to be moved, moving from the top down will destroy data in the top part of the buffer.

UnBuf

The UnBuf routine performs the inverse of a RePack. This code splits a single data area into the three buffer areas: the top of the file buffer, the screen buffer, and the bottom of the file buffer. UnBuf is shown in Listing 6.42.

Lines 2–3 lock the file buffer. Lines 4–10 compute the address of the first byte of data in the buffer. The global variable SRow indicates the row number of the first row to appear in the screen buffer. Lines 12–13 calculate the number of *unpacked* bytes in the top area of the screen buffer. Calculating the number of *packed* bytes requires a knowledge of the packed format.

Lines 15–29 move the packed data to the top of the file buffer. The Macintosh packing algorithm compresses adjacent bytes with the same value. The data is divided into groups that are less than 128 bytes long. The first byte in each group is a count byte. Count bytes are one less than the actual number of bytes represented by the group. A negative count byte indicates that the next byte is to be replicated. The number of replicated bytes is the absolute value of the count byte plus 1. Count bytes with the value −128 (80 hex) are ignored. A positive count byte indicates a run of unequal bytes. For a count

```
PackU   PUSHALL                   ;     Save caller's registers
        MOVE.L  UnDoBuf(A5),A0    ;     A0 -> Undo handle
        MOVE.L  A0,A3             ;     Save handle
        _HLock                    ;     Lock it down, too
        MOVE.L  (A0),A1           ;     A1 -> Undo buffer
        MOVE.L  A1,A4             ;     Save this
        MOVE.L  ScrBuf(A5),A0     ;     A0 -> Handle
        MOVE.L  A0,A2             ;     Save handle
        _HLock                    ;     Lock it down
        MOVE.L  (A0),A0           ;     A0 -> Unpacked data
        MOVE.W  SSize(A5),D0      ;     D0 = Count
        MOVE.L  A1,A6             ;     Calculate
        ADDA.L  UBufSiz(A5),A6    ;             limit
        BSR     Pack              ;     Pack it
        SUB.L   A4,A1             ;     Compute byte count
        MOVE.L  A1,(SP)           ;     Put it in D0 (via POPALL)
        MOVE.L  A2,A0             ;     Unlock
        _HUnLock                  ;           all
        MOVE.L  A3,A0             ;               locked
        _HUnLock                  ;                     buffers
        POPALL                    ;     Restore registers
        RTS                       ;     Return to caller
```

Listing 6.40 *– The PackU routine* byte of *n*, there are *n* + 1 data bytes that follow. Figure 6.31 illustrates the relationship between packed and unpacked data.

Lines 16–17 fetch the next count byte from the buffer and store it in the destination. Lines 19–22 handle a positive count byte, moving the data bytes that follow into the buffer. Lines 23–25 handle repeated bytes, moving the repeated data byte and negating the count byte. Lines 26–28 subtract the number of unpacked bytes moved from the total number required. The loop should terminate after moving exactly the number of bytes required (that is, D2 should be zero) at line 29. The BNE instruction branches to an alert routine if this is not the case. Lines 33–34 compute the byte count in the first part of the file buffer.

Lines 35–42 set up a call to UnPack at line 43. This converts the data to be displayed into unpacked format in the screen buffer. Line 44 computes the address of the first byte in the last part of the file buffer. Lines 44–45 correct the count in FBEnd to the proper number of bytes left in the end of the file buffer.

Lines 48–50 unlock the locked buffers. Line 51 packs the screen buffer into the undo buffer. This is necessary because the undo buffer may have data in it that reflect a part of the document which is no longer in the window. Lines 52–53 restore the registers and branch to the ReDraw routine. This causes the correct part of the document to appear in the window.

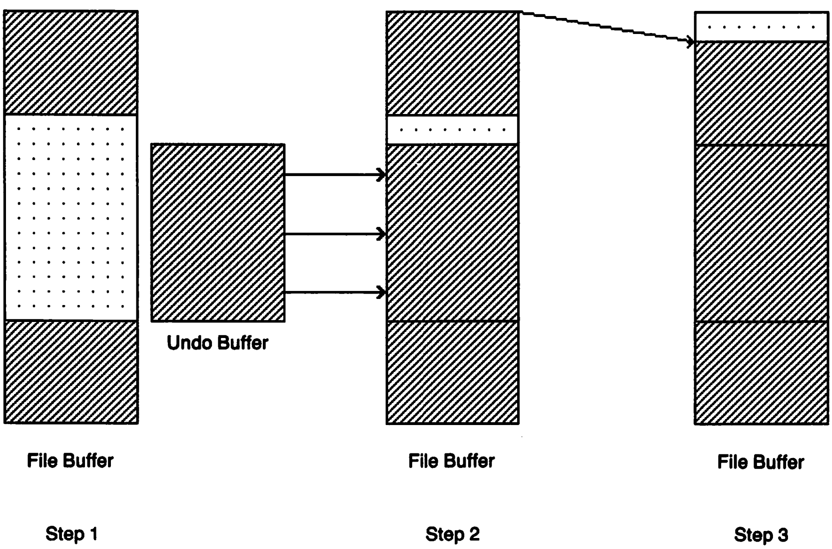


Figure 6.30 – *File buffer consolidation*

Draw Data Area

Listing 6.43 contains data definitions for variables used only by the draw routines. The data items have been explained in the text accompanying previous listings.

MacDoodle Resource File Definitions

To build a Macintosh application, you need both source code and resource definitions. This section describes the different resource types required for MacDoodle. These definitions are contained in the file MacDoodle.R.

```
RePack  PUSHALL                   ;      Save all registers
        BSR     PackU             ;      Pack screen to undo buffer
        MOVE.L  UnDoBuf(A5),A0    ;      A0 ->
        MOVE.L  (A0),A0           ;            Undo buffer
        MOVE.L  FileBuf(A5),A1    ;      A1 ->
        MOVE.L  (A1),A1           ;            File buffer
        MOVE.L  A1,A2             ;      Save this
        CLR.L   D1                ;      Clear D1 to convert
        MOVE.W  FBEnd(A5),D1      ;      D1 = # bytes in bottom area
        SUB.L   FBufSiz(A5),D1    ;      Subtract buffer size
        NEG.L   D1                ;      D1 = Buffer offset
        SUB.W   D0,D1             ;      Compute new buffer offset
        LEA     0(A1,D1.W),A1     ;      A1 -> New buffer offset
        MOVE.L  A1,A4             ;      Save this
        SUBQ.W  #1,D0             ;      Adjust D0 for DBRA now
        CMP.W   FBStart(A5),D1    ;      Overlap?
        BGE     @1                ;      No, it's OK
        StopAlert #BuffErr        ;      Out of memory,
        EXIT                      ;                  all is lost
@1      MOVE.B  (A0)+,(A1)+       ;      Move in undo buffer
        DBRA    D0,@1             ;      until done
        SUB.W   FBStart(A5),D1    ;      Subtract start length
        ANDI.L  #$FFFF,D1         ;      Mask off upper part
        SUB.L   FBufSiz(A5),D1    ;      Compute
        NEG.L   D1                ;            end length
        MOVE.W  D1,FBEnd(A5)      ;      Store new count
        MOVE.W  FBStart(A5),D0    ;      D0 = # bytes to move
        BEQ     @3                ;      EQ => Top region empty
        LEA     0(A2,D0.W),A2     ;      A2 -> End of start region
        SUBQ.W  #1,D0             ;      DBRA-Adjusted
        CLR.W   FBStart(A5)       ;      No bytes here now
@2      MOVE.B  -(A2),-(A4)       ;      Move in
        DBRA    D0,@2             ;            start of buffer
@3      POPALL                    ;      Restore registers
        RTS                       ;      Back to caller
```

***Listing 6.41** – The RePack routine*

```
UnBuf   PUSHALL                     ;     Save the registers
        MOVE.L  FileBuf(A5),A0      ;     A0 -> File buffer handle
        _HLock                      ;     Lock it
        MOVE.L  (A0),A1             ;     A1 ->  File buffer
        MOVE.L  A1,A2               ;     Save for later
        CLR.L   D1                  ;     Load
        MOVE.W  FBEnd(A5),D1        ;             byte count
        SUB.L   FBufSiz(A5),D1      ;     Compute
        NEG.L   D1                  ;             new
        LEA     0(A1,D1.W),A0       ;                   offset
        MOVE.L  A0,A4               ;     Remember this
        MOVE.W  SRow(A5),D2         ;     D2 =
        MULS.W  #SWidth,D2          ;             # bytes
        BEQ     @5                  ;     If EQ, do unpack
        CLR.L   D3                  ;     D3 = temp
@1      MOVE.B  (A0)+,D3            ;     Fetch count byte
        MOVE.B  D3,(A1)+            ;     Store in buffer top
        BMI     @3                  ;     < 0 means repeated byte
        MOVE.W  D3,D4               ;     Copy count byte
@2      MOVE.B  (A0)+,(A1)+         ;     Move to
        DBRA    D4,@2               ;             buffer top
        BRA     @4                  ;     Branch to end of loop
@3      NEG.B   D3                  ;     Correct count
        BMI     @1                  ;     Ignore hex 80 bytes
        MOVE.B  (A0)+,(A1)+         ;     Move data byte
@4      ADDQ.W  #1,D3               ;     Bump count (Un-DBRA adjust)
        SUB.L   D3,D2               ;     Subtract item count
        BGT     @1                  ;     > 0 means go get next
        BNE     Void                ;     A terrible error!
;
;       Now unpack file buffer to screen
;
@5      SUBA.L  A2,A1               ;     Compute
        MOVE.W  A1,FBStart(A5)      ;             top byte count
        MOVE.L  A0,-(SP)            ;     Save A0 for a bit
        MOVE.L  ScrBuf(A5),A0       ;     A0 -> Screen buffer handle
        _HLock                      ;     Lock it
        MOVE.L  (A0),A1             ;     A1 -> Screen buffer
        MOVE.L  (SP)+,A0            ;     Restore A0
        MOVE.W  SSize(A5),D0        ;     D0 = # rows to unpack
        MOVE.L  A1,A6               ;     Calculate
        ADDA.L  SBufSiz(A5),A6      ;                limit
        BSR     UnPack              ;     Unpack to screen buffer
        SUB.L   A4,A0               ;     Size of start + unpacked area
        MOVE.L  A0,D0               ;     Move to data register
        SUB.W   D0,FBEnd(A5)        ;     Result is new  size
        MOVE.L  ScrBuf(A5),A0       ;     Unlock
        _HUnLock                    ;             all
        MOVE.L  FileBuf(A5),A0      ;                   locked
        _HUnLock                    ;                              handles
        BSR     PackU               ;     Make undo buffer correct now
        POPALL                      ;     Pop the registers
        BRA     ReDraw              ;     Redraw the screen
```

***Listing 6.42** – The UnBuf routine*

Output File

The first nonblank and noncomment line in a resource definition file defines the output file for RMAKER. Listing 6.44 shows this definition for MacDoodle.

The notation !MacDoodle means that the resources are to be appended to the file named MacDoodle. To build MacDoodle, you normally run the linker to produce this file, and then you run RMAKER to append the resources.

MacDoodle Icons

Macintosh applications can specify icons to be used for the application file and for document files with the application's signature. Figure 6.32 shows the icons for MacDoodle.

Listing 6.45 contains the definitions necessary to define the MacDoodle icons. The Finder looks at each application on the disk to see if it has a bundle resource. The bundle bit in the Finder flags word for the file must be set for the Finder to recognize the bundle resource. The bundle resource is a data area that specifies other resources in the application file which define the icons. Lines 6–12 define the bundle resource for MacDoodle. This bundle resource links together the other resources in Listing 6.45. Line 8 defines the resource type and ID of a version number resource. The version number resource must have the application's signature as its resource type. This is DOOD for MacDoodle. The resource ID is 0.

Two other resource types are required to complete the bundling process. A file reference resource (FREF) maps a file type onto a number called a *local ID*. An icon list resource (ICN#) maps the local ID onto an icon to be displayed. There are normally two FREF and two ICN# resources: one for the application file and one for documents processed by the application. The local IDs are the Finder's method of relating the

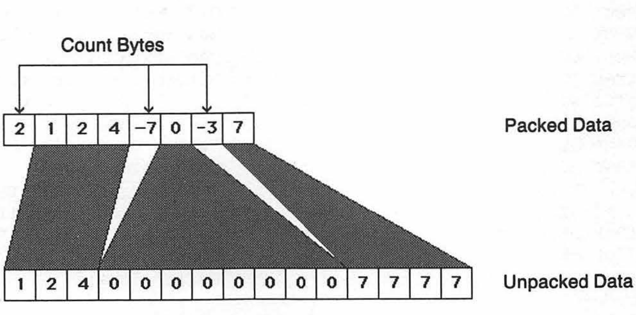


***Figure 6.31** – Packed and unpacked data format*

two resources. All resources in Listing 6.45 associated with the application file type have the same local ID, as do all resources associated with the document file type.

Lines 9–10 give the resource ID of the icon list. Line 10 maps local ID 0 to resource ID 128 and local ID 1 to resource ID 129. Lines 11–12 give the resource IDs of file references for the application and document icons. Local ID 0 maps onto FREF ID 128, and local ID 1 onto FREF ID 129.

The FREF resources map a file type onto a local ID. There are two such resources used in MacDoodle: one for the application file and one for document files. These have local IDs 0 and 1 (resource IDs 128 and 129), respectively. The file type for an application is APPL. This is mapped onto local ID 0 in line 16. The file type for a document processed by MacDoodle is PNTG. Line 18 maps this file type onto local ID 1.

The version data is defined in lines 23–26. The type statement in line 23 uses a GNRL (general) construction. The GNRL mechanism allows you to define resources with arbitrary format. The .P specification at line 25 specifies that the next line is to be placed in the file as a Pascal string.

The icon lists referenced in lines 9–10 are defined in lines 32–81. Line 10 maps local ID 0 to resource ID 128 and local ID 1 to resource ID 129. Resource ID 128 defines the icon for use with the application file.

Each icon definition consists of a 32 × 32-bit image for the icon itself plus another 32 × 32-bit image for the icon mask. The icon mask determines which parts of the background will show through when the icon is placed on top of another object. Also, an icon is highlighted by XORing the icon with the mask. Normally, the mask is the same shape as the icon and completely black.

```
Point1: DS.L   1            ;      Temp point
Point2: DS.L   1            ;      Second temp point
TmpRect DS.L   2            ;      Temporary rectangle
OldRect DS.L   2            ;      Old rectangle for boxes, ovals
NewRect DS.L   2            ;      New rectangle for boxes, ovals
OldPen: DS.W   9            ;      Pen state structure
SBitMap DS.L   1            ;      Bitmap structure for screen buffer
SRBytes DS.W   1            ;              RowBytes word
SBounds DS.W   4            ;              Bounds rectangle
SrcRect DS.W   4            ;      Source rectangle
PortAdr DS.L   1            ;      GrafPort address
ScrType DS.W   1            ;      Remember original type here
BuffCnt DS.W   1            ;      # characters in buffer
BuffPtr DS.L   1            ;      -> Next free slot in buffer
TBufCnt DS.B   1            ;      Count byte for draw string
TBuffer DS.B   TBuffSize    ;      Buffer itself
```

***Listing 6.43** – The Draw data area*

The .H directive at lines 34 and 62 specifies that the resource data that follows is in hex. Lines 35–42 define the MacDoodle application icon, and lines 46–53 define the mask. Lines 63–70 define the document icon, and lines 74–81 define the mask.

Apple supplies a program called the Icon Editor with a package called the Software Supplement. This program has a 32 × 32-bit version of MacPaint FatBits for drawing icons. It can also output the data in hexadecimal. Using this program can save a lot of time in defining icons.

Menus

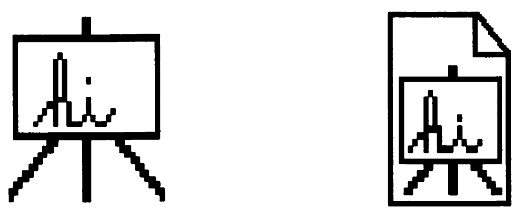
Listing 6.46 contains the resource definitions for the MacDoodle menus. Menus used by the run-time library described in Chapter 8 must have resource IDs numbered consecutively from 1.

The first line after the resource ID for each menu gives the menu title. Subsequent lines contain the text for menu choices. The "(-" construct indicates a line drawn across the menu. This line occupies a menu choice, but it cannot be selected.

Lines 6–9 define the Apple menu, which contains the About dialog choice and the desk accessories. The "\14" at line 7 is the name of the menu. Using a backslash (\) character in any string takes the next two characters as hexadecimal digits, and inserts a character with that hexadecimal value. A value of 14 hex generates the Apple symbol on the screen.

```
*
* This is the resource file for the MacDoodle Program
*

!MacDoodle
```

Listing 6.44 – *Output file definition*



MacDoodle   Document File

Figure 6.32 – *The MacDoodle icons*

```

*
*       Bundle resource definition.  Specifies use of the
*       MacDoodle private icon to the Finder.
*
Type BNDL
    ,128
DOOD 0
ICN#
0 128 1 129
FREF
0 128 1 129

Type FREF
    ,128
APPL 0          ;; Application maps to Icon #0
    ,129
PNTG 1          ;; Document maps to Icon #1

*
*       Version number resource
*
Type DOOD = GNRL
     ,0
.P
MacDoodle Program Version 1.0

*
*       MacDoodle icon definition
*

Type ICN# = GNRL
     ,128
.H
00018000 00018000 00018000 7FFFFFFE
40000002 40000002 40200002 40500002
40500002 40500002 40508002 40500002
40500002 40708002 40C88002 41488402
42494802 444E3002 40000002 7FFFFFFE
00C18300 01818180 030180C0 06018060
0C018030 18018018 3001800C 60018006
C0018003 80018001 00000000 00000000
*
*       Icon mask
*
00018000 00018000 00018000 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
00FFFF00 01FFFF80 03FFFFC0 07FFFFE0
0FFFFFF0 1FFFFFF8 3FFFFFFC 7FFFFFFE
FFFFFFFF FFFFFFFF 00000000 00000000

```

Listing 6.45 – *Icon definition*

```
*
* Document icon definition
*

Type ICN# = GNRL
     ,129
.H
1FFFF800 10000C00 10000A00 10000900
10000880 10000840 10000820 10000FF0
10000010 10018010 10018010 17FFFFD0
14000050 14100050 14280050 14280050
14280050 14284050 14280050 143C4050
14644050 14A4A450 15271850 14000050
17FFFFD0 10318C10 10618610 10C18310
11818190 130180D0 10000010 1FFFFFF0
*
*       Document icon mask
*
1FFFF800 1FFFFC00 1FFFFE00 1FFFFF00
1FFFFF80 1FFFFFC0 1FFFFFE0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
```

***Listing 6.45** – Icon definition (continued)*

```
*
*       Menu definitions
*

Type MENU
  ,1
\14
 About MacDoodle ...
 (-

  ,2
File
  New/N
  Open/R
  Save/W
  Save As ...
(-
  Print
```

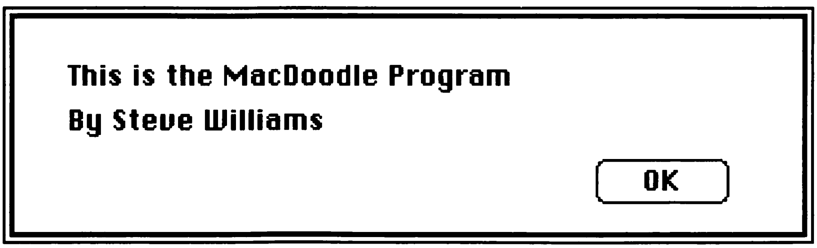
***Listing 6.46** – MacDoodle menu definitions*

```
(-
  Quit

  ,3
Edit
  Undo
(-
  Cut/X
  Copy/C
  Paste/V

  ,4
Draw
  Lines
  Freehand
  Boxes
  Ovals
  Text
(-
  Eraser
(-
  Erase Screen

  ,5
PenSize
   1 x 1
   2 x 2
   3 x 3
   4 x 4
   5 x 5
   6 x 6
   7 x 7
   8 x 8
   9 x 9
  10 x 10
  11 x 11
  12 x 12
  13 x 13
  14 x 14
  15 x 15
  16 x 16

  ,6
Font
  Chicago
  New York
  Geneva
  Monaco
  Venice
  London
  Athens
  San Francisco
  Toronto
  Cairo
  Los Angeles
```

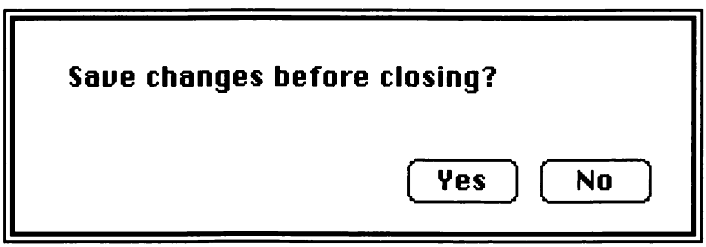
Listing 6.46 *– MacDoodle menu definitions (continued)*

```
  Seattle

  ,7
Style
  Plain/P
  <BBold/B
  <IItalic/I
  <UUnderline/U
  <OOutline/O
  <SShadow/S
  Compressed
  Expanded
(-
   9 Point
  10 Point
  12 Point
  14 Point
  18 Point
  20 Point
  24 Point
  36 Point
  72 Point
```

Listing 6.46 **–** *MacDoodle menu definitions (continued)*



About MacDoodle Dialog (ID = 1)



Save Changes Dialog (ID = 2)

Figure 6.33 **–** *MacDoodle dialog boxes*

```
Type DLOG
  ,1

100 100 190 450
Visible  NoGoAway
1
0
1

Type DITL
  ,1
3

StaticText
15 20 36 350
This is the MacDoodle program

StaticText
35 20 56 350
By Steve Williams

Button
60 260 80 320
OK

Type DLOG
  ,2

100 100 190 400
Visible  NoGoAway
1
0
2


Type DITL
  ,2
3

Button
60 175 80 225
Yes

Button
60 235 80 285
No

StaticText
15 20 36 325
Save changes before closing?
```

Listing 6.47 *– MacDoodle dialog definitions*

The resource file defines two items in this menu: the About dialog and a line at the second menu choice. The run-time library adds the desk accessory names to this menu during program initialization. The code that does this appears in Listing 8.4 (Chapter 8).

Command key equivalents to menu selection are defined by placing a slash (/) character and the desired key character at the end of the selection name. For instance, the New file selection in line 13 can also be invoked by the Command-N key sequence.

You may also specify text attributes for the menu selection items. Lines 79–83 illustrate how to do this. Using a less-than symbol (<), you can assign a single attribute to the text string. These are the allowable attribute sequences:

<B	Bold
<I	Italic
<U	Underline
<O	Outline
<S	Shadow

There is no sequence for Compressed or Expanded text attributes. You cannot set more than one attribute using this technique.

Dialogs

MacDoodle has two dialog boxes, one for the About dialog and one to prompt the user to save changes to the document. These are shown in Figure 6.33. Listing 6.47 contains the resource definitions for these two items.

Defining a dialog box requires two resources: a DLOG resource, which defines the box, and a DITL (Dialog ITem List) resource, which defines what goes in the box. Lines 1–8 define the DLOG resource for the About dialog, and lines 10–24 define the associated DITL resource.

A DLOG resource definition consists of five lines (blank lines are not significant). The first line (line 4 in Listing 6.47) gives the rectangle on the screen where the box is to appear. This rectangle is represented by global coordinates. The next line (5) gives the attributes of the box. The three lines that follow (6–8) give the definition procedure ID, the reference value, and the resource ID of the associated DITL definition.

Two of four possible attributes can be specified for a dialog box. The box can be either visible or invisible, which determines whether the box is drawn initially. The dialog box can have a box similar to a window's box. Specify GoAway if you want the dialog box to have a close box, or NoGoAway.

The definition procedure ID is 1 for a standard dialog box, like the ones in Figure 6.33. The reference constant is normally unused, but it still must be specified. The DITL resource associated with the first dialog has resource ID 1.

Lines 10–24 define the DITL resource for the About dialog. Line 11 indicates the resource ID. Line 12 specifies the number of items in the item list. Lines 14–16 define the first item. StaticText means a message that cannot be edited by the user. Line 15 gives a rectangle in which the message is to appear. This rectangle is in the local coordinates of the dialog box. Line 16 contains the actual text. Lines 18–19 define the second line of text in the dialog. Lines 22–24 define the OK button. Line 23 gives the rectangle containing the button, again in local coordinates. Line 24 contains the text to appear inside the button.

Lines 26–50 define the second dialog box. This is very similar to the About box, except that there are two buttons. The order of the items in the DITL resource determines the value that will be returned by the DIALOG macro when the user makes a selection. In this case, the Yes button returns a value of 1 and the No button returns a value of 2. Static text items occupy a position in determining this value. A DIALOG macro that specifies the About box would return a value of 3.

Alerts

Listing 6.48 shows a sample alert resource definition. Line 2 gives the resource ID for this alert. Line 3 gives the rectangle for the alert box in screen coordinates. Line 4 contains the resource ID of the associated DITL resource. Line 5 contains the stages word (in hexadecimal). Alert stages are illustrated in Figure 6.17.

The DITL resource appears in Lines 7–21. Note that the button is the first entry. Since the default button can be only the first or second item, you must be careful when ordering items in a DITL resource associated with an alert.

Main Window

Listing 6.49 contains the definition for the main document window. The run-time library requires that the window have resource ID 1.

Lines 8–9 define the window boundaries (in global coordinates) and attributes. Window attribute specifications are the same as the dialog attributes discussed previously. Line 10 gives the definition procedure ID. This is 0 for a standard document window. Line 11 contains the reference constant. This is not used by MacDoodle.

The scroll bars are defined next. Lines 17–24 define the vertical scroll bar, and lines 26–33 define the horizontal scroll bar. Lines 19 and 28 define a control title. This is a field required by RMAKER that is unused by scroll bars. Any string can be used. Lines 20 and 29 define the coordinates for the scroll bar in the window's local coordinate scheme. Lines 21 and 30 define the control attributes. Control attributes can be either visible or invisible. Lines 22 and 31 give the procedure definition ID, which is 16 for scroll bars. Lines 23 and 32 contain the reference constant. Lines 24 and 33 contain the minimum, maximum, and initial values for each scroll bar.

Sample Cursor

Listing 6.50 contains a definition of a sample cursor. A cursor resource consists of two 16 × 16-bit images and a point. The first bit image defines the cursor and the second defines the cursor mask. The point defines the mouse coordinates within the cursor.

The cursor is defined with the GNRL type, using hexadecimal data. Lines 10–25 define the cursor outline, and lines 27–42 define the mask. Lines 45–46 define the mouse point within the cursor.

This cursor is the 8 × 8 eraser cursor. The resource ID is 23 (decimal) or 17 hex. The code in Listing 6.17 selects this cursor when Eraser is selected on the Draw menu and 8 × 8 is selected on the PenSize menu. The remaining cursor definitions are left as an exercise.

```
Type ALRT
   ,129
50 50 160 325
3
5555

Type DITL
   ,3
3

Button
85 205 105 265
QUIT

StaticText
20 60 40 225
MAJOR PROGRAM BUG

StaticText
50 20 70 225
This should "Never Happen"
```

Listing 6.48 – *Sample alert definition*

Run-Time Library Configuration

MacDoodle requires a version of the run-time library that is different from the one used by the programs we wrote earlier. Listing 6.51 shows the technique used to configure the RTL.

Line 4 disables the use of the Text Edit package, so that the document window can contain graphics. Line 5 enables the use of scroll bars on the window, and line 6 specifies that the application does not have a separate resource file. These options are explained in Chapter 8.

The Linker Command File

Listing 6.52 shows the MacDoodle linker command file. MacDoodle requires the use of a few additional linker features. Lines 1–6 list the files

```
*
*        Main window definition
*

Type WIND
   ,1

40 5 300 500
Visible GoAway
0
0

*
*        Scroll bar controls for main window
*

Type     CNTL
    ,256(4)
x
    0 480 245 495
    Visible
    16
    0
    0 720 0

Type     CNTL
    ,257(4)
x
    245 0 260 480
    Visible
    16
    0
    0 576 0
```

Listing 6.49 *– MacDoodle window definition* to be included in the link, including the PrLink file from the Macintosh Development System. Line 7 specifies the entry point for the program. The XXSTART label is generated by the PROGRAM macro in Listing 6.1. Line 8 sets the bundle bit in the Finder file flags word for the file. Line 9 sets the Finder file type and application signature words. A file type of

```

*
*       8 x 8 Eraser cursor
*

Type CURS = GNRL
     ,23
.H
*       16 words of cursor pattern
FF00
8100
8100
8100
8100
8100
8100
FF00
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
FE00
FE00
FE00
FE00
FE00
FE00
FE00
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
```

Listing 6.50 *– Sample cursor definition*

APPL denotes an application file. DOOD is the application signature for MacDoodle.

SUMMARY

This chapter contains a brief overview of the techniques used to write Macintosh applications. We have covered a number of important topics:

- The use of the QuickDraw graphics package
- Memory layout and memory management
- Resources
- Windows, dialogs, and alerts
- Menu operations
- Events

```
;
;        MacDoodle RTL.  Configures the RTL code for use with MacDoodle.
;
TEXTRTL EQU     -1              ;       Not text RTL
CONTROL EQU      0              ;       Does have window controls
RESFILE EQU     -1              ;       No separate resource file
Include RTLcode.Asm             ;       Include RTL
        END
```

***Listing 6.51** – MacDoodle RTL configuration file*

```
MacDoodle
MacDoodleDraw
MacDoodleMenu
MacDoodleData
MacDoodleRTL
PrLink
!XXSTART
/Bundle
/Type 'APPL' 'DOOD'
$
```

***Listing 6.52** – The MacDoodle linker command file*

The most important part of the chapter is the example program. This is a piece of working code that performs most of the commonplace operations in the Macintosh environment.

The Macintosh system has many more facilities than we have used here. Most of the system services have several options that give the application a good deal more flexibility than we have presented. However, you can write fairly sophisticated programs using just the material in this chapter.

EXERCISES

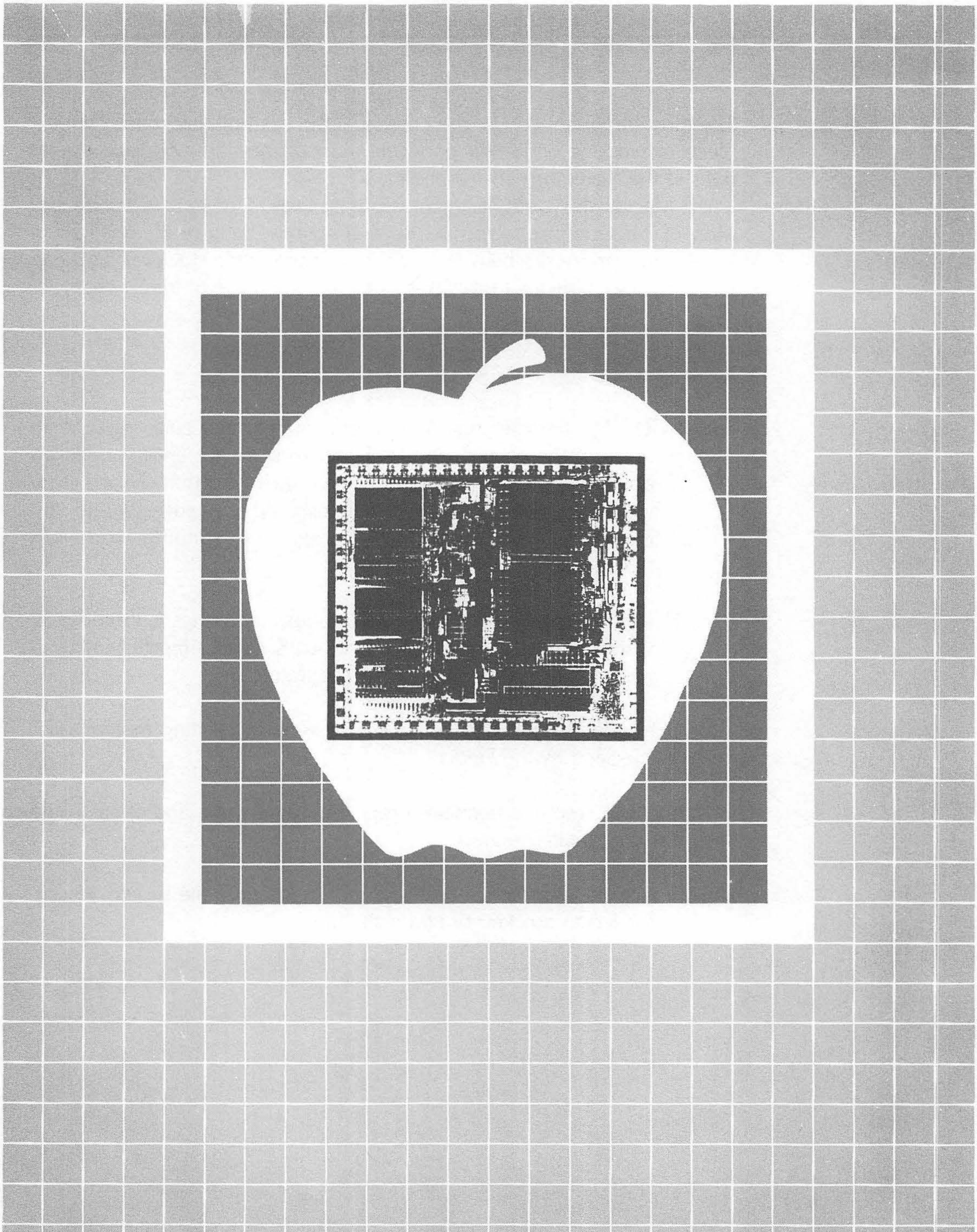
1. The MacDoodle application uses a complicated buffering scheme to avoid the need for a 56K buffer to store the entire unpacked document. Using such a buffer would mean that the program would not run on a 128K Macintosh. What changes would you make if you could restrict the program to running on a 512K Macintosh?

2. Replace the UnPack and Pack routines in Listings 6.38 and 6.39 with routines that make no system calls. What simplifications can you make to other parts of the program?

3. Modify the code in Listing 6.5 to support printing multiple files specified by the Finder.

4. Write resource definitions for all of the cursors and alerts in the MacDoodle program.

5. Which resources in the MacDoodle resource file should be preloaded to quicken program startup?



Chapter 7

Exception Processing

Consider nothing as impossible before it has come to pass.

—Confucius

INTRODUCTION

This chapter will teach you what a machine *exception* is and the programming techniques you can use to process exceptions. We will discuss the types of exceptions possible on the 68000 chip and how each is handled by the Macintosh.

WHAT IS AN EXCEPTION?

An *exception* is the machine's ability to interrupt what it is doing, do something else, and (if necessary) return to the interrupted task. There are two types of exceptions on the 68000:

1. Exceptions caused by I/O devices, called *interrupts*
2. Exceptions caused by internal operations, such as program errors or the *TRAP* instructions

The exception mechanism is commonly used to interweave I/O processing with computation. Because I/O devices are typically much slower than the CPU, the CPU can spend most of its time waiting for the I/O device. The use of exceptions allows programs to make use of that idle CPU time to perform computations, without in any way delaying or interrupting the I/O task at hand. This effectively means that programs can perform both I/O and computations simultaneously, reducing the time required to perform a task.

General Exception Processing

There are four steps in using the external exception mechanism:

1. The program starts an I/O operation and begins doing computations.
2. When the I/O is complete, an exception occurs, causing the computational work to be suspended.
3. If there is more I/O to be done, the program starts another I/O operation.
4. Computation can then resume until another exception occurs.

This type of I/O is commonly called *interrupt-driven I/O.* The Macintosh uses interrupt-driven I/O for the keyboard, mouse, disks, and serial ports.

Exceptions that result from internal operations act like subroutine calls. The TRAP instructions are used to call the operating system in many 68000 computers; the Macintosh uses the illegal instructions with op codes A000 hex and greater instead, as described in Chapter 5. From the standpoint of the application, the instruction that causes the exception resembles a BSR or JSR instruction. Execution resumes at the location following the TRAP or illegal instruction.

In many cases, it is not desirable to resume execution following an internal exception. Most internal exceptions result from programming errors, such as accessing nonexistent memory, division by zero, or a privilege violation. Continuing the program after such an error is almost certain to produce more errors. The Macintosh displays a "System Error" message after this type of error, and you must reboot the Macintosh to recover from such an error.

Vectors

Each possible exception is associated with a unique longword in memory called a *vector.* There are 256 possible vectors, numbered from 0 to 255. Each vector contains the address of the routine that processes the exception. Vectors are organized contiguously in memory, starting at absolute address 0. The address of a vector is the vector number times 4. Internally generated exceptions use dedicated, or preassigned, vector numbers. There are also seven dedicated vector numbers for I/O, called the *autovector interrupts*. In addition, a mechanism exists for I/O devices to specify vectors to be used for I/O interrupts.

Table 7.1 lists the vector numbers that are preassigned by the 68000. Locations 0 and 4 are used for the initial stack and program counter when the processor is first powered up or when the RESET signal is applied. The forward button on the Macintosh Programmer's Switch panel (described in the Macintosh owner's manual) causes a RESET exception.

Vector (decimal)	Address (hex)	Function
0	0	RESET initial SSP (supervisor stack pointer)
1	4	RESET initial PC (program counter)
2	8	BUSERR (nonexistent memory)
3	C	Address (boundary) error
4	10	Illegal instruction
5	14	Zero divide
6	18	CHK instruction
7	1C	TRAPV instruction
8	20	Privilege violation
9	24	TRACE
10	28	Line 1010 emulator
11	2C	Line 1111 emulator
12–14	30–38	Unassigned (reserved)
15	3C	Uninitialized interrupt vector
16–23	40–5C	Unassigned (reserved)
24	60	Spurious interrupt
25–31	64–7C	Level 0–7 autovector interrupts
32–47	80–BF	TRAP 0–15 instruction vectors
48–63	C0–FC	Unassigned (reserved)
64–255	100–3FF	User interrupt vectors

***Table 7.1** – Preassigned Vectors*

A BUSERR (bus error) indicates a program reference to a memory location that does not exist. (This is colloquially known as "missing the bus.") A reference to a word or longword at an odd-numbered address causes an addressing-error exception. Illegal instructions cause exceptions through vector numbers 4, 10, or 11. The op codes Axxx and Fxxx cause exceptions that trap through the line 1010 and line 1111 vectors (10 and 11, respectively). All other illegal instructions (for example, an op code of 4AFC) use vector 4.

Division by zero causes an exception through vector 5. The CHK and TRAPV exceptions are caused by the CHK and TRAPV instructions (see Chapter 3). These instructions trap through vectors 6 and 7.

A privilege-violation exception results from executing a privileged instruction while the processor is in user mode. If bit 13 of the status register is not set before executing a privileged instruction, the 68000 takes an exception through vector 8. All Macintosh software runs in supervisor mode, so this exception should not occur on a Macintosh.

Macintosh debugger programs use the TRACE exception vector (9) for executing single instructions in a program to be debugged. These are the steps usually used in single-stepping:

1. Push the program counter of the instruction to be debugged onto the stack.

2. Push the status register with the trace bit (bit 15) set.

3. Execute an RTE instruction.

The processor will execute one instruction with the trace bit set and then trap through the TRACE vector. Any instruction that affects all bits of the status register can set the trace bit.

Whenever a memory violation (BUSERR) occurs during an external interrupt, the spurious interrupt (vector 24) is used. This indicates a hardware malfunction on the Macintosh.

Vector 15 is used by Motorola support chips for the 68000. The Macintosh does not use these chips. An interrupt through this vector also indicates a hardware malfunction.

Vectors 12–14, 16–23, and 48–63 are reserved for use by Motorola on future 68000-family chips. Vectors 64–255 are reserved for use by Motorola customers (like Apple) for devices connected to the 68000. None of these vectors is used by the Macintosh.

WHAT HAPPENS DURING AN EXCEPTION?

When the 68000 recognizes an exception condition, several things occur:

1. The current values of the program counter (which normally points to the next instruction to be executed) and status register are pushed onto the supervisor-mode stack.
2. The trace bit in the status register is turned off, and the supervisor bit is turned on. This prevents a TRACE exception and forces the 68000 into supervisor state. For external interrupts, the interrupt mask in the status register is also updated.
3. For a BUSERR or addressing-error exception, extra information is pushed onto the stack.
4. The program counter is loaded from the appropriate vector, and execution begins at this address.

The routine whose address is contained in the vector is called an *exception handler.* This routine normally saves the registers on the stack, performs some action, restores the registers, and executes an RTE instruction. Thus, the 68000 provides the ability to interrupt a program and later resume executing the program with no noticeable effect, other than increased processing time. This ability is normally used with interrupt-driven I/O.

RESET

There is one exception that does not behave this way. A special pin on the 68000 chip called RESET causes a special exception to take place. A signal placed on the RESET pin causes the processor to load the supervisor stack pointer from location 0 and the program counter from location 4. This provides a mechanism for starting the 68000 in a known state. RESET is normally used for the *bootstrap* button on 68000 microcomputers; that is, it provides both a mechanism for starting the 68000 when power is applied and the ability to recover from catastrophic software failures. The button labeled RESET on the Macintosh Programmer's Switch panel activates the 68000 RESET signal.

BUSERR and Addressing-Error Exceptions

Vectors 2 and 3 are used for errors detected in references to memory. The BUSERR exception (vector 2) indicates that the program has referenced memory that does not exist.

An addressing-error exception means that the program has referenced a memory word or longword at an odd address. If a program references a memory word or longword at an odd address that is also nonexistent (i.e., both bus and addressing-error conditions), the processor will detect the addressing error first, and only the addressing-error exception will take place.

These two exceptions differ from all other exceptions in that the processor puts extra information on the stack. When the exception handler is entered, the stack appears as shown in Figure 7.1.

The first word on the stack contains information about the type of memory access that caused the fault. Bits 5 through 15 of this word are undefined. If the access error occurred during a memory read, the R/W bit is 1.

Bits	15 … 5	4	3	2 1 0
A7	//////////////////////	R/W	I/N	Access
+2	High word of erroneous address			
+4	Low word of erroneous address			
+6	First word of instruction			
+8	Status register			
+10	Program counter high word			
+12	Program counter low word			

***Figure 7.1** – The stack after bus or addressing error*

If the access error occurred during a memory write, the R/W bit is 0.

The I/N bit is 0 if the processor was processing an instruction and 1 if the error was detected by an external device. (Most of these errors result from instruction processing.) The *access code* contained in the lower three bits describes the type of memory access being performed. Table 7.2 shows the type of memory access indicated by each access code.

These bits tell you two things: whether the processor was referencing data or program instructions and whether the processor was in supervisor mode or user mode. Data references include all accesses to memory that use all addressing modes except the two PC-relative modes. Program references include instruction accesses to memory that use the PC-relative addressing modes as well as accesses to memory that fetch instruction words.

The next two words on the stack give the address where the fault occurred. The processor saves a copy of the op code of the instruction that caused the fault in the next word on the stack. As with all other exceptions, the status register and program counter are present. The value stored as the program counter is advanced from the beginning of the instruction by two to ten bytes. (The smallest 68000 instruction is two bytes long; the largest is ten bytes.)

If the error occurred while an instruction was being fetched, the stored program counter will be in the vicinity of the previous instruction. Normally, this error is caused by taking a wild branch. In this case, the program counter and op code word on the stack indicate the branch instruction rather than the erroneous address of the attempted transfer.

Code	Type of Access
000	Unassigned
001	User mode data reference
010	User mode program reference
011	Unassigned
100	Unassigned
101	Supervisor mode data reference
110	Supervisor mode program reference
111	Interrupt acknowledge

Table 7.2 – *Memory Access Codes*

The erroneous address words on the stack will contain the erroneous instruction address (see Figure 7.1).

If the processor encounters a second BUSERR or addressing error during the processing of one of these errors, a situation called a *double bus fault* occurs. This occurs if the supervisor stack pointer is corrupt and the processor is unable to save any information on the stack. The processor halts and can be restarted only by using the RESET input on the chip.

Illegal Instruction Exceptions

Whenever the 68000 fetches an op code that cannot be interpreted as a legal 68000 instruction, an exception through vectors 4, 10, or 11 occurs. Vectors 10 and 11 are used for op codes that have 1010 or 1111 in the high-order four bits. These are called the *A-line* and *F-line* traps, respectively. Vector 4 is used for all other illegal instructions:

- Illegal op codes other than Axxx or Fxxx
- Illegal addressing modes, such as a PC-relative operand specified as an instruction destination
- Illegal addressing mode or instruction combinations, such as byte operations on address registers

Illegal instructions can be used to simulate extended 68000 instructions. The Macintosh uses the op code Axxx (the A-line trap) in this fashion. (Fxxx is also often used for this purpose on 68000 machines.) To simulate an extended instruction, first define an illegal op code pattern for this instruction. Then write an exception handler for the appropriate illegal instruction vector that simulates the action of the instruction. A program can then make use of the instruction without knowing that the instruction is simulated. There is, of course, a significant cost in terms of processing speed.

This technique is often used to simulate optional hardware extensions to a processor's instruction set, such as floating point. In this way you can write a program that uses floating-point operations and run that program on any machine, whether or not it has the floating-point hardware. The only difference is that the program will run much faster on a machine with the optional hardware.

The Macintosh uses the A-line trap for all Toolbox and operating system calls. The application views the trap as a subroutine call into the operating system. Other 68000 systems use the TRAP instructions for calling the operating system.

TRAP Exceptions

The notion of an operating-system call instruction is similar to software simulation of illegal instructions. The TRAP instructions are frequently used for operating system calls. The Macintosh does not use TRAP instructions, however. The program first issues an operating-system call instruction; then the operating system performs some function and returns an indication of its success or failure to the program.

The program can treat these operating system calls as if they were single instructions. The operating system can provide a set of functions, sometimes called the *extended instruction set,* which is the same on machines that have substantially different hardware. This is the appeal of standard operating systems, which allow the same applications software to run unchanged on many different machines.

The 68000 has 16 operating-system call instructions, TRAP 0 through TRAP 15. Illegal instructions can be used to extend this set almost infinitely. The TRAP instructions use vectors 32 through 47. The advantage of having a large number of such instructions is that the operating system can dedicate a TRAP instruction to a frequently used operating system service and reduce the number of instructions required to perform this service. Less frequently called services can be invoked by requiring the application to load a function code into a register before performing the TRAP instructions. This function code specifies which operating system routine is desired.

The TRAP instruction allows a convenient transition between user mode and supervisor mode. On 68000 systems other than the Macintosh, applications normally run in user mode, while the operating system normally runs in supervisor mode. To enter the operating system, the TRAP instruction automatically places the processor into supervisor mode.

When the RTE instruction is used to return to the user program, the user program's status register is loaded from the stack, causing the processor to go back to user mode. This mechanism also makes it possible for both supervisor-mode and user-mode programs to call the operating system, or even for the operating system to call itself.

Another advantage of the TRAP instruction is that the application program need not know the location of the operating system. Many systems require that the application program jump to some fixed address to call the operating system. The TRAP instruction allows the location of the operating system to change without affecting the application program.

Exceptions Used for Debuggers

Macintosh debuggers provide two mechanisms for controlling program execution: breakpoints and single-instruction execution. These mechanisms make use of two 68000 exceptions: the illegal-instruction and TRACE exceptions.

To set a breakpoint in a program being debugged, the most common technique is to save the instruction at the breakpoint location and place an illegal instruction at that location. The debugger then allows the program to execute until an illegal-instruction exception occurs. This technique will fail if a breakpoint is set in the middle of a multiword instruction, if the breakpoint is never executed, or if the program uses the instruction as data. Also, continuing from a breakpoint requires that the instruction at the breakpoint be single-stepped before program execution resumes.

Single-stepping an instruction involves setting the trace bit in the status register. The most common method is to stack the program counter and status register, set the trace bit in the stacked status register, and execute an RTE instruction. A TRACE exception will occur immediately following the execution of the target instruction. There are some side effects of this technique, however:

- Because an exception clears the trace bit, an exception caused by the instruction being traced causes the debugger to lose control unless the debugger receives control when exceptions occur.
- A-line instructions that call an operating-system or Toolbox function appear as a single instruction.
- Tracing an RTE instruction causes the debugger to lose control because the RTE instruction reloads the status register. Tracing an instruction that reloads the status register has the same effect. These instructions include MOVE to SR, ANDI to SR, and EORI to SR.
- Tracing a MOVE from SR instruction can cause the program to malfunction, because the trace bit will be set in the copy of the status register that the program receives. If the program compares this copy without masking the trace bit, it could execute incorrectly.

Because all Macintosh software executes in supervisor mode, you must exercise care in tracing the instructions that reload the status register.

Other Error Exceptions

The TRAPV, CHK, and zero-divide exceptions are also mechanisms that detect malfunction. These exceptions are used to assist the application in detecting problems with overflow, array subscript range, and division by zero.

The application program may need to regain control after one of these exceptions in order to print out a message that identifies the error and its cause. High-level language programs may have a way of identifying the routine and its line number in the source program. Operating systems usually have some mechanism that allows the application program to regain control following an error exception.

Privilege-Violation Exceptions

A user-mode program that attempts to execute a privileged instruction causes a privilege-violation exception through vector 8. This usually indicates an error condition. With some computer systems, however, you can use the privilege-violation exception to execute multiple supervisor-mode programs. The multiple supervisor-mode programs run in user state and a supervisor-mode monitor simulates the action of all the privileged-mode instructions. This technique is sometimes known as the *virtual machine* technique.

Virtual machines are used to run multiple operating systems on a single computer. A hardware device called a *memory management unit* (MMU) simulates different memory spaces for each operating system. This allows each system to have its own vector area as well as other dedicated memory locations. Because the operating systems run in user mode, they do not interfere with each other.

The 68000 chip is capable of running a virtual machine system with a single exception—the MOVE from SR instruction is not privileged. If software that needs to run in supervisor state uses the MOVE from SR instruction and looks at the supervisor bit, it may malfunction. On the 68010 chip, MOVE from SR is privileged.

Because all Macintosh software runs in supervisor mode, a privilege-violation exception is an indication of a software error. The normal cause for this type of exception is an RTE instruction executed with the wrong data on the stack. This loads an incorrect value into the status register, possibly putting the processor into user mode. The next privileged instruction causes a privilege-violation exception.

Interrupts

Exceptions that come from external sources are often called *interrupts.* The 68000 provides two techniques for external devices to interrupt the CPU: vectored interrupts and autovectored interrupts.

An external device can use three pins on the 68000 chip to cause an interrupt. These pins are called IPL0, IPL1, and IPL2. IPL stands for *interrupt priority level.* These three inputs to the 68000 comprise a three-bit code used to request interrupts. The special code 000 means no interrupt. Combinations 1 through 7 request an interrupt with priorities 1–7. The 68000 chip recognizes an interrupt if the *interrupt mask* (contained in status register bits 8–10) is less than the priority presented on the IPL0–2 pins. Thus, an interrupt mask of 0 allows all interrupts, and an interrupt mask of 7 allows no interrupts (except level 7).

When an interrupt occurs, the interrupt mask in the status register is set to the priority level of the interrupt. This prevents that interrupt, or any interrupt with the same priority, from recurring until the interrupt mask is set to a lower value by the exception handling routine.

Interrupt level 7 is a special case because it cannot be masked off. This interrupt level is normally reserved for extremely high-priority devices or for a "panic button" that can be used to recover from a runaway program. A level-7 interrupt is sometimes called a *nonmaskable interrupt* or NMI.

An interrupt requested at a lower level than the current processor priority remains pending until the operating system lowers the processor priority. The maximum amount of time an interrupt can remain pending is called *interrupt latency.* This time is determined by the maximum number of instructions a program can execute with interrupts masked off. Several applications, including instrumentation, industrial automation, and communications, require rapid interrupt response. In programming interrupt-driven software, there are usually restrictions on the amount of time interrupts can be disabled.

When the 68000 recognizes an interrupt, an additional control pin can be used to request an *autovectored interrupt.* Autovectored interrupts use vectors 25 through 31 for interrupt levels 1–7. If an autovectored interrupt is not requested, the 68000 reads a vector number from the device requesting the interrupt. Deciding which type of interrupt to use is the prerogative of the engineer who designs the 68000 computer system. Autovectored interrupts have the advantage of requiring less hardware (you don't need the extra control pin), but they are slower and require more sophisticated software. Vectored interrupts provide better interrupt response time at the cost of extra hardware. The Macintosh designer chose to use autovectored interrupts.

The relationship between a device's interrupt vector and its interrupt priority level is dependent on the physical connection between the 68000 and the hardware device. This relationship differs for each type (and possibly model) of computer. Interrupt programming usually requires different coding for each machine on which the program runs. (It is possible to write a fixed program that processes software exceptions, such as BUSERR, addressing error, and so on, because these are characteristics of the 68000 chip and do not change from machine to machine.)

The Macintosh hardware uses the autovectored interrupts for all external devices. The devices are connected as follows:

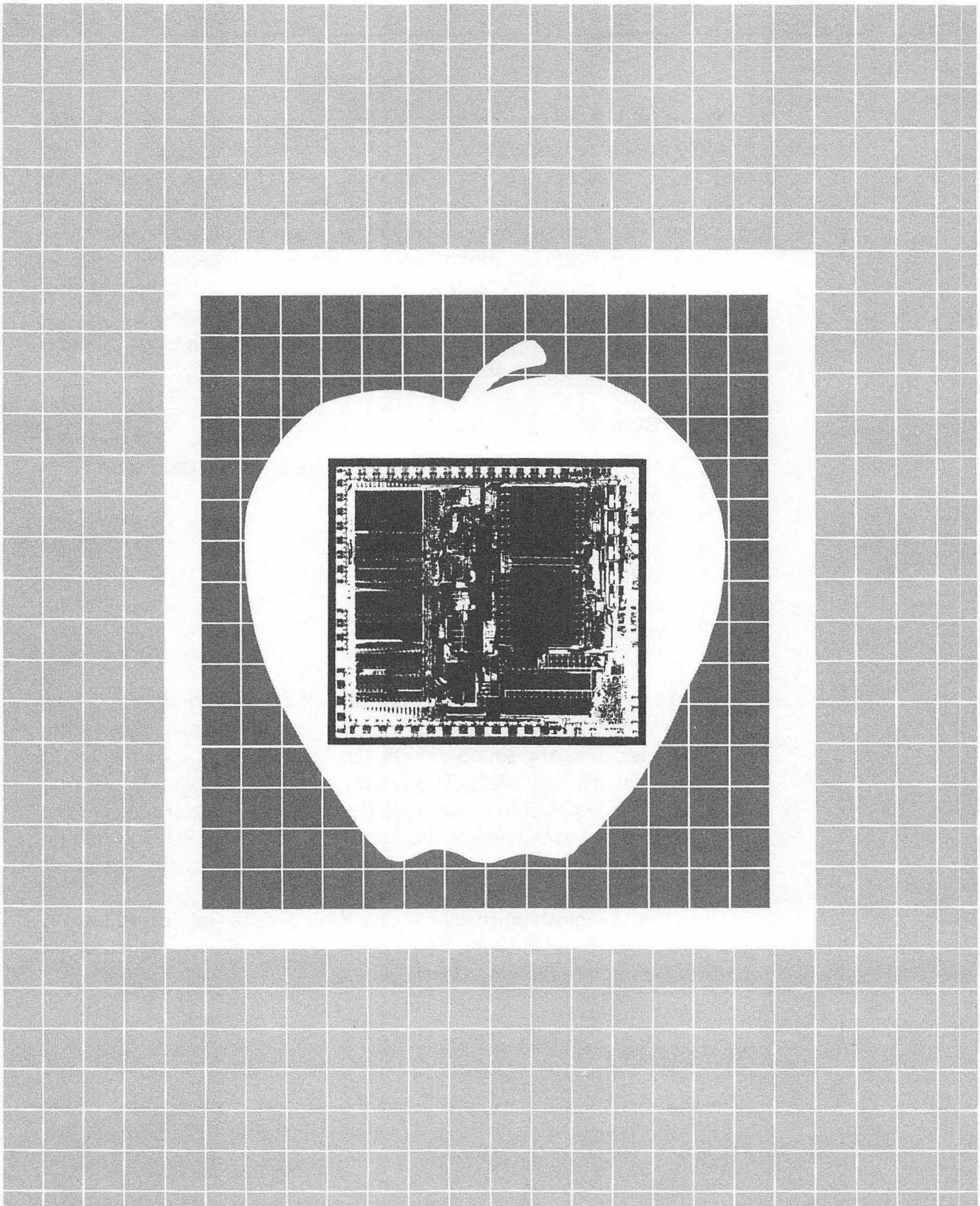
IPL	Devices
1	Clock, vertical retrace, keyboard, mouse, timer, disk, sound
2	Serial ports
3	Both 1 and 2
4	INTERRUPT button
5	Both 4 and 1
6	Both 4 and 2
7	All three devices: 4, 2, and 1

There are three IPL lines into the 68000 chip. A chip called the VIA (Versatile Interface Adapter) handles the clock, vertical retrace, keyboard, mouse, timer, disk, and sound devices. The VIA interrupt is connected to the low-order IPL line (IPL0). The serial ports are controlled by a single chip, the SCC (Serial Communications Controller). The interrupt from this chip is connected to the middle IPL line (IPL1). The INTERRUPT button on the Macintosh Programmer's Switch panel is connected to the high-order IPL line (IPL2).

Thus, the Macintosh normally has only three possible interrupts: level 1, level 2, and level 4. However, if two or more of the devices interrupt, an interrupt on one of the other levels occurs.

SUMMARY

In this chapter we have presented the exception mechanism of the 68000 and learned how it relates to the Macintosh. This information is important for writing device drivers or for debugging programs that cause exceptions.



Chapter 8

Run-Time Library Source

Don't ignore details. Lick them.

—William B. Given, Jr.

INTRODUCTION

This chapter describes the source code for the run-time library used in all of the previous programs in this book. There are two sections of this code. The first section, found in file MACLIB.ASM in Appendix E, is a set of macros that are called by the user program. The second section, found in files RTLcode.asm and RTLdata.asm in Appendix E, consists of the subroutines called by these macros.

The Macintosh program-level interface is often criticized as being too "low level." The Macintosh system designers, in their desire to give the application writer total flexibility, have created a system that requires a large number of system calls to do anything. The run-time library offers a collection of simpler calls that perform the most common combinations of the low-level calls.

There is also a large amount of code that is required as overhead in Macintosh applications. The run-time library attempts to isolate and standardize this software so that it need not be recoded for each new application.

The benefits to the application writer are twofold. First, there are large areas of the Macintosh system software with which the application writer need not be concerned. This saves time in learning how to write applications. Second, the macro calls to the run-time library are easier to code than direct calls to the Macintosh system routines. The resulting smaller programs are easier to write and modify.

The costs are also twofold. First, the application writer loses a certain amount of flexibility in writing an application. Many of the optional features available through direct calls are not available with the run-time macros. Second, there is a small memory overhead for programs that do not use all of the available features. You can eliminate these problems by modifying the source to the run-time library. By creating a custom version of the run-time code for a given application, you can change certain features to match the requirements of your application. You can also reduce memory overhead by taking out unused routines.

The run-time library is a set of reusable software components. It reduces the time required to write an application by cutting both learning time and coding time. This is accomplished through standardization. Each run-time call gives you the most common way of performing that particular function. You invoke run-time functions with a single-line macro call. You can easily rewrite run-time functions to work differently should your application require it.

PROGRAM INITIALIZATION

Programs using the run-time library begin execution with the PROGRAM macro. This macro defines the label XXSTART, which you used in linker command files. Listing 8.1 contains the PROGRAM macro.

This macro has three parameters: the name of the program, a debug flag, and the number of menus present in the program's resource file. The first instruction executed by the program is the JSR to a routine called XXINIT. This is a run-time routine that performs one-time initialization for the program. Following the JSR instruction is a Pascal-format string (lines 23–28) that gives the name of the program. If the name is not specified in the macro call, the string "No Name" is used instead. At the next even address following the name string, there is a word that contains the number of menus in the program menu bar. This word is set to zero if the macro parameter is omitted on the macro call. Following this word is another word that contains the first instruction to be executed by the program. This word contains either hex 4E71 or FF01, depending on the setting of the debug parameter. A value of FF01 invokes the debugger when the program starts. The 4E71 value (a no-operation instruction) causes the program to begin normally.

The code invoked by the PROGRAM macro resides in the run-time library. The PROGRAM macro calls the routine at label XXINIT. This routine performs the following functions:

1. It initializes the Macintosh Toolbox routines.
2. It opens the resource file.

3. It builds and displays the menu bar.
4. It creates a window on the screen.
5. It adds scroll bars to the window.
6. It creates a text record for the window.

```
;        PROGRAM macro.  This macro is used to define an application.
;        It must be the first code-generating statement in the program.
;
;        Usage:
;
;        PROGRAM name,debug,menus
;
;        Where "name" is the application's name (used to find the
;        resource file, for instance).
;
;        "Debug" is either "DEBUGON" or "DEBUGOFF", to make the program's
;        first instruction an illegal instruction ($FF01) or a NOP.
;        This may later be patched.
;
;        "Menus" is the number of menus in the main menu bar, or the
;        "DEFAULT" token.
;
         STRING_FORMAT 3         ; Set all strings to Pascal standard
         .MACRO  PROGRAM         ; name,menus,debug =
         XDEF    XXSTART         ; Entry point name
         XREF    XXINIT          ; Initializer
XXSTART  JSR     XXINIT          ; Call init routine
         IF      '%1' <> ''      ; Name specified
         DC.B    '%1'            ; Yes
         ELSE                    ;
         DC.B    'No Name'       ; Default
         ENDIF
         .ALIGN  2               ; Force word alignment
         IF      '%3' <> ''      ; Menus?
         DC.W    %3              ; Do specified number
         ELSE                    ; Not specified
         DC.W    0               ; Use default
         ENDIF
         IF      '%2' <> ''      ; Debug specified?
         DC.W    %2              ; Yes
         ELSE                    ; Not specified
         DC.W    $4E71           ; Use NOP instruction
         ENDIF
         .ENDM
DEFAULT  EQU     0               ; Default parameter
DEBUGON  EQU     $FF01           ; Debug option
DEBUGOFF EQU     $4E71           ; No debug option
```

***Listing 8.1** – The PROGRAM macro*

Initializing the Toolbox Routines

Listing 8.2 shows the code required to initialize the Macintosh Toolbox routines. The debug instruction at line 2 is used for debugging changes to the XXINIT routine. Note that the semicolon at the beginning of the line deletes this instruction. A breakpoint in the PROGRAM macro is executed only after the initialization code is run. Removing the semicolon at the beginning of line 2 allows use of the debugger before any initialization code runs.

Lines 3–4 initialize the QuickDraw graphics package. QuickDraw requires a scratch area in memory. The linker reserves the first 256 bytes of the uninitialized data area for this purpose. The PEA instruction specifies the address to be used. QuickDraw uses offsets −205 to +3 from the address you specify. This call was not documented correctly in early versions of *Inside Macintosh*.

Lines 5–7 initialize the sections of the Macintosh ROM that handle fonts, windows, and menus, respectively. Lines 8–9 initialize the dialog manager, which handles alerts and dialogs. The longword parameter to this routine is the address of a restart procedure. If the program causes a system error and the user selects Restart from the dialog box, this procedure will be called. Using a zero for this parameter means that the Restart selection will be disabled, and the user must reboot after a system error.

Line 10 initializes the Text Edit routines in ROM. Text Edit is used by the PRINT and INPUT macros, as well as by the dialog manager.

Opening the Resource File

The run-time library supports either a combined executable file, containing both code and other resources, or separate files, one for code and

```
 1: XXINIT:                          ;    Called by PROGRAM macro
 2: ;         DC.W    $FF01          ;    For debugging only!!
 3:           PEA     -4(A5)         ;    QuickDraw scratch area
 4:           _InitGraf              ;    Init QuickDraw
 5:           _InitFonts             ;    Init font package
 6:           _InitWindows           ;    Init windows package
 7:           _InitMenus             ;    Init menu package
 8:           CLR.L   -(SP)          ;    No restart procedure
 9:           _InitDialogs           ;    Init dialog package
10:           _TEInit                ;    Init text edit package
```

Listing 8.2 *- Toolbox initialization* the other for all the other resources. You will normally use separate files to save time while a program is being developed, combining them when the program is completed.

For combined resource files, the run-time library does not try to open a separate resource file. You can remove the code that opens the resource file with conditional assembly in applications that don't use a separate resource file. The symbol RESFILE should be equated to zero if the code is to be included and nonzero if the code is to be excluded.

For separate code and resource files, the code in Listing 8.3 opens the appropriate resource file. This code tries opening two files: Program.RSRC and RTL.RSRC. Program is the name specified in the PROGRAM macro call.

Lines 1–8 build a file name string consisting of the program name and the ending .RSRC. An area called Scratch is used to build the name. Lines 9–15 move register A2 to point to the word containing the number of program menus. This is outside of any conditional assembly, as it must be done in all cases. Lines 16–21 attempt to open the constructed file name. The routine called to open the resource file scans all mounted volumes

```
 1:          IF       RESFILE = 0      ;     Only if resource file separate
 2:          MOVE.L   (SP),A2          ;     A2 -> Name string
 3:          LEA      Scratch(A5),A0   ;     A0 -> Target count byte
 4:          LEA      1(A0),A1         ;     A1 -> Destination
 5:          BSR      MoveCnt          ;     Move in count-delimited string
 6:          LEA      RSRC,A2          ;     A2 -> ".RSRC" string
 7:          BSR      MoveCnt          ;     Move in this field
 8:          ENDIF                     ;     Part 1 of resource conditional
 9:          MOVE.L   (SP),A2          ;     Fetch string address again
10:          CLR.L    D7               ;     Load long
11:          MOVE.B   (A2),D7          ;                unsigned byte
12:          MOVE.L   D7,D6            ;     Save this value
13:          ADDQ.L   #2,D7            ;     Adjust to
14:          BCLR.L   #0,D7            ;                word boundary
15:          ADD.L    D7,A2            ;     A2 -> Menu word
16:          IF       RESFILE = 0      ;     Start part 2 of resource conditional
17:          ADD.W    #RSRCLEN,D6      ;     Add in ".RSRC" length
18:          MOVE.B   D6,(A0)          ;     Update length count
19:          BSR      OpenRsrc         ;     Try to open file
20:          MOVE.W   D0,ResFD(A5)     ;     Save it for later
21:          BPL      ResOK            ;     If > 0, then file opened OK
22:          LEA      Canned,A0        ;     Try default name
23:          BSR      OpenRsrc         ;     Attempt open
24:          MOVE.W   D0,ResFD(A5)     ;     Save it for later
25:          BMI      ResNFG           ;     Couldn't open, quit
26: ResOK:                             ;     Resource file open
27:          ENDIF                     ;     RESFILE = 0
```

***Listing 8.3** – Opening an external resource file* for the file. Should this open fail, the code in lines 22–27 will attempt to open the file RTL.RSRC. This feature allows you to write programs that use a single standard resource file, like the programs in Chapter 4.

Setting Up the Menu Bar

The next step in the initialization process is setting up and displaying the menu bar. Menus are numbered from 1 on the menu bar. The RTL code forces each menu description in the resource file to have a resource ID equal to the position of the menu in the menu bar. The standard Macintosh menu scheme applies. Menu number 1 has the "About" description and the desk accessories. Menu number 2 is the File menu, which must contain a Quit choice. Menu number 3 is the Edit menu, which must contain Undo, Cut, Copy, and Paste choices. All other menus are application specific. The menu initialization code is shown in Listing 8.4.

Lines 1–5 load the number of menus in D6 and the starting menu resource ID in D7. By default, there are three menus, the Apple, File, and Edit menus. As explained above, the first menu has a resource ID of 1.

```
 1:          MOVE.W  (A2)+,D6         ;       Fetch # of menus in resource file
 2:          BNE     GotMenus         ;       NE => Default not chosen
 3:          MOVE.W  #DEFMENU,D6      ;       Select default
 4: GotMenus:                         ;
 5:          MOVE.W  #MENURS,D7       ;       Menu resource number
 6: MenuLoop:                         ;       Loop on menus
 7:          CLR.L   -(SP)            ;       Clear space for menu handle
 8:          MOVE.W  D7,-(SP)         ;       Push resource number
 9:          _GetRMenu                ;       Get menu handle
10:          CLR.W   -(SP)            ;       Insert after flag
11:          _InsertMenu              ;       Insert current menu
12:          ADDQ.L  #1,D7            ;       Bump resource number
13:          CMP.W   D6,D7            ;       Done?
14:          BLE     MenuLoop         ;       Not yet, repeat until all inserted
15: ;
16: ;        Add desk accessories into first menu (Apple)
17: ;
18:          CLR.L   -(SP)            ;       Room for menu handle result
19:          MOVE.L  #'MENU',-(SP)    ;       Indicate resource type
20:          MOVE.W  #MENURS,-(SP)    ;       Menu #1
21:          _GetResource             ;       Fetch menu handle
22:          MOVE.L  #'DRVR',-(SP)    ;       Accessory type
23:          _AddResMenu              ;       Add to menu
24:          _DrawMenuBar             ;       Paint bar on screen
```

Listing 8.4 – *Menu initialization code*

The loop from lines 6–14 is executed once per menu in the resource file. The _GetRMenu reads a menu from the resource file into memory, returning a handle on the stack. The _InsertMenu call takes a menu handle and a menu number. Note that the handle returned by _GetRMenu is passed directy to _InsertMenu without popping it off the stack. This coding technique saves a few bytes of memory. The second parameter to _InsertMenu is a menu resource ID of a menu already in the menu bar. The menu being inserted will appear before the menu that is already present. Specifying a zero menu ID means that the menu being inserted will be inserted following all other menus in the menu bar. The RTL code uses this feature to build the menu bar in menu resource ID order.

Lines 18–23 add the desk accessories to the first menu. The _GetResource call returns a handle to this menu, and the _AddResMenu appends the names of all resources with the ID DRVR to this menu. This resource type is normally found only in the System resource file. Note that a _GetResource call is used to find the menu handle, instead of a _GetRMenu call. *Inside Macintosh* recommends that _GetRMenu be called only once per menu.

At line 24, the menu initialization is complete. The _DrawMenuBar call draws the menu bar on the screen.

Creating a Window

Following menu bar initialization, the next step is to create a document window. The RTL procedure for doing this is shown in Listing 8.5.

```
CLR.L   -(SP)              ;     Room for window pointer
MOVE.W  #WINDOWR,-(SP)     ;     Window resource number
PEA     WArea(A5)          ;     -> Window space
MOVE.L  #ONTOP,-(SP)       ;     Code for on top of others
_GetNewWindow              ;     Create the window
MOVE.L  (SP),A3            ;     A3 -> window record
MOVE.L  A3,WindowPtr(A5);        Save window record address
_SetPort                   ;     Make window the Q/D current port
MOVE.W  #monaco,-(SP)      ;     Set font to Monaco
_TextFont                  ;
MOVE.W  #9,-(SP)           ;     Use 9 point (80 cols on MAC)
_TextSize                  ;
MOVE.L  (SP)+,A0           ;     Get string address from stack
BSR     XXWTITLE           ;     Set the window title
```

Listing 8.5 *– Creating a new window*

The window description is contained in the resource file. The _GetNewWindow call takes a window resource ID, a memory area address, and a variable that indicates whether the window is to be in front of or behind existing windows on the screen. This call draws the window on the screen and returns a pointer to be used in other calls that reference the window.

The memory area must be word-aligned and 292 bytes long. If you pass a null (zero) pointer for the storage area, the window manager will allocate the storage dynamically. The window storage area will then be locked in memory, however. This can cause memory to become fragmented. Fragmented memory can cause your program to run out of memory prematurely. You should therefore avoid letting the window manager allocate the window storage dynamically by allocating space for the window in your program. Use a DS directive to allocate this space in the global area, so it will not be stored in the application file.

The positioning parameter is the window behind which the new window will appear. To place the new window in front of all others, use a value of −1. A value of 0 places the window behind all other windows. A value of 1 places the new window behind the top-most window on the screen. A value of 2 places the new window behind two other windows, and so on.

Lines 6–8 of Listing 8.5 copy the window pointer to a location in the RTL data area and set the QuickDraw port to the new window. Note that the return parameter from the _GetNewWindow call is left on the stack to serve as an input parameter for the _SetPort call.

Lines 9–12 set the text font and size to 9-point Monaco. Monaco is a fixed-width font, and the 9-point size gives approximately 80 screen columns. This simplifies screen output for small programs. The application can change these values at any time, so subsequent routines in the runtime library do not rely on them.

Lines 13–14 cause the name given in the PROGRAM macro to appear as the window title. The address on the stack is the return address from the JSR XXINIT instruction in the PROGRAM macro. The XXWTITLE routine will be described later.

Attaching Controls to a Window

A normal document window has four controls: a vertical scroll bar, a horizontal scroll bar, a grow box, and a close box. The close box is drawn by the _GetNewWindow call. The other controls must be installed by the application. Listing 8.6 shows the RTL code for installing controls on an existing window.

Lines 2–6 create the vertical scroll bar. The _GetNewControl call adds a new control to an existing window. This routine takes the control resource ID and the window address as input. It returns a handle to the new control. Lines 7–11 create the horizontal scroll bar in a similar fashion.

The grow box is a square in the lower-right corner of the window. Lines 12–13 draw the box. The _DrawGrowIcon call takes only the window address as a parameter.

Line 15 calls a routine to determine the size of the window. The application needs to know the original size to correctly display any initial information. Listing 8.7 shows the SizeW routine.

A window data structure has a handle to a region that describes the content area of the window. Within the region data structure there is a rectangle that gives the size of the window's content region (including the scroll bars and grow box). Lines 2–5 isolate this rectangle into registers D0 and D1. If the window has controls, lines 7–10 correct the rectangle to exclude the controls.

Lines 12–17 convert the rectangle from global coordinates (bitmap data structure) to local coordinates (GrafPort data structure). The converted rectangle is stored in a local area. Lines 18–20 set the GrafPort's clipping rectangle to the area inside the controls. This prevents the application from drawing over the control area.

As you will see later, the clipping region must be set to include the controls when you are altering them. However, normal drawing requires that the clipping region exclude the controls.

Text Edit Initialization

A program that uses the PRINT and INPUT macros must create a Macintosh Text Edit record. Use of the Text Edit routines simplifies the PRINT and INPUT support code. The Text Edit record is a data structure required by the Text Edit routines. The code that creates and activates the Text Edit record is shown in Listing 8.8. A Text Edit record is created using the _TENew call, which requires a destination rectangle and a view rectangle. For our purposes, these rectangles are the local variables DestRect and ViewRect created in Listing 8.7. Lines 6–8 set a special field in the new Text Edit record, which disables wrapping long lines. The scrolling code in the PRINT macro depends on this. Lines 9–10 activate the Text Edit record.

Final Initialization

Listing 8.9 shows the code that executes prior to starting the user program. The first two lines get rid of all events in the event queue in order to make behavior during startup more predictable.

When the Finder launches an application, the cursor is set to the watch symbol. The _InitCursor call at line 3 changes the cursor to the standard arrow symbol. The location xCursor contains the ID of the present cursor. A zero indicates the arrow. The location Cursor indicates the cursor to be

```
        IF      CONTROL = 0     ;       If window controls
        CLR.L   -(SP)           ;       Space for result
        MOVE.W  #VscrID,-(SP)   ;       Push resource ID
        MOVE.L  A3,-(SP)        ;       Push window pointer
        _GetNewControl          ;       Add control to window
        MOVE.L  (SP)+,Vscroll(A5);      Save VScroll handle
        CLR.L   -(SP)           ;       Space for result
        MOVE.W  #HscrID,-(SP)   ;       Push resource ID
        MOVE.L  A3,-(SP)        ;       Push window pointer
        _GetNewControl          ;       Add control to window
        MOVE.L  (SP)+,Hscroll(A5);      Save HScroll handle
        MOVE.L  A3,-(SP)        ;       Push window pointer
        _DrawGrowIcon           ;       Draw grow box
        ENDIF                   ;       CONTROL = 0
        BSR     SizeW           ;       Size the window
```

Listing 8.6 *– Adding controls to a window*

```
SizeW   MOVEM.L D0-D2/A0-A3,-(SP);      Save registers we use
        MOVE.L  WindowPtr(A5),A3;       A3 -> window structure
        MOVE.L  contRgn(A3),A0  ;       A0 -> content region (handle)
        MOVE.L  (A0),A0         ;       A0 -> content region (pointer)
        MOVEM.L RgnBBox(A0),D0/D1;      D0,D1 = GrafPort rectangle
        IF      CONTROL = 0     ;       If doing control stuff
        SUB.W   #ScrollW,D1     ;       Subtract
        SWAP    D1              ;               off
        SUB.W   #ScrollW,D1     ;                   scroll
        SWAP    D1              ;                           region
        ENDIF                   ;       CONTROL = 0
        MOVEM.L D0/D1,DestRect(A5);     Set dest rect
        PEA     DestRect(A5)    ;       Convert
        _GlobalToLocal          ;               to local coords
        PEA     DestRect+4(A5)  ;       Both points
        _GlobalToLocal          ;                   in rectangle
        MOVEM.L DestRect(A5),D0/D1;     Get normalized rectangle
        MOVEM.L D0/D1,ViewRect(A5);     Store as view rectangle
        PEA     ViewRect(A5)    ;       Set
        _ClipRect               ;               clip region
        MOVEM.L (SP)+,D0-D2/A0-A3;      Restore the registers
        RTS                     ;       Back to caller
```

Listing 8.7 *– Computing the size of a window* used inside the window on the screen. This is initialized to the I-beam symbol at line 7.

When the user program starts, the window pointer is in register A0 and the window size is in register D0. Lines 8–9 set up these values. The JMP instruction at line 10 returns to the debug word in the PROGRAM macro. Following the execution of that word, the application begins.

RUN-TIME LIBRARY CONVENTIONS

The rest of the run-time library, after the initialization code, is called by macros in the user program. These macros follow certain conventions:

1. There are at most two return values: in D0 and D1.
2. All registers that don't return values are preserved.
3. Errors are indicated by a negative value in D0.

```
        IF      TEXTRTL = 0     ;       If no text, don't create TE
        CLR.L   -(SP)           ;       Space for text handle result
        PEA     DestRect(A5)    ;       -> Destination rectangle area
        PEA     ViewRect(A5)    ;       -> View rectangle area
        _TENew                  ;       New text record
        MOVE.L  (SP),A3         ;       A3 -> -> text record
        MOVE.L  (A3),A3         ;       Dereference
        MOVE.B  #-1,teCROnly(A3);       Specify no wrap mode
        MOVE.L  (SP),THandle(A5) ;      Store text handle
        _TEActivate             ;       Activate text record
        ENDIF
```

Listing 8.8 *– Creating a Text Edit record*

```
 1:     MOVE.L  #$0000FFFF,D0   ;       Mask for all events
 2:     _FlushEvents            ;       Get rid of all events
 3:     _InitCursor             ;       Make cursor the arrow
 4:     CLR.W   xCursor(A5)     ;       Indicate last cursor set
 5:     MOVE.L  (SP)+,EXITAddr(A5);     Save Finder's return address
 6:     MOVE.L  SP,ORGSP(A5)    ;       and stack pointer contents
 7:     MOVE.W  #IBeam,Cursor(A5);      Set I-beam cursor
 8:     MOVE.L  WindowPtr(A5),A0;       Set return register
 9:     MOVE.L  ViewRect+4(A5),D0;      Return window size, also
10:     JMP     (A2)            ;       return to application
```

Listing 8.9 *– Final initialization code*

Values are passed to the run-time routines in registers. This means that the macro used to invoke a routine must save and restore the registers used to pass parameters.

The external names for the run-time routines begin with XX and end with the macro name. Thus, the routine for the OPEN macro is named XXOPEN. This avoids conflict with user-selected names.

User-callable routines inside the run-time library usually begin with a SAVE macro, which stores all the registers in a global memory area. To return to the user, one of three macros is used: the RETURN macro (return value in D0), the RETURNP macro (return values in D0 and D1), or the RETURNALL macro (no return value). Listings for these macros can be found in Appendix E.

The run-time services can be grouped into the following four categories:

1. Interaction routines, such as PRINT and INPUT
2. Event routines, such as GETEVENT and CHKABORT
3. File routines, such as OPEN and READ
4. Menu routines, such as CHECK and STYLE

The remainder of this chapter will be devoted to explaining the macros and run-time code that implement these services.

INTERACTION ROUTINES

The interaction routines consist of all the run-time services that can alter the appearance of the screen:

1. ALERT, STOPALERT, NOTEALERT, CAUTIONALERT
2. DIALOG
3. INPUT
4. PRINT
5. SETVMIN, SETVMAX, SETVVAL, SETHMIN, SETHMAX, SETHVAL
6. SCURSOR

ALERT

All the ALERT macros have nearly identical coding. Listing 8.10 shows the ALERT macro. This macro invokes a Macintosh trap routine directly.

There is no supporting routine in the run-time library. The parameters to the macro are the resource ID of the alert template and an optional procedure address that is called for every event received while the alert is in progress.

The STOPALERT, NOTEALERT, and CAUTIONALERT macros are identical, except for the trap number. Trap numbers are given in Table 8.1.

The value returned by the trap (on the stack) is the item number in the alert template that was selected by the user. This number is placed in D0 to be compatible with the other run-time macros.

DIALOG

The DIALOG macro is similar to the ALERT macros in that it creates a display on the screen and returns an item number in D0. The DIALOG macro does not permit an event-filter procedure, however. Listing 8.11 shows the DIALOG macro.

The macro puts the resource ID of the DIALOG template into D0.W and calls the run-time routine XXDIALOG. Listing 8.12 shows the code for XXDIALOG.

Lines 3–7 create a new dialog record. The _GetNewDialog call takes a dialog template resource ID, a memory area for the dialog record, and a positioning parameter. This call is very similar to the _GetNewWindow call described earlier. Note that a zero is used for the memory area, indicating that it is to be allocated dynamically. This will not cause problems because the storage will be deallocated almost immediately. The _GetNewDialog call returns the address of the dialog record.

Line 9 sets the QuickDraw active port to the newly allocated dialog record. Note that again we save an instruction by leaving the return value

```
        .MACRO  Alert
        MOVEM.L D1-D2/A0-A1,-(SP) ;      Save registers
        CLR.W   -(SP)             ;      Room for result
        MOVE.W  %1,-(SP)          ;      Push resource ID
        IF      '%2' <> ''        ;      If procedure specified
        PEA     %2                ;      Load pointer
        ELSE                      ;      No procedure
        CLR.L   -(SP)             ;      Push 0 long
        ENDIF                     ;
        DC.W    $A985             ;      _Alert trap
        MOVE.W  (SP)+,D0          ;      Result to D0
        MOVEM.L (SP)+,D1-D2/A0-A1 ;      Unsave registers
        .ENDM
```

***Listing 8.10** – The ALERT macro*

from one call on the stack to serve as a parameter to the next call.

If the Text Edit routines are in use, they must be deactivated before the dialog can take place. This is done in lines 11–12.

The actual dialog takes place in lines 14–16. The _ModalDialog call takes a filter procedure and the address of a word as parameters. The word is filled in with the item number selected by the user. The dialog record must represent the top-most window on the screen.

Lines 17–18 free up the dialog storage. Line 19 places the return value in register D0. Lines 20–21 reset the active QuickDraw port to the standard window. Activate and update events for the window will be generated automatically. This causes the application to redraw the area of the screen covered by the dialog box.

INPUT

The INPUT macro (Listing 8.13) reads a line of text from the keyboard. The requesting program receives this text in the form of a Pascal string.

The macro loads the input area address into A0 and the length of the area into D0. If no length is specified, 80 (decimal) is assumed.

The macro calls the run-time routine XXINPUT to read the line from the keyboard. Listing 8.14 contains the code for the XXINPUT routine.

Macro	**Trap Name**	**Trap Number**
ALERT	_Alert	$A985
STOPALERT	_StopAlert	$A986
NOTEALERT	_NoteAlert	$A987
CAUTIONALERT	_CautionAlert	$A988

***Table 8.1** – Alert Trap Numbers*

```
        .MACRO  DIALOG
        MOVE.W  %1,D0           ;       Load ID
        XREF    XXDIALOG        ;       For the linker
        JSR     XXDIALOG        ;       Call RTL routine
        .ENDM
```

***Listing 8.11** – The DIALOG macro*

Lines 4–6 ensure that the byte count parameter is a legal value. The character count for a Pascal string cannot exceed one byte (0–255). This code simply replaces a value greater than 255 with 255.

Lines 8–12 place the buffer address in A2 and the count in D4. These registers are not modified across system calls. They are called *nonvolatile* registers. Register A3 is used as a pointer to the next byte that will receive a character. Register D3 is used to count the number of characters placed in the buffer.

Line 14 gets a keyboard event from the event queue. The operation of the NEXTEVT macro will be described in "The Event System" section of this chapter. Line 15 rejects null events. A null event is an indication that no character is present. Lines 16–17 reject characters typed with the Command key down. Note that all other events (such as menu selection) are ignored by this code.

Lines 19–26 handle backspacing. The Backspace key on the Macintosh generates a character with the value 8. Characters less than backspace are rejected by lines 19–20. Lines 22–23 reject a backspace character when there are no characters in the buffer (that is, the user tried to back up past the beginning). Lines 24–26 handle a valid backspace character by deleting the last character typed.

Lines 28–29 check the character to see if it is a carriage-return (decimal 15) character. If the character is not a carriage return and there is room in

```
XXDIALOG                              ;
          MOVEM.L D1/D6/A0-A1,-(SP);      Push volatile registers
          CLR.L   -(SP)            ;      Dialog pointer return area
          MOVE.W  D0,-(SP)         ;      Dialog box resource ID
          CLR.L   -(SP)            ;      No dialog storage
          MOVE.L  #ONTOP,-(SP)     ;      On top of all
          _GetNewDialog            ;      Create a new dialog box
          MOVE.L  (SP),D6          ;      Handle to D6
          _SetPort                 ;      Make it QuickDraw port
          IF      TEXTRTL = 0      ;      If text style RTL
          MOVE.L  THandle(A5),-(SP) ;     Deactivate
          _TEDeActivate            ;              text window
          ENDIF                    ;      Text
          CLR.L   -(SP)            ;      No filter procedure
          PEA     ItemHit(A5)      ;      -> Item hit data
          _ModalDialog             ;      Wait for "OK" button
          MOVE.L  D6,-(SP)         ;      Push dialog box handle
          _DisposDialog            ;      And close it
          MOVE.W  ItemHit(A5),D0   ;      Load return register
          MOVE.L  WindowPtr(A5),-(SP);    Reset
          _SetPort                 ;              port address
          MOVEM.L (SP)+,D1/D6/A0-A1;      Restore registers
          RTS                      ;      Back to caller
```

Listing 8.12 *– The XXDIALOG routine* the buffer, lines 34–49 put the character in the user's buffer and on the screen. The _TEKey system call takes a Text Edit handle (returned from _TENew) and a character. This call adds the character into the Text Edit display so that PRINT and INPUT statements can be used together.

Lines 40–46 store the character count in the first byte of the user's buffer. The unused part of the buffer is filled with binary zeros. Lines 47–48 print a carriage return on the screen, using the code for the PRINT macro.

PRINT

The PRINT macro displays a string (in Pascal format) on the screen. This macro is shown in Listing 8.15. The address of the string to be printed is loaded into register A4. The macro then calls the routine XXPRINT to output the string to the screen using the Text Edit routines. If the second parameter is omitted, the macro again calls XXPRINT with A4 pointing to a string containing only a carriage return.

The XXPRINT routine is shown in Listing 8.16. This routine uses the Text Edit routines to print an entire string. It is surprising how much of this code is required to support scrolling. Note that the code is included in the run-time library only if the Text Edit package is in use. Applications that do not use Text Edit (such as MacDoodle) cannot use the PRINT macro.

Lines 4–14 count the number of carriage returns in the string. If there is none, the code at lines 15–16 causes the scrolling code to be skipped. Lines 17–22 compute in D5 the height of the window in pixels. Line 23 determines how many lines will fit in the window by dividing the height in pixels (from the window record) by the number of pixels per line (from the Text Edit record). Lines 24–28 determine whether the string will fit on the screen without scrolling. If the string will not fit, lines 29–34 scroll the screen an appropriate number of pixels. The _TEScroll call takes three

```
        .MACRO  INPUT           ;Address,length =
        XREF    XXINPUT         ;Define for linker
        MOVEM.L A0/D0,-(SP)     ;Save parameter registers
        LEA     %1,A0           ;-> Return buffer
        IF      '%2' = ''       ;Not specified?
        MOVEQ.L #80,D0          ;Default to 80
        ELSE                    ;Use specified
        MOVE.L  %2,D0           ;
        ENDIF                   ;
        JSR     XXINPUT         ;Call RTL routine
        MOVEM.L (SP)+,A0/D0     ;Restore registers
        .ENDM
```

Listing 8.13 *– The INPUT macro*

```
         IF      TEXTRTL = 0     ;         If text style RTL
         XDEF    XXINPUT
XXINPUT  SAVE                    ;         Save user's registers
         CMP.W   #255,D0         ;         D0 > 1 byte?
         BLE     CountOK         ;         No, count is OK
         MOVE.W  #255,D0         ;         Use only 255
CountOK                          ;
         MOVE.W  D0,D4           ;         Preserve in nonvolatile reg
         MOVE.L  A0,A2           ;         Save buffer pointer
         MOVE.L  A2,A3           ;         in nonvolatile registers
         TST.B   (A3)+           ;         -> First character position
         CLR.W   D3              ;         = Character count so far...
InputLoop                        ;         Here to get next character
         NEXTEVT #KeyMask        ;         Get next keyboard event
         BNE     InputLoop       ;         No, just ignore
         BTST    #ComKey,Modify(A5);       Command key down?
         BEQ     InputLoop       ;         Yes, ignore
         MOVE.W  Message+2(A5),D0;         Character to D0
         CMP.B   #BS,D0          ;         Backspace?
         BLT     InputLoop       ;         Invalid character
         BNE     NotBS           ;         Check for control chars
         TST.W   D3              ;         Any characters?
         BEQ     InputLoop       ;         No, invalid character
         SUBQ.W  #1,D3           ;         Decrement count
         TST.B   -(A3)           ;         Remove from buffer
         BRA     DoKey           ;         Do TEKey for screen update
NotBS:                           ;         Here to see if control character
         CMP.B   #CR,D0          ;         Was it carriage return?
         BEQ     IsCR            ;         Yes, store count and return
         CMP.B   #BL,D0          ;         Less than space?
         BLT     InputLoop       ;         Yes, error
         CMP.W   D3,D4           ;         Room in buffer?
         BLE     InputLoop       ;         No, don't do anything
         MOVE.B  D0,(A3)+        ;         Store in buffer
         ADDQ.L  #1,D3           ;         Increment count
DoKey:   MOVE.W  D0,-(SP)        ;         Now put on screen
         MOVE.L  THandle(A5),-(SP);
         _TEKey                  ;         and in TE buffer
         BRA     InputLoop       ;         Get next character
IsCR:                            ;         Here when CR typed
         MOVE.B  D3,(A2)         ;         Store the count byte
         SUB.W   D3,D4           ;         How many bytes remain?
         BLE     PrintCR         ;         None, bail out
         SUB.W   #1,D4           ;         Adjust for DBRA
ZapBuf:  CLR.B   (A3)+           ;         Clean out
         DBRA    D4,ZapBuf       ;                rest of buffer
PrintCR  LEA     XXCR,A4         ;         A4 -> CR sequence
         JMP     DoPrint         ;         Print CR, then return
         ENDIF                   ;         TEXTRTL
```

***Listing 8.14** – The XXINPUT routine* parameters: the number of horizontal pixels to scroll, the number of vertical pixels to scroll, and the Text Edit record handle returned by _TENew. Positive values for the number of pixels to scroll move the text right or down. Negative values move the text left or up. In the XXPRINT code, the value in D5 at line 27 will be negative, causing the text on the screen to scroll upwards.

After the TEScroll call, the screen has the correct appearance. However, the text that was scrolled off the top of the screen is still in memory. There is no support in the Text Edit package for deleting lines that scroll off the top of the screen. This means that a program that does a lot of PRINT calls will eventually run out of memory. The code at lines 35–57 deletes the lines that were scrolled off the top of the screen. The Text Edit package has two rectangles associated with a given Text Edit record: a destination rectangle, which encloses all the text in the record, and a view rectangle, which encloses only the text visible on the screen. Lines 35–38 reset the destination rectangle to the view rectangle.

The Text Edit record stores the character offset of the beginning of each line in successive words, starting at offset TELines. The text itself is stored separately. Lines 39–53 delete the text scrolled off the top of the screen from the text buffer. Lines 54–57 call the _TESetText routine to replace the old text with the shortened text.

After the scrolling is complete, lines 59–62 output the text. The _TEInsert call inserts a string of characters into a Text Edit record and onto the screen. This call has three parameters: the string address, the number of characters, and the Text Edit record handle.

SCURSOR

The SCURSOR macro is used to change the shape of the mouse cursor on the screen. This macro takes a single parameter: the resource ID of the

```
        .MACRO  PRINT           ;String,crflag =
        XREF    XXPRINT         ;Define for linker
        MOVE.L  A4,-(SP)        ;Save A4
        LEA     %1,A4           ;A4 -> output string
        JSR     XXPRINT         ;Call output string
        IF      '%2' = ''       ;If carriage return desired
        XREF    XXCR            ;External string
        LEA     XXCR,A4         ;A4 -> carriage-return sequence
        JSR     XXPRINT         ;Output it
        ENDIF
        MOVE.L  (SP)+,A4        ;Restore A4
        .ENDM
```

Listing 8.15 *– The PRINT macro*

```
         IF      TEXTRTL = 0     ;      Junk whole routine if graphics
         XDEF    XXPRINT         ;      Define for linker
XXPRINT  SAVE                    ;      Save registers
DoPrint  CLR.L   D6              ;      Clear a temporary
         MOVE.B  (A4)+,D6        ;      Fetch count byte
         MOVE.L  A4,A1           ;      A1 -> Text
         MOVE.L  D6,D0           ;      D0 = Count
         SUBQ.L  #1,D0           ;      Adjust for DBRA count
         BLT     TYPEDone        ;      Done if < 0
         CLR.L   D7              ;      Initialize counter
LookCR:  CMPI.B  #CR,(A1)+       ;      Is it a <CR>?
         BNE     TYPEDBR         ;      No, don't bump count
         ADDQ.L  #1,D7           ;      Bump count
TYPEDBR  DBRA    D0,LookCR       ;      Repeat till count exhausted
         TST.W   D7              ;      Any line control chars?
         BEQ     DoType          ;      No, skip next
         MOVE.L  THandle(A5),A1  ;      A1 -> Text handle
         MOVE.L  (A1),A1         ;      A1 -> Actual text record
         CLR.L   D5              ;      Clear out D5 long
         MOVE.L  WindowPtr(A5),A0;      A0 -> Window record
         MOVE.W  PortRect+4(A0),D5;     D5 = Rectangle
         SUB.W   PortRect+0(A0),D5;                     size (vertical)
         DIVS    teLineHite(A1),D5;     D5 = # lines in dest rectangle
         SUB.W   teNLines(A1),D5 ;      How many already?
         BGT     GotSome         ;      Still some left at bottom
         CLR.L   D5              ;      Correct for none left
GotSome  SUB.W   D7,D5           ;      How many lines do we need?
         BGE     DoType          ;      None
         MOVE.W  D5,D0           ;      Copy to scratch register
         MULS.W  teLineHite(A1),D0;     Multiply to convert to pixels
         CLR.W   -(SP)           ;      No horizontal scroll
         MOVE.W  D0,-(SP)        ;      Push vertical scroll
         MOVE.L  THandle(A5),-(SP);     Push text handle
         _TEScroll               ;      Scroll it
         MOVE.L  THandle(A5),A1  ;      A1 -> Text handle
         MOVE.L  (A1),A1         ;      A1 -> Actual text record
         MOVE.L  teViewRect(A1),teDestRect(A1);  Adjust vertical view for deletion
         MOVE.L  teViewRect+4(A1),teDestRect+4(A1);      "
         NEG.W   D5              ;      D5 = # lines to delete
         SUB.W   D5,teNlines(A1) ;      Mark as deleted
         ADD.W   D5,D5           ;      Convert to index
         MOVE.W  teLines(A1,D5.W),D0;   First character to remain
         MOVE.L  teTextH(A1),A2  ;      A2 = Text handle
         MOVE.L  (A2),A2         ;      A2 -> Text start (destination)
         LEA     (A2,D0.W),A3    ;      A3 -> Source
         MOVE.W  teLength(A1),D1 ;      Count of characters to move
         SUB.W   D0,D1           ;      Fix for deletion
         EXT.L   D1              ;      Out to long
         MOVE.W  D1,D0           ;      Adjust
         SUBQ.W  #1,D0           ;              for DBRA
         MOVE.L  A2,A0           ;      Save address in A0
TypeDel  MOVE.B  (A3)+,(A0)+     ;      Move a byte
         DBRA    D0,TypeDel      ;      Until done
```

***Listing 8.16** – The XXPRINT routine*

```
         MOVE.L  A2,-(SP)         ;      -> Text
         MOVE.L  D1,-(SP)         ;      = Character count
         MOVE.L  THandle(A5),-(SP);      -> -> Text record
         _TESetText               ;      "Replace" text in TE record
DoType:                           ;      Here to begin output
         MOVE.L  A4,-(SP)         ;      Push character pointer
         MOVE.L  D6,-(SP)         ;      Push length
         MOVE.L  THandle(A5),-(SP);      and text handle
         _TEInsert                ;      Output the text
TYPEDone:                         ;      Here when done
         RETURNALL                ;      Restore registers
         ENDIF                    ;      TEXTRTL
```

Listing 8.16 *– The XXPRINT routine (continued)*

cursor desired. As explained in Chapter 6, it is possible to use this macro for the standard cursors as well as any cursor you define in a resource file. Listing 8.17 contains the SCURSOR macro definition.

The macro loads the cursor resource ID into D0.W and calls the run-time routine XXCURSOR. The code for this routine is shown in Listing 8.18.

Lines 2–4 check the last cursor displayed against the requested cursor. The routine returns immediately if the desired cursor is already being displayed. Lines 6–10 display the arrow cursor using the _InitCursor call. Other cursors are displayed by the code in lines 12–17. The _GetCursor call fetches a cursor from the resource file and returns a handle to the memory copy. The _SetCursor call displays a cursor, given its address. Note the conversion from handle to address at line 16.

SETVMIN, SETVMAX, SETVVAL, SETHMIN, SETHMAX, and SETHVAL

These routines set the values associated with a scroll bar. They are all essentially the same; only the system trap changes. Listing 8.19 shows the SETVMIN macro.

This macro puts the desired value in D0.W and calls the appropriate run-time routine. The run-time routine is shown in Listing 8.20.

Note that much of this code is shared between the XXSETVMIN and XXSETHMIN routines. They differ only in the control ID passed to the _SetMinCtl trap. The other routines are similar, differing only in trap name. The trap names appear in Table 8.2.

TRACK

The TRACK macro is used when a user program receives a page or line scroll event from the run-time library GETEVENT macro. This macro causes a procedure to be called repetitively, as long as the user holds down the mouse button in the control that causes the scrolling event. The TRACK macro code is shown in Listing 8.21.

The macro loads the control handle into register D0 and the procedure address into A0. It then calls the run-time routine XXTRACK (Listing 8.22). The XXTRACK routine calls an internal procedure, IncControls, which resets the window's clipping region to include the scroll bars. If you

```
        .MACRO  SCURSOR
        MOVE.L  D0,-(SP)        ;       Save register
        MOVE.W  %1,D0           ;       Set ID
        XREF    XXCURSOR        ;       RTL routine
        JSR     XXCURSOR        ;       Call RTL
        MOVE.L  (SP)+,D0        ;       Restore register
        .ENDM
```

Listing 8.17 *– The SCURSOR macro*

```
XXCURSOR:
        CMP.W   xCursor(A5),D0  ;       Need to do this?
        BNE     @1              ;       Yes
        RTS                     ;       No
@1:     MOVEM.L D0-D2/A0-A2,-(SP);      Save registers
        MOVE.W  D0,xCursor(A5)  ;       Remember for next time
        TST.W   D0              ;       Arrow?
        BNE     @2              ;       No, do it differently
        _InitCursor             ;       Arrow done thusly
        BRA     CDone           ;       Finished
@2:                             ;       Must get from resource file
        CLR.L   -(SP)           ;       Space for cursor handle
        MOVE.W  D0,-(SP)        ;       Push cursor ID
        _GetCursor              ;       Fetch cursor handle
        MOVE.L  (SP)+,A0        ;       A0 -> cursor pointer
        MOVE.L  (A0),-(SP)      ;       Push actual address
        _SetCursor              ;       Do the set
CDone:                          ;       Here with correct cursor
        MOVEM.L (SP)+,D0-D2/A0-A2;      Restore registers
        RTS                     ;       Return to caller
```

Listing 8.18 *– The XXCURSOR routine* invoke a system call that affects the controls before doing this, the screen appearance will not change.

The system call _TrackControl takes three parameters: the control ID (from the macro), the original mouse point (from the event that caused the scroll), and the user's procedure address (from the macro). When _TrackControl finishes, the ExcControls routine resets the window's clip region to exclude the control area. This prevents the applications from drawing on top of the scroll bars.

```
        .MACRO   SETVMIN         ;
        MOVEM.L  D0,-(SP)        ;    Push registers
        MOVE.W   %1,D0           ;    D0 = Value
        XREF     XXSETVMIN       ;    For the linker
        JSR      XXSETVMIN       ;    Call RTL routine
        MOVEM.L  (SP)+,D0        ;    Pop registers
        .ENDM
```

Listing 8.19 *– The SETVMIN macro*

```
XXSETVMIN                                  ;    Set upper limit
        SAVE                               ;    Save registers
        MOVE.L  VScroll(A5),-(SP)          ;    Push control handle
        BRA     SetMin                     ;    Do the SetMin
XXSETHMIN                                  ;    Set left limit
        SAVE                               ;    Save registers
        MOVE.L  HScroll(A5),-(SP)          ;    Push control handle
SetMin  MOVE.W  D0,-(SP)                   ;    and value
        _SetMinCtl                         ;    Set it
        RETURNALL                          ;    Back to caller
```

Listing 8.20 *– The XXSETVMIN and XXSETHMIN routines*

```
        .MACRO   TRACK
        MOVEM.L  D0/A0,-(SP)     ;    Save the registers
        MOVE.L   %1,D0           ;    D0 gets handle
        LEA      %2,A0           ;    A0 gets procedure
        XREF     XXTRACK         ;    For the linker
        JSR      XXTRACK         ;    Call RTL routine
        MOVEM.L  (SP)+,D0/A0     ;    Restore the registers
        .ENDM
```

Listing 8.21 *– The TRACK macro*

THE EVENT SYSTEM

The run-time library event-handling code handles a large number of things that a Macintosh application is expected to perform. This code handles many events, such as opening desk accessories, directly. The run-time library also partially processes other events, such as menu selection, scrolling, and window sizing. Still other events, such as disk insertion and network events, are passed back to the application with no processing. Placing this code in the run-time library means that the application writer need not be aware that it is present. The following macros are event related:

1. The GETEVENT macro, which returns the next event
2. The CHKABORT macro, which checks for an ABORT event

The GETEVENT routine is the most complicated of these routines.

Macro Name	Run-Time Name	Control ID	Trap Name
SETVMIN	XXSETVMIN	VScroll(A5)	_SetMinCtl
SETVMAX	XXSETVMAX	VScroll(A5)	_SetMaxCtl
SETVVAL	XXSETVVAL	VScroll(A5)	_SetCtlValue
SETHMIN	XXSETHMIN	HScroll(A5)	_SetMinCtl
SETHMAX	XXSETHMAX	HScroll(A5)	_SetMaxCtl
SETHVAL	XXSETHVAL	HScroll(A5)	_SetCtlValue

Table 8.2 – *Scroll Bar Value Routines*

```
        XDEF    XXTRACK          ;       For the linker
XXTRACK SAVE                     ;       Save the registers
        BSR     IncControls      ;       Clip to allow drawing controls
        CLR.W   -(SP)            ;       Room for result
        MOVE.L  D0,-(SP)         ;       Push control handle
        MOVE.L  Point(A5),-(SP)  ;       and original mouse point
        MOVE.L  A0,-(SP)         ;       and user's procedure
        _TrackControl            ;       Do the track
        ADDQ.L  #2,SP            ;       Pop return
        BSR     ExcControls      ;       Reclip for no controls
        RETURN                   ;       and return
```

Listing 8.22 – *The XXTRACK routine*

GETEVENT

The GETEVENT macro returns the next event number in D0.W. Additional parameters may be returned in D1.L and the high word of D0. The code for this macro is shown in Listing 8.23.

The macro has one parameter: the resource ID of a cursor to be used when the mouse is inside the content region of the active window. D0.W receives the cursor ID, and the macro calls the run-time routine XXEVENT. This routine is shown in Listing 8.24.

Lines 3–16 select the proper cursor, depending on whether the mouse is in the application window. The _GetMouse system call takes the address of a point and returns the mouse location (a point) in that location. The _PtInRect system call takes a point and the address of a rectangle and returns a word that is nonzero if the point is in the rectangle and zero otherwise. The mouse form is set to the specified cursor inside the window and the arrow cursor outside the window, using the XXCURSOR routine described previously.

Line 17 fetches the next event from the Macintosh event queue. The NEXTEVT macro is explained below. The event call fills in a memory area called an *event record,* which has the structure shown in Figure 8.1.

The event type field is a number between 0 and 15. The event time and mouse position fields give the time of day in binary and the mouse position at the time of the event. The message and modifier fields are dependent on the event type code. The event codes returned by the Macintosh Toolbox routine are shown in Table 8.3.

Lines 19–50 of Listing 8.24 perform a 16-way branch based on the value of the event type field. Before investigating each of these 16 branches, let's look first at the NEXTEVT macro and supporting code. Listing 8.25 shows the NEXTEVT macro and its supporting run-time routine.

The NEXTEVT macro loads a word parameter into D0 and calls NextEvent. The word parameter is a bitmask indicating the type of event being

```
        .MACRO  GETEVENT        ;       cursor =
        IF      '%1' <> ''      ;       Specified?
          MOVE.W %1,D0          ;       Yes, use specified
        ELSE                    ;       Wasn't
          MOVE.W #1,D0          ;       Use I-beam cursor
        ENDIF                   ;       Cursor
        XREF    XXEVENT         ;       For the linker
        JSR     XXEVENT         ;       Call RTL routine
        .ENDM
```

***Listing 8.23** – The GETEVENT macro*

```
XXEVENT:                           ;       
         SAVE                      ;       Save registers
         MOVE.W  D0,Cursor(A5)     ;       Set default cursor
DoEvent  PEA     Point(A5)         ;       Push mouse point address
         _GetMouse                 ;       Fetch mouse coordinates
         CLR.W   -(SP)             ;       Room for result
         MOVE.L  Point(A5),-(SP)   ;       Push point
         PEA     DestRect(A5)      ;       Push rect address
         _PtInRect                 ;       See if mouse in window
         TST.W   (SP)+             ;       Is mouse in window?
         BNE     ChangeCurs        ;       Yes, change the cursor
         CLR.L   D0                ;       Set to arrow (0)
         BRA     DoCurs            ;       Set the cursor
ChangeCurs:                        ;       Here if need to change cursor
         MOVE.W  Cursor(A5),D0     ;       Change to requested
DoCurs   BSR     XXCURSOR          ;       Change the cursor
         NEXTEVT #AllMask          ;       Get new event
         BEQ     NoEvent           ;       None there, try again
         MOVE.W  What(A5),D0       ;       D0 = Event type
         ADD.W   D0,D0             ;       Multiply by 2
         MOVE.W  Eventtab(D0),D0   ;       D0 = Offset
         JMP     Event(D0)         ;       Jump to event routine
Event:                             ;       Base point for offset table
;
;        Define a macro for event table entries
;
MACRO    ETAB    addr =            ;       Entry macro
         DC.W    {addr}-XXBASE     ;       Generate an offset word
         |
;
;        Define the event table itself
;
EventTab                           ;       Offset table
XXBASE   SET     *                 ;       Here is the base value
         ETAB    NullEvent         ;       0 => Null event, ignore
         ETAB    MouseDown         ;       1 => Mouse-down event
         ETAB    MouseUp           ;       2 => Mouse-up event
         ETAB    KeyDown           ;       3 => Key press (down)
         ETAB    KeyUp             ;       4 => Key release (up)
         ETAB    AutoKey           ;       5 => Autokey
         ETAB    Update            ;       6 => Update event
         ETAB    Disk              ;       7 => Disk
         ETAB    Activate          ;       8 => Activate
         ETAB    Abort             ;       9 => Abort
         ETAB    Network           ;       A => Network
         ETAB    Driver            ;       B => Driver
         ETAB    Appl              ;       C => Application #1
         ETAB    App2              ;       D => Application #2
         ETAB    App3              ;       E => Application #3
         ETAB    App4              ;       F => Application #4
```

***Listing 8.24** – The XXEVENT routine* sought. This can be used to eliminate undesirable events, as in the XXINPUT routine where only key-down events are useful.

The NextEvent routine does more than simply return the next event. Since this routine is called very often, it performs the normal chores

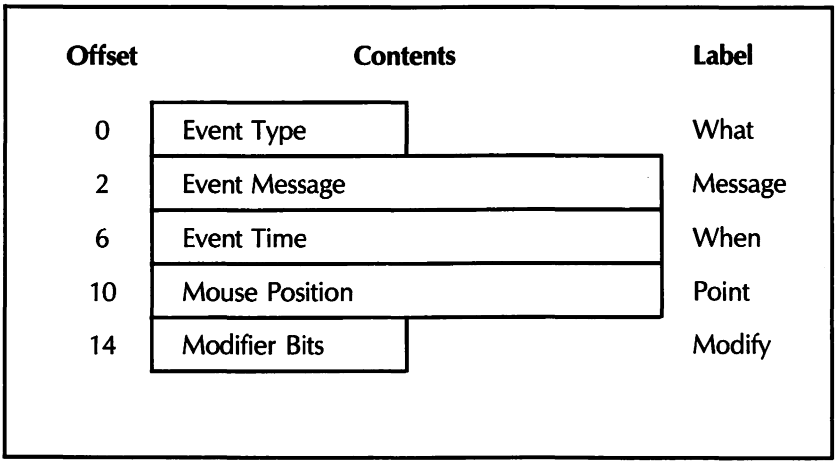


Figure 8.1 *– Macintosh event record*

Code	Meaning
0	No event is present
1	Mouse button was pressed (mouse down)
2	Mouse button was released (mouse up)
3	A key was pressed (key down)
4	A key was released (key up)
5	Repeat keypress (autokey)
6	Window update
7	Disk insertion
8	Activate window
9	Abort
10	Network event
11	Driver event
12	Application event #1
13	Application event #2
14	Application event #3
15	Application event #4

Table 8.3 *– Macintosh Event Types*

expected of an idling Macintosh application. Lines 9–10 call the TEIdle procedure, which causes the cursor to flash. Line 12 calls the _SystemTask function, which gives CPU time to any desk accessories that may be active.

Lines 14–18 actually fetch the next event from the queue. The _GetNextEvent system call takes the event mask and event record address as parameters and returns an event ID. The ID is zero if no event is available. The event record is filled in if the ID is nonzero.

We will now discuss the remaining code for the GETEVENT macro. Listing 8.24 contains a 16-way branch that is based on the return value from the NEXTEVT macro. Routines invoked by this branch handle the following events:

- Events passed back to the application and not processed by the application
- Mouse-down events, including menu selection, opening a desk accessory, scrolling, dragging the window, changing the window size, and closing the window
- Update events
- Activate events

We will also discuss the CHKABORT macro, which uses the NEXTEVT macro.

Events Passed to the Application

Many events are passed directly to the application without further processing. Listing 8.26 shows the values returned to the application program. The events in lines 1–11 all return the event code in the low word of D0, the message longword in D1.L, and the modifier word in the high word of D0.

Other events are screened before they are passed to the application. For instance, a key-down event is checked to see if the key pressed is actually a menu selection.

Mouse-Down Events

By far, the most difficult device to support is the Macintosh mouse. The mouse-down event is tricky to handle. Depending on the position of the mouse, this event can signal one of seven different activities:

1. If the mouse is in the desk area, the mouse-down event is meaningless and should be ignored.

2. If the mouse is in the menu area, the user wants to select a menu item.
3. If the mouse is in a desk accessory window, the event must be passed on to the accessory.
4. If the mouse is in the content area of the window, the event must be passed on to the application.

```
          MACRO    NEXTEVT Evmask   =
          MOVE.W   {Evmask},D0      ;        Load mask register
          JSR      NextEvent        ;        Get next event
          |

NextEvent:                          ;        Called internally only
          MOVE.L   D0,-(SP)         ;        Save D0 for a bit....
          IF       TEXTRTL = 0      ;        If text type RTL
          MOVE.L   THandle(A5),-(SP);        Push text record handle
          _TEIdle                   ;        Call text edit idle proc
          ENDIF                     ;        TEXTRTL
          _SystemTask               ;        Let system tasks run
          MOVE.L   (SP)+,D0         ;        Recover mask
          CLR.W    -(SP)            ;        Make room for event ID
          MOVE.W   D0,-(SP)         ;        Push event mask
          PEA      EventRecord(A5)  ;        Push event record address
          _GetNextEvent             ;        Fetch next Mac event
          MOVE.W   (SP)+,D0         ;        Event ID to D0
          RTS                       ;        Return
```

Listing 8.25 – *The NEXTEVT macro and the NextEvent routine*

```
NullEvent
Disk
Abort
MouseUp
KeyUp
Network
Driver
Appl
App2
App3
App4
UserEvt MOVE.W   Modify(A5),D0    ;        Modify word
        SWAP     D0               ;        To high word of D0
        MOVE.W   What(A5),D0      ;        Set return code
        MOVE.L   Message(A5),D1   ;        Message to D1.L
        RETURNP                   ;        Restore and return
```

Listing 8.26 – *Passing events to the application*

5. If the mouse is in the title bar of the window, the user wants to move the window.
6. If the mouse is in the grow box of the window, the user wants to change the size of the window.
7. If the mouse is in the close box of the window, the user wants to close the window.

The code that evaluates a mouse-down event is shown in Listing 8.27.

The _FindWindow call determines which window, and which part of that window, contains a given point. The integer returned by _FindWindow indicates the area of the window, as shown in lines 13–19 of Listing 8.27. Values of 0 and 7 are discarded. Values 1–6 are processed by the run-time library.

Menu Selection A mouse-down event in the menu bar means that a menu selection must be performed. The run-time library classifies menu selections into one of four groups:

1. A menu selection that is passed back to the application
2. An About . . . dialog selection from the Apple menu, which is processed by the run-time library

```
 1: MouseDown                         ;
 2:         CLR     -(SP)             ;      Clear a space for result
 3:         MOVE.L  Point(A5),-(SP)   ;      Mouse at time of event
 4:         PEA     WWindow(A5)       ;      Event window
 5:         _FindWindow               ;      Find the window
 6:         MOVE.W  (SP)+,D0          ;      Fetch index
 7:         AND.W   #7,D0             ;      Force to range 0-7
 8:         ADD.W   D0,D0             ;      Multiply by 2
 9:         MOVE.W  WTable(D0),D0     ;      Fetch offset
10:         JMP     WTable(D0)        ;      Jump to appropriate routine
11: WTable:                           ;      Start dispatch table
12: XXBASE  SET     *                 ;      Define macro base here
13:         ETAB    Doevent           ; 0    Mouse in desk area someplace
14:         ETAB    InMenu            ; 1    Mouse in menu area
15:         ETAB    SystemEvent       ; 2    Mouse in system window
16:         ETAB    Content           ; 3    Window content area
17:         ETAB    Drag              ; 4    In drag area
18:         ETAB    Grow              ; 5    In grow area
19:         ETAB    GoAway            ; 6    In close box
20:         ETAB    Doevent           ; 7    (Undefined)
```

Listing 8.27 – *Mouse-down events*

3. Starting a desk accessory, which is handled by the run-time library
4. The Copy, Cut, and Paste selections in the Edit menu (if the run-time library is configured to use the Text Edit package), which are also handled by the run-time library

Most menu selections fall into the first group. The first part of the menu code is shown in Listing 8.28.

Lines 1–9 illustrate how keyboard equivalents to menu selections work. The _MenuKey system call takes a key value and returns a longword. The high-order word of the return value is the menu number, and the low-order word is the item number in that menu.

Using the mouse to select a menu item involves the _MenuSelect call. This call takes the mouse point from the event as input and returns the

```
 1: KeyDown
 2: AutoKey
 3:         MOVE.W  Message+2(A5),D1;       Key to D1.W
 4:         BTST    #ComKey,Modify(A5);     Is Command key down?
 5:         BEQ     UserEvt         ;       No, give key to user
 6:         CLR.L   -(SP)           ;       Room for result
 7:         MOVE.W  D1,-(SP)        ;       Push character
 8:         _MenuKey                ;       See if a command
 9:         BRA     DoMenu          ;       Go do the menu
10: InMenu:                         ;
11:         CLR.L   -(SP)           ;       Clear space for return value
12:         MOVE.L  Point(A5),-(SP) ;       Push mouse point
13:         _MenuSelect             ;       Perform menu selection
14: DoMenu  CLR.W   -(SP)           ;       Unhilight menus
15:         _HiLiteMenu             ;       This does all
16:         MOVE.L  (SP),D1         ;       Both to D1
17:         MOVE.W  (SP)+,D6        ;       Menu number    (from _MenuSelect)
18:         MOVE.W  (SP)+,D7        ;       Menu selection (from _MenuSelect)
19:         TST.L   D1              ;       Really a choice?
20:         BEQ     DoEvent         ;       No, just kidding
21:         MOVE.L  #MenuChoice,D0  ;       D0 = menu choice return code
22:         CMP.W   #APPLE,D6       ;       Apple menu?
23:         BEQ     Inapple         ;       Yes, do this one ourselves
24:         CMP.W   #EDIT,D6        ;       No, how about the Edit menu?
25:         BEQ     InEdit          ;       Yes, handle this one too
26:         RETURNP                 ;       Restore registers; return
27: Inapple:                        ;       Here if it's a choice we handle ....
28:         CMP.W   #ABOUT,D7       ;       Was it the "About" choice?
29:         BNE     DeskAcc         ;       No, must be a desk accessory
30:         MOVE.W  D7,D0           ;       Set parameter register
31:         BSR     XXDIALOG        ;       Do the dialog
32:         BRA     DoEvent         ;       and do another event
```

Listing 8.28 – *Menu selection code* same longword as _MenuKey. After line 13, the code is the same, regardless of the method used to select the entry.

Both the _MenuKey and _MenuSelect calls leave the menu highlighted in the menu bar. Lines 14–15 unhighlight the menu. This is a deviation from standard Macintosh practice. Normally, you don't unhighlight the menu until you have finished performing the action associated with the menu selection. Since the run-time library will not regain control when the application finishes processing the selection, the unhighlighting must be done in advance.

Lines 16–26 determine whether the menu selection is to be returned directly to the user, or whether the run-time library will process the selection. The run-time library handles all events in the Apple menu and some events in the Edit menu.

Lines 28–32 handle the About . . . choice in the Apple menu. This is a dialog, identical to the one invoked by the DIALOG macro. The dialog template in the resource file must have a resource ID of 1.

The other selections in the Apple menu are desk accessories. The code to open a desk accessory appears in Listing 8.29. Lines 1–4 get a handle to the Apple menu on the stack. Lines 5–8 fetch the name associated with the menu selection into a scratch area. The return value of the _GetResource call is used as the first parameter to the _GetItem call. The returned string is in Pascal format.

The desk accessory is opened with the _OpenDeskAcc call at line 13. This call takes the name as input and returns a handle to the open accessory. Note that the current GrafPort is saved and restored around the _OpenDeskAcc call. This is advisable since the accessory may not restore the GrafPort before returning.

Desk accessories are parasites. They rely on the menu bar and the event code of the current application. The application must issue a _SystemTask request frequently to allow the accessory to run. The application must support an Edit menu selection in an accessory window via the _SysEdit call. The accessory receives mouse-down event notification via the _SystemClick call.

The code for the _SysEdit and _SystemClick support is shown in Listing 8.30. Lines 1–8 invoke _SysEdit. Note that the parameter to _SysEdit is zero-relative instead of one-relative like all other menu-related calls. The return value is either zero or nonzero, indicating whether the active window belongs to a desk accessory. This value is ignored by the run-time library.

Lines 9–13 illustrate how to call _SystemClick. You supply the event record address and the window pointer as parameters. The desk accessory receives an appropriate event call.

The code for handling the Edit menu appears in Listing 8.31. Lines 2–6 decide whether the Edit choice is destined for the application or a desk accessory. If the Text Edit package is in use, the _TECut, _TECopy, and _TEPaste system calls handle the Cut, Copy, and Paste selections. Lines 8–10 return a choice larger (lower on the menu) than Paste directly to the application. Lines 11–18 handle the Cut selection. Lines 21–23 handle the Copy selection. Lines 25–27 handle the Paste selection.

If the run-time library is not configured to use the Text Edit package, the code at lines 29–33 returns the Edit menu selection directly to the application.

Content Area or Scroll Bar Click A mouse-down event in the content area of a window invokes the code in Listing 8.32. Lines 2–4 reject windows

```
DeskAcc:                               ;
         CLR.L    -(SP)                ;     Clear return space
         MOVE.L   #'MENU',-(SP)        ;     Indicate menu resource
         MOVE.W   #APPLE,-(SP)         ;     Push menu ID
         _GetResource                  ;     Get menu handle
         MOVE.W   D7,-(SP)             ;     Push menu selection
         PEA      Deskname(A5)         ;     -> Acc name area
         _GetItem                      ;     Fetch desk accessory name
         PEA      GPort(A5)            ;     Push longword address
         _GetPort
         CLR.W    -(SP)                ;     Space for handle
         PEA      DeskName(A5)         ;     -> Filled-in name area
         _OpenDeskAcc                  ;     Open the accessory
         TST.W    (SP)+                ;     Discard handle
         MOVE.L   GPort(A5),-(SP)      ;     Push old GrafPort address
         _SetPort                      ;     Restore it
         BRA      Doevent              ;     and back for another event
```

Listing 8.29 – *Opening a desk accessory*

```
SystemEdit:                            ;
         MOVE.W   D7,D0                ;     D0 = menu choice
         SUBQ.L   #1,D0                ;     Decrement
         CLR.W    -(SP)                ;     Space for result
         MOVE.W   D0,-(SP)             ;     Choice word
         _SysEdit                      ;     Do it
         MOVE.W   (SP)+,D0             ;     Result to D0
         BRA      Doevent              ;     Back for next event
SystemEvent:                           ;
         PEA      EventRecord(A5)      ;     -> Event not ours
         MOVE.L   WWindow(A5),-(SP)    ;     Window handle
         _SystemClick                  ;     Call O/S
         BRA      Doevent              ;     Back for another event
```

Listing 8.30 – *Calling _SysEdit and _SystemClick*

```
InEdit:                               ;       Cut/Copy/Paste
          CLR.L    -(SP)              ;       Clear return area
          _FrontWindow                ;       Return front window pointer
          MOVE.L   (SP)+,A0           ;       Window handle to A0
          CMP.L    WindowPtr(A5),A0   ;       Our window?
          BNE      SystemEdit         ;       No, it's a system window
          IF       TEXTRTL = 0        ;       If TEXT RTL only
          CMP.W    #PASTE,D7          ;       Bigger than Paste?
          BLE      DoEdit             ;       No, it's OK
          RETURNP                     ;       Restore; return
DoEdit:   CMP.W    #COPY,D7           ;       Bigger than Copy value?
          BGE      DoCopy             ;       Yes, keep looking
;
;         Must Be a Cut Command
;
          MOVE.L   THandle(A5),-(SP)  ;       Push text handle
          _TECut                      ;       Do the Cut
          BRA      Doevent            ;       Back for more
DoCopy:   CMP.W    #PASTE,D7          ;       Is it a Paste?
          BEQ      DoPaste            ;       Yes
          MOVE.L   THandle(A5),-(SP)  ;       Push text handle
          _TECopy                     ;       Do the Copy
          BRA      Doevent            ;       And back for more
DoPaste:                              ;       Here for Paste
          MOVE.L   Thandle(A5),-(SP)  ;       Push text handle
          _TEPaste                    ;       Do it
          BRA      Doevent            ;       Get next event
          ELSE                        ;       Graphics mode
          MOVE.L   #MenuChoice,D0     ;       Set return code
          MOVE.W   D6,D1              ;       Copy choice words
          SWAP     D1                 ;            to
          MOVE.W   D7,D1              ;       return register
          RETURNP                     ;       and return menu choice
          ENDIF
```

Listing 8.31 *– Edit menu code*

other than the application window. Note that desk accessory windows are handled by the SystemEvent routine, which is selected by the code in Listing 8.27. This code will not cause a desk accessory event to be lost. Lines 5–6 select the window (make it active) if it is behind another window. Lines 7–8 convert the mouse point in the event record from global (screen) coordinates to local (window) coordinates. If the Text Edit package is in use, lines 10–13 invoke the _TEClick routine, which handles mouse-down events. This routine has three parameters: the mouse point in local coordinates, an extend flag, and the Text Edit record handle. The extend flag indicates that the current selection is to be expanded. The run-time library always sets this flag to FALSE (zero).

```
Content:                              ;
        MOVE.L  WWindow(A5),A0        ;    A0 = window handle
        CMP.L   WindowPtr(A5),A0      ;    Ours?
        BNE     Doevent               ;    No, get next event
        MOVE.L  A0,-(SP)              ;    Stack window handle
        _SelectWindow                 ;    Select window
        PEA     Point(A5)             ;    Stack point address
        _GlobalToLocal                ;    Convert to window coords
        IF      TEXTRTL = 0           ;    If text type RTL
        MOVE.L  Point(A5),-(SP)       ;    Push point in local coords
        CLR.W   -(SP)                 ;    No extend flag
        MOVE.L  THandle(A5),-(SP)     ;    Stack text handle
        _TEClick                      ;    Do text edit click
        BRA     Doevent               ;    Get next event
        ELSE                          ;    Graphics mode
```

Listing 8.32 *– Mouse down in window content area (text mode)*

When the window has controls, a click in the content area could be in the scroll bars. The code for this situation is shown in Listing 8.33.

The call to routine IncControls sets the window's clipping region to include the scroll bars. You must do this before you attempt any operations on the controls. The _FindControl call in lines 3–9 determines whether the mouse point (now in local coordinates) is in one of the controls. This system call takes the mouse point, the window address, and the address of a control handle as parameters. If the mouse point is inside a control, the control handle is set to indicate which control. The return value (a word) indicates which part of the control contains the mouse point, as shown in the following list:

Value	Control Area
0	Not in any control, or in an inactive control
10	In a simple button
11	In a check box or other button
20	In up (left) arrow of a scroll bar
21	In down (right) arrow of a scroll bar
22	In page up (left) region of a scroll bar
23	In page down (right) region of a scroll bar
129	In thumb area of a scroll bar

If the mouse point is not in any control area, the point is returned to the user as a mouse-down event. If the point is in a scroll bar region (codes 20–23), the event is returned as a scroll event. The event code is 19 (decimal),

```
        IF      CONTROL = 0         ;       Only if we have controls
        BSR     IncControls         ;       Include controls first
        CLR.W   -(SP)               ;       Room for result
        MOVE.L  Point(A5),-(SP)     ;       Push point
        MOVE.L  WindowPtr(A5),-(SP);        Push window address
        PEA     CHandle(A5)         ;       Push control handle
        _FindControl                ;       Fetch the control
        MOVE.W  (SP)+,D0            ;       Get part code
        BEQ     NoControl           ;       None, give point to user
        MOVE.L  CHandle(A5),D1      ;       Fetch control handle
        CMP.W   #inThumb,D0         ;       Special case?
        BEQ     @2                  ;       Yes, go handle
        SUB.W   #inUpButton,D0      ;       Force to range 0-3
        CMP.L   VScroll(A5),D1      ;       Was it?
        BEQ     @1                  ;       Yes, give to user
        ORI.W   #4,D0               ;       Or in horizontal bit
@1      SWAP    D0                  ;       Set return
        MOVE.W  #ScrollWin,D0       ;                       register
        BRA     @4                  ;       Back to user
;
;       If in thumb, we can do the tracking for the user program.
;
@2      MOVE.L  D1,D4               ;       Preserve control ID
        CLR.W   -(SP)               ;       Room for result
        MOVE.L  D1,-(SP)            ;       Push control ID
        MOVE.L  Point(A5),-(SP)     ;       Push original point
        CLR.L   -(SP)               ;       No action procedure
        _TrackControl               ;       Track it
        TST.W   (SP)+               ;       For real?
        BNE     @3                  ;       Yes, process
        BSR     ExcControls         ;       Reset clip region
        BRA     DoEvent             ;       Get an event
@3                                  ;
;
;       User changed the value.  Fetch new setting.
;
        CLR.W   -(SP)               ;       Room for result
        MOVE.L  D4,-(SP)            ;       Push control handle
        _GetCtlValue                ;       Fetch the value
        MOVE.W  (SP)+,D0            ;       to
        SWAP    D0                  ;               return register
        MOVE.L  D4,D1               ;       Reset parameter register
        MOVE.W  #TScrollV,D0        ;       Assume vertical
        CMP.L   VScroll(A5),D1      ;       Good assumption?
        BEQ     @4                  ;       Yes
        MOVE.W  #TScrollH,D0        ;       No, fix it
@4      BSR     ExcControls         ;       Exclude controls
        RETURNP                     ;       Done, return to caller
NoControl                           ;       Here if user point
        BSR     ExcControls         ;       Exclude controls
        ENDIF                       ;       CONTROL = 0
;
```

Listing 8.33 – *Mouse down in window content area (graphics mode)*

```
53: ;        Here if mouse is in the REAL content area.  Pass the event
54: ;        back to the application.
55: ;
56:          MOVE.W  What(A5),D0     ;         Set mouse-down return code
57:          MOVE.L  Point(A5),D1    ;         Return point in local coords
58:          RETURNP                 ;         Back to user
59:          ENDIF                   ;
```

***Listing 8.33** – Mouse down in window content area (graphics mode) (continued)*

with the high word of D0 containing a value from 0–7 indicating the type of scrolling to be performed:

Value	Scroll Type
0	Line scroll up
1	Line scroll down
2	Page scroll up
3	Page scroll down
4	Line scroll left
5	Line scroll right
6	Page scroll left
7	Page scroll right

Note that bit 0 of this value determines scrolling direction. An even scrolling value indicates a scroll up (left), and an odd scrolling value indicates a scroll down (right). Similarly, bit 1 distinguishes between a line scroll and a page scroll, and bit 2 distinguishes between a vertical scroll and a horizontal scroll. Lines 10–19 compute this value. The user program should issue a call to the TRACK macro to continue scrolling until the mouse button is released.

If the mouse is down in the thumb area, the run-time library returns a thumb scroll event (D0.W = 20 for a vertical scroll, 21 for a horizontal scroll). The final value of the thumb is returned in the high word of D0. D1 contains the control handle on return to the application.

Lines 23–29 perform the _TrackControl call necessary to allow the user to change the thumb position on the screen. This system call returns the part code (described previously) of the control if the mouse remains in the control. A zero is returned otherwise, indicating that the user moved the mouse outside the control area. Lines 30–32 handle this case, resetting the window's clipping region to exclude the controls, and ignoring the event.

Lines 37–48 handle the other case, where the user actually moves the thumb box on the screen. The _GetCtlValue call takes a control handle and returns the value of the control. This value is placed in the high word of D0 at lines 40–41. The application sets the minimum and maximum values for the controls using the SETVMIN, SETVMAX, SETHMIN, and SETHMAX macros. Lines 42–46 set the low word of D0 to the appropriate event code, depending on which scroll bar was affected. Lines 47–48 reset the window clipping region and return the new control value to the user.

Lines 49–59 return a mouse-down event to the application. If the mouse was not pressed in a control, or if there are no controls, the run-time library passes the event directly back to the application.

Dragging the Window A mouse-down event in the window title bar means the user is trying to drag the entire window. The code required to support this is shown in Listing 8.34.

The _DragWindow system call does all the required work. This system call takes as parameters the window pointer, the mouse point from the event record, and a bounds rectangle. The bounds rectangle specifies limits within which the window can be moved.

Changing the Window Size Listing 8.35 contains the code for altering window size. This code is activated when the mouse is pressed in the window's grow box, as determined by the _FindWindow call in Listing 8.27.

Lines 3–9 issue a _GrowWindow system call. This call requires a window pointer, the mouse point from the event record, and a bounds rectangle. The bounds rectangle establishes limits on the minimum and maximum size of the window. The first point in the rectangle gives the minimum size, and the second point gives the maximum size. _GrowWindow returns a point containing the new size of the window. If this return value is zero, the window did not change size.

```
Drag:                               ;
        MOVE.L  WWindow(A5),-(SP);      Push window handle
        MOVE.L  Point(A5),-(SP) ;       Push mouse point
        PEA     DWBounds        ;       and bounds rectangle
        _DragWindow             ;       Do the drag
        BRA     Doevent         ;       and get a new event
```

Listing 8.34 – *Window dragging code*

```
Grow:                             ;
        IF      CONTROL = 0       ;       Only if controls in place
        CLR.L   -(SP)             ;       Room for result
        MOVE.L  WWindow(A5),-(SP);        Push window handle
        MOVE.L  Point(A5),-(SP)   ;       and mouse point
        PEA     GWBounds          ;       and bounds rectangle
        _GrowWindow               ;       Do the grow
        MOVE.L  (SP)+,D5          ;       D5 = New size
        BEQ     DoEvent           ;       If EQ, didn't change
;
;       First, generate an update for the old scroll bar region.
;
        PEA     ViewRect(A5)      ;       Convert ViewRect to
        _LocalToGlobal            ;               global coordinates
        MOVEM.L ViewRect(A5),D3/D4;       Fetch rectangle
        MOVEM.L ViewRect(A5),D0/D1;       twice
;
;       Invalidate VScroll bar
;
        MOVE.W  D1,D0             ;       Copy right to left
        ADD.W   #ScrollW,D1       ;       Adjust right for scroll bar width
        SWAP    D1                ;       Adjust
        SUB.W   #ScrollW,D1       ;               bottom
        SWAP    D1                ;                       for grow box
        MOVEM.L D0/D1,ViewRect(A5);       Now
        PEA     ViewRect(A5)      ;               make
        _InvalRect                ;                       invalid
;
;       Now, invalidate HScroll bar
;
        SWAP    D3                ;       Invert
        SWAP    D4                ;               rectangle
        MOVE.W  D4,D3             ;       Copy bottom to top
        ADD.W   #ScrollW,D4       ;       Adjust bottom for scroll width
        SWAP    D3                ;       Re-
        SWAP    D4                ;               swap
        ADD.W   #ScrollW,D4       ;       Adjust right for grow box
        MOVEM.L D3/D4,ViewRect(A5);       Now
        PEA     ViewRect(A5)      ;               make
        _InvalRect                ;                       invalid
;
;       Next, actually resize the window on the screen
;
        MOVE.L  WWindow(A5),-(SP);        Push window handle
        MOVE.L  D5,-(SP)          ;       Push size
        MOVE.W  #-1,-(SP)         ;       Push TRUE value
        _SizeWindow               ;       Update screen to new size
        BSR     SizeW             ;       Size the window
;
;       Erase the new window content area
;
        PEA     ViewRect(A5)      ;       Push view rectangle address
        _EraseRect                ;       Erase it
```

***Listing 8.35** – Changing the size of a window*

```
;
;        Here we move the scroll bars to their new location.  First,
;        hide both controls.
;
         MOVE.L  VScroll(A5),-(SP);      Hide
         _HideControl            ;               VScroll bar
         MOVE.L  HScroll(A5),-(SP);      Hide
         _HideControl            ;               HScroll bar
;
;        Next move and resize VScroll bar
;
         MOVE.L  VScroll(A5),-(SP);      Move VScroll
         MOVE.W  ViewRect+6(A5),-(SP);   "Right" - scroll width
         MOVE.W  ViewRect+0(A5),-(SP);   "Top"
         _MoveControl            ;       Move the scroll bar
         MOVE.L  VScroll(A5),-(SP);      Now resize VScroll bar
         MOVE.W  #ScrollW,-(SP)  ;       Width
         MOVE.W  ViewRect+4(A5),-(SP);   Length
         _SizeControl            ;       Do the size
;
;        Now move and resize HScroll bar
;
         MOVE.L  HScroll(A5),-(SP);      Move HScroll
         MOVE.W  ViewRect+2(A5),-(SP);   "Left"
         MOVE.W  ViewRect+4(A5),-(SP);   "Bottom" - scroll width
         _MoveControl            ;       Do the move
         MOVE.L  HScroll(A5),-(SP);      Push control handle
         MOVE.W  ViewRect+6(A5),-(SP);   Width
         MOVE.W  #ScrollW,-(SP)  ;       Length
         _SizeControl            ;       Size it
;
;        Now show both controls
;
         MOVE.L  VScroll(A5),-(SP);      Show
         _ShowControl            ;               VScroll bar
         MOVE.L  HScroll(A5),-(SP);      Show
         _ShowControl            ;               HScroll bar
;
;        Now pass the size information to the application.
;
         MOVE.L  ViewRect+4(A5),D1;      Pick up new size (clipped)
         MOVE.L  #GrowWin,D0     ;       Set return code
         RETURNP                 ;       Give back to user
         ELSE                    ;       CONTROL = 0
         BRA     DoEvent         ;       No grows allowed
         ENDIF                   ;       CONTROL = 0
```

Listing 8.35 *– Changing the size of a window (continued)*

_GrowWindow drags an outline of the window around on the screen, as shown in Figure 8.2. When the mouse button is released, the outline disappears and _GrowWindow returns the new size to the application.

Lines 13–40 mark the region occupied by the scroll bars and the grow box as invalid. This is necessary for proper operation of overlapped windows. (When the top window shrinks, any underlying window must be redisplayed.) The view rectangle for the window gives the bounds of the content area of the window (exclusive of scroll bars) in local coordinates. Lines 13–16 convert this rectangle to global (screen) coordinates and make copies in D0/D1 and D3/D4.

Lines 20–24 calculate the rectangle occupied by the old vertical scroll bar on the screen. Lines 25–27 mark the rectangle as invalid. Lines 31–37 calculate the rectangle occupied by the combination of the old horizontal scroll bar and grow box. Lines 38–40 mark this area on the screen as invalid.

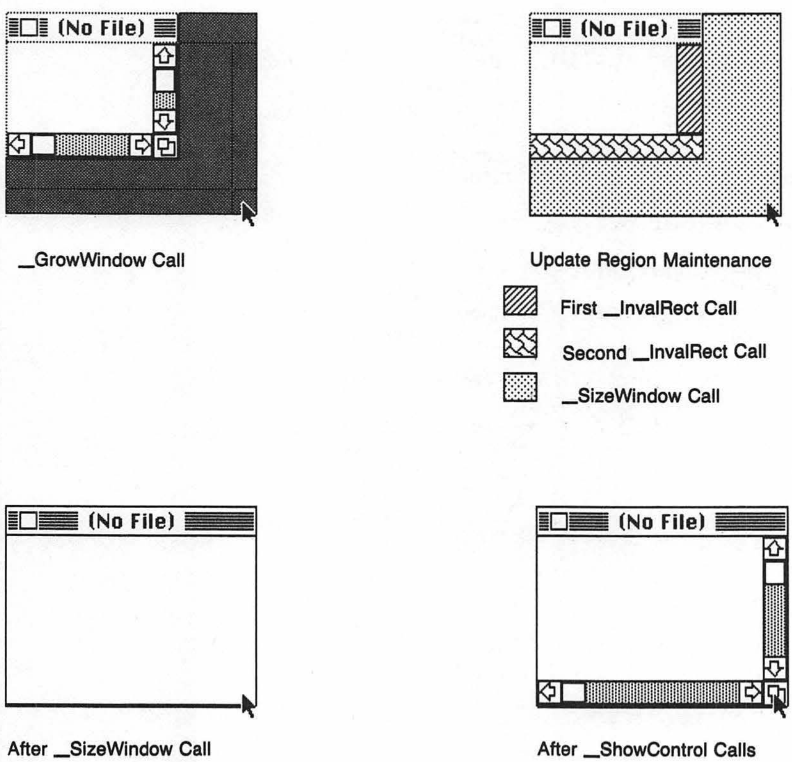


Figure 8.2 – *Enlarging a window*

Lines 44–47 actually change the size of the window. The _SizeWindow call redraws the window on the screen. The scroll bars and grow box are not drawn, however. _SizeWindow requires a window address, the size returned by _GrowWindow, and a word that indicates whether the update region should automatically be updated. This word is nonzero for updating. The area between the old and new window sizes is added to the update region. The call to SizeW at line 48 updates the internal variables in the run-time library to reflect the change in window size. The SizeW routine appears in Listing 8.7.

Lines 52–53 erase the content region (not including the scroll bars) of the resized window. You must at least erase the area occupied by the old grow box when enlarging a window. The run-time code requires that the application redraw the window contents.

Lines 58–61 issue the _HideControl call for both scroll bars. Lines 65–68 move the vertical scroll bar to its new location. The _MoveControl call takes the control handle and the coordinates of the new location as parameters. Note that the coordinates are specified in reverse order (horizontal first) for this call. Lines 69–72 specify the size of the scroll bar. The _SizeControl call takes the control handle, width, and length as parameters. Lines 76–83 repeat the move and size operations for the horizontal scroll bar.

Lines 87–90 issue _ShowControl calls for both scroll bars. This causes the scroll bars to reappear in their new locations on the screen.

Lines 94–96 return the size window event to the application. D1.L contains the new window size (not including the scroll bars). Lines 97–99 ignore the event if the window has no controls.

Listing 8.36 contains the code for the IncControls and ExcControls routines, which include and exclude the controls in the window clipping area. Lines 5–6 add the width of the scroll bar to the bottom and right words of the rectangle. Lines 7–8 set the clip rectangle to this rectangle. Lines 12–19 reset the clip rectangle to the saved rectangle.

Closing the Window A mouse click in the window's close box is handled by the code in Listing 8.37. The _TrackGoAway call highlights the close box as shown in Figure 8.3. If the user moves the mouse outside the close box before releasing the button, _TrackGoAway unhighlights the box. Moving the mouse back into the box highlights it again.

Lines 2–6 perform the _TrackGoAway call. The parameters are the window address and the mouse point from the event record. A word is returned. If this word is nonzero, the user released the button inside the close box, and a close-window event is returned to the application at lines 8–9. A zero value indicates that the user released the button outside the close box. The event is ignored in this case.

```
IncControls                         ;       Include controls
        MOVEM.L D0-D2/A0-A2,-(SP);          Preserve registers
        MOVEM.L DestRect(A5),D0/D1;         Pick up old rect
        MOVEM.L D0/D1,ViewRect(A5);         Reset to clipped
        ADD.W   #ScrollW,ViewRect+4(A5);  Adjust
        ADD.W   #ScrollW,ViewRect+6(A5);             clip
        PEA     ViewRect(A5)        ;                      rectangle
        _ClipRect                   ;
        MOVEM.L (SP)+,D0-D2/A0-A2;          Restore registers
        RTS                         ;       Back to caller

ExcControls                         ;       Exclude
        MOVEM.L D0-D2/A0-A2,-(SP);          Preserve registers
        MOVEM.L DestRect(A5),D0/D1;         Pick up old rect
        MOVEM.L D0/D1,ViewRect(A5);         Reset to clipped
        PEA     ViewRect(A5)        ;       Push rectangle address
        _ClipRect                   ;       Set to clipped
        MOVEM.L (SP)+,D0-D2/A0-A2;          Restore registers
        RTS                         ;       Back to caller
```

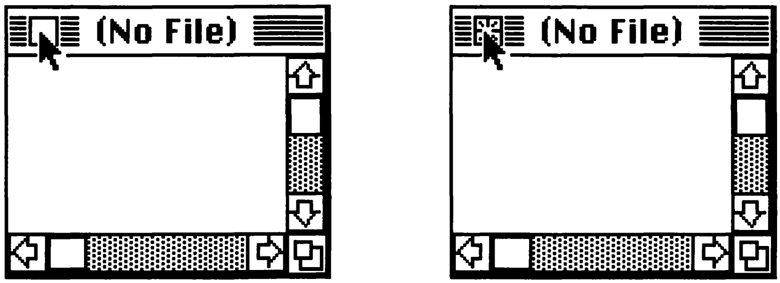
***Listing 8.36** – IncControls and ExcControls routines*

Update Events

The result of the _InvalRect calls in Listing 8.36 is to generate a Macintosh *update event.* When an area of the screen is marked as invalid, an update event is generated for all windows in the invalid area. In many cases, only a part of each window is affected. However, most applications update all of the window in response to any update event. This is the case for all programs in this book.

Listing 8.38 shows the code executed by the run-time library in response to an update event. Lines 2–10 handle the update completely if the run-time code is using the Text Edit package. The actual update must be bracketed with _BeginUpDate and _EndUpDate calls, as with the MacDoodle program in Chapter 6. For a Text Edit window, you need call only the _TEUpdate routine. This call requires a rectangle and the Text Edit handle for the affected window. The rectangle is in the local coordinates of the window, and it can be used to update only a part of the window if desired. The ViewRect structure specified in line 5 causes the entire window to be updated.

For applications using graphics, the update event is handled by the application. If the window has controls, however, the run-time library draws the controls before giving the event to the application. Lines 12–17 redraw all the window's controls. Note that the clipping region must be



Before After

Figure 8.3 *– The effects of _TrackGoAway*

set to include the controls during this operation, as evidenced by the calls to IncControls and ExcControls before and after drawing the controls.

Activate Events

A window becomes the active window by means of a Macintosh *activate event.* An activate event can also deactivate a window. Listing 8.39 shows the code executed in response to an activate event.

Lines 2–3 reject activate events for windows other than the application window. Bit 0 of the event record "modify" word determines whether the event is an activate or deactivate request. If the bit is clear, the event is a deactivate request. Lines 12–14 make this determination.

Lines 15–31 process activate requests. If the window is a text window, the _TEActivate system call is required, as shown in lines 11–14. You must then issue a _SelectWindow call to highlight the window and a _SetPort call to make the window the active QuickDraw port. Finally, if the window has scroll bars, you must issue a _ShowControl call for each scroll bar.

```
GoAway:
        CLR.W   -(SP)              ;     Room for result
        MOVE.L  WWindow(A5),-(SP)  ;     Push window handle
        MOVE.L  Point(A5),-(SP)    ;     Push mouse point
        _TrackGoAway               ;     See if legitimate
        TST.W   (SP)+              ;     Pop result
        BEQ     Doevent            ;     User changed mind
        MOVE.L  #CloseWin,D0       ;     Set special return code
        RETURNP                    ;     Bail out
```

Listing 8.37 *– The close-window routine*

```
Update:                             ;       Just update text part
          IF       TEXTRTL = 0      ;       If text only
          MOVE.L   WindowPtr(A5),-(SP) ;    Warn
          _BeginUpDate              ;               everyone first
          PEA      ViewRect         ;       Now repaint
          MOVE.L   THandle(A5),-(SP) ;      text window
          _TEUpdate                 ;       thusly
          MOVE.L   WindowPtr(A5),-(SP) ;    Now signal end
          _EndUpdate                ;                       of update
          BRA      Doevent          ;       Done
          ELSE                      ;       Graphics mode
          BSR      IncControls      ;       Include controls in window
          MOVE.L   WindowPtr(A5),-(SP);     Push window pointer
          _DrawControls             ;       Draw the controls
          MOVE.L   WindowPtr(A5),-(SP);     Push window pointer
          _DrawGrowIcon             ;       Draw grow box
          BSR      ExcControls      ;       Take controls back out
          BRA      UserEvt          ;       Let user handle
          ENDIF                     ;
```

Listing 8.38 *– Update event handler*

Lines 35–47 handle deactivate requests. If the window is a Text Edit window, you must call _TEDeActivate to deactivate the Text Edit Record. If the window has controls, you must call _HideControl for each scroll bar. Note that there is no counterpart to the _SelectWindow or _SetPort calls in the Activate code.

CHKABORT

The CHKABORT routine used in the Dump program in Chapter 4 tests to see if the next event is an abort event. This routine is also part of the event system. This code consists of the CHKABORT macro and a run-time support routine called XXABORT. The CHKABORT macro is shown in Listing 8.40. It is simply a JSR to routine XXABORT. The code for this routine appears in Listing 8.41.

XXABORT gets the next event from the event queue. Only three types of events are allowed by the bit mask AbtEvt: key down, autokey, and abort. If the event returned is an abort event or if the key pressed is Command-Period, a value of −1 is returned to the user program at line 13. Otherwise, a value of zero is returned at line 15.

```
Activate                        ;       Activate window request
        MOVE.L  Message(A5),A0  ;       A0 = window pointer

        CMP.L   WindowPtr(A5),A0 ;      Is it ours?
        BNE     DoEvent         ;       No, get next event
        IF      CONTROL = 0     ;       Controls?
        BSR     IncControl      ;       Yes, include them
        ENDIF                   ;       CONTROL
;
;       (De)activate the text window
;
        MOVE.W  Modify(A5),D0   ;       Activate or deactivate?
        BTST.L  #0,D0           ;       Bit 0 tells us
        BEQ     Deactivate      ;       Clear => Deactivate
        IF      TEXTRTL = 0     ;       Don't activate TE if not needed
        MOVE.L  THandle(A5),-(SP) ;     Push text handle
        _TEActivate             ;       Activate text window
        ENDIF                   ;
        MOVE.L  Message(A5),-(SP) ;     Push window pointer
        _SelectWindow           ;       Select this window as active
        MOVE.L  Message(A5),-(SP) ;     Push window pointer
        _SetPort                ;       Make the Q/D active port
        IF      CONTROL = 0     ;       If window has controls
        MOVE.L  VScroll(A5),-(SP);      Show
        _ShowControl            ;               scroll
        MOVE.L  HScroll(A5),-(SP);                      bars
        _ShowControl            ;
        BRA     EndAct          ;       Do end stuff
        ELSE                    ;       CONTROL = 0
        BRA     DoEvent         ;       Fetch next event
        ENDIF                   ;       CONTROL
;
;       Here if deactivate specified
;
Deactivate:                     ;
        IF      TEXTRTL = 0     ;       Don't deactivate TE if no text
        MOVE.L  THandle(A5),-(SP) ;     Push text handle
        _TeDeActivate           ;       and deactivate it
        ENDIF
        IF      CONTROL = 0     ;       If scroll bars
        MOVE.L  VScroll(A5),-(SP);      Hide
        _HideControl            ;               scroll
        MOVE.L  HScroll(A5),-(SP);                      bars
        _HideControl            ;
EndAct: BSR     ExcControls     ;       Exclude controls
        ENDIF                   ;       CONTROL = 0
        BRA     Doevent         ;       Get next event
```

Listing 8.39 – *Activate event handler*

```
         MACRO   CHKABORT       =
         XREF    XXABORT        ;        For the linker
         JSR     XXABORT        ;        Call abort check routine
         |
```

***Listing 8.40** – The CHKABORT macro*

```
XXABORT:                        ;
         SAVE                   ;        Save registers
         NEXTEVT #AbtMask       ;        Get abort event, if any
         BEQ     NoEvent        ;        No event, return
         CMPI.W  #AbtEvt,What(A5);       Was an abort?
         BEQ     AbtEvent       ;        Yes, quit
         CMPI.B  #'.',Message+3(A5);     Check
         BNE     NoEvent        ;            for
         BTST    #ComKey,Modify(A5);             Command-
         BEQ     NoEvent        ;                        Period
AbtEvent                        ;
         MOVEQ.L #-1,D0         ;        Set return code
         RETURN                 ;        Quit
NoEvent  CLR.L   D0             ;        Set no event return code
         RETURN                 ;        Quit
```

***Listing 8.41** – The XXABORT routine*

FILE ROUTINES

The run-time library supports several calls relating to the Macintosh file system:

1. The OPEN, OPENR, and POPEN calls open an existing file.
2. The CREATE and PCREATE calls create a new file.
3. The CLOSE call closes an open file.
4. The READ and READLN calls read data from a file.
5. The WRITE call writes data to a file.
6. The SEEK call allows random access to a file.
7. The DELETE call removes an existing file.
8. The GETFINFO and SETFINFO calls get and set the file information.

Macintosh I/O Parameter Block

Most of these routines use a Macintosh data structure known as an *I/O parameter block*, or *IOPB*. This structure was partially described in Chapter 4. Figure 8.4 shows the structure of an I/O parameter block.

Offset	Contents	MDS Label
0	I/O Link Address	ioLink
4	I/O Type	ioType
6	I/O Trap	ioTrap
8	I/O Command Address	ioCmdAddr
12	Application Completion Routine Address	ioCompletion
16	I/O Result Code	ioResult
18	File Name Address	ioFilename
22	Volume Reference Number	ioVRefNum
24	I/O Reference Number	ioRefNum
26	I/O File Type	ioFileType
27	I/O Permission	ioPermssn
28	File Private 522-byte Buffer Address	ioOwnBuf
32	Read/Write Buffer Address	ioBuffer
36	Read/Write Byte Count Requested	ioByteCount
40	Read/Write Byte Count Completed	ioNumDone
44	File Positioning Mode/End-of-Line Character	ioPosMode
46	File Position Offset	ioPosOffset

Figure 8.4 *– The I/O parameter block*

The column labeled MDS Label contains the standard name for each field using the MDS equate files. The IOPB names are found in the file SysEqux.D. The first four fields of the IOPB are used internally by the operating system. The application need only reserve the space for these fields.

The *ioCompletion* field is used for asynchronous I/O. You can specify a routine to be called when the operation is complete. To specify that no routine is to be called, zero this field. The run-time library does not use this feature.

The *ioResult* field is returned by the system. This word contains either an error code or a positive result. Macintosh error codes are always less than zero.

The *ioFileName* field is specified by the application on an open, create, delete, or file information request. This field should contain the address of a Pascal-format (leading count byte) string. You can specify a volume name before the file name, separating the two by a colon if desired.

The volume reference number (*ioVRefNum*) is set by the application before an open, create, delete, or file information request. This is an alternate way of specifying a drive on which to find the file. If this field is zero, the default drive is used. This field is set by the operating system on a successful operation. Specifying a drive name as part of the file name overrides the volume reference number.

The I/O reference number (*ioRefNum*) contains a reference number on a file open or create operation. This number is set by the application on a read, seek, or write request.

The I/O file type (*ioFileType*) and I/O permission (*ioPermssn*) bytes are additional parameters for open requests. The file type is set to zero by most applications on create and open requests. The I/O permission byte is an indication of what requests will be attempted after the file is open. This byte can have the following values:

Value	**Meaning**
0	Request type is unknown
1	Only read requests will be attempted
2	Only write requests will be attempted
3	Both read and write requests will be attempted

This byte provides the means to detect whether you can perform the operations you desire on the file at open time. For instance, writing on a locked file or disk could be detected at open time rather than at the first write call. The run-time library, however, always sets this byte to zero.

The private buffer address, *ioOwnBuf,* provides a means of speeding up file access. If this buffer is not specified, a buffer shared by the entire disk will be used. If you have many files open at once, you can achieve a significant performance improvement by using private file buffers.

Read and write requests use the buffer address (*ioBuffer*) as the address of the first byte to be transferred. The byte count requested (*ioByteCount*) is the number of bytes to be transferred. The operating system will set the byte count completed (*ioNumDone*) to the number of bytes actually transferred. If the number of bytes requested is not equal to the number transferred, an error occurred if the request was a write request. This situation may not indicate an error with read requests, however, if you are reading the last bytes in the file (that is, you ask for 100 bytes but there are only 50 left).

The *ioPosMode* field indicates the meaning of the file position offset field (*ioPosOffset*) in requests that require a file offset. The SEEK macro uses this field. You can specify an offset relative to the beginning of the file, the current file position, or the end of the file. In addition, if bit 7 of this word is set on a read request, the upper byte is taken to be an end-of-line character. The READLN macro uses this feature.

Opening a File

To access the data within a file, you must open the file. A file open operation establishes a link between an I/O reference number and a file name on the disk. The run-time library offers three open macros: OPEN, OPENR, and POPEN.

OPEN and OPENR

Listing 8.42 contains the source to the OPEN macro. The OPENR macro is identical, except for the name of the run-time support routine.

The run-time routine for the OPEN macro is called XXOPEND. For the OPENR macro, the run-time routine is XXOPENR. These macros have three parameters: the file name address, a buffer area, and a volume reference number. The file name is passed in A0, the buffer address in A1, and the volume reference in D1. A zero is passed for missing parameters. The file name must always be specified, however. The I/O reference number is returned in the low-order word of D0. The volume reference number is returned in the low-order word of D1.

Listing 8.43 shows the run-time support routine. The Gopen function starting at line 11 is common to the OPEN, OPENR, and GETFINFO routines. It is also called internally by the OpenRsrc routine, which is called from the run-time initialization. The Gopen routine receives the file name address in A0 and the address of an open function in A6. Gopen tries to open the named file on every mounted drive in the system.

```
        .MACRO   OPEN                ; filename,buffer,volref =
        MOVEM.L  D1/A0/A1,-(SP)      ; Save registers
        LEA      %1,A0               ; A0 -> file name
        LEA      %2,A1               ; A1 -> buffer area
        IF       '%3' = ''
        CLR.W    D1                  ; Do vol scan
        ELSE                         ; Don't
        MOVE.W   %3,D1               ;
        ENDIF
        XREF     XXOPEND             ; RTL routine name
        JSR      XXOPEND             ; Call RTL
        MOVEM.L  (SP)+,D1/A0-A1      ; Restore registers
        .ENDM
```

Listing 8.42 *– The OPEN macro*

Lines 2–3 set up the Gopen parameters to open the data fork of a file. Lines 11–16 attempt to open the file on the default drive (or the drive whose volume reference is contained in D1). If this open fails and the volume reference number was not specified, a volume scan is attempted.

The volume scan begins at line 17. D6 is used to hold the byte count of the specified file name, and A3 is used to point to an operating system data structure called a *volume control block,* or *VCB.* Lines 20–21 test the drive number in the VCB. If this number is zero, the disk is not in any drive. No open is attempted for such a disk. Lines 22–28 move the volume name from the VCB into a run-time library scratch area called FScratch. Line 29 appends a colon character to the volume name. Lines 30–33 append the file name to this string, so that the complete string is

```
vol:filename
```

where "vol" is the volume name and "filename" is the name supplied by the application. Lines 34–38 attempt to open the file name thus constructed. If successful, the BGT at line 38 succeeds, and Gopen exits back to the application. Otherwise, the code at lines 39–42 attempt the next volume. If there is no next volume, the link word in the VCB is zero, and a -1 is returned to the application at line 43.

You could use the volume reference number instead of constructing the file name string, except that the _OpenResFile trap used to open a resource file does not allow specification of a volume reference number.

Listing 8.44 contains the code for the TryDopen routine, which is called by Gopen to open the data fork of a file. The _Open trap at line 9 is a register-based call to the operating system. Register A0 contains the address of an I/O parameter block (IOPB) on entry. For the _Open trap,

```
XXOPEND SAVE                        ;       Entry point
        LEA     TRYDopen,A6         ;       A6 -> Open function
        MOVE.L  A1,Filebuf(A5)      ;       Save buffer address
;
;       General open function. Called by resource and data open
;       front ends.
;
;       A6 -> Routine for open
;       A0 -> File name
;
Gopen:  MOVE.L  A0,A4               ;       Save file name pointer
        JSR     (A6)                ;       Try to open as is
        TST.W   D0                  ;       Success?
        BGT     RQuit               ;       Yes, return
        TST.W   D1                  ;       Volume specified?
        BNE     RQuit               ;       Yes, fail
        CLR.L   D6                  ;       Load
        MOVE.B  (A4),D6             ;               length byte
        MOVE.L  VCBQHdr+2,A3        ;       A3 -> VCB entry
RVScan: TST.W   VCBDrvNum(A3)       ;       Is volume on line?
        BEQ     RNextV              ;       No, get next
        LEA     FScratch+1(A5),A1;          A1 -> scratch area
        LEA     VCBVN(A3),A2        ;       A2 -> volume name
        CLR.L   D7                  ;       Fetch
        MOVE.B  (A2),D7             ;               count byte
        ADD.L   #1,D7               ;       Increment for ":" character
        ADD.W   D6,D7               ;       Compute total string length
        BSR     MoveCnt             ;       Move in volume name
        MOVE.B  #':',(A1)+          ;       Move in separator
        MOVE.L  A4,A2               ;       A2 -> file name
        BSR     MoveCnt             ;       Move file name in, too
        LEA     Fscratch(A5),A0     ;       A0 -> composite name
        MOVE.B  D7,(A0)             ;       Plunk down length byte
        CLR.W   D1                  ;       Default volume number
        JSR     (A6)                ;       Try to open that
        MOVE.W  VCBVREFNum(A3),D1;          Return volume ref number in D1
        TST.W   D0                  ;       Success?
        BGT     RQuit               ;       Yes, stop
RNextV:                             ;       No, chain to next volume
        MOVE.L  (A3),A3             ;       A3 -> next in chain
        MOVE.L  A3,D1               ;       Move to D register
        BNE     RVScan              ;       Try again
        MOVE.L  #-1,D0              ;       Failed utterly
RQuit:  RETURNP                     ;       Restore registers; return
```

Listing 8.43 – *The XXOPEND and Gopen routines*

you must fill in the file name, volume reference number, file type, permission, and private buffer fields. After the _Open, any error code will be in the ioResult field of the IOPB. If this field is zero, the I/O reference number for the file will be in the ioRefNum field.

```
TRYDopen:                          ;       A0 -> file name
         MOVE.L  A0,ioFileName+IOPB(A5); Put in file name
         LEA     IOPB(A5),A0       ;       A0 -> Parameter block
         CLR.L   ioCompletion(A0);         No completion routine
         MOVE.W  D1,ioVRefNum(A0);         Set volume ID
         CLR.W   ioFileType(A0)    ;       Default version
         CLR.B   ioPermssn(A0)     ;       Whatever we can get
         MOVE.L  Filebuf(A5),ioOwnBuf(A0); Special access buffer
         _Open                     ;       Try it
         LEA     IOPB(A5),A0       ;       A0 -> Parameter block again
         MOVE.W  ioResult(A0),D0   ;       Fetch error code
         BNE     OpenFail          ;       IF NE, back to caller
         MOVE.W  ioRefNum(A0),D0   ;       Get return parameter
OpenFail:                          ;       Common exit point
         EXT.L   D0                ;       Convert to long
         RTS                       ;       and back to caller
```

***Listing 8.44** – The TryDopen routine*

As mentioned before, the code for the OPENR macro is nearly identical. The run-time routine is named XXOPENR, and the trap name for the open is _OpenRF. The routine passed to Gopen is called TryRopen. This function opens the resource fork of a file for I/O.

OpenRsrc

Opening an application resource file is a completely different operation. This function requires the _OpenResFile trap. The code that performs this operation is shown in Listing 8.45.

Lines 2–6 are called from the initialization code in Listing 8.3. The routine called from Gopen issues the _OpenResFile trap at line 10. The word returned from this function is an I/O reference number. The run-time library uses it simply as an indication of whether the open attempt was successful.

POPEN

The POPEN macro uses a feature of the Macintosh system called the *standard file package.* A Macintosh *package* is loaded on demand as an overlay from the System file. It is not in the ROM. The code for the POPEN macro is shown in Listing 8.46.

The POPEN macro has four parameters: the address of a type array, the address of a file extension string, the address of a buffer to receive the file name, and the address of a file buffer. These parameters are loaded into address registers in lines 3–6. The run-time support routine is called XXPOPEN. This routine is shown in Listing 8.47.

```
 1: OpenRsrc:                          ;      Entry point
 2:         SAVE                       ;      Save registers
 3:         LEA     TRYRESOP,A6        ;      -> Open function
 4:         CLR.L   Filebuf(A5)        ;      No special file buffer
 5:         CLR.W   D1                 ;      Default volume search
 6:         BRA     Gopen              ;      Go do it
 7: TRYRESOP:                          ;      A0 -> File name
 8:         CLR.W   -(SP)              ;      Clear out room
 9:         MOVE.L  A0,-(SP)           ;      Push file name
10:         _OpenResFile               ;      Open
11:         MOVE.W  (SP)+,D0           ;      Fetch file handle/error code
12:         RTS                        ;      Return to caller
```

***Listing 8.45** - The OpenRsrc routine*

```
 1:         .MACRO  POPEN          ; types,ext,namebuf,buffer =
 2:         MOVEM.L A1-A3,-(SP)    ; Save registers
 3:         LEA     %1,A3          ; A3 -> Type array
 4:         LEA     %2,A0          ; A0 -> Extension string
 5:         LEA     %3,A2          ; A2 -> Buffer for name
 6:         LEA     %4,A1          ; A1 -> buffer
 7:         XREF    XXPOPEN        ; For the linker
 8:         JSR     XXPOPEN        ; Call RTL routine
 9:         MOVEM.L (SP)+,A1-A3    ; Restore registers
10:         .ENDM
11: Cancelled EQU   -1             ; Code for cancelled by user
```

***Listing 8.46** - The POPEN macro*

Registers A0 and A1 are saved in global storage by lines 2–3. The call to the standard file package starts at line 5. The first parameter is the point at which the upper-left corner of the window will appear. The run-time library uses an arbitrary value of (100,100). The second parameter is the address of a function that examines each file for placement in the window. The third parameter is the number of types in the type array (−1 if there is no array). The fourth parameter is the actual type array address. Lines 8–13 load these two parameters from the single address given by the macro. The fifth parameter is the address of a dialog hook procedure to capture and filter events during the package operation. This parameter is always set to zero, to indicate no procedure. The sixth parameter is the address of a standard file reply record, as shown in Figure 8.5. The last parameter is the number of the routine desired. For the POPEN macro, it is the SFGet routine (value of 2).

```
 1: XXPOPEN:                          ;      Entry point
 2:         SAVE                      ;      Save caller's registers
 3:         MOVE.L  A0,ExtPtr(A5)     ;      Save the extension pointer
 4:         MOVE.L  A1,Filebuf(A5)    ;      Place buffer in magic location
 5:         MOVE.L  SfPoint,-(SP)     ;      Push point for box upper left
 6:         CLR.L   -(SP)             ;      Push null prompt string
 7:         PEA     Extfilter         ;      Use standard filter procedure
 8:         MOVE.W  #-1,-(SP)         ;      Push default (no type filtering)
 9:         MOVE.L  A3,D0             ;      Test for
10:         BEQ     NoTypes           ;              types there
11:         MOVE.W  (A3)+,(SP)        ;      Replace default with actual number
12: NoTypes:                          ;
13:         MOVE.L  A3,-(SP)          ;      Push type list address (or nil)
14:         CLR.L   -(SP)             ;      No dialog hook procedure either
15:         PEA     SFReply(A5)       ;      Push reply area pointer
16:         MOVE.W  #SFGet,-(SP)      ;      Push routine indicator
17:         _Pack3                    ;      Try the file get
18:         TST.B   Good(A5)          ;      Test return code
19:         BNE     NotCancelled      ;      NE => Not cancelled
20:         MOVE.L  #Cancel,D0        ;      Return
21:         BRA     XXPFAIL           ;      Failed, return
22: NotCancelled:                     ;      Here if OK
23:         MOVE.W  VolRef(A5),D1     ;      Set volume ID
24:         LEA     Fname(A5),A0      ;      A0 -> File name
25:         JSR     TryDOpen          ;      Try to open
26:         TST.W   D0                ;      Worked?
27:         BMI     XXPFAIL           ;      No, quit now
28:         MOVE.B  Fname(A5),(A2)+   ;      Move in count byte
29:         MOVE.L  A2,A1             ;      A1 -> Destination
30:         LEA     Fname(A5),A2      ;      A2 -> Source
31:         JSR     MoveCnt           ;      Move in name
32:         MOVE.W  ioVRefNum(A0),D1;        Set volume ref number
33: XXPFAIL:
34:         MOVE.L  WindowPtr(A5),-(SP);     Push GrafPort address
35:         _SetPort                  ;      Restore port
36:         RETURNP                   ;      Restore registers and return
```

***Listing 8.47** – The XXPOPEN routine*

The _Pack3 call at line 17 invokes the standard file package. Line 18 tests the return value. If it is nonzero, a file was selected. Otherwise, a value of −1 is returned to the application at line 20.

After selecting the file, we must open it. Lines 23–27 call the TryDOpen routine, which opens the data fork of the file. On a successful open, lines 28–31 place the file name in the application buffer. Line 32 places the volume reference number in D1 for return to the application.

The extension filter routine specified in line 7 is shown in Listing 8.48. This routine is called by the routine invoked by the _Pack3 trap using standard Pascal parameter conventions. There is one input parameter, the address of an I/O parameter block containing the file information.

Offset	Contents	MDS Label
0	Reply Data Valid	rGood
1		
2	File Type (4 bytes)	rType
6	Volume Reference Number (word)	rVolume
8	File Version (Word)	rVersion
10	File Name (64-byte Pascal string)	rName

***Figure 8.5** - Standard file reply record*

The routine returns a single word, which is zero if the file is to be displayed and nonzero otherwise.

Lines 2–3 return zero for every file if no extension was specified in the POPEN macro call. Lines 5–15 compute the address of the first byte of the file name to compare. The specified extension string is compared with the last bytes in the file name at lines 16–26. Lines 18–22 convert lowercase characters in the file name to uppercase for comparison. Lines 27–30 return an indication that the file should be displayed, while lines 31–34 return the opposite indication.

Creating a File

The run-time library supports two methods of creating a file. When the name of the file is specified by the application, the CREATE macro places a new file on the disk and opens the data fork. When the user specifies the file name, the PCREATE macro performs the user dialog, creates the file, and opens the data fork.

CREATE

Listing 8.49 shows the code for the CREATE macro. The macro has three parameters: a file name address, a file buffer address, and a volume reference number. The string DEFAULT can be used for either the buffer address or the volume reference number, or these parameters can be omitted.

```
ExtFilter:                          ;    Filter file names
          TST.L   ExtPtr(A5)        ;    Any to filter?
          BEQ     Success           ;    No, just bail out
          MOVE.L  ExtPtr(A5),A0     ;    A0 -> Extension string
          MOVE.L  4(SP),A1          ;    A1 -> IOPB from SF package
          MOVE.L  ioFileName(A1),A1;     A1 -> File name area
          CLR.L   D0                ;    Load extension
          MOVE.B  (A0)+,D0          ;                     count byte
          CLR.L   D1                ;    Load file name
          MOVE.B  (A1)+,D1          ;                     count byte
          SUB.W   D0,D1             ;    Adjust count
          BLT     Failure           ;    Lengths are incompatible
          LEA     0(A1,D1.W),A1     ;    Point A1 to comparison byte
          SUBQ.W  #1,D0             ;    Adjust for DBRA
          BLE     Failure           ;    If count <= 0, quit
ExtLoop:                            ;    Now compare byte by byte
          MOVE.B  (A1)+,D1          ;    Fetch character
          CMPI.B  #'a',D1           ;    Lower-
          BLT     DoCompare         ;            case?
          CMPI.B  #'z',D1           ;
          BGT     DoCompare         ;
          BCLR    #5,D1             ;    Convert to uppercase
DoCompare:                          ;    Here to compare
          CMP.B   (A0)+,D1          ;    Compare next
          BNE     Failure           ;    If NE, quit
          DBRA    D0,ExtLoop        ;    Repeat for count
Success:                            ;    Here if match
          MOVE.W  #0,8(SP)          ;    Load Boolean TRUE
          MOVE.L  (SP)+,(SP)        ;    Move return -> parameter
          RTS                       ;    Exit
Failure:                            ;    Here if no match
          MOVE.W  #-1,8(SP)         ;    Load Boolean FALSE
          MOVE.L  (SP)+,(SP)        ;    Move return -> parameter
          RTS                       ;    Exit
```

Listing 8.48 *– The ExtFilter routine*

The file name parameter is mandatory. Line 3 loads the file name address into A0. Lines 4–8 place the buffer address (or zero if none was specified) into A1. Lines 9–17 set D1 to the volume reference number. Note the nested conditional assembly. The third parameter can be blank, or it can be the string DEFAULT, both of which are special cases. If some other string is specified, line 15 loads the value into D1.W.

The run-time routine called by the CREATE macro is XXCREATE. This code is shown in Listing 8.50. The routine TryCreate, which is shared by the CREATE and PCREATE code, actually does the work. Lines 7–14 set up the IOPB fields. Line 15 deletes any existing file of the same name, and line 16 creates a new file. This is done so that the application need not call the DELETE macro before calling CREATE. The PCREATE macro prompts the user before destroying an existing file.

```
        .MACRO   CREATE             ; filename,buffer,volref =
        MOVEM.L  A0-A1,-(SP)        ; Save registers
        LEA      %1,A0              ; A0 -> File name
        IF       '%2' <> ''         ; Buffer specified?
        LEA      %2,A1              ; A1 -> Buffer
        ELSE                        ; No buffer
        SUBA.L   A1,A1              ; Clear A1
        ENDIF                       ;
        IF       '%3' = ''          ; Vol ref specified?
          CLR.W D1                  ; No, use zero
        ELSE                        ; Specified,
          IF    '%3' = 'DEFAULT'; take the default?
            CLR.W       D1          ; Do vol scan
           ELSE                     ; Don't
            MOVE.W      %3,D1       ; use specified
          ENDIF                     ; = DEFAULT
        ENDIF                       ; Specified
        XREF     XXCREATE           ; RTL routine name
        JSR      XXCREATE           ; Do the create
        MOVEM.L  (SP)+,A0-A1        ; Restore the registers
        .ENDM
```

Listing 8.49 *– The CREATE macro*

Line 17 gets the file information. Note that A0 is preserved by these calls, so it is not necessary to reload it every time. Lines 18–19 set the Finder file type and application signature, and line 20 writes the corrected file information out to disk. Applications that use a different file type (like MacDoodle) are required to do a SETFINFO call to change the file information from this default. Lines 21–27 open the file and return the I/O reference number in D0.W and the volume reference number in D1.W.

PCREATE

The PCREATE macro code is shown in Listing 8.51. This macro has four address parameters: a prompt, a default file name, a buffer for the actual file name, and a file buffer. These are loaded into address registers A0–A3 in lines 3–6.

The run-time routine for the PCREATE macro is called XXPCREATE. The XXPCREATE code is shown in Listing 8.52. The XXPCREATE routine calls the same _Pack3 trap that the POPEN routine uses. It takes six parameters:

1. The point at which the dialog box is to be displayed
2. The address of the prompt string
3. The address of the default name

```
 1: XXCREATE:                          ;
 2:         SAVE                       ;      Save callers registers
 3:         JSR     TryCreate          ;      Call internal routine
 4:         RETURNP                    ;      Restore all but D0/D1
 5:
 6: TryCreate:                         ;      Here if locally called
 7:         MOVE.L  A0,ioFileName+IOPB(A5); Set file name
 8:         MOVE.L  A1,A3              ;      Save A1
 9:         LEA     IOPB(A5),A0        ;      A0 -> IOPB
10:         CLR.L   ioCompletion(A0);         No completion routine
11:         MOVE.W  D1,ioVRefNum(A0);         Specified volume
12:         CLR.W   ioFileType(A0)     ;      Default version
13:         CLR.B   ioPermssn(A0)      ;      Whatever we can get
14:         CLR.L   ioOwnBuf(A0)       ;      No special access buffer
15:         _Delete                    ;      Get rid of existing version
16:         _Create                    ;      Do it
17:         _GetFileInfo               ;      Get file information
18:         MOVE.L  #'TEXT',ioFlUsrWds(A0); Set file type
19:         MOVE.L  #'EDIT',ioFlUsrWds+4(A0); Set application
20:         _SetFileInfo               ;      Set the info
21:         MOVE.L  A3,ioOwnBuf(A0)    ;      Set buffer address
22:         _Open
23:         MOVE.W  ioResult(A0),D0    ;      Get error code
24:         EXT.L   D0                 ;      Extend to long
25:         BNE     CreateNFG          ;      Failed if nonzero
26:         MOVE.W  ioRefNum(A0),D0    ;      Get reference number
27:         MOVE.W  ioVRefNum(A1),D1;         Return volume reference number
28: CreateNFG:                         ;      Common exit point
29:         RTS                        ;      Return to caller
```

***Listing 8.50** – The XXCREATE routine*

4. The address of the dialog hook procedure (always set to zero—no procedure—by the run-time library)

5. The address of the reply buffer (see Figure 8.4)

6. The constant 1, indicating the put file procedure of the package

The standard file package takes care of the user dialog, including the prompt for overwriting an existing file. On return, the file name is in the reply area. Lines 16–19 call the TryCreate routine in Listing 8.50 to create the file. Lines 22–25 copy the file name from the standard file buffer to the application-supplied area. Line 26 copies the volume reference to D1.W for return to the application. The XXPFAIL label is contained in Listing 8.47.

Closing a File

The CLOSE macro disconnects an application from a file opened by an OPEN, OPENR, POPEN, CREATE, or PCREATE call. The code for this macro is shown in Listing 8.53.

The macro has a single parameter, the I/O reference number, which is passed to the run-time library in D0.W. The run-time routine is XXCLOSE and appears in Listing 8.54.

Lines 3–4 set up the IOPB (only one field is needed). The _Close trap closes the file. Line 6 calls an internal routine that instructs the operating system to write any data still in memory out to the disk. This internal routine is part of the EXIT code; it is contained in Listing 8.75.

Transferring Data

Data is copied between application memory and the file system via the READ, READLN, and WRITE macros. The file must have been opened by one of the OPEN or CREATE routines discussed previously.

READ and READLN

The READ and READLN macros are shown in Listings 8.55 and 8.56, respectively. Each macro requires an I/O reference number, a buffer address, and a buffer length. These are placed in registers D0.W, A0, and D1.L, respectively. Register D2.W receives a zero in the READ macro and the constant 3456 (decimal) in the READLN macro.

Both of these macros call the run-time support routine XXREAD, which appears in Listing 8.57. Lines 2–7 set up the IOPB for the read, storing the registers (which contain the macro parameters) in the appropriate places

```
        MACRO   PCREATE prompt,defname,namebuf,buffer   =
        MOVEM.L A0-A3,-(SP)      ; Save registers
        LEA     {prompt},A0      ; A0 -> Prompt string
        LEA     {defname},A3     ; A3 -> Default name
        LEA     {namebuf},A2     ; A2 -> Name buffer area
        LEA     {buffer},A1      ; A1 -> 522-byte buffer
        XREF    XXPCREATE        ; For the linker
        JSR     XXPCREATE        ; Call RTL routine
        MOVEM.L (SP)+,A0-A3      ; Restore registers
        |
```

Listing 8.51 – *The PCREATE macro*

in the IOPB. D2.W is stored in the ioPosMode offset of the IOPB. A value of zero in this field indicates that the read is terminated by the byte count. The read may also be terminated by the end of the file, if there are not enough data left in the file to satisfy the request. If bit 7 of the ioPosMode field is set, the high byte of this word is taken as a terminator character.

```
XXPCREATE:                              ;       Entry point
          SAVE                          ;       Save registers
          MOVE.L  A1,Filebuf(A5)        ;       Save buffer address
          MOVE.L  SFPoint,-(SP)         ;       Point for dialog box
          MOVE.L  A0,-(SP)              ;       Prompt string
          MOVE.L  A3,-(SP)              ;       Default name
          CLR.L   -(SP)                 ;       No dialog hook procedure
          PEA     SFReply(A5)           ;       -> Reply area
          MOVE.W  #SFPut,-(SP)          ;       Routine selector
          _Pack3                        ;       Invoke package
          TST.B   Good(A5)              ;       Test return code
          BNE     NoCancel              ;       NE => Not cancelled
          MOVE.L  #Cancel,D0            ;       Return
          BRA     XXPFAIL               ;       Failed, return
NoCancel:                               ;       Here if OK
          MOVE.W  VolRef(A5),D1         ;       Set volume ID
          LEA     Fname(A5),A0          ;       A0 -> File name
          MOVE.L  Filebuf(A5),A1        ;       Restore A1
          JSR     TryCreate             ;       Try to create it
          TST.W   D0                    ;       Worked?
          BMI     XXPFAIL               ;       No, quit now
          MOVE.B  Fname(A5),(A2)+       ;       Move in count byte
          MOVE.L  A2,A1                 ;       A1 -> Destination
          LEA     Fname(A5),A2          ;       A2 -> Source
          JSR     MoveCnt               ;       Move in name
          MOVE.W  ioVRefNum(A0),D1;             Set volume ref number
          BRA     XXPFAIL               ;       Go exit
```

***Listing 8.52** – The XXPCREATE routine*

```
          MACRO   CLOSE   filenum =
          MOVE.L  D0,-(SP)          ; Save D0
          MOVE.W  {filenum},D0      ; D0 = File number to close
          XREF    XXCLOSE           ; Define RTL routine name
          JSR     XXCLOSE           ; Call RTL
          MOVE.L  (SP)+,D0          ; Restore D0
          |
```

***Listing 8.53** – The CLOSE macro*

```
        XDEF     XXCLOSE         ;       For linker
XXCLOSE SAVE                     ;       Save caller's registers
        LEA      IOPB(A5),A0     ;       A0 -> Parameter block
        MOVE.W   D0,ioRefNum(A0) ;       Set reference number
        _Close                   ;       Close the file
        BSR      DoFlush         ;       Flush all volumes
        RETURNALL                ;       Return to caller
```

Listing 8.54 *- The XXCLOSE routine*

```
        MACRO    READ     filenum,buffer,count =
        MOVEM.L  D1-D2/A0,-(SP)  ; Save registers
        MOVE.W   {filenum},D0    ; Load file number
        LEA      {buffer},A0     ; A0 -> Buffer address
        MOVE.L   {count},D1      ; D1 = Byte count
        CLR.W    D2              ; Indicate no newline
        XREF     XXREAD          ; Define RTL routine name
        JSR      XXREAD          ; Call RTL
        MOVEM.L  (SP)+,D1-D2/A0  ; Restore registers
        |
```

Listing 8.55 *- The READ macro*

```
        MACRO    READLN   filenum,buffer,count =
        MOVEM.L  D1-D2/A0,-(SP)  ; Save registers
        MOVE.W   {filenum},D0    ; Load file number
        LEA      {buffer},A0     ; A0 -> Buffer address
        MOVE.L   {count},D1      ; D1 = Byte count
        MOVE.W   #3456,D2        ; CR terminator + flag
        XREF     XXREAD          ; Define RTL routine name
        JSR      XXREAD          ; Call RTL
        MOVEM.L  (SP)+,D1-D2/A0  ; Restore registers
        |
```

Listing 8.56 *- The READLN macro*

The read may then also be terminated if this character is encountered. A value of 3456 (D80 hex) causes the read to terminate at the next carriage return. This reads one line from a Macintosh textfile.

Lines 11–16 check for an error on the read operation. If an end-of-file error was returned but some bytes were transferred, no error is returned.

A subsequent READ macro will return the end-of-file error. Line 17 loads the actual number of bytes read into D0.L for return to the application.

WRITE

A Macintosh file write operation is very similar to a read operation. There is no support for terminator characters, however, so there is no WRITELN macro. Listing 8.58 contains the WRITE macro code.

Registers D0.W, A0, and D1.L are used for the same parameters as the READ macro. Register D2 is not used by the WRITE code. The run-time routine is called XXWRITE and appears in Listing 8.59.

```
 1: XXREAD: SAVE                      ;      Save caller's registers
 2:         LEA     IOPB(A5),A1       ;      A1 -> Parameter block
 3:         MOVE.W  D0,ioRefNum(A1)   ;      Store file reference number
 4:         MOVE.L  D1,ioByteCount(A1);      Store byte count
 5:         MOVE.L  A0,ioBuffer(A1)   ;      and buffer address
 6:         MOVE.W  D2,ioPosMode(A1);        Specify sequential read
 7:         MOVE.L  A1,A0             ;      A0 -> Parameter block
 8:         _Read                     ;      Issue the read
 9:         LEA     IOPB(A5),A0       ;      A0 -> Parameter block again
10:         MOVE.W  ioResult(A0),D0   ;      Fetch result code
11:         EXT.L   D0                ;      Extend to long
12:         BEQ     Noerr             ;      No error
13:         CMPI.L  #eofERR,D0        ;      End of file error?
14:         BNE     ReadDone          ;      No, return error
15:         TST.L   ioNumDone(A0)     ;      Any bytes read?
16:         BEQ     ReadDone          ;      No, return error
17: Noerr:  MOVE.L  ioNumDone(A0),D0;        Not (really) an error, return byte count
18: ReadDone:                         ;      Here when done
19:         RETURN                    ;      Restore the registers
```

***Listing 8.57** – The XXREAD routine*

```
        MACRO   WRITE   filenum,buffer,count =
        MOVEM.L D1-D2/A0,-(SP)  ; Save registers
        MOVE.W  {filenum},D0    ; Load file number
        LEA     {buffer},A0     ; A0 -> Buffer address
        MOVE.L  {count},D1      ; D1 = Byte count
        XREF    XXWRITE         ; Define RTL routine name
        JSR     XXWRITE         ; Call RTL
        MOVEM.L (SP)+,D1-D2/A0  ; Restore registers
        |
```

***Listing 8.58** – The WRITE macro*

Lines 2–7 set up the IOPB for the _Write trap. Following the write, lines 9–12 check for errors and line 13 loads the byte count into D0.L if no error occurred. Note that it is not necessary to check for a shortened return on writes.

SEEK

The SEEK macro allows applications to access file data in any order. This macro requires the I/O reference number, an offset, and a sense word, which are placed in registers D0.W, D1.L, and D2.W, respectively. This macro is presented in Listing 8.60.

The run-time routine for the SEEK macro is XXSEEK, shown in Listing 8.61. Lines 1–7 set up the IOPB for the _SetFPos (set file position) trap.

```
 1: XXWRITE SAVE                      ;      Save caller's registers
 2:         LEA     IOPB(A5),A1       ;      A1 -> Parameter block
 3:         MOVE.W  D0,ioRefNum(A1)   ;      Store file reference number
 4:         MOVE.L  D1,ioByteCount(A1);      Store byte count
 5:         MOVE.L  A0,ioBuffer(A1)   ;      and buffer address
 6:         CLR.W   ioPosMode(A1)     ;      Indicate sequential
 7:         MOVE.L  A1,A0             ;      A0 -> Parameter Block
 8:         _Write                    ;      Issue the write
 9:         LEA     IOPB(A5),A0       ;      A0 -> Parameter block again
10:         MOVE.W  ioResult(A0),D0   ;      Fetch result code
11:         EXT.L   D0                ;      Extend to long
12:         BNE     WriteDone         ;      NE => Error, done
13:         MOVE.L  ioNumDone(A0),D0; ;      No error, return actual byte count
14: WriteDone:                        ;      Here when done
15:         RETURN                    ;      Restore the registers
```

Listing 8.59 *– The XXWRITE routine*

```
MACRO    SEEK    ioref,offset,sense =
MOVEM.L  D1-D2,-(SP)      ; Save input registers
MOVE.W   {ioref},D0       ; Load ioref #
MOVE.L   {offset},D1      ; Load offset
MOVE.W   {sense},D2       ; and sense word
XREF     XXSEEK           ; For the linker
JSR      XXSEEK           ; Call RTL routine
MOVEM.L  (SP)+,D1-D2      ; Restore input registers
|
```

Listing 8.60 *– The SEEK macro*

Lines 5–6 translate the sense value from a UNIX-compatible to a Macintosh-compatible value. Lines 9–12 check for errors, and lines 13–15 fetch the resulting absolute seek value. This allows a single macro to both get and set the file position. Seeking 0 bytes from the current position returns the current position.

Deleting a File

The DELETE macro removes an existing file from a disk. This macro appears in Listing 8.62. It has two parameters: a file name address and a volume reference number. Register A0 receives the file name address, and register D1.W receives the volume reference number.

The run-time routine is called XXDELETE and is shown in Listing 8.63. This routine uses the Gopen volume scan routine described earlier. The TryDel routine is called by Gopen to attempt a delete. Lines 9–12 set up the IOPB for the _Delete trap at line 13. Lines 14–15 retrieve the return code and place it in D0.L.

Getting and Setting File Information

The GETFINFO and SETFINFO macros get and set the Macintosh file information. These macros use an 80-byte buffer, as described in Chapter 4.

```
 1: XXSEEK: SAVE                       ;     Seek on an open file
 2:         LEA     IOPB(A5),A0        ;     A1 -> Parameter block
 3:         MOVE.W  D0,ioRefNum(A0)    ;     Store file reference number
 4:         MOVE.L  D1,ioPosOffset(A0);      Store offset
 5:         ANDI.W  #3,D2              ;     Insure proper range
 6:         MOVE.B  SeekTab(D2),D2     ;     Load seek table equivalent
 7:         MOVE.W  D2,ioPosMode(A0);        Set proper field
 8:         _SetFPos                   ;     Do the set
 9:         LEA     IOPB(A5),A0        ;     A0 -> Parameter block
10:         MOVE.W  ioResult(A0),D0    ;     Fetch error code
11:         EXT.L   D0                 ;     Out to long
12:         BMI     SeekDone           ;     < 0 means error
13:         _GetFPos                   ;     Fetch new pointer
14:         LEA     IOPB(A5),A0        ;     A0 -> Parameter block
15:         MOVE.L  ioPosOffset(A0),D0;      Load return value
16: SeekDone:                          ;     Here to return to caller
17:         RETURN                     ;     Restore caller's registers
18: SeekTab DC.B    1,3,2              ;     Seek translation table
```

Listing 8.61 – *The XXSEEK routine*

```
          .MACRO   DELETE
          MOVEM.L  A0/D1,-(SP)       ; Save input registers
          LEA      %1,A0             ; Load file name address
          IF       '%2' <> ''        ; Is name null?
          MOVE.W   %2,D1             ; Load volume reference
          ELSE                       ; Not null,
          CLR.W    D1                ;                make it zero
          ENDIF                      ;
          XREF     XXDELETE          ; For the linker
          JSR      XXDELETE          ; Call RTL routine
          MOVEM.L  (SP)+,A0/D1       ; Restore registers
          .ENDM
```

Listing 8.62 *– The DELETE macro*

GETFINFO

The GETFINFO macro returns the file information for an existing file. This macro has three parameters: a file name address, the address of an 80-byte buffer, and a volume reference number. The volume reference number can be omitted. The GETFINFO macro definition appears in Listing 8.64.

Lines 3–4 place the file name address in register A0 and the buffer address in register A1. Lines 5–13 place the volume reference number in D1.W. This parameter can be omitted or set to the string DEFAULT, either

```
XXDELETE:                         ;
        SAVE                      ;       Save the registers
        LEA     TryDel,A6         ;       Do volume scan for file
        BRA     Gopen             ;       Do general open stuff
;
;       TryDel routine.  Attempts to delete a file.
;
TryDel:                           ;
        MOVE.L  A0,IOPB+ioFileName(A5); Stuff file name
        LEA     IOPB(A5),A0       ;       A0 -> IOPB now
        MOVE.W  D1,ioVRefNum(A0); Clear vol ref field
        CLR.W   ioFDirIndex(A0)   ;       Clear directory index field
        _Delete                   ;       Try it
        MOVE.W  ioResult(A0),D0   ;       Load return code
        EXT.L   D0                ;       Extend to long
        RTS                       ;       and bail out
```

Listing 8.63 *– The XXDELETE routine*

```
        .MACRO  GETFINFO          ; filename,buffer,volref =
        MOVEM.L A0/A1/D1,-(SP)    ; Save registers
        LEA     %1,A0             ; A0 -> File name
        LEA     %2,A1             ; A1 -> Buffer area
        IF      '%3' <> ''        ; Vol ref specified?
          IF    '%3' = 'DEFAULT'
            CLR.W      D1         ; Do vol scan
          ELSE                    ; Don't
            MOVE.W %3,D1          ; Do specific vol
          ENDIF                   ; DEFAULT
        ELSE                      ;
          CLR.W D1                ; Not specified
        ENDIF                     ; Specified
        XREF    XXGETF            ; RTL routine name
        JSR     XXGETF            ; Call RTL
        MOVEM.L (SP)+,A0/A1/D1    ; Restore registers
        .ENDM
```

Listing 8.64 *- The GETINFO macro*

of which will place a zero in D1.W. The run-time routine is called XXGETF and is shown in Listing 8.65.

The XXGETF routine uses the Gopen code described in Listing 8.43. Gopen calls the label TryGetF at line 9 to attempt to get the information for a file. Lines 10–13 set up the IOPB for the _GetFileInfo trap at line 14. A successful trap causes the code at lines 17–22 to move the entire 80-byte IOPB back to the application-specified buffer. The I/O reference number or error code is loaded into D0.W before return.

SETFINFO

The SETFINFO macro performs the inverse of GETFINFO. The 80-byte file information area is associated with an existing file on the disk. A subsequent GETFINFO will return the updated information. The SETFINFO macro appears in Listing 8.66.

This macro has the same parameters as the GETFINFO macro: a file name address, a buffer address, and a volume number. Lines 2–4 place the file name address in register A0 and the buffer address in register A1. The volume reference number can be omitted or set to the string DEFAULT, either of which places a zero in D1.W. A nonblank volume reference number different from DEFAULT is placed in D1.W at line 9. The run-time routine is called XXSETF and appears in Listing 8.67.

Lines 2–6 move the 80-byte area into the IOPB in the run-time library data area. Line 7 places the file name address into the IOPB in preparation for the _SetFileInfo trap at line 10. Lines 11–13 return to the application.

```
XXGETF:                              ;        Entry point
          SAVE                       ;        Save caller's registers
          LEA     TryGetF,A6         ;        Do volume scan for file
          MOVE.L  A1,FileBuf(A5)     ;        Save A1 in global area
          BRA     Gopen              ;        Do general open stuff
;
;         TryGetF routine.  Attempts to get file info for file.
;
TryGetF:                             ;
          MOVE.L  A0,IOPB+ioFileName(A5); Stuff file name
          LEA     IOPB(A5),A0        ;        A0 -> IOPB now
          MOVE.W  D1,ioVRefNum(A0);           Clear vol ref field
          CLR.W   ioFDirIndex(A0) ;           Clear directory index field
           _GetFileInfo              ;        Try it
          MOVE.W  ioResult(A0),D0 ;           Error?
          BNE     GETNFG             ;        Yes, return error
          MOVE.W  #ioFQElSize-1,D0;           Load count
          LEA     IOPB(A5),A2        ;        A2 -> IOPB
          MOVE.L  FileBuf(A5),A1     ;        A1 -> User's buffer
GetMove:                             ;        Move it in
          MOVE.B  (A2)+,(A1)+        ;        Stuff a byte
          DBRA    D0,GetMove         ;        Until done
          MOVE.W  ioRefNum(A0),D0 ;           Load ref number
GETNFG:   RTS                        ;        and bail out
```

Listing 8.65 *– The XXGETF routine*

```
          .MACRO  SETFINFO        ; filename,buffer,volref =
          MOVEM.L A0/A1/D1,-(SP)  ; Save registers
          LEA     %1,A0           ; A0 -> File name
          LEA     %2,A1           ; A1 -> Buffer area
          IF      '%3' <> ''      ; Vol ref specified?
            IF    '%3' = 'DEFAULT'
              CLR.W       D1      ; Do vol scan
            ELSE                  ; Don't
              MOVE.W      %3,D1   ; Do specific vol
            ENDIF                 ; DEFAULT
          ELSE                    ;
            CLR.W D1              ; Not specified
          ENDIF                   ; Specified
          XREF    XXSETF          ; RTL routine name
          JSR     XXSETF          ; Call RTL
          MOVEM.L (SP)+,A0/A1/D1  ; Restore registers
          .ENDM
```

Listing 8.66 *– The SETINFO macro*

```
XXSETF: SAVE                        ;       Save registers
        LEA     IOPB(A5),A2         ;       A2 -> IOPB
        MOVE.W  #ioFQElSize-1,D0;           D0 = Count
SetMove:                            ;       Move parameters back
        MOVE.B  (A1)+,(A2)+         ;       Move a byte
        DBRA    D0,SetMove          ;       Loop till done
        MOVE.L  A0,IOPB+ioFileName(A5);     Stuff file name
        LEA     IOPB(A5),A0         ;       A0 -> IOPB now
        MOVE.W  D1,ioVRefNum(A0);           Set volume reference number
        _SetFileInfo                ;       Try it
        MOVE.W  ioResult(A0),D0     ;       Get result code
        EXT.L   D0                  ;       Extend to long
        RETURN                      ;       and bail out
```

Listing 8.67 *- The XXSETF routine*

MENU ROUTINES

The run-time library supports several routines that assist the application in handling Macintosh menus:

1. The CHECK and UNCHECK macros indicate which items in a menu are currently active.
2. The ENABLE and DISABLE macros dynamically control whether an item in a menu can be selected.
3. The STYLE macro sets the type style of a menu item.

Each of these routines will now be described in detail.

CHECK and UNCHECK

The source for the CHECK and UNCHECK macros appears in Listing 8.68. Each of these macros has two parameters: a menu resource ID and an item number. These are placed in D0.W and D1.W, respectively.

The run-time routines are XXCHECK and XXUNCHECK; these appear in Listing 8.69. Lines 4–5 and 8 place the appropriate character in register D2. A check mark has a value of 12 hex, and a zero value causes the check mark to be erased. Lines 9–13 perform a _GetResource call to return the menu handle for the menu ID passed in D0.W. Lines 15–17

```
          .MACRO   CHECK
          MOVEM.L  D0-D1,-(SP)      ;       Push registers
          MOVE.W   %1,D0            ;       Menu to D0
          MOVE.W   %2,D1            ;       Item to D1
          XREF     XXCHECK          ;       For the linker
          JSR      XXCHECK          ;       Call RTL
          MOVEM.L  (SP)+,D0-D1      ;       Restore registers
          .ENDM

          .MACRO   UNCHECK
          MOVEM.L  D0-D1,-(SP)      ;       Push registers
          MOVE.W   %1,D0            ;       Menu to D0
          MOVE.W   %2,D1            ;       Item to D1
          XREF     XXUNCHECK        ;       For the linker
          JSR      XXUNCHECK        ;       Call RTL
          MOVEM.L  (SP)+,D0-D1      ;       Restore registers
          .ENDM
```

Listing 8.68 – *The CHECK and UNCHECK macros*

```
          XDEF     XXCHECK          ;       For the
          XDEF     XXUNCHECK        ;             linker
XXCHECK   SAVE                      ;       Save registers
          MOVE.W   #CheckMark,D2    ;       Set mark character
          BRA      DoCheck          ;       Go do it
XXUNCHECK                           ;       Here to undo it
          SAVE                      ;       Save registers
          MOVE.W   #noMark,D2       ;       Set blank (null) mark
DoCheck   MOVE.W   D1,D3            ;       Preserve item number
          CLR.L    -(SP)            ;       Result area
          MOVE.L   #'MENU',-(SP)    ;       Specify resource type
          MOVE.W   D0,-(SP)         ;       Convert
          _GetResource              ;             ID
                                    ;                 to handle (SP)
          MOVE.W   D3,-(SP)         ;       Push item number
          MOVE.W   D2,-(SP)         ;       Push mark character
          _SetItmMark               ;       Mark it
          RETURN                    ;       and return
```

Listing 8.69 – *The XXCHECK and XXUNCHECK routines*

perform a _SetItmMark call, which requires the menu handle, item number, and mark character as parameters. The mark charater is placed in front of the menu selection. Note that the return value from _GetResource is simply left on the stack to serve as a parameter to _SetItmMark.

The following characters are legal in a _SetItmMark call:

Value (hex)	Character
0	Blank, removes any existing mark
11	Command key symbol
12	Check mark
13	Diamond symbol
14	Apple symbol

ENABLE and DISABLE

The ENABLE and DISABLE macros are similar to CHECK and UNCHECK. Listing 8.70 contains the ENABLE and DISABLE macro definitions. The parameters are the same as for CHECK and UNCHECK: a menu resource ID and an item number. These are loaded into D0.W and D1.W, respectively.

The run-time routines are XXENABLE and XXDISABLE, shown in Listing 8.71. Register D3.W is used as a flag to indicate whether an ENABLE or DISABLE operation is in progress. This flag is set to 1 for ENABLE and 0 for DISABLE. Lines 10–14 call _GetResource to get the menu handle, which is left on the stack to serve as a parameter to the next system call. Lines 15–22 execute either an _EnableItem or _DisableItem system call to complete the operation.

```
        .MACRO   ENABLE
        MOVEM.L  D0-D1,-(SP)     ;       Push registers
        MOVE.W   %1,D0           ;       Menu to D0
        MOVE.W   %2,D1           ;       Item to D1
        XREF     XXENABLE        ;       For the linker
        JSR      XXENABLE        ;       Call RTL
        MOVEM.L  (SP)+,D0-D1     ;       Restore registers
        .ENDM

        .MACRO   DISABLE
        MOVEM.L  D0-D1,-(SP)     ;       Push registers
        MOVE.W   %1,D0           ;       Menu to D0
        MOVE.W   %2,D1           ;       Item to D1
        XREF     XXDISABLE       ;       For the linker
        JSR      XXDISABLE       ;       Call RTL
        MOVEM.L  (SP)+,D0-D1     ;       Restore registers
        .ENDM
```

***Listing 8.70** – The ENABLE and DISABLE macros*

STYLE

The STYLE macro sets the style of the text in a menu selection. The macro definition appears in Listing 8.72. This macro has three parameters: a menu resource ID, an item number, and a style word. These are placed in registers D0.W, D1.W, and D2.W, respectively. The style word is explained in Chapter 6.

```
         XDEF     XXENABLE          ;       For the
         XDEF     XXDISABLE         ;                linker
XXENABLE                            ;       Enable entry point
         SAVE                       ;       Save registers
         MOVE.W   #1,D3             ;       Set flag register
         BRA      XXENDIS           ;       Branch around disable entry
XXDISABLE                           ;       Here to disable
         SAVE                       ;       Save the registers
         CLR.W    D3                ;       Clear flag register
XXENDIS  MOVE.W   D1,D4             ;       Preserve item number
         CLR.L    -(SP)             ;       Room for handle return
         MOVE.L   #'MENU',-(SP)     ;       Resource type
         MOVE.W   D0,-(SP)          ;       Resource ID
         _GetResource               ;       Read resource into memory
         MOVE.W   D4,-(SP)          ;       Push item number
         TST.W    D3                ;       Enable?
         BEQ      DoDisable         ;       No, disable
         _EnableItem                ;       Do the enable
         RETURN                     ;       Back to caller
DoDisable                           ;       Here to disable
         _DisableItem               ;       Disable it
         RETURN                     ;       Back to caller
```

***Listing 8.71** – The XXENABLE and XXDISABLE routines*

```
         .MACRO   STYLE
         MOVEM.L  D0-D2,-(SP)       ;       Push registers
         MOVE.W   %1,D0             ;       Menu to D0
         MOVE.W   %2,D1             ;       Item to D1
         MOVE.W   %3,D2             ;       Style word to D2
         XREF     XXSTYLE           ;       For the linker
         JSR      XXSTYLE           ;       Call RTL
         MOVEM.L  (SP)+,D0-D2       ;       Restore registers
         .ENDM
```

***Listing 8.72** – The STYLE macro*

The run-time routine is XXSTYLE and appears in Listing 8.73. Lines 5–9 perform a _GetResource call to return the menu handle on the stack. This handle is the first parameter to the _SetItmStyle call at line 13. Other parameters are the item number and style word.

MISCELLANEOUS RUN-TIME LIBRARY ROUTINES

There are three routines that do not fall into any of the categories previously explained:

1. The EXIT routine, which returns to the Finder
2. The QUIT routine, which exits after a Quit menu choice
3. The WTITLE routine, which sets the window title

EXIT

The EXIT macro definition appears in Listing 8.74. It is a simple jump to the run-time routine XXEXIT. The XXEXIT code appears in Listing 8.75.

Line 1 calls the DoFlush routine to ensure that all disk data in memory are written to disk before exiting. Lines 2–3 retrieve the original stack pointer and the return address for the Finder. The JMP instruction at line 4 terminates the program.

The DoFlush routine at line 9 traverses the volume control block list in a manner similar to the Gopen routine used to search all mounted disks.

```
 1:         XDEF    XXSTYLE         ;       For the linker
 2: XXSTYLE                         ;
 3:         SAVE                    ;       Save registers
 4:         MOVE.W  D1,D3           ;       Preserve item number
 5:         MOVE.W  D2,D4           ;       and style word
 6:         CLR.L   -(SP)           ;       Result area
 7:         MOVE.L  #'MENU',-(SP)   ;       Specify resource type
 8:         MOVE.W  D0,-(SP)        ;       Convert
 9:         _GetResource            ;               ID
10:                                 ;                   to handle (SP)
11:         MOVE.W  D3,-(SP)        ;       Push item number
12:         MOVE.W  D4,-(SP)        ;       Push style word
13:         _SetItmStyle            ;       Change it
14:         RETURN                  ;       and return
```

Listing 8.73 – *The XXSTYLE routine*

```
        MACRO   EXIT    =
        XREF    XXEXIT          ; For linker
        JMP     XXEXIT          ; Call RTL exit routine
        |
```

Listing 8.74 *– The EXIT macro*

```
 1: XXEXIT: BSR     DoFlush            ;       Flush vols first
 2:         MOVE.L  EXITAddr(A5),A0 ;       A0 -> Exit routine
 3:         MOVE.L  ORGSP(A5),SP       ;       Take stack to original
 4:         JMP     (A0)               ;       Quit
 5: ;
 6: ;       DoFlush routine.  Flushes all volumes.
 7: ;
 8: ;
 9: DoFlush MOVE.L  VCBQHdr+2,A3       ;       A3 -> VCB entry
10: EVScan: TST.W   VCBDrvNum(A3)      ;       Is volume on line?
11:         BEQ     ENextV             ;       No, get next
12:         LEA     IOPB(A5),A0        ;       A0 -> I/O parameter block
13:         MOVE.L  VCBVN(A3),ioVNPtr(A0);     Set volume name
14:         CLR.W   ioVDrvNum(A0)      ;       Specify no drive number
15:         _FlushVol                  ;       Flush the volume
16: ENextV:                            ;       No, chain to next volume
17:         MOVE.L  (A3),A3            ;       A3 -> Next in chain
18:         MOVE.L  A3,D1              ;       Move to D reg
19:         BNE     EVScan             ;       Try again
20:         RTS                        ;       Back to caller
```

Listing 8.75 *– The XXEXIT routine*

The _FlushVol call at line 15 uses an IOPB, which contains the volume name address. The address of volume name in the volume control block is specified in line 13. This routine is also called by the XXCLOSE routine in Listing 8.54.

QUIT

The QUIT macro is shown in Listing 8.76. This macro issues a GETEVENT call and checks the return for the first selection from the second (File) menu. If GETEVENT returns anything other than this event, the macro simply calls GETEVENT again. When the user selects Quit, the macro issues an EXIT call.

```
        MACRO   QUIT           =
@1:     GETEVENT               ;       Get next event
        CMP.W   #16,D0         ;       Menu choice
        BNE     @1             ;       No, get next event
        CMP.L   #$20001,D1     ;       File menu/Quit?
        BNE     @1             ;       No, get next
        EXIT                   ;       Now quit
        |
```

***Listing 8.76** – The QUIT macro*

Note that this macro is useful only if the File menu contains Quit as its first choice. This is the case only for simple programs, such as those contained in Chapter 4.

WTITLE

The WTITLE macro (Listing 8.77) changes the title in the current application window. The macro loads the address of the new title string into register A0 and calls the run-time routine XXWTITLE (Listing 8.78). The _SetWTitle trap at line 6 of the XXWTITLE routine takes the window address and title string as parameters.

SUMMARY

The code in this chapter illustrates a methodology for reducing the complexity of writing a Macintosh application. It simplifies the use of the most common functions, at the expense of program flexibility.

You have seen how to write code that you can reuse with each new program you write, as well as how to add a macro library interface that

```
        .MACRO  WTitle
        MOVE.L  A0,-(SP)       ;       Save A0
        LEA     %1,A0          ;       A0 -> new string
        XREF    XXWTITLE       ;       Call
        JSR     XXWTITLE       ;               RTL routine
        MOVE.L  (SP)+,A0       ;       Pop A0
        .ENDM
```

***Listing 8.77** – The WTITLE macro*

```
        XDEF    XXWTITLE        ;       For the linker
XXWTITLE                        ;
        SAVE                    ;       Save the registers
        MOVE.L  WindowPtr(A5),-(SP);    Push window address
        MOVE.L  A0,-(SP)        ;       and string address
        _SetWTitle              ;       Set the title
        RETURNALL               ;       Return to caller
```

***Listing 8.78** - The XXWTITLE routine*

makes using this code simpler. Programming effort can be measured in terms of the number of lines of code you have to write. Since you can treat the macro calls as a single line of code, they represent a considerable reduction in length and complexity over the 40–60 lines that some of them actually take.

In addition, the event system (the GETEVENT macro) standardizes a large amount of code that every Macintosh application must contain. Code that relates to other windows on the screen, such as desk accessories, is completely isolated in the run-time library. Menu selection is considerably less complicated, as are window sizing and window positioning. Scrolling and window closing are also aided, though to a lesser extent.

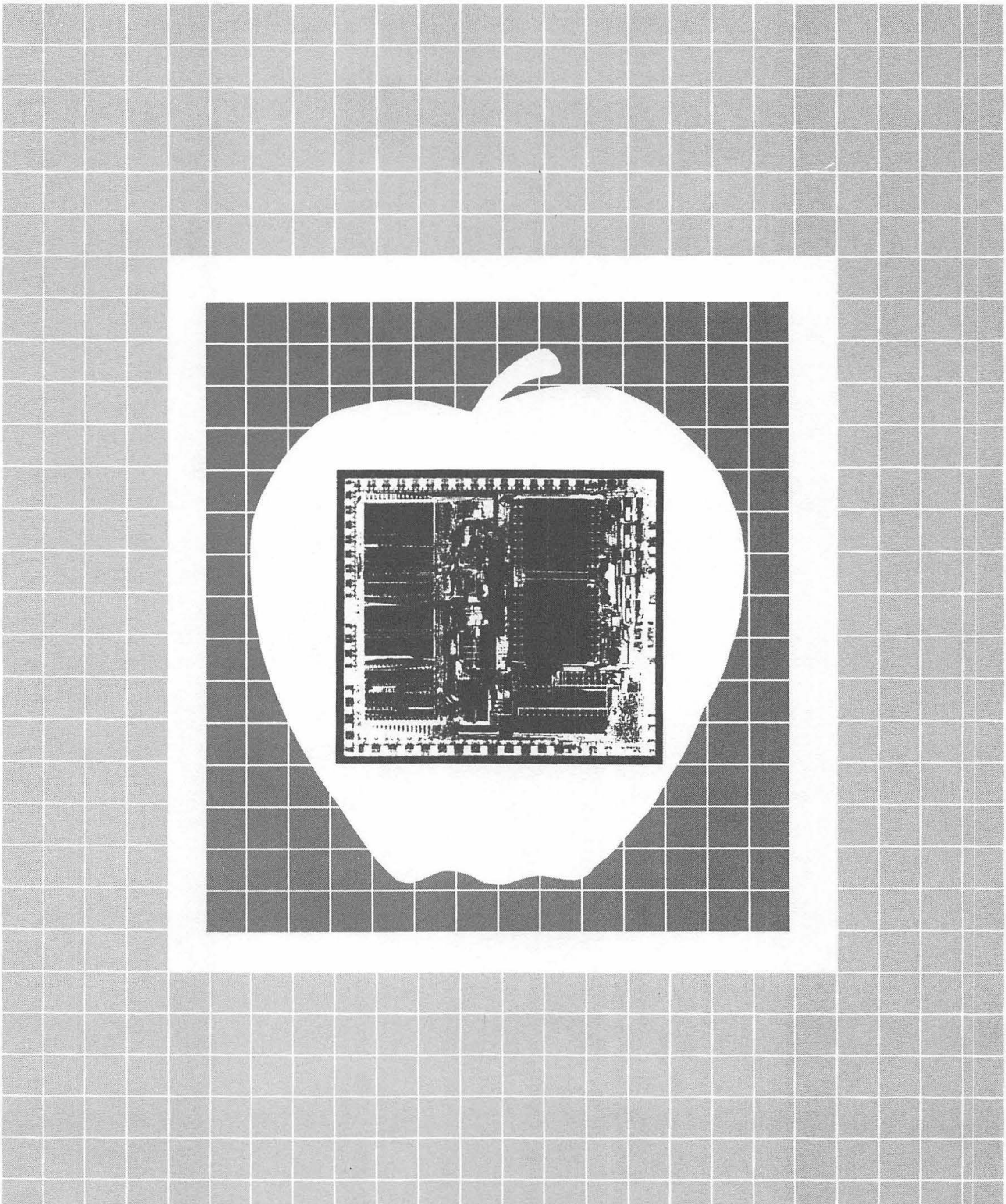
There is considerable room for expansion. For instance, scrolling and sizing could be made to work on text windows, and multiwindow capability could be added. See the exercises for more information.

EXERCISES

1. There is a system trap called _ExitToShell, which takes no parameters. Executing this trap causes a return to the Finder. Modify the initialization code in Listing 8.2 to specify a restart procedure that calls the _ExitToShell trap.
2. Modify the PRINT macro so that only one call is made to the run-time library.
3. Write the MoveCnt routine.
4. Modify the menu routines to call a common subroutine that issues the _GetResource call.
5. Write the routines from Table 8.2 that are not shown in Listing 8.20.

The following are research projects, meaning that the effort involved is not trivial and that the answers are not in the back of the book:

6. Implement scrolling and sizing for text windows. This means that you should not delete the lines scrolled off the top of the window, but allow the user to scroll back through them, if desired.
7. Redesign the library macros to allow the use of multiple application windows. Include support for opening and closing windows. (You'll need a copy of *Inside Macintosh* for this one.)
8. Modify the library so that it can be used with your favorite language compiler.



Appendix A

Answers to Exercises

Problems are the price of progress.

—Charles Kettering

This appendix gives the answers to the Exercise questions found at the end of each chapter.

CHAPTER 1

1. There are many correct solutions to this question. Here is one:
   1. Select the highest place value from the table that will divide into the number to be converted. Let the converted number be the initial remainder.
   2. Calculate the new quotient and remainder when the current remainder is divided by the present table entry.
   3. If place value table entries remain, repeat step 2 with the next table entry and the remainder just calculated.
   4. Read the answer as the successive quotients.
2. The flowchart for the above would look like Figure A.1.
3. The machine language program would look like Table A.1. You must load the contents of location 105 first because the difference to be computed is (location 105 − location 104). The subtraction

instruction works by subtracting memory from the register. Therefore, you must have the contents of location 105 in the register to perform the subtraction operation.

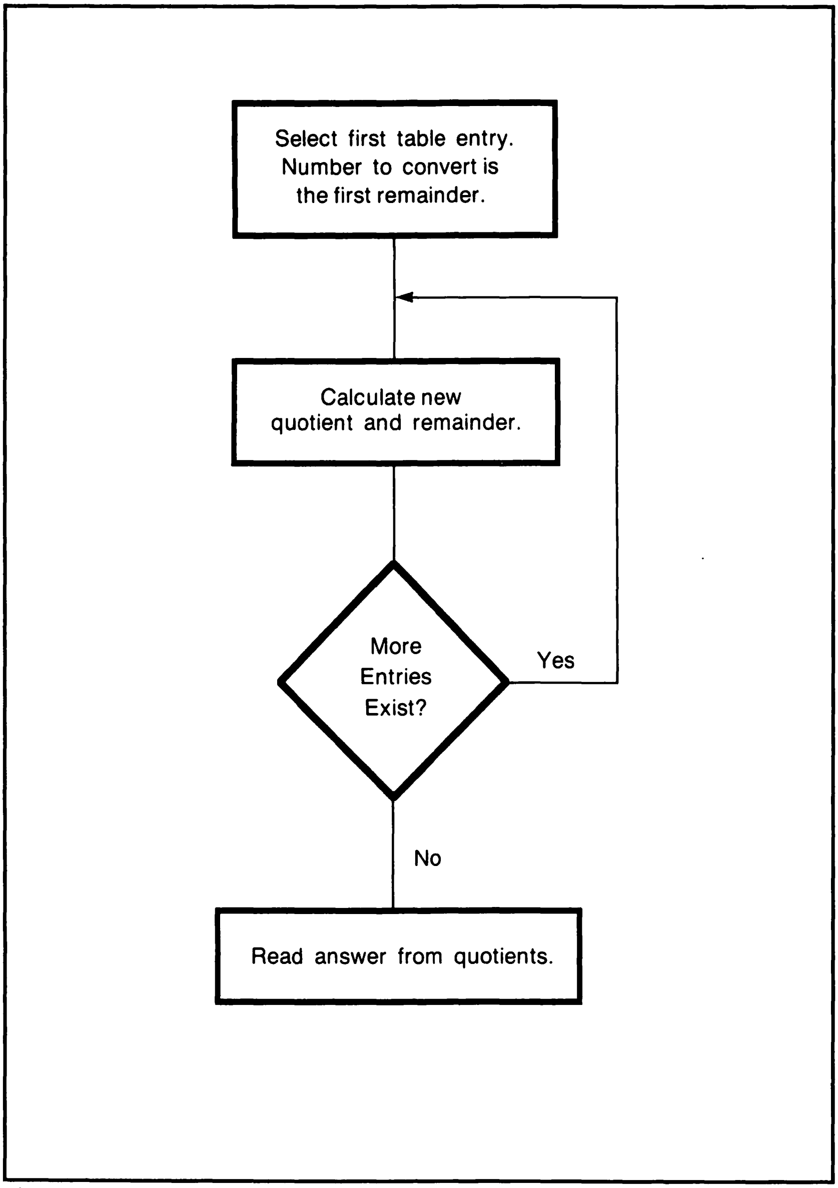


Figure A.1 *– Flowchart for Exercise 2*

4. Moving to location 200 yields the results in Table A.2.

5. This is the assembly language version:

```
        LOAD    A,Y
        SUB     A,X
        STORE   A,Z
        STOP
X:      DC      300
Y:      DC      400
Z:      DC      0
```

Location	Contents	Instruction
200	1205	Load A from location 205
201	5204	Subtract location 204
202	3206	Store into location 206
204	0300	<Data>
205	0400	<Data>
206	0000	<Data>

Table A.1 – *Machine Language Program at Location 100*

Location	Contents	Instruction
100	1105	Load A from location 105
101	5104	Subtract location 104
102	3106	Store into location 106
104	0300	<Data>
105	0400	<Data>
106	0000	<Data>

Table A.2 – *Machine Language Program at Location 200*

6. To add the first five integers, a modified version of the previous program will work:

```
        LOAD    A,A
        ADD     A,B
        ADD     A,C
        ADD     A,D
        ADD     A,E
        STORE   A,F
        STOP
A:      DC      1
B:      DC      2
C:      DC      3
D:      DC      4
E:      DC      5
F:      DC      0
```

7. Conversion to hex and binary is shown in Table A.3.

8. Conversion to binary and decimal is shown in Table A.4.

Decimal	Hex	Binary
273	111	0001 0001 0001
421	1A5	0001 1010 0101
1024	400	0100 0000 0000
100	64	0000 0110 0100

Table A.3 *– Conversion to Hex and Binary*

Hex	Binary	Decimal
ABE	0000 1010 1011 1110	2750
100	0000 0001 0000 0000	256
64	0000 0000 0110 0100	100
1024	0001 0000 0010 0100	4132
505	0000 0101 0000 0101	1285

Table A.4 *– Conversion to Binary and Decimal*

9. Complements are shown in Table A.5.

10. Binary operations are shown in Table A.6.

11. Shift and Rotate Tables:

Logical Shifts

Times	**Left**	**Right**
0	1111 1111 (FF hex)	1111 1111 (FF hex)
1	1111 1110 (FE hex)	0111 1111 (7F hex)
2	1111 1100 (FC hex)	0011 1111 (3F hex)
3	1111 1000 (F8 hex)	0001 1111 (1F hex)
4	1111 0000 (F0 hex)	0000 1111 (0F hex)
5	1110 0000 (E0 hex)	0000 0111 (07 hex)
6	1100 0000 (C0 hex)	0000 0011 (03 hex)
7	1000 0000 (80 hex)	0000 0001 (01 hex)
8	0000 0000 (00 hex)	0000 0000 (00 hex)

Number	One's Complement	Two's Complement
0ABE	F541	F542
0100	FEFF	FF00
0064	FF9B	FF9C
1024	EFDB	EFDC
0505	FAFA	FAFB

Table A.5 – *Complements*

Number	Pair	AND	OR	XOR	ADD	C
A5A5	5A5A	0000	FFFF	FFFF	FFFF	0
FFFF	0001	0001	FFFF	FFFE	0000	1
1234	4321	0220	5335	5115	5555	0

Table A.6 – *Binary Operations*

Times	Left	Right
0	0101 0101 (55 hex)	0101 0101 (55 hex)
1	1010 1010 (AA hex)	0010 1010 (2A hex)
2	0101 0100 (54 hex)	0001 0101 (15 hex)
3	1010 1000 (A8 hex)	0000 1010 (0A hex)
4	0101 0000 (50 hex)	0000 0101 (05 hex)
5	1010 0000 (A0 hex)	0000 0010 (02 hex)
6	0100 0000 (40 hex)	0000 0001 (01 hex)
7	1000 0000 (80 hex)	0000 0000 (00 hex)
8	0000 0000 (00 hex)	0000 0000 (00 hex)

Arithmetic Shifts

Times	Left	Right
0	1111 1111 (FF hex)	1111 1111 (FF hex)
1	1111 1110 (FE hex)	1111 1111 (FF hex)
2	1111 1100 (FC hex)	1111 1111 (FF hex)
3	1111 1000 (F8 hex)	1111 1111 (FF hex)
4	1111 0000 (F0 hex)	1111 1111 (FF hex)
5	1110 0000 (E0 hex)	1111 1111 (FF hex)
6	1100 0000 (C0 hex)	1111 1111 (FF hex)
7	1000 0000 (80 hex)	1111 1111 (FF hex)
8	0000 0000 (00 hex)	1111 1111 (FF hex)

Times	Left	Right
0	0101 0101 (55 hex)	0101 0101 (55 hex)
1	1010 1010 (AA hex)	0010 1010 (2A hex)
2	0101 0100 (54 hex)	0001 0101 (15 hex)
3	1010 1000 (A8 hex)	0000 1010 (0A hex)
4	0101 0000 (50 hex)	0000 0101 (05 hex)
5	1010 0000 (A0 hex)	0000 0010 (02 hex)
6	0100 0000 (40 hex)	0000 0001 (01 hex)
7	1000 0000 (80 hex)	0000 0000 (00 hex)
8	0000 0000 (00 hex)	0000 0000 (00 hex)

Rotates

Times	C	Left	Right	C
0	0	1111 1111 (FF hex)	1111 1111	0 (FF hex)
1	1	1111 1110 (FE hex)	0111 1111	1 (7F hex)
2	1	1111 1101 (FD hex)	1011 1111	1 (BF hex)

3	1	1111 1011 (FB hex)	1101 1111 1 (DF hex)
4	1	1111 0111 (F7 hex)	1110 1111 1 (EF hex)
5	1	1110 1111 (EF hex)	1111 0111 1 (F7 hex)
6	1	1101 1111 (DF hex)	1111 1011 1 (FB hex)
7	1	1011 1111 (BF hex)	1111 1101 1 (FD hex)
8	1	0111 1111 (7F hex)	1111 1110 1 (FE hex)
9	0	1111 1111 (FF hex)	1111 1111 0 (FF hex)

Times	**C**	**Left**	**Right C**
0	0	0101 0101 (55 hex)	0101 0101 0 (55 hex)
1	0	1010 1010 (AA hex)	0010 1010 1 (2A hex)
2	1	0101 0100 (54 hex)	1001 0101 0 (95 hex)
3	0	1010 1001 (A9 hex)	0100 1010 1 (4A hex)
4	1	0101 0010 (52 hex)	1010 0101 0 (A5 hex)
5	0	1010 0101 (A5 hex)	0101 0010 1 (52 hex)
6	1	0100 1010 (4A hex)	1010 1001 0 (A9 hex)
7	0	1001 0101 (95 hex)	0101 0100 1 (54 hex)
8	1	0010 1010 (2A hex)	1010 1010 0 (AA hex)
9	0	0101 0101 (55 hex)	0101 0101 0 (55 hex)

CHAPTER 2

1. The instructions would result in this illegal instruction:

   ```
   MOVE.B D0,A0
   ```

   This instruction is illegal because address registers can only be accessed as words or longwords. Give yourself extra points if you knew that an illegal-instruction exception would occur.

   ```
   MOVE.W D0,A0
   ```

   This instruction causes the contents of register A0 to become FFFF8000. Remember that word moves to an address register cause sign extension.

   ```
   MOVE.B D0,(A0)+
   ```

   The byte at location 1000 becomes 0. Register A0 becomes 1001.

   ```
   MOVE.B D0,-(A7)
   ```

The byte at memory location 10000 becomes 0. The contents of register A7 become FFFE. Remember that A7 is the hardware stack pointer and is incremented or decremented by 2 in byte operations.

2. These are the instructions as modified:

Instruction	**Hex**	**Changes**
ADD.L D1,D0	D081	D0=01234566
ADD.L A1,D0	D089	D0=01234566
ADD.L (A1),D0	D091	D0=01234566
ADD.L (A1)+,D0	D099	D0=01234566
ADD.L −(A1),D0	D0A1	D0=01234566
ADD.L 4(A1),D0	D0A9 0004	D0=01234566
ADD.L 4(A1,A2.L),D0	D0B1 A804	D0=01234566
ADD.L $1000,D0	D0B8 1000	D0=01234566
ADD.L $10000,D0	D0B9 0001 0000	D0=01234566
ADD.L $100(PC),D0	D0BA 0100	D0=01234566
ADD.L $10(PC,A1.L)	D0BB 9810	D0=01234566
ADD.L #$10002000,D0	D0BC 1000 2000	D0=11234566

3. The pre-decrement and post-increment modes can be used to implement a stack as follows:

   PUSH operation is MOVE.s xxx,(An)+
   POP operation is MOVE.s −(An),xxx

   The difference between this technique and a stack that grows toward lower addresses is that the address register no longer contains the address of the top item of the stack. Instead, the register points to the next stack location to be used. By changing the addressing modes to pre-increment and post-decrement (+(An) and (An)−), the stacking would be equivalent to the present technique.

CHAPTER 3

1–2. There aren't any answers back here. Either you can use your system or you can't. If you can't, you should first learn how. The basic mechanical steps are essential, and you will learn to program faster if you can do the mechanics well.

3. The ADDX and SUBX instructions perform arithmetic on multiple memory locations. You must start with the least significant digit in doing arithmetic. Comparisons, however, must be done starting with the most significant digit.

4. The RTE instruction reloads the status register (SR). This changes the trace bit in the status register. Normally, the trace bit is cleared by an RTE instruction. This prevents the instruction from being traced and causes the debugger to lose control. Other instructions that can clear the trace bit will have the same effect: MOVE to SR, ANDI to SR, and EORI to SR.

CHAPTER 4

1. The long division routine is shown in Listing A.1. Handling negative numbers is not required for the next question. The disadvantage to

```
;
;       32-bit division routine
;       Enter with:
;         D0.L = dividend
;         D1.L = divisor
;       Exit with:
;         D0.L = quotient
;         D1.L = remainder
;
        XDEF    LDiv
LDiv    MOVEM.L D2-D3,-(SP)    ;       Save some working registers
        CLR.L   D2             ;       Temporary quotient
        CLR.L   D3             ;       Sign flag
        TST.L   D0             ;       Dividend > 0?
        BGE     @1             ;       Yes
        NEG.L   D0             ;       Force positive
        ADDQ.L  #1,D3          ;       Increment sign flag
@1      TST.L   D1             ;       Divisor > 0?
        BGE     @2             ;       Yes
        NEG.L   D1             ;       Force positive
        ADDQ.L  #1,D3          ;       Increment sign flag
@2      CMP.L   D0,D1          ;       Dividend : divisor
        BGT     @3             ;       If divisor > dividend, done
        SUB.L   D1,D0          ;       Subtract divisor from dividend
        ADDQ.L  #1,D2          ;       Increment quotient
        BRA     @2             ;       Loop till done
@3      BTST    #0,D3          ;       Need negative quotient?
        BEQ     @4             ;       No, both positive
        NEG.L   D2             ;       Negate quotient
        NEG.L   D0             ;       Negate remainder
@4      MOVE.L  D0,D1          ;       This is remainder
        MOVE.L  D2,D0          ;       This is quotient
        MOVEM.L (SP)+,D2-D3    ;       Restore working registers
        RTS                    ;       Return to caller
```

Listing A.1 *– The LDIV routine*

doing division in this manner is the excessive amount of time required to divide a very large number by a very small one. It can take several minutes for a single division operation.

2. The modified binary-to-decimal ASCII-conversion routine is shown in Listing A.2.

   Listing A.3 illustrates the use of a divisor table in converting binary to decimal. This is the conventional method and is much faster than the algorithm in Listing A.2.

```
;
;        BinDec routine.  Converts binary to decimal ASCII.
;
;        Enter with:
;
;           D0.W =  Number to convert
;           A0.L -> 10-byte output area
;
         XDEF     BinDec
         XREF     LDiv
BinDec:  MOVEM.L  D0-D2/A0,-(SP)  ;       Preserve registers
         MOVE.B   #' ',D1         ;       Assume positive number
         TST.L    D0              ;       Is number negative?
         BPL      NotNeg          ;       No, use ' ' sign
         MOVE.B   #'-',D1         ;       Number is negative, use '-'
         NEG.L    D0              ;       Force positive
NotNeg:  MOVE.B   D1,(A0)+        ;       Put sign in buffer
         ADDA.L   #9,A0           ;       A0 -> 1 beyond last digit
         MOVE.W   #8,D2           ;       DBRA-adjusted loop count
Loop:    MOVE.L   #10,D1          ;       Set quotient
         JSR      LDiv            ;       Divide by 10
         MOVE.B   D1,-(A0)        ;       Put in buffer (0-9 binary)
         ADD.B    #'0',(A0)       ;       Add ASCII offset (0-9 ASCII)
         DBRA     D2,Loop         ;       Do all digits
         MOVEM.L  (SP)+,D0-D2/A0  ;       Restore registers
         RTS                      ;       Return to caller
```

Listing A.2 *– Modified BinDec routine*

```
;
;        BinDec routine.  Converts binary to decimal ASCII.
;
;        Enter with:
;
;           D0.W =  Number to convert
```

Listing A.3 *– The BinDec routine using a divisor table (continued)*

```
;           A0.L -> 10-byte output area
;
        XDEF    BinDec
BinDec  MOVEM.L D0-D3/A0-A1,-(SP);      Preserve registers
        MOVE.W  #9,D2           ;       Loop count
        LEA     DivTab,A1       ;       A1 -> Table of divisors
        CLR.L   D3              ;       Assume positive number
        TST.L   D0              ;       Is number negative?
        BPL     @1              ;       No
        SUBQ.L  #1,D3           ;       Number is negative, use '-'
        NEG.L   D0              ;       Force positive
        SUBQ.W  #1,D2           ;       Down count
        ADDQ.L  #4,A1           ;       A1 -> Next divisor
        MOVE.B  #' ',(A0)+      ;       Clobber first byte of output area
@1      CMP.L   (A1)+,D0        ;       Number too big?
        BHI     Stars           ;       Yes, fill with asterisks
@2      CMP.L   (A1)+,D0        ;       Find
        BGE     @3              ;               first digit
        MOVE.B  #' ',(A0)+      ;       Space fill
        DBRA    D2,@2           ;       Loop till done
        MOVE.B  #'0',-(A0)      ;       Zero is special case
        BRA     DoRet           ;       Done
@3      SUBQ.L  #4,A1           ;       A1 -> First divisor
        TST.L   D3              ;       Negative?
        BEQ     @4              ;       No
        MOVE.B  #'-',-1(A0)     ;       Set sign
@4      CLR.L   D3              ;       Clear divisor
@5      CMP.L   (A1),D0         ;       Compare divisor to dividend
        BLT     @6              ;       Done with this one
        SUB.L   (A1),D0         ;       Subtract
        ADDQ.L  #1,D3           ;       Increment quotient
        BRA     @5              ;       Loop till done
@6      ADDQ.L  #4,A1           ;       A1 -> Next divisor
        MOVE.B  D3,(A0)         ;       Set
        ADD.B   #'0',(A0)+      ;               digit
        DBRA    D2,@4           ;       Loop till done
        BRA     DoRet           ;       Done
Stars                           ;       Here if number too big
@1      MOVE.B  #'*',(A0)+      ;       Fill
        DBRA    D2,@1           ;               with asterisks
DoRet   MOVEM.L (SP)+,D0-D3/A0-A1;      Restore registers
        RTS                     ;       Return
DivTab  DC.L  10000000000       ;       First number too large
        DC.L   1000000000       ;       Highest value allowed
        DC.L    100000000       ;
        DC.L     10000000       ;
        DC.L      1000000       ;
        DC.L       100000       ;
        DC.L        10000       ;
        DC.L         1000       ;
        DC.L          100       ;
        DC.L           10       ;
        DC.L            1       ;
```

***Listing A.3** – The BinDec routine using a divisor table*

3. The finished conversion program looks like Listing A.4. Notice that the only change required is the expansion of the size of the decimal output conversion area.

4. The hex conversion routine looks like Listing A.5. A slightly tricky piece of code to yield the proper binary nibble uses register D1 as the factor to be subtracted from the ASCII byte.

5. The program for converting hex to decimal can be derived from the earlier solution for converting the longword decimal to hex, as shown in Listing A.6.

6. The modified DecBin routine appears in Listing A.7.

```
;
;         DecHex program.  This program converts decimal numbers
;         to hex.  Numbers are input from the keyboard and output
;         to the screen.
;
          Include MACLIB.ASM
          XREF    BinHex          ;        Hex output conversion
          XREF    BinDec          ;        Decimal output conversion
          XREF    DecBin          ;        Decimal input conversion
          PROGRAM DecHex
Loop:     PRINT   'Enter Decimal Number: ',$ ; Prompt for input
          INPUT   InBuff(A5)      ;        Get a line of input
          TST.B   InBuff(A5)      ;        Check for null line
          BEQ     Done            ;        If null, quit
          LEA     InBuff+1(A5),A0 ;        A0 -> Digits input
          JSR     DecBin          ;        Convert to binary
          LEA     DecBuf,A0       ;        A0 -> Decimal output area
          JSR     BinDec          ;        Convert to decimal ASCII
          LEA     HexBuf,A0       ;        A0 -> Hex output area
          JSR     BinHex          ;        Convert to hex ASCII
          PRINT   OutLine         ;        Type the answer
          BRA     Loop            ;        Do it again
Done:     EXIT                    ;        Back to Finder
          STRING_FORMAT 0         ;        No count bytes
OutLine   DC.B    EndLine-DecBuf  ;        Count Byte
DecBuf:   DC.B    'XXXXXXXXXX Decimal is '
HexBuf:   DC.B    'XXXXXXXX Hex'
EndLine:                          ;        Tag for end of line
Inbuff:   DS.B    81              ;        Input area
          ENDPROGRAM
```

***Listing A.4** – The modified DecHex program*

7. The entire Dump program appears in Appendix E.
8. The listing program source file is shown in Listing A.8.

CHAPTER 5

1. The Fibonacci routine is shown in Listing A.9.

```
;
;       This subroutine converts hex ASCII to longword binary.
;
;       Enter with:
;
;               A0 -> Hex string
;
;       Exit with:
;
;               D0.L = Converted number
;
;       Conversion terminates on first nonhexadecimal character.
;       No overflow detection.
;
        XDEF    HexBin
HexBin  MOVEM.L D1-D2/A0,-(SP)  ;       Save temporary registers
        CLR.L   D0              ;       Clear out accumulator
Loop    CLR.L   D1              ;       Zero out D1
        CMPI.B  #'9',(A0)       ;       Upper decimal bound
        BHI     NotDec          ;       Not a decimal digit
        CMPI.B  #'0',(A0)       ;       Lower decimal bound
        BLO     NotHex          ;       Not a hex (or decimal) digit
        MOVE.B  #'0',D1         ;       Correction factor
        BRA     GotDig          ;       It's a hex digit
NotDec  CMPI.B  #'A',(A0)       ;       Lower UC hex bound
        BLO     NotHex          ;       Not a hex digit
        CMPI.B  #'F',(A0)       ;       Upper UC hex bound
        BHI     NotUC           ;       Not uppercase hex
        MOVE.B  #'A'-10,D1      ;       Correction factor
        BRA     GotDig          ;       It's a hex digit
NotUC   CMPI.B  #'a',(A0)       ;       Test lower LC bound
        BLO     NotHex          ;       Not hex
        CMPI.B  #'f',(A0)       ;       Test upper LC bound
        BHI     NotHex          ;       Not hex
        MOVE.B  #'a'-10,D1      ;       Correction factor
GotDig  CLR.L   D2              ;       Clear scratch register
        MOVE.B  (A0)+,D2        ;       Pick up digit
        SUB.L   D1,D2           ;       Convert to binary
        LSL.L   #4,D0           ;       Multiply by 16
        ADD.L   D2,D0           ;       Add in new digit
        BRA     Loop            ;       Repeat till done
NotHex  MOVEM.L (SP)+,A0/D1-D2  ;       Pop registers
        RTS                     ;       Return
```

***Listing A.5** – The HexBin routine*

2. The code in Listing A.9 can be called directly from C. Listing A.10 contains the BASIC version.

   Listing A.11 contains the Pascal version. This routine uses the Pascal calling sequence to accomplish the recursion, rather than the optimized assembly language in Listings A.9 and A.10. Note the overhead involved in passing the parameters on the stack.

CHAPTER 6

1. Restricting MacDoodle to run on a 512K Macintosh means that all of the code dealing with dynamically allocated buffers becomes much simpler. You would allocate only the file buffer dynamically. This buffer would be required only during the reading and writing process. The screen and undo buffers would be

```
;
;       HexDec program.  This program converts hex numbers
;       to decimal.  Numbers are input from the keyboard
;       and output to the screen.
;
include MACLIB.ASM
        XREF    BinHex          ;   Hex output conversion
        XREF    BinDec          ;   Decimal output conversion
        XREF    HexBin          ;   Hex input conversion
        PROGRAM DecHex          ;
Loop:   PRINT   'Enter Hex Number: ',$ ; Prompt for input
        INPUT   InBuff(A5)      ;   Get a line of input
        TST.B   InBuff(A5)      ;   Check for null line
        BEQ     Done            ;   If null, quit
        LEA     InBuff+1(A5),A0 ;   A0 -> Digits input
        JSR     HexBin          ;   Convert to binary
        LEA     HexBuf,A0       ;   A0 -> Hex output area
        JSR     BinHex          ;   Convert to hex ASCII
        LEA     DecBuf,A0       ;   A0 -> Hex output area
        JSR     BinDec          ;   Convert to decimal ASCII
        PRINT   OutLine         ;   Type the answer
        BRA     Loop            ;   Do it again
Done:   EXIT                    ;   Back to Finder
        STRING_FORMAT 0         ;   No count bytes
OutLine DC.B    EndLine-HexBuf  ;   Count byte
HexBuf: DC.B    'XXXXXXXX Hex is '
DecBuf: DC.B    'XXXXXXXXXX Decimal'
EndLine:                        ;   Tag for end of line
Inbuff: DS.B    81              ;   Input area
        ENDPROGRAM
```

***Listing A.6** – The HexDec program*

```
;
;       DecBin routine.  Converts decimal ASCII to longword binary.
;
;       Enter with:
;
;          A0.L -> Input string
;
;       Exit with:
;
;          D0.L =  Converted number
;
;       Conversion stops at the first nondecimal digit.  No
;       overflow or error detection.
;
;       Revision history:
;
;<01>   20-Jan-85       Added support for negative numbers
;                       and leading spaces
;
;
        XDEF    DecBin
DecBin: MOVEM.L D1-D2/A0,-(SP)  ;<01>   Preserve registers
LBlanks CMPI.B  #' ',(A0)       ;<01>   Leading space
        BNE     CheckMinus      ;<01>   No, check for negative #
        TST.B   (A0)+           ;<01>   Advance to next character
        BRA     LBlanks         ;<01>   Repeat till nonblank
CheckMinus:                     ;<01>   Here when nonblank
        CLR.L   D2              ;<01>   Clear flag
        CMPI.B  #'-',(A0)+      ;<01>   Is negative?
        BEQ     GetNum          ;<01>   Yes, continue
        TST.B   -(A0)           ;<01>   No, fix A0
        MOVE.L  #1,D2           ;<01>   Set positive flag
GetNum: CLR.L   D0              ;       Clear accumulator
Loop:   CMPI.B  #'9',(A0)       ;       Check upper bound
        BHI     NotDec          ;       Not a decimal digit
        CMPI.B  #'0',(A0)       ;       Check lower bound
        BLO     NotDec          ;       Not a decimal digit
        LSL.L   #1,D0           ;       Multiply current value x 2
        MOVE.L  D0,D1           ;       Save this value
        LSL.L   #2,D0           ;       Multiply again x 4
        ADD.L   D1,D0           ;       32-bit multiply x 10
        MOVE.B  (A0)+,D1        ;       Fetch digit from memory
        ANDI.L  #15,D1          ;       Isolate low-order 4 bits
        ADD.L   D1,D0           ;       Add to accumulator
        BRA     Loop            ;       Back for next digit
NotDec: TST.L   D2              ;<01>   Positive number?
        BNE     GoBack          ;<01>   Yes, don't touch
        NEG.L   D0              ;<01>   No, make negative
GoBack: MOVEM.L (SP)+,D1-D2/A0  ;<01>   Restore registers
        RTS                     ;       Return to caller
        END
```

***Listing A.7** – The modified DecBin routine*

```
;
;       Lister program.  This program generates a numbered listing file
;       from a .asm file
;
;
BufLen  EQU     256                             ;       Bytes in line buffer
Include MACLIB.ASM                              ;       Include macro library
        XREF    BinDec                          ;       Decimal print routine
        PROGRAM Lister                          ;
Start:                                          ;       Here for repeats
        BSR     OpenFiles                       ;       Open source and listing files
        MOVE.L  #1,D7                           ;       Start with line 1
ReadSrc:                                        ;       Here to read from source file
        LEA     LineBf(A5),A0                   ;       A0 -> Buffer
        MOVE.W  #BufLen-1,D0                    ;       D0 = DBRA-adjusted count
ZapBuf: CLR.B   (A0)+                           ;       Clear a byte
        DBRA    D0,ZapBuf                       ;       Repeat till done
        READLN  SrcFD(A5),LineBf+8(A5),#BufLen; Read source line
        TST.L   D0                              ;       Any bytes?
        BLE     EOF                             ;       No, quit
        MOVE.L  D0,D2                           ;       Save byte count
        ADDQ.L  #8,D2                           ;       8 bytes for line #, extra space.
        LEA     LineBf(A5),A0                   ;       A0 -> Output area
        MOVE.L  D7,D0                           ;       Move line # to D0
        JSR     BinDec                          ;       Call conversion routine
        MOVE.B  #' ',1(A0)                      ;       Blank first digit
        MOVE.W  #'  ',6(A0)                     ;       Blank out next 2 cols
        WRITE   OutFD(A5),LineBF(A5),D2 ;       Do the write
        ADDQ.L  #1,D7                           ;       Bump line count
        BRA     ReadSrc                         ;       Back for more
EOF:    CLOSE   SrcFD(A5)                       ;       And source file
        CLOSE   OutFD(A5)                       ;       And output file
        BRA     Start                           ;       Do it again
;
;       Subroutine Openfiles.  This routine opens the listing and corresponding
;       source file.
;
;       Enter with:
;
;               (none)
;
;       Exit with:
;
;               OutFD(A5) = Listing file descriptor
;               SrcFD(A5) = Source  file descriptor
;
OpenFiles:                                      ;
        POPEN   Types,'.ASM',Scratch(A5),SrcBuf(A5)
        MOVE.W  D0,SrcFD(A5)                    ;       Save returned file descriptor
        BMI     NoGood                          ;       No good
        LEA     Scratch(A5),A0                  ;       A0 -> Source file name
        CLR.W   D0                              ;       D0 =
        MOVE.B  (A0)+,D0                        ;               Count byte
        MOVE.L  A0,A1                           ;       Save this
        SUBQ.W  #1,D0                           ;       Adjust for DBRA
FIndDot:                                        ;       Find "." in file name
```

***Listing A.8** – The Lister program*

```
        CMPI.B  #'.',(A0)+          ;       Scan for
        BEQ     FoundDot            ;                "."
        DBRA    D0,FindDot          ;                        character
        TST.B   (A0)+               ;       Correct for no "."
FoundDot:                           ;       Here when found or none
        TST.B   -(A0)               ;       A0 -> "." character
SrcOK:  MOVE.B  #'.',(A0)+          ;       Move
        MOVE.B  #'L',(A0)+          ;               in
        MOVE.B  #'I',(A0)+          ;                       .LIS
        MOVE.B  #'S',(A0)+          ;                               extension
        PCREATE 'Output File',Scratch(A5),Scratch(A5),OutBuf(A5)
        MOVE.W  D0,OutFD(A5)        ;       Store file descriptor
        BMI     NoGood              ;       Check for cancel
        RTS                         ;       and quit
NoGood: EXIT                        ;       Back to Finder if no good
;
;       Lister Data area.
;
;       Initialized Data:
;
        STRING_FORMAT   0           ;       No count bytes, please
Types:  DC.W    1                   ;       1 type to follow
        DC.L    'TEXT'              ;       Type is 'text'

;
;       Uninitialized Data:
;
SrcFD:  DS.W    1                   ;       Source file descriptor
OutFD:  DS.W    1                   ;       Output file descriptor
SrcBuf: DS.B    522                 ;       Source file buffer
OutBuf: DS.B    522                 ;       Output file buffer
Scratch DS.B    256                 ;       Scratch area
LineBf: DS.B    BufLen              ;       Line buffer area
        .Align  2                   ;       Align to even boundary
        ENDPROGRAM
```

***Listing A.8** – The Lister program (continued)*

```
;
;       Recursive Fibonacci routine
;
;       Enter with number in D0.W
;       Exit  with answer in D0.W
;
        XDEF    Fib
Fib     CMP.W   #1,D0           ;       Easy?
        BGT     DoFib           ;       No, recurse for it
        RTS                     ;       F(1) is 1
DoFib   MOVE.W  D0,-(A7)        ;       Save input value
        SUB.W   #1,D0           ;       Take
        BSR     Fib             ;               F(n-1)
```

***Listing A.9** – The recursive Fibonacci routine*

```
        MOVE.W  D0,-(A7)        ;    Save the result
        MOVE.W  2(A7),D0        ;    Get original number
        SUB.W   #2,D0           ;    Take
        BSR     Fib             ;         F(n-2)
        ADD.W   (A7)+,D0        ;    Calculate F(n-1) + F(n-2)
        ADD.L   #2,A7           ;    Pop original number
        RTS                     ;    Done
```

***Listing A.9** – The recursive Fibonacci routine (continued)*

```
;
;       Recursive Fibonacci routine (BASIC version)
;
;       Call with:
;
;               CALL BFIB(num%,ans%)
;
;       Where:
;
;               num% is the number desired
;               ans% is the resulting Fibonacci number
;
        INCLUDE BASIC.D         ;    Include BASIC defs
bfib    JSR     GetNextLibArg(A5);   Fetch next parameter
        JSR     IntegerArg(A5)  ;    Convert parameter
        MOVE.W  D3,-(SP)        ;    Save number
        JSR     GetNextLibArg(A5);   Fetch next parameter
        CMP.W   #$3,D0          ;    Integer variable?
        BNE     @1              ;    No, bail out
        MOVE.W  (SP)+,D0        ;    Number to D0
        BSR     Fib             ;    Call recursive routine
        MOVE.W  D0,(A2)         ;    Set return variable
        RTS                     ;    Return
@1:                             ;    Here if parameter mismatch
        MOVE.W  #5,D2           ;    "Illegal Function Call"
        JSR     BasicError(A5)  ;    Bail out
;
;       The original Fibonacci routine
;
Fib     CMP.W   #1,D0           ;    Easy?
        BGT     DoFib           ;    No, recurse for it
        RTS                     ;    F(1) is 1
DoFib   MOVE.W  D0,-(A7)        ;    Save input value
        SUB.W   #1,D0           ;    Take
        BSR     Fib             ;         F(n-1)
        MOVE.W  D0,-(A7)        ;    Save the result
        MOVE.W  2(A7),D0        ;    Get original number
        SUB.W   #2,D0           ;    Take
        BSR     Fib             ;         F(n-2)
        ADD.W   (A7)+,D0        ;    Calculate F(n-1) + F(n-2)
        ADD.L   #2,A7           ;    Pop original number
        RTS                     ;    Done
```

***Listing A.10** – The BASIC Fibonacci routine* allocated with DS directives. There would be no packing and unpacking operations during scrolling. The program itself would become much simpler. (This is the reason why many Macintosh applications run only on 512K machines.)

2–4. The answers to these questions appear in the MacDoodle listings in Appendix E.

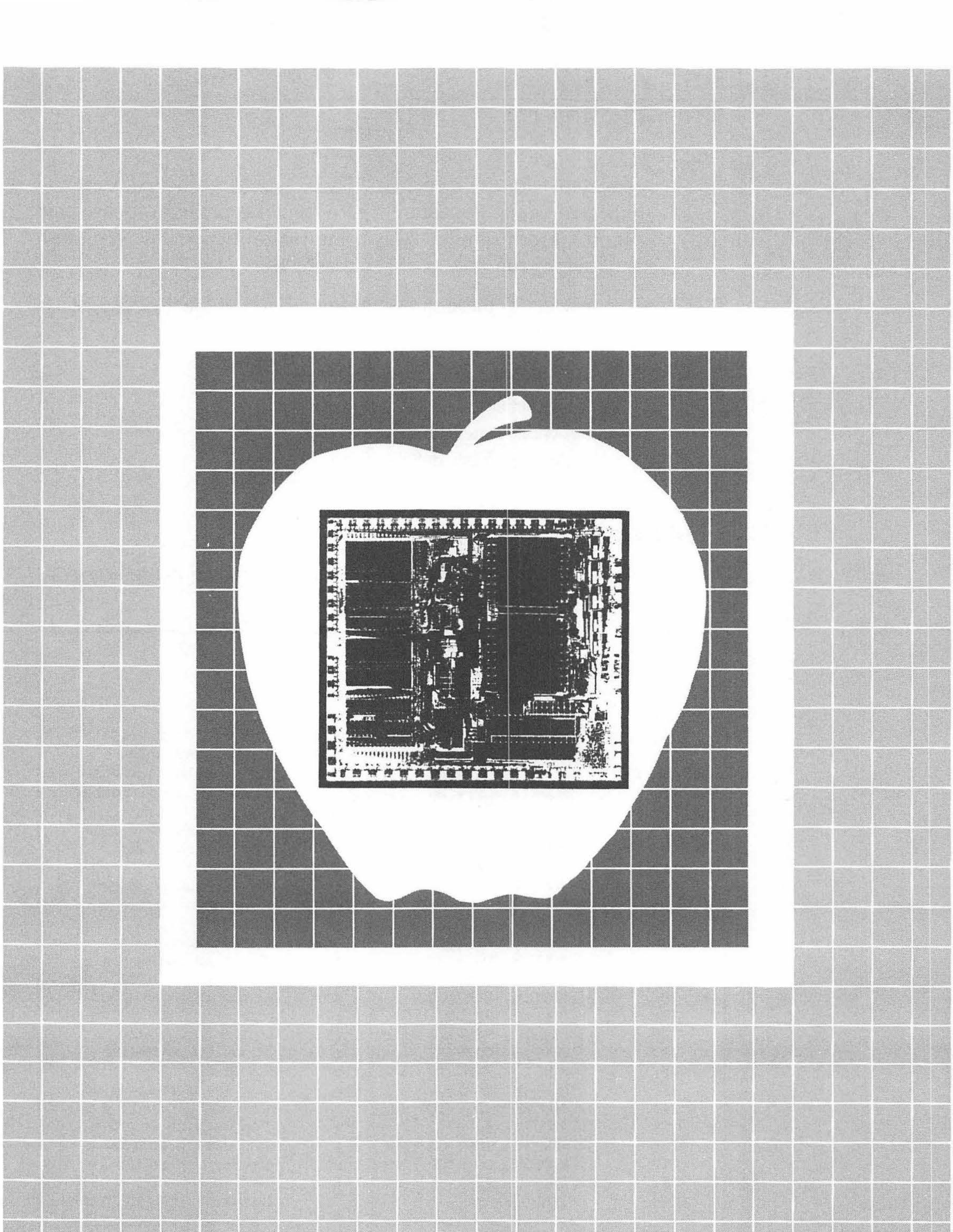
5. These resources should be preloaded:
   a. Menu definitions
   b. Main window definition
   c. Window scroll bar definitions

CHAPTER 8

The answers to all Exercises in Chapter 8 are contained in the listings in Appendix E.

```
;
;               Fibonacci program (Pascal version)
;
                .PROC   FibAsm          ;       Procedure statement
                .DEF    fib             ;       Make accessible
fib             MOVE.L  (SP)+,A1        ;       A1 =  Return address
                MOVE.W  (SP)+,D0        ;       D0 =  Original number
                CMP.W   #1,D0           ;       Easy?
                BGT     DoFib           ;       No, recurse for it
                MOVE.W  D0,(SP)         ;       Set return value
                JMP     (A1)            ;       F(1) is 1
DoFib           MOVE.L  A1,-(SP)        ;       Save return address
                MOVE.W  D0,-(A7)        ;       Save input value
                SUB.W   #1,D0           ;       Take F(n-1)
                CLR.W   -(SP)           ;       Space for result
                MOVE.W  D0,-(A7)        ;       Function parameter
                BSR     Fib             ;       Recurse
                MOVE.W  2(A7),D0        ;       Get original number
                SUB.W   #2,D0           ;       Take F(n-2)
                CLR.W   -(SP)           ;       Space for result
                MOVE.W  D0,-(A7)        ;       Function parameter
                BSR     Fib             ;       Recurse
                MOVE.W  (A7)+,D0        ;       Fetch returned value
                ADD.W   (A7)+,D0        ;       Calculate F(n-1) + F(n-2)
                ADD.L   #4,A7           ;       Pop original number
                MOVE.L  (SP)+,A1        ;       Pop return address
                MOVE.W  D0,-(SP)        ;       Set return value
                JMP     (A1)            ;       Return
                .END                    ;       Mandatory END statement
```

***Listing A.11** – The Pascal Fibonacci routine*



Appendix **B**

Macintosh Character Set

There is always one moment when the door opens and lets the future in.

—Graham Greene

The Macintosh uses a variation on the ASCII (American Standard Code for Information Interchange) character set. Table B.1 gives the decimal and hexadecimal values for each character in the standard ASCII character set.

The ASCII character set consists of seven-bit characters. The high-order bit of a byte containing an ASCII character is normally zero.

The Macintosh, however, uses all eight bits of the character value. Figure B.1 shows the Macintosh character set. This table contains all 256 characters in the Macintosh Chicago font. Each character is identified by two hex digits. The first hex digit appears in the first column of the table, and the second hex digit appears in the first row of the table. To find the character corresponding to the hex value 12, find the intersection of the row labeled 1 and the column labeled 2 (the check mark character). The box symbol indicates a character that is not in the font. Note that there is no easy way (or any way at all) to generate some of these font symbols from the keyboard.

Dec	Hex	Character	Dec	Hex	Character
0	00	CTL-@ NULL	34	22	"
1	01	CTL-A SOH	35	23	#
2	02	CTL-B STX	36	24	$
3	03	CTL-C ETX	37	25	%
4	04	CTL-D EOT	38	26	&
5	05	CTL-E ENQ	39	27	'
6	06	CTL-F ACK	40	28	(
7	07	CTL-G BEL	41	29	)
8	08	CTL-H BS	42	2A	*
9	09	CTL-I HT	43	2B	+
10	0A	CTL-J LF	44	2C	,
11	0B	CTL-K VT	45	2D	-
12	0C	CTL-L FF	46	2E	.
13	0D	CTL-M CR	47	2F	/
14	0E	CTL-N SO	48	30	0
15	0F	CTL-O SI	49	31	1
16	10	CTL-P DLE	50	32	2
17	11	CTL-Q DC1 (XON)	51	33	3
18	12	CTL-R DC2	52	34	4
19	13	CTL-S DC3 (XOFF)	53	35	5
20	14	CTL-T DC4	54	36	6
21	15	CTL-U NAK	55	37	7
22	16	CTL-V SYN	56	38	8
23	17	CTL-W ETB	57	39	9
24	18	CTL-X CAN	58	3A	:
25	19	CTL-Y EM	59	3B	;
26	1A	CTL-Z SUB	60	3C	<
27	1B	CTL-[ESC	61	3D	=
28	1C	FS	62	3E	>
29	1D	GS	63	3F	?
30	1E	RS	64	40	@
31	1F	US	65	41	A
32	20	Space	66	42	B
33	21	!	67	43	C

Table B.1 – *The Standard ASCII Character Set*

Dec	Hex	Character	Dec	Hex	Character
68	44	D	98	62	b
69	45	E	99	63	c
70	46	F	100	64	d
71	47	G	101	65	e
72	48	H	102	66	f
73	49	I	103	67	g
74	4A	J	104	68	h
75	4B	K	105	69	i
76	4C	L	106	6A	j
77	4D	M	107	6B	k
78	4E	N	108	6C	l
79	4F	O	109	6D	m
80	50	P	110	6E	n
81	51	Q	111	6F	o
82	52	R	112	70	p
83	53	S	113	71	q
84	54	T	114	72	r
85	55	U	115	73	s
86	56	V	116	74	t
87	57	W	117	75	u
88	58	X	118	76	v
89	59	Y	119	77	w
90	5A	Z	120	78	x
91	5B	[	121	79	y
92	5C	\	122	7A	z
93	5D	]	123	7B	{
94	5E	^	124	7C	|
95	5F	_	125	7D	}
96	60	‘	126	7E	
97	61	a	127	7F	DEL

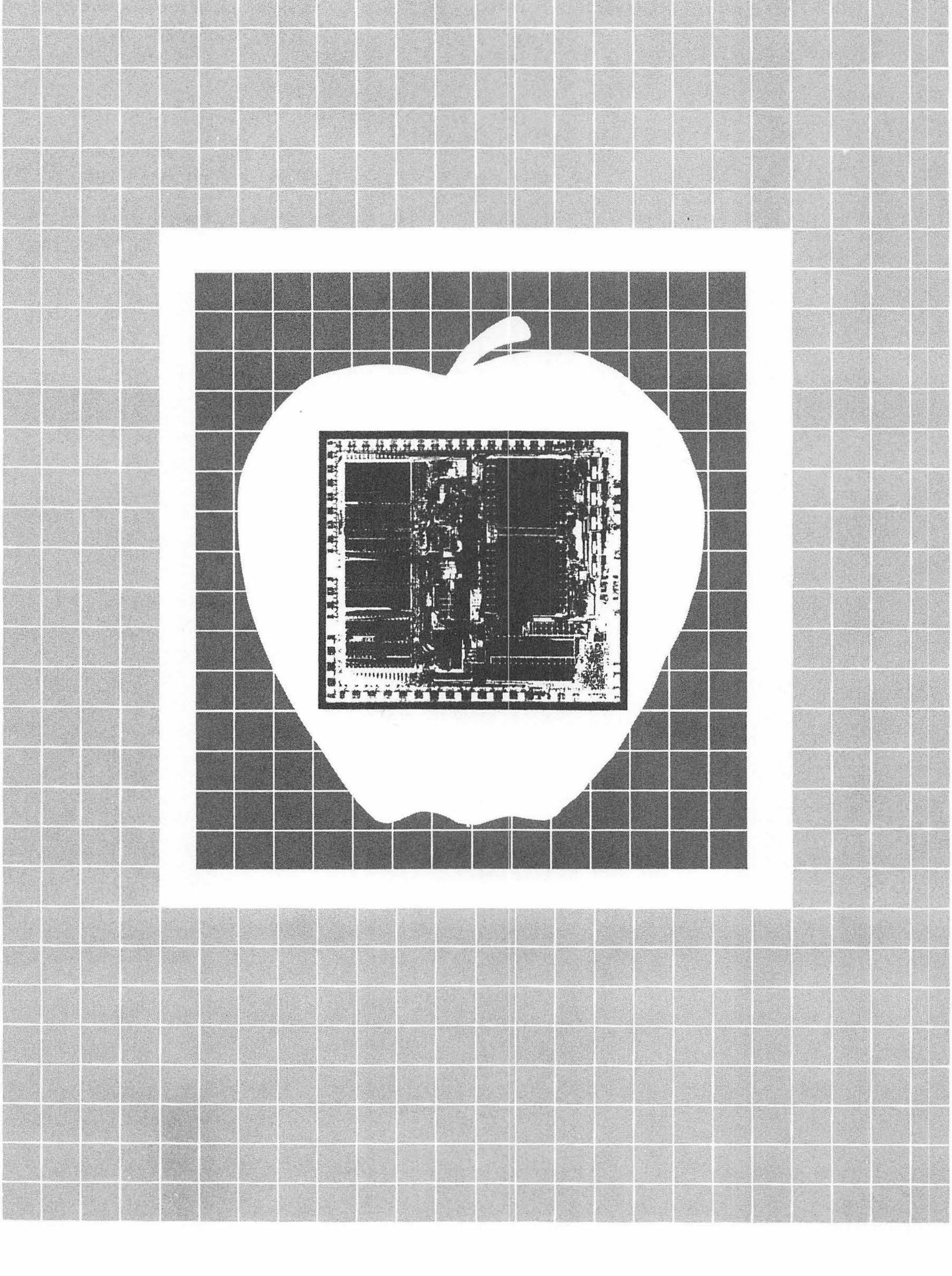
Table B.1 *– The Standard ASCII Character Set (continued)*

Second Digit

First Digit

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		□	□	□	□	□	□	□	□		□	□	□		□	□
1	□	⌘	✓	◆		□	□	□	□	□	□	□	□	□	□	□
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
9	ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
A	†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø
B	∞	±	≤	≥	¥	µ	∂	Σ	Π	π	∫	ª	º	Ω	æ	ø
C	¿	¡	¬	√	ƒ	≈	∆	«	»	…		À	Ã	Õ	Œ	œ
D	–	—	“	”	‘	’	÷	◊	ÿ	□	□	□	□	□	□	□
E	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
F	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□

Figure B.1 *– The Macintosh character set*



Appendix **C**

Macro Library Summary

Give me the right word and the right accent and I will move the world.

—Joseph Conrad

This appendix is a quick reference for the macros used in writing programs that work with the run-time library. These macros are defined in the source file MACLIB.ASM.

ALERT

Syntax: ALERT ID

Parameter: ID

Data Type: Word

Purpose: Resource ID of alert template

Description: The ALERT macro causes an alert box to be displayed on the Macintosh screen. Alert boxes are shown in Figure 6.16 (Chapter 6). The box drawn by the ALERT macro has no icon in the upper-left corner.

CAUTIONALERT

Syntax: CAUTIONALERT ID

Parameter: ID

Data Type: Word

Purpose: Resource ID of alert template

Description: The CAUTIONALERT macro causes an alert box to be displayed on the Macintosh screen. Alert boxes are shown in Figure 6.16 (Chapter 6). The box drawn by the CAUTIONALERT macro has the question mark icon in the upper-left corner.

CHECK

Syntax: CHECK menuID,selection

Parameters	Data Type	Purpose
menuID	Word	Resource ID of a menu in the menu bar
selection	Word	Item number within the menu

Description: The CHECK macro places a check mark in front of the item indicated by the selection parameter. Menu items are numbered consecutively from 1. The menuID is the resource ID of the menu on which the item is to be found. Menus used with the run-time library must also be numbered consecutively from 1.

CHKABORT

Syntax: CHKABORT

Parameters: None

Return Values: D0.L is nonzero if the user wishes to stop the program. D0.L is zero otherwise.

Description: The CHKABORT macro determines whether the Macintosh user has typed Command-Period, indicating a desire to stop the program.

CLOSE

Syntax: CLOSE refnum

Parameter: refnum

Data Type: Word

Purpose: File reference number returned by CREATE, OPEN, OPENR, or POPEN macros

Description: The CLOSE macro terminates operations on a file. The file must have been opened by one of the macros listed above.

CREATE

Syntax: CREATE filename[,buffer][,volref]

Parameters	Data Type	Purpose
filename	Pascal string	Name of the file to be created
buffer	Address	522-byte area for file buffer
volref	Word	Volume reference number for file

Return Values: D0.W contains the file reference number for the open file.

Description: The CREATE macro places a new file on the disk and opens the data fork for reading or writing. Any previously existing file with the same name on the same disk is deleted. The buffer parameter is the optional address of a 522-byte area used by the operating system for operations on this file. Use of this buffer will significantly improve the performance of I/O operations on this file. The volume reference number is an optional indicator of the disk on which the file is to be placed. If this parameter is not specified, the file will be placed on the default drive.

DELETE

Syntax: DELETE filename[,volref]

Parameters	Data Type	Purpose
filename	Pascal string	Name of file to be deleted
volref	Word	Optional drive specifier

Return Values: D0.W = 0 if no error; D0.W = error code if DELETE is unsuccessful.

Description: The DELETE macro removes an existing file from the specified drive. If the volume reference number is not specified, all drives will be searched for the indicated file.

DIALOG

Syntax: DIALOG ID

Parameter: ID

Data Type: Word

Purpose: Resource ID of dialog template

Return Values: D0.W = item number selected by the user.

Description: The DIALOG macro invokes the Macintosh modal dialog routine. A dialog box is placed on the screen, and the user must select an item before the program will continue. The item number selected is returned in D0.W. Item numbers are defined by the order in which the dialog items are specified in the program's resource file. Items are numbered consecutively from 1.

DISABLE

Syntax: DISABLE menuID,selection

Parameters	Data Type	Purpose
menuID	Word	Resource ID of a menu in the menu bar
selection	Word	Item number within the menu

Description: The DISABLE macro inhibits selection of the menu item indicated by the selection parameter. Menu items are numbered consecutively from 1. The menuID is the resource ID of the menu on which the item is to be found. Menus used with the run-time library must also be numbered consecutively from 1.

ENABLE

Syntax: ENABLE menuID,selection

Parameters	Data Type	Purpose
menuID	Word	Resource ID of a menu in the menu bar
selection	Word	Item number within the menu

Description: The ENABLE macro allows selection of the menu item indicated by the selection parameter, undoing the action of any previous DISABLE macro. Menu items are numbered consecutively from 1. The menuID is the resource ID of the menu on which the item is to be found. Menus used with the run-time library must also be numbered consecutively from 1.

ENDPROGRAM

Syntax: ENDPROGRAM

Parameters: None

Description: The ENDPROGRAM macro is the last line in the program source file, other than an END statement. This macro generates no code and serves only to ensure the correct format of quoted strings declared as instruction operands.

EXIT

Syntax: EXIT

Parameters: None

Description: The EXIT macro causes a return to the Finder. The only action taken before invoking the Finder is to ensure that all data output to disk files is actually written to disk.

GETEVENT

Syntax: GETEVENT [cursor]

Parameter: cursor

Data Type: Word

Purpose: Resource ID of cursor

Return Values:

D0 Low	D0 High	D1 High	D1 Low	Event
00	–	–	–	No event is ready
01	–	Mouse V	Mouse H	Mouse down (D1 = point)
02	–	–	–	Mouse up
03	Modifier	–	Character	Key down
04	–	–	–	Key up
05	Modifier	–	Character	(Repeated) Key down
06	–	Window address		Update-window event
07	–	–	Drive #	Disk-inserted event
08	–	Window address		Activate event
09	–	–	–	Abort event
10	–	–	–	Network event
11	–	–	–	Driver event
12–15	–	–	–	Application event 1–4
16	–	Menu #	Item #	Menu selection
17	–	–	–	Close-window event
18	–	Height	Width	Resize-window event
19	SType	Control ID		Plain scroll event
20	Value	Control ID		Vertical thumb scroll
21	Value	Control ID		Horizontal thumb scroll

Description: The GETEVENT macro fetches the next event from the run-time library. On return, the low-order word of D0 contains the event type. The high word of D0 and all of D1 contain additional return parameters that depend on the event type. The cursor parameter specifies a cursor to be used when the mouse is inside the application window. If this parameter is omitted, the I-beam (text) cursor will be used. The arrow cursor is used outside the application window. These are the standard cursors:

Code	Resulting Cursor
0	Arrow
1	I-beam (text)

2	Cross
3	Plus
4	Watch

GETFINFO

Syntax: GETFINFO filename,buffer[,volref]

Parameters	Data Type	Purpose
filename	Pascal string	Name of file for which information is desired
buffer	Address	80-byte area to receive information
volref	Word	Optional drive specifier

Return Values: D0.W = 0 if no error; D0.W < 0 if GETFINFO fails.

Description: The GETFINFO macro returns the file information for the specified file. The format of this information is shown in Figure 4.7 (Chapter 4).

INPUT

Syntax: INPUT buffer[,length]

Parameters	Data Type	Purpose
buffer	Address	Memory area to receive data
length	Word	Number of data bytes in buffer area

Description: The INPUT macro reads a line of keyboard input and places the characters read in the input buffer. Unused bytes in the input buffer are set to zero. If the length parameter is not specified, a length of 80 is assumed.

The input buffer is formatted as a Pascal string. The first byte of the buffer contains the number of characters read. This count byte is not included in the length parameter. For instance, if you do not specify the length parameter, a length of 80 data bytes is assumed. You would need a data area of at least 81 data bytes to allow for the count byte.

NOTEALERT

Syntax: NOTEALERT ID

Parameter: ID

Data Type: Word

Purpose: Resource ID of alert template

Return Values: D0.W = item number selected by the user.

Description: The NOTEALERT macro causes an alert box to be displayed on the Macintosh screen. Alert boxes are shown in Figure 6.16 (Chapter 6). The box drawn by the NOTEALERT macro has the asterisk icon in the upper-left corner.

OPEN

Syntax: OPEN filename[,buffer][,volref]

Parameters	Data Type	Purpose
filename	Pascal string	Name of file to open
buffer	Address	Optional 522-byte file buffer
volref	Word	Optional drive specifier

Return Values: D0.W = file reference number if no error; D0.W < 0 if OPEN is unsuccessful.

Description: The OPEN macro connects the application to the data fork of an existing file. The file reference number returned by OPEN identifies the file on subsequent CLOSE, READ, READLN, WRITE, and SEEK operations.

OPENR

Syntax: OPENR filename[,buffer][,volref]

Parameters	Data Type	Purpose
filename	Pascal string	Name of file to open
buffer	Address	Optional 522-byte file buffer
volref	Word	Optional drive specifier

Return Values: D0.W = file reference number if no error; D0.W < 0 if OPENR is unsuccessful.

Description: The OPENR macro connects the application to the resource fork of an existing file. The file reference number returned by OPENR identifies the file on subsequent CLOSE, READ, READLN, WRITE, and SEEK operations.

PCREATE

Syntax: PCREATE prompt,defname,namebuf,buffer

Parameters	Data Type	Purpose
prompt	Pascal string	Prompt string for dialog box
defname	Pascal string	Initial file name for dialog box
namebuf	Address	Buffer area for final file name
buffer	Address	522-byte area for file buffer

Return Values: D0.W contains the file reference number for the open file. D0.W < 0 if PCREATE failed. D1.W contains the volume reference number.

Description: The PCREATE macro places a new file on the disk and opens the data fork for reading or writing. Any previously existing file with the same name on the same disk is deleted. This macro uses the Macintosh standard file package to allow the user to specify the file name and drive. The defname parameter specifies a name that can be edited by the user. The namebuf parameter is the address of an area that will receive the final file name chosen by the user. The buffer parameter is the optional address of a 522-byte area used by the operating system for operations on this file. Use of this buffer will significantly improve the performance of I/O operations on this file.

POPEN

Syntax: POPEN types,ext,namebuf,buffer

Parameters	Data Type	Purpose
types	Address	File type array
ext	Pascal string	File extension string
namebuf	Address	Buffer area for final file name
buffer	Address	522-byte area for file buffer

Return Values: D0.W contains the file reference number for the open file. D0.W < 0 if POPEN failed. D1.W contains the volume reference number.

Description: The POPEN macro connects the application with the data fork of an existing file. This macro uses the Macintosh standard file package to allow the user to specify the file name and drive. The types parameter is the address of an array that gives the acceptable file types. Any file that does not have one of these types will not appear in the selection window presented to the user. This array has a 16-bit count followed by 32-bit file types. If this parameter is not specified, all file types will be selected. The ext parameter is a Pascal string that specifies the final characters in each file name selected for display. You can use this feature to select all files ending in .asm, for instance. All letters in the ext string must be uppercase. The namebuf parameter is the address of an area that will receive the final file name chosen by the user. The buffer parameter is the optional address of a 522-byte area used by the operating system for operations on this file. Use of this buffer will significantly improve the performance of I/O operations on this file.

PRINT

Syntax: PRINT string[,crflag]

Parameters	Data Type	Purpose
string	Pascal string	Characters to output to the screen
crflag	(none)	No line advance if nonblank

Description: The PRINT macro displays a string of text in the application window. The string can have embedded carriage-return characters, which will produce multiple lines. If the crflag parameter is omitted, a carriage return will be appended to the end of the string. Any nonblank crflag parameter will inhibit this final line advance.

PROGRAM

Syntax: PROGRAM [name][,debug][,menus]

Parameters	Data Type	Purpose
name	Quoted string	Name to appear in program window
debug	(none)	DEBUGON to use debugger
menus	16-bit constant	Number of menus in resource file

Return Values: A0.L = window address; D0.L = window size.

Description: The PROGRAM macro immediately precedes the first instruction to be executed in the application. The name parameter specifies a name to appear as the title of the window created during initialization. The name is also used in locating the resource file for the application. If the file Name.RSRC exists, it will be used as the application's resource file. The debug parameter is the string DEBUGON; it is used when you want to enter the debugger just prior to executing the first instruction following the PROGRAM macro. The menu parameter is the number of menus in the resource file. Menu resource IDs must be numbered consecutively from 1.

QUIT

Syntax: QUIT

Parameters: None

Description: The QUIT macro waits for a Quit choice from the File menu. This macro calls GETEVENT repeatedly until the first item is selected from the second menu. After the selection is made, the macro exits to the Finder.

READ

Syntax: READ refnum,buffer,count

Parameters	Data Type	Purpose
refnum	Word	Reference number of open file
buffer	Address	Memory address to receive data from file
count	Longword	Number of bytes to transfer

Return Values: D0.L = number of bytes actually transferred; D0.L < 0 if READ is unsuccessful.

Description: The READ macro transfers data from an open file into application memory. The transfer from the file begins immediately following the last transfer unless the file pointer was altered by a SEEK macro. On return, D0.L contains the number of bytes actually read. This may be less than the number requested if the number of bytes left before the end of the file is less than the count parameter.

READLN

Syntax: READLN refnum,buffer,count

Parameters	Data Type	Purpose
refnum	Word	Reference number of open file
buffer	Address	Memory address to receive data from file
count	Longword	Maximum number of bytes to transfer

Return Values: D0.L = number of bytes actually transferred; D0.L < 0 if READLN is unsuccessful.

Description: The READLN macro transfers data from an open file into application memory. The transfer from the file begins immediately following the last transfer unless the file pointer was altered by a SEEK macro. The transfer ends on the next carriage-return character in the file or when the count is satisfied. On return, D0.L contains the number of bytes actually read.

SCURSOR

Syntax: SCURSOR ID

Parameter: ID

Data Type: Word

Purpose: Resource ID of desired cursor

Description: The SCURSOR changes the mouse cursor to a shape contained in the application or system resource file. The system file contains the following standard cursors:

Code	Resulting Cursor
0	Arrow
1	I-beam (text)
2	Cross
3	Plus
4	Watch

SEEK

Syntax: SEEK refnum,offset,sense

Parameters	Data Type	Purpose
refnum	Word	Reference number for open file
offset	Longword	File offset
sense	Word	Interpretation of offset parameter: 0 From beginning of file 1 From current position 2 From end of file

Return Values: D0.L contains resulting file position.

Description: The SEEK macro specifies the file location at which the next transfer operation will begin. An attempt to position the file pointer beyond the end of file places it one byte beyond the end.

SETFINFO

Syntax: SETFINFO filename,buffer[,volref]

Parameters	Data Type	Purpose
filename	Pascal string	Name of file for which information is desired
buffer	Address	80-byte area containing information
volref	Word	Optional drive specifier

Return Values: D0.W = 0 if no error; D0.W < 0 if SETFINFO failed.

Description: The SETFINFO macro modifies the file information for the specified file. The format of this information is shown in Figure 4.7 (Chapter 4).

SETHMAX

Syntax: SETHMAX value

Parameter: value

Data Type: Word

Purpose: New maximum value

Description: The SETHMAX macro sets the maximum value associated with the horizontal scroll bar. This is the value returned in a scrolling event when the document is positioned all the way to the right.

SETHMIN

Syntax: SETHMIN value

Parameter: value

Data Type: Word

Purpose: New minimum value

Description: The SETHMIN macro sets the minimum value associated with the horizontal scroll bar. This is the value returned in a scrolling event when the document is positioned all the way to the left.

SETHVAL

Syntax: SETHVAL value

Parameter: value

Data Type: Word

Purpose: New horizontal value

Description: The SETHVAL macro sets the thumb value associated with the horizontal scroll bar. Setting the value causes the thumb to be repositioned.

SETVMAX

Syntax: SETVMAX value

Parameter: value

Data Type: Word

Purpose: New maximum value

Description: The SETVMAX macro sets the maximum value associated with the vertical scroll bar. This is the value returned in a scrolling event when the document is positioned all the way at the bottom.

SETVMIN

Syntax: SETVMIN value

Parameter: value

Data Type: Word

Purpose: New minimum value

Description: The SETVMIN macro sets the minimum value associated with the vertical scroll bar. This is the value returned in a scrolling event when the document is positioned all the way at the top.

SETVVAL

Syntax: SETVVAL value

Parameter: value

Data Type: Word

Purpose: New vertical value

Description: The SETVVAL macro sets the thumb value associated with the vertical scroll bar. Setting the value causes the thumb to be repositioned.

STOPALERT

Syntax: STOPALERT ID

Parameter: ID

Data Type: Word

Purpose: Resource ID of alert template

Return Values: D0.W = item number selected by the user.

Description: The STOPALERT macro causes an alert box to be displayed on the Macintosh screen. Alert boxes are shown in Figure 6.16 (Chapter 6). The box drawn by the STOPALERT macro has the exclamation point icon in the upper-left corner.

STYLE

Syntax: STYLE menuID,selection,style

Parameters	Data Type	Purpose
menuID	Word	Resource ID of a menu in the menu bar
selection	Word	Item number within the menu
style	Word	Text style mask

Description: The STYLE macro changes the text style of the item indicated by the selection parameter. Menu items are numbered consecutively from 1. The menuID is the resource ID of the menu on which the item is to be found. Menus used with the run-time library must also be numbered consecutively from 1. The style word contains the text attributes, as described in Figure 6.10 (Chapter 6).

TRACK

Syntax: TRACK controlID,scrollproc

Parameters	**Data Type**	**Purpose**
controlID	Longword	Scroll bar ID from scrolling event
scrollproc	Address	Pointer to function handling scrolling

Description: The TRACK macro causes line and page scrolls to repeat until the mouse button is released. This macro handles highlighting of the appropriate controls while the mouse button remains down. The scrollproc procedure should perform a single instance of the desired scrolling action. The TRACK macro calls this procedure repeatedly until the user releases the mouse button.

UNCHECK

Syntax: UNCHECK menuID,selection

Parameter	**Data Type**	**Purpose**
menuID	Word	Resource ID of a menu in the menu bar
selection	Word	Item number within the menu

Description: The UNCHECK macro removes any check mark in front of the item indicated by the selection parameter. Menu items are numbered consecutively from 1. The menuID is the resource ID of the menu on which the item is to be found. Menus used with the run-time library must also be numbered consecutively from 1.

WRITE

Syntax: WRITE refnum,buffer,count

Parameters	Data Type	Purpose
refnum	Word	Reference number of open file
buffer	Address	Memory address containing data for file
count	Longword	Number of bytes to transfer

Return Values: D0.L = number of bytes actually transferred; D0.L < 0 if WRITE is unsuccessful.

Description: The WRITE macro transfers data from application memory to an open file. The transfer to the file begins immediately following the last transfer unless the file pointer was altered by a SEEK macro. On return, D0.L contains the number of bytes actually written. If this number is less than the number requested, an error occurred.

WTITLE

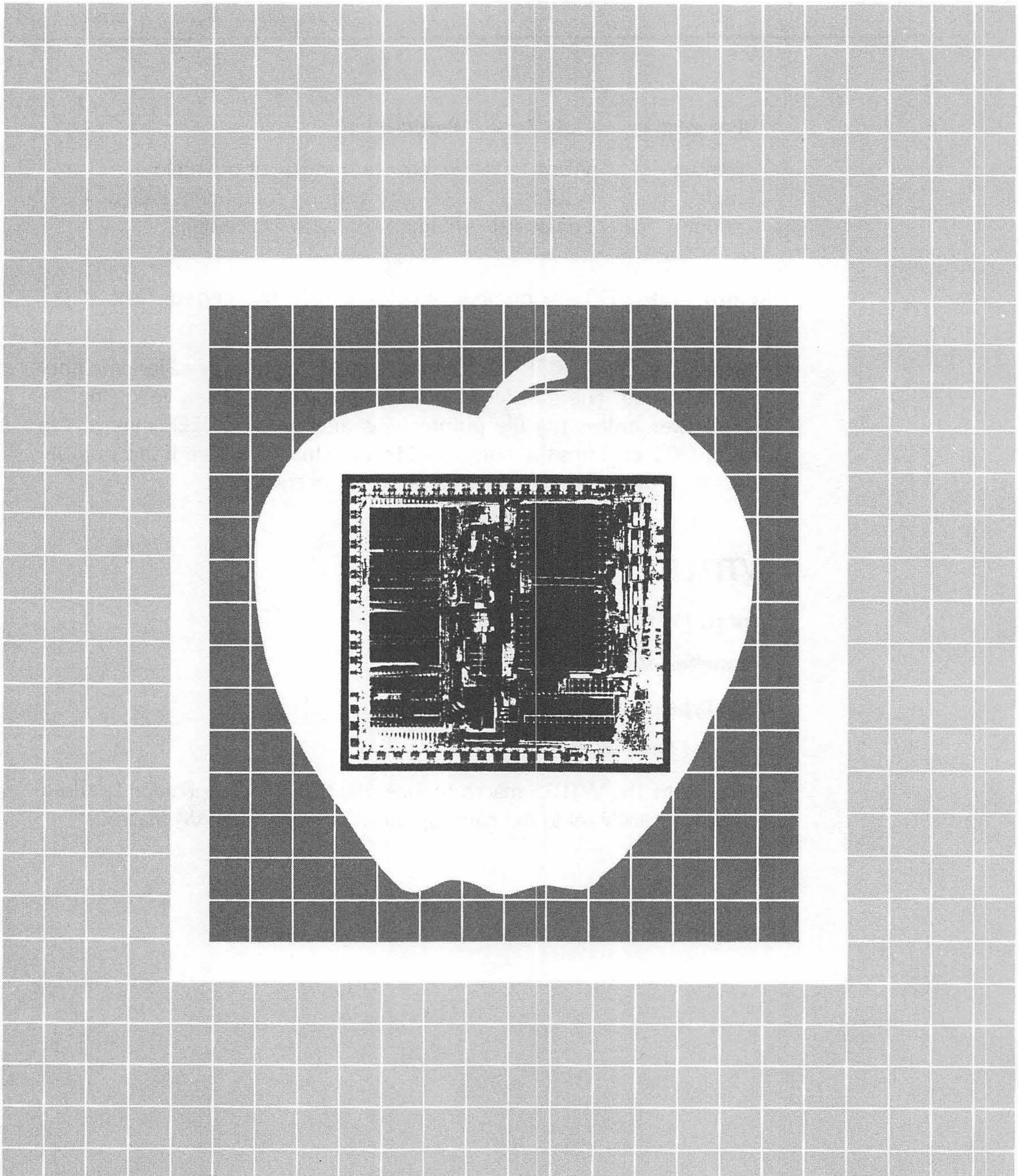
Syntax: WTITLE string

Parameter: string

Data Type: Pascal string

Purpose: New window title

Description: The WTITLE macro replaces the title in the application window. The title is initially set to the name specified in the PROGRAM macro.



Appendix **D**

68000 Quick Reference

A man is only as good as the tool he uses.

—Goethe

EFFECTIVE ADDRESS SUMMARY

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction Format:	Op Code				////////////////						Effective			Address		
											← Mode →			← Reg. →		

Addressing Mode Name	Syntax	Mode	Register
Data Register Direct	Dn	000	Data Register
Address Register Direct	An	001	Address Register
Address Register Indirect	(An)	010	Address Register
Address Register Indirect Postincrement	(An)+	011	Address Register
Address Register Indirect Predecrement	−(An)	100	Address Register
Address Register Indirect with Displacement	w(An)	101	Address Register
Address Register Indirect with Index	b(An,Rx)	110	Address Register
Absolute Short	w	111	000
Absolute Long	l	111	001
Program Counter with Displacement	w(PC)	111	010
Program Counter with Index	b(PC,Rx)	111	011
Immediate	#x	111	100
Status Register	SR	111	100
Condition Code Register	CCR	111	100

Legend:

Dn	Data Register	(n is 0-7)
An	Address Register	(n is 0-7)
b	08-bit constant	
w	16-bit constant	
l	32-bit constant	
x	8-, 16-, or 32-bit constant	
Rx	Index Register Specification, one of:	

Dn.W	Low 16 bits of Data Register
Dn.L	All 32 bits of Data Register
An.W	Low 16 bits of Address Register
An.L	All 32 bits of Address Register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Index Extension Word:	A	Register			Sz	0	0	0	8-bit Displacement							

"A" is the type of index register: 1 for an address register, 0 for a data register. "Sz" is 1 for a long index, 0 for word.

OPERATION CODE SUMMARY

Bits 12 through 15	Operation
0000	Bit Manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc/DBcc
0110	Bcc / BSR
0111	MOVEQ
1000	OR / DIV / SBCD
1001	SUB / SUBX
1010	(unassigned)
1011	CMP / EOR
1100	AND / MUL / ABCD / EXG
1101	ADD / ADDX
1110	ASL/ASR/ROL/ROR/ROXL/ROXR
1111	(unassigned)

CONDITION CODE NOTATION

Symbol	Meaning
*	Set according to result of operation
–	Not affected
0	Cleared
1	Set
U	Outcome (state after operation) undefined
I	Set by immediate data

NUMERICAL INSTRUCTION SUMMARY

Instruction — ***Machine Code Format*** — ***Condition Codes***

ORI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	0	0	0	0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

BTST
(Dynamic)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	D Reg.			1	0	0	Effective			Address		

X	N	Z	V	C
-	-	*	-	-

BCHG
(Dynamic)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	D Reg.			1	0	1	Effective			Address		

X	N	Z	V	C
-	-	*	-	-

BCLR
(Dynamic)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	D Reg.			1	1	0	Effective			Address		

X	N	Z	V	C
-	-	*	-	-

BSET
(Dynamic)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	D Reg.			1	1	1	Effective			Address		

X	N	Z	V	C
-	-	*	-	-

MOVEP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	D Reg.			1	Dr	Sz	0	0	1	Address Register		

X	N	Z	V	C
-	-	-	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
ANDI	0000	0	0	1	0	Size		Effective Address						-	*	*	0	0
SUBI	0000	0	1	0	0	Size		Effective Address						*	*	*	*	*
ADDI	0000	0	1	1	0	Size		Effective Address						*	*	*	*	*
BTST (Static)	0000	1	0	0	0	0	0	Effective Address						-	-	*	-	-
BCHG (Static)	0000	1	0	0	0	0	1	Effective Address						-	-	*	-	-
BCLR (Static)	0000	1	0	0	0	1	0	Effective Address						-	-	*	-	-
BSET (Static)	0000	1	0	0	0	1	1	Effective Address						-	-	*	-	-

Instruction	Machine Code Format	Condition Codes

EORI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	1	0	1	0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

CMPI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	1	1	0	0	Size		Effective			Address		

X	N	Z	V	C
*	*	*	*	*

MOVES
(68010)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	1	1	1	0	Size		Effective			Address		

X	N	Z	V	C
-	-	-	-	-

MOVE.B

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0001	Register			Mode			Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	*	*	-	-

MOVEA.L

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0010	Register			0	0	1	Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	-	-	-	-

MOVE.L

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0010	Register			Mode			Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	*	*	0	0

Instruction	Machine Code Format													Condition Codes				
	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
MOVEA.W	0011	Register			0	0	1	Mode			Register			-	-	-	-	-
		← Destination →						← Source →										

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
MOVE.W	0011	Register			Mode			Mode			Register			-	*	*	0	0
		← Destination →						← Source →										

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
NEGX	0100	0	0	0	0	Size		Effective Address						*	*	*	*	*

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
MOVE From SR	0100	0	0	0	0	1	1	Effective Address						-	-	-	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
CHK	0100	D Reg.			1	1	0	Effective Address						-	*	U	U	U

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
LEA	0100	A Reg.			1	1	1	Effective Address						-	-	-	-	-

Instruction	***Machine Code Format***	***Condition Codes***

CLR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	0	1	0	Size		Effective			Address		

X	N	Z	V	C
-	0	1	0	0

MOVE From CCR *(68010)*

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	0	1	0	1	1	Effective			Address		

X	N	Z	V	C
-	-	-	-	-

NEG

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	1	0	0	Size		Effective			Address		

X	N	Z	V	C
*	*	*	*	*

MOVE to CCR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	1	0	0	1	1	Effective			Address		

X	N	Z	V	C
I	I	I	I	I

NOT

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	1	1	0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

MOVE to SR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	1	1	0	1	1	Effective			Address		

X	N	Z	V	C
I	I	I	I	I

Instruction	Machine Code Format	Condition Codes

NBCD

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	0	0	Effective Address					

X	N	Z	V	C
*	U	*	U	*

SWAP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	0	1	0	0	0	D Reg.		

X	N	Z	V	C
-	*	*	0	0

PEA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	0	1	Effective Address					

X	N	Z	V	C
-	-	-	-	-

EXT.W

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	1	0	0	0	0	D Reg.		

X	N	Z	V	C
-	*	*	0	0

MOVEM
(Regs to EA)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	1	Sz	Effective Address					

X	N	Z	V	C
-	-	-	-	-

EXT.L

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	1	1	0	0	0	D Reg.		

X	N	Z	V	C
-	*	*	0	0

TST

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	1	0	Size		Effective Address					

X	N	Z	V	C
-	*	*	0	0

Instruction ***Machine Code Format*** ***Condition Codes***

TAS

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	1	0	1	1	Effective Address					

X	N	Z	V	C
-	*	*	0	0

ILLEGAL

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	1	0	1	1	1	1	1	1	0	0

X	N	Z	V	C
-	-	-	-	-

MOVEM
(EA to Regs)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	0	0	1	Sz	Effective Address					

X	N	Z	V	C
-	-	-	-	-

TRAP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	0	0	Vector			

X	N	Z	V	C
-	-	-	-	-

LINK

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	0	1	0	A Reg.		

X	N	Z	V	C
-	-	-	-	-

UNLK

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	0	1	1	A Reg.		

X	N	Z	V	C
-	-	-	-	-

MOVE
to USP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	0	0	A Reg.		

X	N	Z	V	C
-	-	-	-	-

Instruction	Machine Code Format													Condition Codes
	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X N Z V C
MOVE from USP	0100	1	1	1	0	0	1	1	0	1	A Reg.			- - - - -
RESET	0100	1	1	1	0	0	1	1	1	0	0	0	0	- - - - -
NOP	0100	1	1	1	0	0	1	1	1	0	0	0	1	- - - - -
STOP	0100	1	1	1	0	0	1	1	1	0	0	1	0	I I I I I
RTE	0100	1	1	1	0	0	1	1	1	0	0	1	1	I I I I I
RTD *(68010)*	0100	1	1	1	0	0	1	1	1	0	1	0	0	- - - - -
RTS	0100	1	1	1	0	0	1	1	1	0	1	0	1	- - - - -

Instruction	Machine Code Format	Condition Codes

TRAPV

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	1	1	0

X	N	Z	V	C
-	-	-	-	-

RTR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	1	1	1

X	N	Z	V	C
I	I	I	I	I

MOVEC
(68010)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	1	0	1	Rd

X	N	Z	V	C
-	-	-	-	-

JSR

15–12	11	10	9	8	7	6	5–0
0100	1	1	1	0	1	0	Effective Address

X	N	Z	V	C
-	-	-	-	-

JMP

15–12	11	10	9	8	7	6	5–0
0100	1	1	1	0	1	1	Effective Address

X	N	Z	V	C
-	-	-	-	-

ADDQ

15–12	11–9	8	7–6	5–0
0101	IData	0	Size	Effective Address

X	N	Z	V	C
*	*	*	*	*

Scc

15–12	11–8	7	6	5–0
0101	Condition	1	1	Effective Address

X	N	Z	V	C
-	-	-	-	-

Instruction	***Machine Code Format***	***Condition Codes***

DBcc

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0101	Condition			1	1	0	0	1	D Reg.			

X	N	Z	V	C
-	-	-	-	-

SUBQ

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0101	Data			1	Size		Effective Address					

X	N	Z	V	C
*	*	*	*	*

Bcc

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0110	Condition				8-bit Displacement							

X	N	Z	V	C
-	-	-	-	-

BSR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0110	0	0	0	1	8-bit Displacement							

X	N	Z	V	C
-	-	-	-	-

MOVEQ

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0111	D Reg.			0	8-bit Data Value							

X	N	Z	V	C
-	*	*	0	0

OR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	D Reg.			Dr	Size		Effective Address					

X	N	Z	V	C
-	*	*	0	0

DIVU

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	D Reg.			0	1	1	Effective Address					

X	N	Z	V	C
-	*	*	*	0

Instruction	***Machine Code Format***	***Condition Codes***

SBCD

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	Dest Reg			1	0	0	0	0	DA	Src Reg		

X	N	Z	V	C
*	U	*	U	*

DIVS

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	D Reg.			1	1	1	Effective Address					

X	N	Z	V	C
-	*	*	*	0

SUB

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1001	D Reg.			Dr	Size		Effective Address					

X	N	Z	V	C
*	*	*	*	*

SUBA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1001	A Reg.			Sz	1	1	Effective Address					

X	N	Z	V	C
-	-	-	-	-

SUBX

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1001	Dest Reg			1	Size		0	0	DA	Src Reg		

X	N	Z	V	C
*	*	*	*	*

CMP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1011	D Reg.			0	Size		Effective Address					

X	N	Z	V	C
-	*	*	*	*

CMPA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1011	A Reg.			Sz	1	1	Effective Address					

X	N	Z	V	C
-	*	*	*	*

Instruction	***Machine Code Format***	***Condition Codes***

EOR

15–12	11 10 9	8	7 6	5 4 3 2 1 0
1011	D Reg.	1	Size	Effective Address

X	N	Z	V	C
-	*	*	0	0

CMPM

15–12	11 10 9	8	7 6	5	4	3	2 1 0
1011	Dest Reg	1	Size	0	0	1	Src Reg

X	N	Z	V	C
-	*	*	*	*

AND

15–12	11 10 9	8	7 6	5 4 3 2 1 0
1100	D Reg.	Dr	Size	Effective Address

X	N	Z	V	C
-	*	*	0	0

MULU

15–12	11 10 9	8	7	6	5 4 3 2 1 0
1100	D Reg.	0	1	1	Effective Address

X	N	Z	V	C
-	*	*	0	0

ABCD

15–12	11 10 9	8	7	6	5	4	3	2 1 0
1100	Dest Reg	1	0	0	0	0	DA	Src Reg

X	N	Z	V	C
*	U	*	U	*

EXG
(2 D regs)

15–12	11 10 9	8	7	6	5	4	3	2 1 0
1100	D Reg.	1	0	1	0	0	0	D Reg.

X	N	Z	V	C
-	-	-	-	-

EXG
(2 A regs)

15–12	11 10 9	8	7	6	5	4	3	2 1 0
1100	A Reg.	1	0	1	0	0	1	A Reg.

X	N	Z	V	C
-	-	-	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
EXG *(A and D reg)*	1100	D Reg.			1	1	0	0	0	1	A Reg.			-	-	-	-	-
MULS	1100	D Reg.			1	1	1	Effective Address						-	*	*	0	0
ADD	1101	D Reg.			Dr	Size		Effective Address						*	*	*	*	*
ADDA	1101	A Reg.			Sz	1	1	Effective Address						-	-	-	-	-
ADDX	1101	Dest Reg			1	Size		0	0	DA	Src Reg			*	*	*	*	*
ASR *(Memory form)*	1110	0	0	0	0	1	1	Effective Address						*	*	*	*	*
ASL *(Memory form)*	1110	0	0	0	1	1	1	Effective Address						*	*	*	*	*

Instruction	Machine Code Format	Condition Codes

LSR (Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	0	1	0	1	1	Effective Address					

X	N	Z	V	C
*	*	*	0	*

LSL (Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	0	1	1	1	1	Effective Address					

X	N	Z	V	C
*	*	*	0	*

ROXR (Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	0	0	1	1	Effective Address					

X	N	Z	V	C
*	*	*	0	*

ROXL (Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	0	1	1	1	Effective Address					

X	N	Z	V	C
*	*	*	0	*

ROR (Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	1	0	1	1	Effective Address					

X	N	Z	V	C
-	*	*	0	*

ROL (Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	1	1	1	1	Effective Address					

X	N	Z	V	C
-	*	*	0	*

Instruction	***Machine Code Format*** 15–12	11 10 9	8	7 6	5	4	3	2 1 0	***Condition Codes*** X	N	Z	V	C
ASR *(Register form)*	1110	Cnt/Reg	0	Size	IR	0	0	D Reg.	*	*	*	*	*
ASL *(Register form)*	1110	Cnt/Reg	1	Size	IR	0	0	D Reg.	*	*	*	*	*
LSR *(Register form)*	1110	Cnt/Reg	0	Size	IR	0	1	D Reg.	*	*	*	0	*
LSL *(Register form)*	1110	Cnt/Reg	1	Size	IR	0	1	D Reg.	*	*	*	0	*
ROXR *(Register form)*	1110	Cnt/Reg	0	Size	IR	1	0	D Reg.	*	*	*	0	*
ROXL *(Register form)*	1110	Cnt/Reg	1	Size	IR	1	0	D Reg.	*	*	*	0	*

Instruction	***Machine Code Format***	***Condition Codes***

ROR
(Register form)

15–12	11 10 9	8	7 6	5	4	3	2 1 0
1110	Cnt/Reg	0	Size	IR	1	1	D Reg.

X	N	Z	V	C
-	*	*	0	*

ROL
(Register form)

15–12	11 10 9	8	7 6	5	4	3	2 1 0
1110	Cnt/Reg	1	Size	IR	1	1	D Reg.

X	N	Z	V	C
-	*	*	0	*

Legend:

Size — Specifies instruction data size:

00 Byte
01 Word
10 Long

Dr — Specifies instruction direction:

0 EA or Memory to Dn
1 Dn to EA or Memory

Sz — Specifies instruction data size:

0 Word
1 Long

Rd — Specifies instruction direction:

0 Control register to General Register
1 General register to Control Register

IData — 3 bits of immediate data:

000 Specifies the value "8".
001–111 Specify values 1–7.

Condition Specifies a branch condition:

Condition	Instruction	Condition	Instruction
0000	BRA	1000	BVC
0001	(NONE)	1001	BVS
0010	BHI	1010	BPL
0011	BLS	1011	BMI
0100	BCC	1100	BGE
0101	BCS	1101	BLT
0110	BNE	1110	BGT
0111	BEQ	1111	BLE

DA Specifies the type of registers used:

0 Data Registers
1 Address Registers

Cnt Reg Specifies a count or a register value:

If the IR field is 0, then 3-bit IData format
If the IR field is 1, then Data Register Number

ALPHABETICAL INSTRUCTION SUMMARY

Instruction ***Machine Code Format*** ***Condition Codes***

ABCD

15–12	11 10 9	8	7	6	5	4	3	2 1 0
1100	Dest Reg	1	0	0	0	0	DA	Src Reg

X	N	Z	V	C
*	U	*	U	*

ADD

15–12	11 10 9	8	7 6	5 4 3 2 1 0
1101	D Reg.	Dr	Size	Effective Address

X	N	Z	V	C
*	*	*	*	*

Instruction | **Machine Code Format** | **Condition Codes**

ADDA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1101	A Reg.			Sz	1	1	Effective			Address		

X	N	Z	V	C
-	-	-	-	-

ADDI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	0	1	1	0	Size		Effective			Address		

X	N	Z	V	C
*	*	*	*	*

ADDQ

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0101	IData			0	Size		Effective			Address		

X	N	Z	V	C
*	*	*	*	*

ADDX

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1101	Dest Reg			1	Size		0	0	RM	Src Reg		

X	N	Z	V	C
*	*	*	*	*

AND

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1100	D Reg.			Dr	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

ANDI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	0	0	1	0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

ASL
(Register form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	Cnt/Reg			1	Size		IR	0	0	D Reg.		

X	N	Z	V	C
*	*	*	*	*

Instruction	Machine Code Format													Condition Codes				
	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
ASL (Memory form)	1110	0	0	0	1	1	1	Effective Address						*	*	*	*	*

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
ASR (Register form)	1110	Cnt/Reg			0	Size		IR	0	0	D Reg.			*	*	*	*	*

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
ASR (Memory form)	1110	0	0	0	0	1	1	Effective Address						*	*	*	*	*

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
Bcc	0110	Condition				8-bit Displacement								-	-	-	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
BCHG (Dynamic form)	0000	D Reg.			1	0	1	Effective Address						-	-	*	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
BCHG (Static form)	0000	1	0	0	0	0	1	Effective Address						-	-	*	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
BCLR (Dynamic form)	0000	D Reg.			1	1	0	Effective Address						-	-	*	-	-

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
BCLR *(Static form)*	0000	1	0	0	0	1	0	Effective Address						-	-	*	-	-
BSET *(Dynamic form)*	0000	D Reg.			1	1	1	Effective Address						-	-	*	-	-
BSET *(Static form)*	0000	1	0	0	0	1	1	Effective Address						-	-	*	-	-
BSR	0110	0	0	0	1	8-bit Displacement								-	-	-	-	-
BTST *(Dynamic form)*	0000	D Reg.			1	0	0	Effective Address						-	-	*	-	-
BTST *(Static form)*	0000	1	0	0	0	0	0	Effective Address						-	-	*	-	-
CHK	0100	D Reg.			1	1	0	Effective Address						-	*	U	U	U

Instruction	Machine Code Format	Condition Codes

CLR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	0	1	0	Size		Effective			Address		

X	N	Z	V	C
-	0	1	0	0

CMP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1011	D Reg.			0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	*	*

CMPA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1011	A Reg.			Sz	1	1	Effective			Address		

X	N	Z	V	C
-	*	*	*	*

CMPI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	1	1	0	0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	*	*

CMPM

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1011	Dest Reg			1	Size		0	0	1	Src Reg		

X	N	Z	V	C
-	*	*	*	*

DBcc

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0101	Condition				1	1	0	0	1	D Reg.		

X	N	Z	V	C
-	-	-	-	-

DIVS

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	D Reg.			1	1	1	Effective			Address		

X	N	Z	V	C
-	*	*	*	0

Instruction	***Machine Code Format***	***Condition Codes***

DIVU

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	D Reg.			0	1	1	Effective			Address		

X	N	Z	V	C
-	*	*	*	0

EOR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1011	D Reg.			1	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

EORI

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0000	1	0	1	0	Size		Effective			Address		

X	N	Z	V	C
-	*	*	0	0

EXG
(2 D regs)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1100	D Reg.			1	0	1	0	0	0	D Reg.		

X	N	Z	V	C
-	-	-	-	-

EXG
(2 A regs)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1100	A Reg.			1	0	1	0	0	1	A Reg.		

X	N	Z	V	C
-	-	-	-	-

EXG
(A and D reg)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1100	D Reg.			1	1	0	0	0	1	A Reg.		

X	N	Z	V	C
-	-	-	-	-

EXT.L

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	1	1	0	0	0	D Reg.		

X	N	Z	V	C
-	*	*	0	0

Instruction	***Machine Code Format***	***Condition Codes***

EXT.W

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	0	0	1	0	0	0	0	D Reg.		

X	N	Z	V	C
-	*	*	0	0

ILLEGAL

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	0	1	0	1	1	1	1	1	1	0	0

X	N	Z	V	C
-	-	-	-	-

JMP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	1	1	Effective Address					

X	N	Z	V	C
-	-	-	-	-

JSR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	1	0	Effective Address					

X	N	Z	V	C
-	-	-	-	-

LEA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	A Reg.			1	1	1	Effective Address					

X	N	Z	V	C
-	-	-	-	-

LINK

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	0	1	0	A Reg.		

X	N	Z	V	C
-	-	-	-	-

LSL
(Register form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	Cnt/Reg			1	Size		IR	0	1	D Reg.		

X	N	Z	V	C
*	*	*	0	*

Instruction	***Machine Code Format***	***Condition Codes***

LSL *(Memory form)*

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	0	1	1	1	1	Effective Address					

X	N	Z	V	C
*	*	*	0	*

LSR *(Register form)*

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	Cnt/Reg			0	Size		IR	0	1	D Reg.		

X	N	Z	V	C
*	*	*	0	*

LSR *(Memory form)*

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	0	1	0	1	1	Effective Address					

X	N	Z	V	C
*	*	*	0	*

MOVE from CCR *(68010)*

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	0	1	0	1	1	Effective Address					

X	N	Z	V	C
-	-	-	-	-

MOVE from SR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	0	0	0	1	1	Effective Address					

X	N	Z	V	C
-	-	-	-	-

MOVE from USP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	0	1	A Reg.		

X	N	Z	V	C
-	-	-	-	-

MOVE to CCR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	1	0	0	1	1	Effective Address					

X	N	Z	V	C
I	I	I	I	I

Instruction	Machine Code Format	Condition Codes

MOVE to SR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	0	1	1	0	1	1	Effective Address					

X	N	Z	V	C
I	I	I	I	I

MOVE to USP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	0	0	A Reg.		

X	N	Z	V	C
-	-	-	-	-

MOVE.B

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0001	Register			Mode			Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	*	*	0	0

MOVE.L

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0010	Register			Mode			Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	*	*	0	0

MOVE.W

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0011	Register			Mode			Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	*	*	0	0

MOVEA.L

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0010	Register			0	0	1	Mode			Register		
	← Destination →						← Source →					

X	N	Z	V	C
-	-	-	-	-

Instruction	***Machine Code Format***	***Condition Codes***

MOVEA.W

15–12	11 10 9	8	7	6	5 4 3	2 1 0
0011	Register	0	0	1	Mode	Register
	← Destination →				← Source	→

X	N	Z	V	C
-	-	-	-	-

MOVEC
(68010)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	1	0	1	Rd

X	N	Z	V	C
-	-	-	-	-

MOVEM
(Regs to EA)

15–12	11	10	9	8	7	6	5 4 3 2 1 0
0100	1	0	0	0	1	Sz	Effective Address

X	N	Z	V	C
-	-	-	-	-

MOVEM
(EA to Regs)

15–12	11	10	9	8	7	6	5 4 3 2 1 0
0100	1	1	0	0	1	Sz	Effective Address

X	N	Z	V	C
-	-	-	-	-

MOVEP

15–12	11 10 9	8	7	6	5	4	3	2 1 0
0000	D Reg.	1	Dr	Sz	0	0	1	Address Register

X	N	Z	V	C
-	-	-	-	-

MOVEQ

15–12	11 10 9	8	7 6 5 4 3 2 1 0
0111	D Reg.	0	8-bit Data Value

X	N	Z	V	C
-	*	*	0	0

MOVES
(68010)

15–12	11	10	9	8	7 6	5 4 3 2 1 0
0000	1	1	1	0	Size	Effective Address

X	N	Z	V	C
-	-	-	-	-

Instruction	Machine Code Format													Condition Codes
	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X N Z V C
MULS	1100	D Reg.			1	1	1	Effective			Address			- * * 0 0
MULU	1100	D Reg.			0	1	1	Effective			Address			- * * 0 0
NBCD	0100	1	0	0	0	0	0	Effective			Address			* U * U *
NEG	0100	0	1	0	0	Size		Effective			Address			* * * * *
NEGX	0100	0	0	0	0	Size		Effective			Address			* * * * *
NOP	0100	1	1	1	0	0	1	1	1	0	0	0	1	- - - - -
NOT	0100	0	1	1	0	Size		Effective			Address			- * * 0 0

Instruction	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
OR	1000	D Reg.			Dr	Size		Effective Address						-	*	*	0	0
ORI	0000	0	0	0	0	Size		Effective Address						-	*	*	0	0
PEA	0100	1	0	0	0	0	1	Effective Address						-	-	-	-	-
RESET	0100	1	1	1	0	0	1	1	1	0	0	0	0	-	-	-	-	-
ROL *(Register form)*	1110	Cnt/Reg			1	Size		IR	1	1	D Reg.			-	*	*	0	*
ROL *(Memory form)*	1110	0	1	1	1	1	1	Effective Address						-	*	*	0	*
ROR *(Register form)*	1110	Cnt/Reg			0	Size		IR	1	1	D Reg.			-	*	*	0	*

Instruction	***Machine Code Format***	***Condition Codes***

ROR
(Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	1	0	1	1	Effective Address					

X N Z V C
\- * * 0 *

ROXL
(Register form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	Cnt/Reg			1	Size		IR	1	0	D Reg.		

X N Z V C
* * * 0 *

ROXL
(Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	0	1	1	1	Effective Address					

X N Z V C
* * * 0 *

ROXR
(Register form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	Cnt/Reg			0	Size		IR	1	0	D Reg.		

X N Z V C
* * * 0 *

ROXR
(Memory form)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1110	0	1	0	0	1	1	Effective Address					

X N Z V C
* * * 0 *

RTD
(68010)

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	1	0	0

X N Z V C
\- - - - -

RTE

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	0	1	1

X N Z V C
I I I I I

Instruction **Machine Code Format** **Condition Codes**

RTR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	1	1	1

X	N	Z	V	C
I	I	I	I	I

RTS

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	1	0	1

X	N	Z	V	C
-	-	-	-	-

SBCD

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1000	Dest Reg			1	0	0	0	0	DA	Src Reg		

X	N	Z	V	C
*	U	*	U	*

Scc

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0101	Condition				1	1	Effective			Address		

X	N	Z	V	C
-	-	-	-	-

STOP

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0100	1	1	1	0	0	1	1	1	0	0	1	0

X	N	Z	V	C
I	I	I	I	I

SUB

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1001	D Reg.			Dr	Size		Effective			Address		

X	N	Z	V	C
*	*	*	*	*

SUBA

15–12	11	10	9	8	7	6	5	4	3	2	1	0
1001	A Reg.			Sz	1	1	Effective			Address		

X	N	Z	V	C
-	-	-	-	-

Instruction	*Machine Code Format*													*Condition Codes*				
	15–12	11	10	9	8	7	6	5	4	3	2	1	0	X	N	Z	V	C
SUBI	0000	0	1	0	0	Size		Effective Address						*	*	*	*	*
SUBQ	0101	Data			1	Size		Effective Address						*	*	*	*	*
SUBX	1001	Dest Reg			1	Size		0	0	DA	Src Reg			*	*	*	*	*
SWAP	0100	1	0	0	0	0	1	0	0	0	D Reg.			-	*	*	0	0
TAS	0100	1	0	1	0	1	1	Effective Address						-	*	*	0	0
TRAP	0100	1	1	1	0	0	1	0	0	Vector				-	-	-	-	-
TRAPV	0100	1	1	1	0	0	1	1	1	0	1	1	0	-	-	-	-	-

Instruction	Machine Code Format	Condition Codes

TST

15–12	11	10	9	8	7 6	5 4 3 2 1 0	X N Z V C
0100	1	0	1	0	Size	Effective Address	- * * 0 0

UNLK

15–12	11	10	9	8	7	6	5	4	3	2 1 0	X N Z V C
0100	1	1	1	0	0	1	0	1	1	A Reg.	- - - - -

EXCEPTION VECTORS

Vector	Address	Use
0	0	RESET Initial SSP
1	4	RESET Initial PC
2	8	BUSERR (Non-existent Memory)
3	C	Address (Boundary) Error
4	10	Illegal Instruction
5	14	Zero Divide
6	18	CHK Instruction
7	1C	TRAPV Instruction
8	20	Privilege Violation
9	24	TRACE
10	28	Line 1010 Emulator
11	2C	Line 1111 Emulator
12–13	30–34	Unassigned (Reserved)
14	38	68010 Stack Format Error
15	3C	Uninitialized Interrupt Vector
16–23	40–5C	Unassigned (Reserved)
24	60	Spurious Interrupt
25–31	64–7C	Level 0–7 Autovector Interrupts
32–47	80–BF	TRAP 0–15 Instruction Vectors
48–63	C0–FC	Unassigned (Reserved)
64–255	100–3FF	User Interrupt Vectors

BUSERR / ADDRESSING ERROR STACK FRAME (68000/68008)

Bits	15 … 5	4	3	2 1 0
A7	//////////////////////	R/W	I/N	Access
+2	High word of erroneous address			
+4	Low word of erroneous address			
+6	First word of instruction			
+8	Status register			
+10	Program counter high word			
+12	Program counter low word			

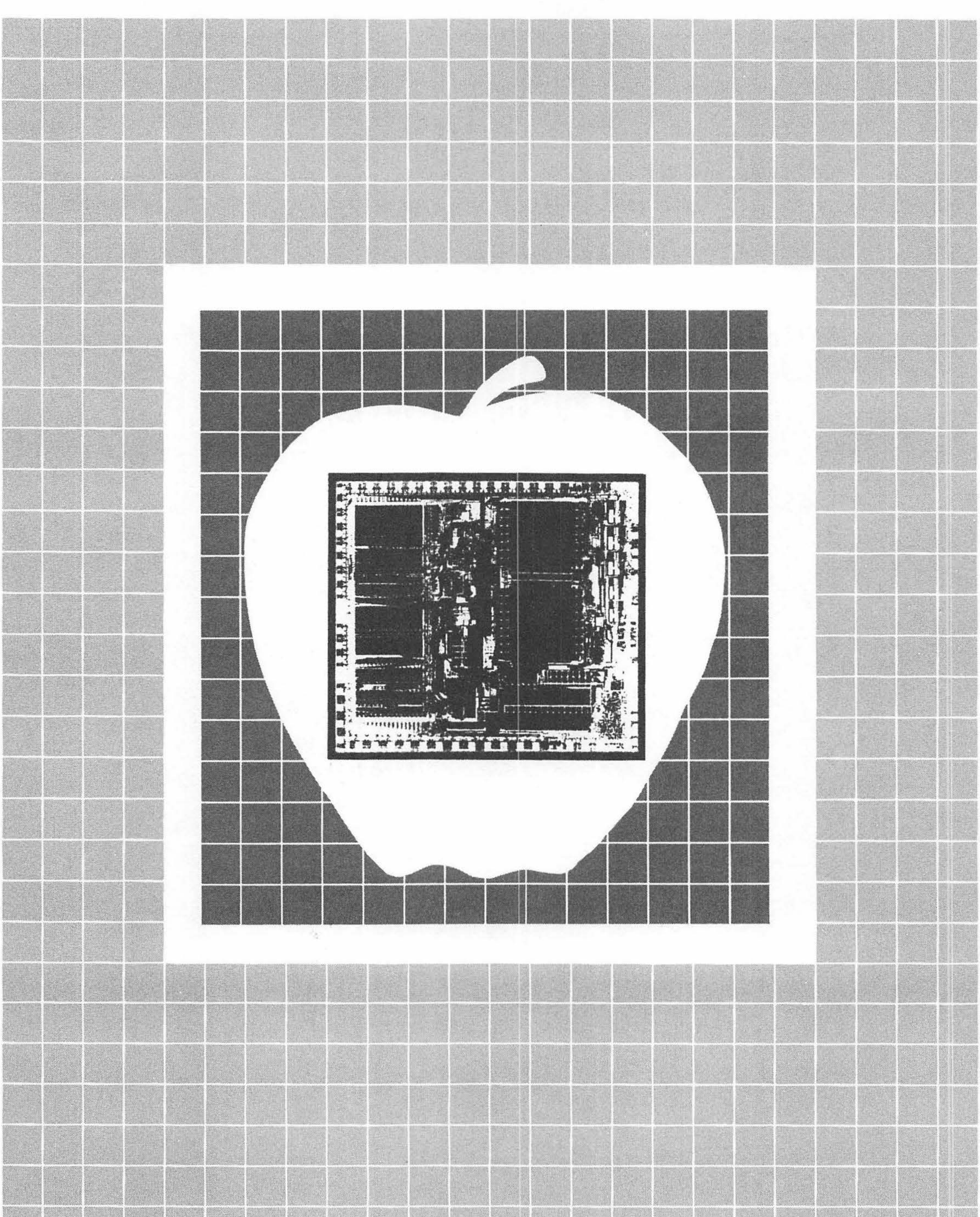
BUSERR / ADDRESSING ERROR STACK FRAME (68010)

Bit	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A7 →	Status Register Before Exception
	Program Counter High Word
	Program Counter Low Word
	1 0 0 0 Vector Offset
	Special Status Word
	Fault Address High Word
	Fault Address Low Word
	Reserved
	Data Output Buffer
	Reserved
	Data Input Buffer
	Reserved
	Instruction Input Buffer
	Internal Information (16 words)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	RR	**	IF	DF	RM	HB	BY	RW	**	**	**	**	**	Function		

SPECIAL STATUS WORD FORMAT

RR	ReRun Bit Set by software for no hardware rerun
IF	Instruction fetch
DF	Data fetch
RM	Read-modify-write cycle
HB	Processor accessing high byte
BY	Byte access
RW	Read / Write bit
Function	Function codes FC0–FC2



Appendix **E**

Source Listings

Bloody instructions, which, being taught, return to plague the inventor.

—Shakespeare (Macbeth)

INTRODUCTION

Many of the programs presented in Chapters 4–6 and 8 are too long to explain completely; to facilitate the presentation, we left out many lines of code from the listings in these chapters. In addition, many exercises call for completion or modification of these programs. This appendix presents the complete source listings of these programs.

To run the programs in Chapter 4 and 6, you will need to build the run-time library. See the section on Chapter 8 in this Appendix for instructions.

To obtain a disk containing the source files for all the programs in this book, send \$19.95 plus \$2.50 postage and handling in check or money order to this address:

MacDoodle Source Disk
P.O. Box 3262
Monterey, CA 93940

California residents add 6 percent sales tax. Make the check payable to *MacDoodle Source Disk.* Please allow 6–8 weeks for delivery.

CHAPTER 4

The Dump program in Chapter 4 consists of three files: Dump.asm, Dump.job, and Dump.link. These are contained in Listings E.1–E.3.

```
;
;       Dump.asm. This program dumps an arbitrary file to an ASCII textfile.
;
;       Version 01.00   18-Feb-85
;
;       This program prompts for a file to be dumped, changes the extension to
;       .dmp, and prints the contents of the file in hex and ASCII in the
;       output file.
;
;       The dump can be repeated as desired.
;
include MACLIB.ASM
;
BufSize EQU     16                      ;       16 bytes per dump line
Space   EQU     $20                     ;       ASCII value for a blank
;
;
;
        PROGRAM DUMP                    ;       Setup program entry
Start:  BSR     OpenFiles               ;       Start with opening input/output
        PRINT   'Dumping ',*            ;       Echo file names
        PRINT   INameBuf(A5),*          ;       to main window
        PRINT   ' To ',*                ;
        PRINT   ONamebuf(A5)            ;
        WRITE   ORefNum(A5),InfoStr,InfLen;     Write "File Information" on file
        BSR     DoFInfo                 ;       Dump file information first
        TST.L   D0                      ;       Successful?
        BNE     DQuit                   ;       No, quit
        WRITE   ORefNum(A5),DataStr,DatLen;     Write "Data Fork" on file
        BSR     DoFork                  ;       Do the data fork
        TST.L   D0                      ;       Successful?
        BNE     DQuit                   ;       No, quit
        CLOSE   IRefNum(A5)             ;       Close the file
        OPENR   INameBuf(A5),IFBuff(A5) ;       Open resource fork now
        MOVE.W  D0,IRefNum(A5)          ;       Save file number
        BGT     ROpenOK                 ;       If GT, then OK
        PRINT   'Unable to Open Resource Fork: ',*
        PRINT   INameBuf(A5)            ;       Identify file name
        CLOSE   ORefNum(A5)             ;       Close output file
        BRA     Start                   ;       To next file
ROpenOK:                                ;       Here if resource open OK
        WRITE   ORefNum(A5),ResStr,ResLen;      Write resource ID
        BSR     DoFork                  ;       Do the dump
DQuit:  CLOSE   IRefNum(A5)             ;       Close the file
        CLOSE   ORefNum(A5)             ;       and output file
        BRA     Start                   ;       Go get next file name
;
;       OpenFiles routine.  Here to open input file and output file using
;       POPEN and PCREATE macros.
;
;       Entry conditions:
;
```

***Listing E.1** – The Dump.asm file*

```
;       <none>
;
;       Exit conditions:
;
;               IRefNum = Input file number
;               ORefNum = Output file number
;
OpenFiles:
        MOVEM.L D0/A0-A1,-(SP)          ;       Save registers
        POPEN   DEFAULT,DEFAULT,INameBuf(A5),IFbuff(A5); Display all files for user
        MOVE.W  D0,IRefNum(A5)          ;       Save file number
        BLT     DoExit                  ;       If > 0, ok
        LEA     INameBuf(A5),A0         ;       A0 -> File name specified by user
        LEA     DefName+1(A5),A1        ;       A1 -> Place to put prompt name
        CLR.L   D0                      ;       Load count
        MOVE.B  (A0)+,D0                ;       D0 = # bytes in file name
        SUBQ.L  #1,D0                   ;       Adjust for DBRA
FindDot:                                ;       Label for DBRA loop
        MOVE.B  (A0),(A1)+              ;       Move a byte
        CMPI.B  #'.',(A0)+              ;       Was it a period?
        BEQ     FoundDot                ;       Yes, break out
        DBRA    D0,FindDot              ;       Loop till run out of name
        MOVE.B  #'.',(A1)+              ;       Move in a period
FoundDot:                               ;       Here when dot found
        MOVE.L  #2,D0                   ;       DBRA-adjusted count
        LEA     'DMP'+1,A0              ;       A0 -> Extension
MoveExt:                                ;       Loop label
        MOVE.B  (A0)+,(A1)+             ;       Move a byte
        DBRA    D0,MoveExt              ;       Loop on count
        LEA     DefName(A5),A0          ;       Find address
        SUBA.L  A0,A1                   ;       Subtract to find length
        MOVE.W  A1,D0                   ;       Move in
        SUB.L   #1,D0                   ;               corrected
        MOVE.B  D0,(A0)                 ;                       count byte
        PCREATE OPrompt,DefName(A5),ONamebuf(A5),OFBuff(A5);    Create dump file
        MOVE.W  D0,ORefNum(A5)          ;       Save output file number
        BGT     CreatOK                 ;       If OK, continue
        CLOSE   IRefNum(A5)             ;       Close input file
        BRA     OpenFiles               ;       Go try next file
CreatOK MOVEM.L (SP)+,D0/A0-A1          ;       Restore registers
        RTS                             ;       Back to main routine
DoExit: EXIT                            ;       Quit routine
;
;       DoFInfo routine.  Dumps out the file information block.
;
;       Entry conditions:
;
;       (none)
;
;       Exit conditions:
;
;       (none)
;
```

***Listing E.1** – The Dump.asm file (continued)*

```
;       Equates for file info:
DirIndex        EQU     28          ;       Directory index
Attrib          EQU     30          ;       File attributes
Version         EQU     31          ;       File version number
FiFType         EQU     32          ;       Finder file type
FiCreate        EQU     36          ;       Finder file creator
FiFlags         EQU     40          ;       Finder flags word
FiIcon          EQU     42          ;       Finder icon location
FiFolder        EQU     46          ;       Finder folder number
FileNum         EQU     48          ;       File number
DataStart       EQU     52          ;       Data fork start block
DataLEOF        EQU     54          ;       Data fork logical EOF
DataPEOF        EQU     58          ;       Data fork physical EOF
RsrcStart       EQU     62          ;       Resource fork start block
RsrcLEOF        EQU     64          ;       Resource fork logical EOF
RsrcPEOF        EQU     68          ;       Resource fork physical EOF
CreateDate      EQU     72          ;       Creation date
ModifyDate      EQU     76          ;       Modification date
DoFInfo MOVEM.L D1-D7/A0-A4,-(SP)   ;       Save caller's registers
        GETFINFO INameBuf(A5),InfoBuf(A5);  Get file info for input file
        LEA     InfoBuf(A5),A3      ;       A3 -> File information
        MOVE.W  DirIndex(A3),D0     ;       D0 = Directory index
        LEA     Diridx,A0           ;       A0 -> Output area
        BSR     HexW                ;       Convert
        MOVE.B  Attrib(A3),D0       ;       D0 = File attribute byte
        LEA     Attr,A0             ;       A0 -> Output area
        BSR     HexB                ;       Convert
        MOVE.B  Version(A3),D0      ;       D0 = File version number
        LEA     Vers,A0             ;       A0 -> Output area
        BSR     HexB                ;       Convert
        MOVE.L  FiFType(A3),D0      ;       D0 = Finder file type
        LEA     FFType,A0           ;       A0 -> Output area
        BSR     HexL                ;       Convert
        LEA     FiFType(A3),A1      ;       A1 -> Binary file type
        LEA     FFTAsc,A0           ;       A0 -> Output area
        MOVEQ.L #4,D0               ;       D0 = Byte count
        BSR     DumpAsc             ;       Convert
        MOVE.L  FiCreate(A3),D0     ;       D0 = Finder file creator
        LEA     FFCreate,A0         ;       A0 -> Output area
        BSR     HexL                ;       Convert
        LEA     FiCreate(A3),A1     ;       A1 -> Creator binary
        LEA     FFCAsc,A0           ;       A0 -> Output area
        MOVEQ.L #4,D0               ;       D0 = Byte count
        BSR     DumpAsc             ;       Convert
        MOVE.W  FiFlags(A3),D0      ;       D0 = Finder flags word
        LEA     FFlags,A0           ;       A0 -> Output area
        BSR     HexW                ;       Convert
        MOVE.L  FiIcon(A3),D0       ;       D0 = Finder icon location
        LEA     FFIloc,A0           ;       A0 -> Output area
        BSR     HexL                ;       Convert
        MOVE.W  FiFolder(A3),D0     ;       D0 = Finder folder number
        LEA     FFNum,A0            ;       A0 -> Output Area
        BSR     HexW                ;       Convert
```

***Listing E.1** – The Dump.asm file (continued)*

```
         MOVE.L  FileNum(A3),D0          ;       D0 = File number
         LEA     OFileNum,A0             ;       A0 -> Output area
         BSR     HexL                    ;       Convert
         MOVE.W  DataStart(A3),D0        ;       D0 = Data fork start block
         LEA     DFStart,A0              ;       A0 -> Output area
         BSR     HexW                    ;       Convert
         MOVE.L  DataLEOF(A3),D0         ;       D0 = Data fork logical EOF
         LEA     DFLEOF,A0               ;       A0 -> Output area
         BSR     HexL                    ;       Convert
         MOVE.L  DataPEOF(A3),D0         ;       D0 = Data fork physical EOF
         LEA     DFPEOF,A0               ;       A0 -> Output area
         BSR     HexL                    ;       Convert
         MOVE.W  RsrcStart(A3),D0        ;       D0 = Resource fork start block
         LEA     RFStart,A0              ;       A0 -> Output area
         BSR     HexW                    ;       Convert
         MOVE.L  RsrcLEOF(A3),D0         ;       D0 = Resource fork logical EOF
         LEA     RFLEOF,A0               ;       A0 -> Output area
         BSR     HexL                    ;       Convert
         MOVE.L  RsrcPEOF(A3),D0         ;       D0 = Resource fork physical EOF
         LEA     RFPEOF,A0               ;       A0 -> Output area
         BSR     HexL                    ;       Convert
         MOVE.L  CreateDate(A3),D0       ;       D0 = Creation date
         LEA     Creation,A0             ;       A0 -> Output area
         BSR     Date                    ;       Convert
         MOVE.L  ModifyDate(A3),D0       ;       D0 = Modification date
         LEA     Modify,A0               ;       A0 -> Output area
         BSR     Date                    ;       Convert
         WRITE   ORefNum(A5),FInfoStr,FinfoLen;  Write converted data to file
         CMP.L   FinfoLen,D0             ;       Check for write error
         BNE     WError                  ;       Had an error, indicate it
         CLR.L   D0                      ;       Clear return register
FIRet:   MOVEM.L (SP)+,D1-D7/A0-A4       ;       Unstack registers
         RTS                             ;       Back to caller
WError:  PRINT   'Write Error on File:',* ;      Print error message
         PRINT   ONameBuf(A5)            ;       Identify file by name
         MOVEQ.L #-1,D0                  ;       Return error code
         BRA     FIRet                   ;       Do return
;
;        DoFork routine.  This routine outputs the contents of an open file
;        in hex to another open file.  Called once for data fork and once
;        for resource fork.
;
;        Entry conditions:
;
;                IRefNum = Input file number
;                ORefNum = Output file number
;
;        Exit conditions:
;
;                <none>
;
DoFork:  MOVEM.L D1-D7/A0-A4,-(SP)       ;       Save registers
         CLR.L   D7                      ;       File address pointer
```

Listing E.1 – *The Dump.asm file (continued)*

```
READXX:                                  ;
        LEA     Buffer(A5),A3            ;       A3 -> Buffer area
        MOVE.W  #BufSize-1,D0            ;       DBRA-adjusted count
Clear:  CLR.B   (A3)+                    ;       Clear a byte
        DBRA    D0,Clear                 ;       Until done
        CHKABORT                         ;       Check for Command-Period
        TST.W   D0                       ;       D0 <> 0 if aborted
        BNE     Next                     ;       Yes, quit
        READ    IRefNum(A5),BUFFER(A5),#BufSize
        MOVE.L  D0,BytesRead(A5)         ;       Remember byte count
        BLE     AtEOF                    ;       None, return to caller
        LEA     DumpLine,A0              ;       A0 -> ASCII area
        MOVE.L  D7,D0                    ;       Address to D0
        JSR     HexL                     ;       Convert to ASCII
        ADD.L   #10,A0                   ;       Skip number, and ": "
        LEA     Buffer(A5),A1            ;       A1 -> Input data
        MOVE.W  #3,D2                    ;       DBRA-adjusted count
Hloop:                                   ;       Conversion loop
        MOVE.L  (A1)+,D0                 ;       Fetch next data longword
        BSR     HexL                     ;       Convert to ASCII
        ADD.L   #9,A0                    ;       Bump past number and space
        DBRA    D2,Hloop                 ;       Repeat as needed
        LEA     Buffer(A5),A1            ;       A1 -> Input data again
        MOVE.W  #BufSize,D0              ;       D0 = # bytes in buffer
        BSR     DumpAsc                  ;       Convert to dump ASCII
        WRITE   ORefNum(A5),DumpLine,Olen;       Write to output file
        CMP.L   Olen,D0                  ;       Was write successful?
        BNE     Werror                   ;       No, quit
        ADD.L   #BufSize,D7              ;       Bump file address
        CMP.L   #BufSize,BytesRead(A5)   ;       Did last read go OK?
        BEQ     READXX                   ;       Yes, back for more
AtEOF:  CLR.L   D0                       ;       Indicate no error
Next:   MOVEM.L (SP)+,D1-D7/A0-A4        ;       Unsave registers
        RTS                              ;       Return to caller
;
;       Hex output conversion routines.
;       HexL,HexW, and HexB convert longword, word, and byte to hex.
;
;       Enter with:
;
;       D0 = Binary number
;       A0 -> Output area
;
;       Exit with:
;
;       (no change to registers)
;
HexL:   MOVE.L  A0,-(SP)                 ;       Convert longword to ASCII
        SWAP    D0                       ;       Do high word
        BSR     HexW                     ;       Convert high word
        LEA     4(A0),A0                 ;       Point to next area
        SWAP    D0                       ;       Restore D0
        BSR     HexW                     ;       Do low word
```

Listing E.1 – The Dump.asm file (continued)

```
        MOVE.L  (SP)+,A0            ;       Restore A0
        RTS                         ;       and return
HexW:                               ;       Convert word to ASCII
        MOVE.L  A0,-(SP)            ;       Save A0
        ROL.W   #8,D0               ;       Do high byte first
        BSR     HexB                ;       Do 1 byte
        LEA     2(A0),A0            ;       A0 -> Next area
        ROL.W   #8,D0               ;       Do low byte
        BSR     HexB                ;       Convert low byte
        MOVE.L  (SP)+,A0            ;       Restore A0
        RTS                         ;       and return
HexB:                               ;       Convert a single byte
        MOVEM.L D1/A1,-(SP)         ;       Save D1, A1
        LEA     HexTab,A1           ;       A1 -> 16 hex digits
        MOVE.B  D0,D1               ;       Copy D0.B
        ANDI.W  #15,D1              ;       0 <= D1.W <= 15
        MOVE.B  0(A1,D1.W),1(A0)    ;       Do low nibble
        MOVE.B  D0,D1               ;       Recopy
        LSR.W   #4,D1               ;       Shift right 1 nibble
        MOVE.B  0(A1,D1.W),(A0)     ;       Store high nibble
        MOVEM.L (SP)+,D1/A1         ;       Restore registers
        RTS                         ;       and return
        STRING_FORMAT   0           ;       No count bytes
HexTab: DC.B    '0123456789ABCDEF'  ;       Hex conversion table
;
;       Dump ASCII conversion routine.  Converts binary to printable ASCII.
;       Unprintable binary characters are converted to periods.
;
;       Enter with:
;
;       A0 -> Output area
;       A1 -> Input data
;       D0 =  Byte count
;
;       Exit with:
;
;       (No change to registers)
;
DumpAsc MOVEM.L D0/A0-A1,-(SP)      ;       Save registers
        SUBQ.W  #1,D0               ;       Adjust for DBRA
DALoop: CMPI.B  #Space,(A1)+        ;       Printable?
        BGE     NoDot               ;       Yes, use character
        MOVE.B  #'.',(A0)+          ;       No, use '.'
        BRA     DADBRA              ;       Do loop end stuff
NoDot:  MOVE.B  -1(A1),(A0)+        ;       Move in actual character
DADBRA: DBRA    D0,DALoop           ;       Repeat until done
        MOVEM.L (SP)+,D0/A0-A1      ;       Restore registers
        RTS                         ;       and return
;
;       Macintosh date conversion routine.  Converts binary date
;       to date/time:  mm/dd/yy hh:mm:ss
;
;       Enter with:
```

***Listing E.1** – The Dump.asm file (continued)*

```
;
;       D0.L = Binary date
;       A0  -> Output area
;
;       Exit with:
;
;       (No change to registers)
;
;       Doesn't work for year 2100 (wrongly considers 2100 a leap year).
;
SecsMin EQU     60                      ;       Seconds/minute
SecsHr  EQU     3600                    ;       Seconds/hour
SecsDay EQU     86400                   ;       Seconds/day
SecsM28 EQU     2419200                 ;       Seconds/28-day month
SecsM29 EQU     2505600                 ;       Seconds/29-day month
SecsM30 EQU     2592000                 ;       Seconds/30-day month
SecsM31 EQU     2678400                 ;       Seconds/31-day month
SecsYr  EQU     31536000                ;       Seconds/(nonleap) year
SecsLYr EQU     31622400                ;       Seconds/leap year
Date:   MOVEM.L D0-D6/A0-A1,-(SP)       ;       Save registers
        MOVE.L  #4,D1                   ;       D1 to be year
LYearLoop:                              ;       Here to do years
        CMP.L   #SecsLYr,D0             ;       (Starts with 1904, a leap year)
        BLO     YearFound               ;       Year is now correct
        SUB.L   #SecsLYr,D0             ;       Subtract off a year
        ADDQ.L  #1,D1                   ;       Bump year counter
        MOVE.W  #2,D3                   ;       Count 3 more nonleap years
YearLoop:                               ;       Loop of nonleap years
        CMP.L   #SecsYr,D0              ;       Another year left?
        BLO     YearFound               ;       No, do the rest
        SUB.L   #SecsYr,D0              ;       Subtract a year
        ADDQ.L  #1,D1                   ;       Bump year counter
        DBRA    D3,YearLoop             ;       Repeat for nonleap years
        BRA     LYearLoop               ;       Repeat till expired
YearFound:                              ;       Here when year found
        LEA     MonthTab,A1             ;       A1 -> Table of secs/month
        MOVE.W  D1,D3                   ;       Copy year to D3
        ANDI.W  #3,D3                   ;       Check for divisible by 4
        BNE     NotLeap                 ;       Not a leap year
        LEA     LMonthTab,A1            ;       A1 -> Secs/month for leap year
NotLeap MOVEQ.L #1,D2                   ;       D2 to be month
MonthLoop:                              ;       Now calculate month number
        CMP.L   (A1),D0                 ;       Enough seconds left?
        BLT     MonthFound              ;       No, month correct
        SUB.L   (A1)+,D0                ;       Subtract off month
        ADDQ.L  #1,D2                   ;       Bump counter
        BRA     MonthLoop               ;       Back for more
MonthFound:                             ;
        MOVEQ.L #1,D3                   ;       D3 will accumulate days
DayLoop:                                ;
        SUB.L   #SecsDay,D0             ;       Subtract # seconds/day
        BLT     DayFound                ;       Found day, quit
        ADDQ.L  #1,D3                   ;       Increment D3
```

***Listing E.1** – The Dump.asm file (continued)*

```
         BRA     DayLoop              ;       Loop till found
DayFound:                             ;       Here when day found
         ADD.L   #SecsDay,D0          ;       Fix what we broke
         DIVS    #SecsHr,D0           ;       Can use 16-bit divide now
         MOVE.W  D0,D4                ;       D4 = hours
         SWAP    D0                   ;       Remainder back to D0
         EXT.L   D0                   ;       Clear high word
         DIVS    #SecsMin,D0          ;       Divide to get mins & secs
         MOVE.W  D0,D5                ;       D5 = Minutes
         SWAP    D0                   ;       D0.W = Seconds
         MOVE.W  D0,D6                ;       Preserve in D6
;
;        We have computed all fields in binary.  Now convert them to ASCII.
;
         MOVE.W  D2,D0                ;       Month to D0
         BSR     TwoDig               ;       Convert
         MOVE.B  #'/',(A0)+           ;       Drop in separator
         MOVE.W  D3,D0                ;       Days to D0
         BSR     TwoDig               ;       Convert
         MOVE.B  #'/',(A0)+           ;       Separator
         MOVE.W  D1,D0                ;       Year to D0
         BSR     TwoDig               ;       Convert
         MOVE.B  #' ',(A0)+           ;       Separate date from time
         MOVE.W  D4,D0                ;       Hours to D0
         BSR     TwoDig               ;       Convert
         MOVE.B  #':',(A0)+           ;       Separator
         MOVE.W  D5,D0                ;       Minutes to D0
         BSR     TwoDig               ;       Convert
         MOVE.B  #':',(A0)+           ;       Separator
         MOVE.W  D6,D0                ;       Seconds to D0
         BSR     TwoDig               ;       Convert
         MOVEM.L (SP)+,D0-D6/A0-A1    ;       Restore registers
         RTS                          ;       Return
;
;        "Thirty days hath September, April, June, and November."
;
;                January Feb.    March   April   May     June
;                July    August  Septem. October Novem.  December
Monthtab DC.L    SecsM31,SecsM28,SecsM31,SecsM30,SecsM31,SecsM30
         DC.L    SecsM31,SecsM31,SecsM30,SecsM31,SecsM30,SecsM31
LMonthtab DC.L   SecsM31,SecsM29,SecsM31,SecsM30,SecsM31,SecsM30
         DC.L    SecsM31,SecsM31,SecsM30,SecsM31,SecsM30,SecsM31
;
;        TwoDig routine.  Converts byte in D0 to two ASCII digits.
;
;        Enter with:
;
;        D0 =  Binary
;        A0 -> Output area
;
;        Exit with:
;
;        A0 -> Next byte beyond output area
```

***Listing E.1** – The Dump.asm file (continued)*

```
;
TwoDig: MOVE.L  D0,-(SP)                ;       Save D0
        EXT.L   D0                      ;       Clear high word
        DIVS    #10,D0                  ;       Divide by 10
        ADD.B   #'0',D0                 ;       Convert to ASCII
        MOVE.B  D0,(A0)+                ;       Place in buffer
        SWAP    D0                      ;       Remainder to low word
        ADD.B   #'0',D0                 ;       Convert to ASCII
        MOVE.B  D0,(A0)+                ;       Place in buffer
        MOVE.L  (SP)+,D0                ;       Restore D0
        RTS                             ;       Quit
;
;
;       Dump program data area.
;
;       Strings that are written to the .DMP file:
;
        STRING_FORMAT 0
InfoStr:  DC.B  13,'File Information:',13
InfEnd:
ResStr:   DC.B  13,'Resource Fork:',13
ResEnd:
DataStr:  DC.B  13,'Data Fork:    ',13
DataEnd:
DumpLine: DC.B  'xxxxxxxx: XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX ................',13
EndDump:
;
;       ASCII output area for file information
;
FInfoStr:
          DC.B  'Directory Index:      '
Diridx:   DC.B  'hhhh',13,'Attributes:          '
Attr:     DC.B  'hh',13,'Version:             '
Vers:     DC.B  'hh',13,'Finder File Type:    '
FFType:   DC.B  'hhhhhhhh '
FFTAsc:   DC.B  '....',13,'Finder File Creator: '
FFCreate: DC.B  'hhhhhhhh '
FFCAsc:   DC.B  '....',13,'Finder Flags:        '
FFlags:   DC.B  'hhhh',13,'Finder Icon Location: '
FFILoc:   DC.B  'hhhhhhhh',13,'Finder Folder Number: '
FFNum:    DC.B  'hhhh',13,'File Number:          '
OFileNum: DC.B  'hhhhhhhh',13,'Data     Fork Start:  '
DFStart:  DC.B  'hhhh Logical EOF: '
DFLEOF:   DC.B  'hhhhhhhh Physical EOF: '
DFPEOF:   DC.B  'hhhhhhhh',13,'Resource Fork Start:  '
RFStart:  DC.B  'hhhh Logical EOF: '
RFLEOF:   DC.B  'hhhhhhhh Physical EOF: '
RFPEOF:   DC.B  'hhhhhhhh',13,'Creation Date:        '
Creation: DC.B  'xx/xx/xx hh:mm:ss',13,'Modification Date:    '
Modify:   DC.B  'xx/xx/xx hh:mm:ss',13
EndFinfo:
;
;       Pascal strings
```

***Listing E.1** – The Dump.asm file (continued)*

```
;
            STRING_FORMAT 3              ;       Inform assembler
OPrompt:    DC.B  'Output File'
            .ALIGN 2
;
;       Other initialized data
;
InfLen:     DC.L  InfEnd-InfoStr         ;       Length of file info ID
ResLen:     DC.L  ResEnd-ResStr          ;       Length of resource fork ID
DatLen:     DC.L  DataEnd-DataStr        ;       Length of data fork ID
Olen:       DC.L  EndDump-DumpLine       ;       Length of output line
FinfoLen:   DC.L  EndFinfo-FinfoStr      ;       Length of file info output
;
;       Uninitialized data
;
Buffer:     DS.W  8                      ;       Data buffer for input file
BytesRead   DS.L  1                      ;       Number of bytes read
ORefNum:    DS.W  1                      ;       Output file number
IRefNum:    DS.W  1                      ;       Input file number
IFbuff:     DS.B  522                    ;       Input file buffer
OFbuff:     DS.B  522                    ;       Output file buffer
InfoBuf:    DS.B  80                     ;       File information buffer
INameBuf:   DS.B  64                     ;       Input file name
ONameBuf:   DS.B  64                     ;       Output file name
DefName:    DS.B  64                     ;       Constructed default name
            ENDPROGRAM
```

***Listing E.1** – The Dump.asm file (continued)*

```
Asm     Dump.Asm        Exec    Edit
Asm     RTL.Asm Exec    Edit
Link    Dump.Link       Exec    Edit
```

***Listing E.2** – The Dump.job file*

```

Dump
RTL
!XXSTART
$
```

***Listing E.3** – The Dump.link file*

CHAPTER 5

The Lisa Pascal sieve program requires several files as part of its build procedure. The complete set of files to build the optimized program appear in Listings E.4–E.7.

```
{$M+}
{$X-}
{$R-}
{$D-}
(****************************************************************************)
(*                                                                          *)
(*   Lisa Pascal version of the sieve of Eratosthenes (optimized)           *)
(*                                                                          *)
(****************************************************************************)
PROGRAM   xsieve;                                   (*                     *)
USES {$U-}                                          (* Standard definitions *)
     {$U OBJ/MemTypes}  MemTypes,                   (* Memory type defs    *)
     {$U OBJ/QuickDraw} QuickDraw,                  (* QuickDraw stuff     *)
     {$U OBJ/OSIntf}    OSIntf,                     (* O/S interface       *)
     {$U OBJ/ToolIntf}  ToolIntf;                   (* Toolbox interface   *)
CONST                                               (* Constant section    *)
     ASIZE = 8190;                                  (* Array size          *)
     ITER = 10;                                     (* Number of iterations *)
     TIMELOC = 362;                                 (* 0x16A in decimal    *)
TYPE                                                (* Type section        *)
     flagtype = array[0..ASIZE] of char;            (* Flag array type     *)
VAR                                                 (*---------------------*)
     i,j,k,count,prime : integer;                   (* 16-bit temporaries  *)
     flags             : flagtype;                  (* Flags array         *)
     start,stop        : longint;                   (* Time variables      *)
     timeptr           : ^longint;                  (* Points to timer long *)
                                                    (*---------------------*)
PROCEDURE setflags(var flag  : flagtype;            (* Declare             *)
                   xcount: integer); external;      (*        setflags proc*)
FUNCTION  sieve    (var flag  : flagtype;           (* Declare             *)
                   xcount: integer) : integer;      (*        sieve funct. *)
                   external;                        (*---------------------*)
BEGIN                                               (* Main program        *)
     InitGraf(@thePort);                            (* Initialize QuickDraw *)
     InitFonts;                                     (* Initialize font mgr *)
     InitWindows;                                   (* Initialize window mgr *)
     FlushEvents(everyEvent,0);                     (* Clear event queue   *)
     InitMenus;                                     (* Initialize menu mgr *)
     TEInit;                                        (* Initialize Text Edit *)
     InitDialogs(NIL);                              (* Initialize dialogs  *)
     InitCursor;                                    (* Set cursor to arrow *)
                                                    (*---------------------*)
     timeptr := pointer(TIMELOC);                   (* -> Tick word        *)
     start   := timeptr^;                           (* Fetch starting value *)
     FOR j := 1 TO ITER DO                          (* Iteration loop      *)
     BEGIN                                          (*                     *)
       setflags(flags,ASIZE);                       (* Set all flag bits   *)
```

Listing E.4 *– Pascal source file for optimized sieve program*

```
        count := sieve(flags,ASIZE);                  (* Do the sieve algorithm*)
      END;                                            (* FOR J := 1 TO ITER    *)
      stop := timeptr^;                               (* Fetch stop time       *)
      writeln(count,' primes ',stop-start,' ticks');(* Output time           *)
 END.                                                 (* Main program          *)
                                                      (************************)
```

Listing E.4 *– Pascal source file for optimized sieve program (continued)*

```
;        Assembler part of optimized PASCAL sieve program
;
          .PROC   SieveAsm       ;       Procedure statement
          .DEF    setflags       ;       Declare global
setflags  MOVE.L (SP)+,A1        ;       A1 = Return address
          MOVE.W (SP)+,D1        ;       D1 = array size
          SUB.W   #1,D1          ;       DBRA-adjust count
          MOVE.L (SP)+,A0        ;       A0 -> Array
@1:       MOVE.B #1,(A0)+        ;       Set flag byte
          DBRA    D1,@1          ;       Loop till done
          JMP     (A1)           ;       Return
;
;        Sieve routine.  Does the main part of the sieve algorithm.
;
          .DEF    sieve          ;       Make accessible
sieve     MOVE.L (SP)+,A1        ;       A1 =  Return address
          MOVE.W (SP)+,D1        ;       D1 =  Array size
          MOVE.L (SP)+,A0        ;       A0 -> Flags array
          MOVEM.W D3-D6,-(SP)    ;       Save registers
          MOVE.W D1,D4           ;       Replicate count
          SUBQ.W #1,D1           ;       DBRA-adjust D1
          CLR.W   D2             ;       D2 =  Count
          CLR.W   D3             ;       D3 =  i
@1:       TST.B   0(A0,D3.W)     ;       If(flags[i])
          BEQ     @4             ;       Not prime, skip this one
          MOVE.W D3,D5           ;       D5 = "prime" variable
          ADD.W   D3,D5          ;       prime = i + i
          ADDQ.W #3,D5           ;                    + 3;
          MOVE.W D3,D6           ;       D6 = "k"
@2:       ADD.W   D5,D6          ;       Add in "prime"
          CMP.W   D4,D6          ;       While (k <= SIZE)
          BGT     @3             ;       Not <=, do next iteration
          CLR.B   0(A0,D6.W)     ;       Flags[k] = 0;
          BRA     @2             ;       Loop again
@3:       ADDQ.W #1,D2           ;       Increment count variable
@4:       ADDQ.W #1,D3           ;       Increment i
          DBRA    D1,@1          ;       Loop till done
          MOVEM.W (SP)+,D3-D6    ;       Restore registers
          MOVE.W D2,(SP)         ;       Set return value
          JMP     (A1)           ;       Return
          .END                   ;       Mandatory END statement
```

Listing E.5 *– Assembler part of optimized sieve program*

```
$EXEC
P{ascal}$M+
xsieve.pas{Source File}
xsieve.lst{Listing File}
xpsieve{Object File}
A{ssemble}xsieve.asm{Source file}
xsieve.asm.lst{Listing File}
xasieve{Object File}
L{ink}?
+X
{No more options}
xpsieve{Pascal Input file}
xasieve{Assembly Input file}
obj/quickdraw
obj/ostraps
obj/tooltraps
obj/pasinit
obj/paslib
obj/paslibasm
obj/rtlib
{no more input}
{list to console}
xsieve{output filename}
R{un}RMaker
xsieve.r{Input file}
R{un}Maccom
F{inder information}Y{es}L{isa to Macintosh}xpsieve.rsrc
xpsieve{Macintosh Filename}
APPL{Application Type}
JUNK{Application Signature}
{No bundle bit}
E{ject}Q{uit}
$ENDEXEC
```

***Listing E.6** – EXEC file for optimized sieve program*

```
xpsieve.rsrc

TYPE JUNK = STR
   ,0
Sievetest V 1.0

Type Code
  xsieve.obj,0
```

***Listing E.7** – Lisa RMAKER input file for sieve program*

CHAPTER 6

The MacDoodle program consists of nine files:

File Name	Listing	Purpose
MacDoodle.asm	E.8	Main program
MacDoodle.equ	E.9	Global equates
MacDoodleData.asm	E.10	Global data
MacDoodleDraw.asm	E.11	Graphics-related code
MacDoodleMenu.asm	E.12	Menu-related code
MacDoodleRTL.asm	E.13	RTL parameter file
MacDoodle.R	E.14	RMAKER input file
MacDoodle.link	E.15	Linker input file
MacDoodle.job	E.16	EXEC input file

```
;
;       MacDoodle.asm.  This program is a rather complete Macintosh application.
;       It features drawing with QuickDraw, text entry, window scrolling and
;       size control, and printing.
;
;       Version 1.00            30-May-1985
;
;       This is a multifile application.  The following files are required:
;
;               MacDoodle.equ           Equates
;               MacDoodleMenu.asm       Menu processors
;               MacDoodleDraw.asm       Drawing/text stuff
;               MacDoodleData.asm       Data file
;               MacDoodle.asm           This file (main program)
;
;               MacDoodleRTL.asm        RTL shell file for MacDoodle RTL
;               MacDoodle.job           EXEC file to build MacDoodle
;               MacDoodle.link          Linker command file
;
        XREF    MenuInit        ;       Menu initialize routine
        XREF    DrawInit        ;       Draw initialize routine
        XREF    Mouse           ;       Mouse-down routine
        XREF    KeyDown         ;       Key-down event
        XREF    ReDraw          ;       Redraw-window routine
        XREF    UpDate          ;       Update-window routine
        XREF    MenuEvt         ;       Menu event
        XREF    CloseEvt        ;       Close-window event
        XREF    ReSize          ;       Resize-window event
        XREF    DoScroll        ;       Line- or page-scroll event
```

Listing E.8 – *The MacDoodle.asm file*

```
         XREF    VScroll         ;       Vertical thumb scroll
         XREF    HScroll         ;       Horizontal thumb scroll
         INCLUDE MACLIB.ASM      ;       Macro definitions
         INCLUDE MacDoodle.Equ   ;       Global equates
         Program MacDoodle,DEBUGOFF,nMenu
;
;        Perform initialization code
;
         MOVE.L  A0,WPointer(A5) ;       Save window address
         MOVE.L  D0,SSize(A5)    ;        "   "    size
         CLR.B   Flags(A5)       ;       Clear flags byte
         JSR     DrawInit        ;       Call initialization
         JSR     MenuInit        ;                          routines
;
;        This is the top of the event loop.  Get an event from the run-time
;        library routine, and call the proper routine to process it.
;
EVLoop:  GETEVENT Cursor(A5)     ;       Get next event
         LSL.W   #2,D0           ;       Multiply by 4
         LEA     JmpTab,A0       ;       A0 -> Jump array
         JSR     0(A0,D0)        ;       Jump to event handler
         BRA     EVLoop          ;       Get next event
;
;        Event Jump Table
;
JmpTab:                          ;       Table of jumps to routine handlers
         JMP     NullEvt         ; 0     No event, get next
         JMP     Mouse           ; 1     Mouse down in window
         JMP     NullEvt         ; 2     Mouse up, ignore
         JMP     KeyDown         ; 3     Key down
         JMP     NullEvt         ; 4     Key up, ignore
         JMP     KeyDown         ; 5     Autokey (same as key down)
         JMP     UpDate          ; 6     Update event
         JMP     NullEvt         ; 7     Disk event, ignore
         JMP     NullEvt         ; 8     Activate event, ignore
         JMP     NullEvt         ; 9     Abort event, ignore
         JMP     NullEvt         ;10     Network event, ignore
         JMP     NullEvt         ;11     Driver event, ignore
         JMP     NullEvt         ;12     Application event #1
         JMP     NullEvt         ;13     Application event #2
         JMP     NullEvt         ;14     Application event #3
         JMP     NullEvt         ;15     Application event #4
         JMP     MenuEvt         ;16     Menu choice
         JMP     CloseEvt        ;17     Close-window event
         JMP     ReSize          ;18     Resize-window event
         JMP     DoScroll        ;19     Scroll event
         JMP     VScroll         ;20     Scroll V with thumb
         JMP     HScroll         ;21     Scroll H with thumb
NullEvt  RTS                     ;       Null event handler
         ENDPROGRAM
```

***Listing E.8** – The MacDoodle.asm file (continued)*

```
;
;       MacDoodle equates.  This file contains most of the symbolic
;       definitions for the MacDoodle program.
;
;       Revision history:
;
;       <01>    31-Aug-85       Install NEWPACK conditional
;
;       Event system equates:
;
EvMenu  EQU     16              ;       Menu choice code
EvClose EQU     17              ;       Close-window choice
;
;       Menu number equates
;
mApple  EQU     1               ;       Apple menu
mFile   EQU     2               ;       File menu
mEdit   EQU     3               ;       Edit menu
mDraw   EQU     4               ;       Draw menu
mPen    EQU     5               ;       PenSize menu
mFont   EQU     6               ;       Text menu
mStyle  EQU     7               ;       Style menu
nMenu   EQU     7               ;       Number of menus
;
CheckM  EQU     $12             ;       Check mark character
CursChar EQU    $5F             ;       Cursor character (underline)
NEWPACK EQU     1               ;<01>   Set to zero to use O/S pack/unpack
;
;       Flags byte definitions
;
F_File   EQU    0               ;       1 => File in window
F_Change EQU    1               ;       1 => Unsaved changes
F_Text   EQU    2               ;       1 => Processing text
;
        XREF    Flags           ;       For the linker
;
;       Macro to set Flags
;
        .MACRO  SetFlag
        BSET    #%1,Flags(A5)   ;       Set the flag
        .ENDM
;
;       Macro to clear Flags
;
        .MACRO  ClrFlag
        BCLR    #%1,Flags(A5)
        .ENDM
;
;       Macro to test Flags
;
        .MACRO  TstFlag
        BTST    #%1,Flags(A5)
        .ENDM
```

Listing E.9 *– The MacDoodle.equ file*

```
;
;       Cursor definitions
;
Arrow   EQU     0               ;       Arrow cursor
IBeam   EQU     1               ;       Text  cursor
Cross   EQU     2               ;       Cross cursor
Plus    EQU     3               ;       Plus  cursor
Watch   EQU     4               ;       Watch cursor
;
;       Data XREFs
;
        XREF    Cursor          ;       Current cursor value
        XREF    PenSize         ;       Pen size value
        XREF    PenType         ;       Pen type value
        XREF    DrwType         ;       Draw type value
        XREF    FBufLen         ;       File buffer data length
        XREF    FileBuf         ;       Paint file buffer
        XREF    FBufSiz         ;       File buffer size
        XREF    UnDoBuf         ;       Undo buffer handle
        XREF    UBufLen         ;       Undo buffer data length
        XREF    UBufSiz         ;       Undo buffer size
        XREF    ScrBuf          ;       Screen buffer
        XREF    SBufSiz         ;       Screen buffer data length
        XREF    FName           ;       File name buffer
        XREF    InfoBuf         ;       File info buffer
        XREF    PType           ;       Type array for POPEN
        XREF    HdrBuf          ;       File header buffer
        XREF    HBufLen         ;       Length of header buffer
        XREF    FBStart         ;       Beginning of file buffer in window
        XREF    FBEnd           ;       End of file buffer in window
        XREF    WPointer        ;       Window pointer
        XREF    SRow            ;       Row number on screen
        XREF    SPoint          ;       Point for (0,0) on screen (in ScrBuf)
        XREF    SSize           ;       Screen size (local coords)
        XREF    FontSel         ;       Font selection number
        XREF    FontNum         ;       Text font number
        XREF    TextFlg         ;       Text flags word
        XREF    TextSiz         ;       Text size in points
        XREF    TextSel         ;       Size selection on menu
        XREF    VolRef          ;       Volume reference number
;
;       TextFlg bit definitions
;
Bold    EQU     0                       ;       Bold bit
Italic  EQU     1                       ;       Italic
ULine   EQU     2                       ;       Underline
OutLine EQU     3                       ;       Outline
Shadow  EQU     4                       ;       Shadow
Cmpress EQU     5                       ;       Compressed
Expand  EQU     6                       ;       Expanded
mBold   EQU     1                       ;       Mask for bold bit
mItalic EQU     2                       ;       Mask for italic bit
mULine  EQU     4                       ;       Mask for underline bit
```

***Listing E.9** – The MacDoodle.equ file (continued)*

```
mOutLine EQU    8               ;       Mask for outline
mShadow  EQU   16               ;       Mask for shadow
mCmpress EQU   32               ;       Mask for compressed
mExpand  EQU   64               ;       Mask for expanded
;
;       Other text definitions
;
BS        EQU   8               ;       Backspace character
CR        EQU   13              ;       Carriage-return character
TBuffSize EQU   512             ;       Input buffer for keys
;
;       Font definitions
;
Chicago EQU     0               ;       System font
NewYork EQU     2               ;
Geneva  EQU     3               ;
Monaco  EQU     4               ;
Venice  EQU     5               ;
London  EQU     6               ;
Athens  EQU     7               ;
SanFran EQU     8               ;
Toronto EQU     9               ;
Cairo   EQU     11              ;
LosAng  EQU     12              ;
Seattle EQU     130             ;
;
;       Macro to construct jump tables
;
        .MACRO  JTAB            ;       Up to 9 labels
        IF      '%1' <> ''      ;       Is parameter nonblank?
        DC.W    %1-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%2' <> ''      ;       Is parameter nonblank?
        DC.W    %2-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%3' <> ''      ;       Is parameter nonblank?
        DC.W    %3-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%4' <> ''      ;       Is parameter nonblank?
        DC.W    %4-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%5' <> ''      ;       Is parameter nonblank?
        DC.W    %5-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%6' <> ''      ;       Is parameter nonblank?
        DC.W    %6-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%7' <> ''      ;       Is parameter nonblank?
        DC.W    %7-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        IF      '%8' <> ''      ;       Is parameter nonblank?
        DC.W    %8-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
```

***Listing E.9** – The MacDoodle.equ file (continued)*

```
        IF      '%9' <> ''      ;       Is parameter nonblank?
        DC.W    %9-JBASE        ;       Offset from table base
        ENDIF                   ;       Parameter nonblank
        .ENDM
;
;       Resource file equates
;
ProgramErr  EQU 129             ;       Program error alert template
Werr        EQU 130             ;       Write error alert template
Cerr        EQU 131             ;       File create error alert
Oerr        EQU 132             ;       File open error alert
BuffErr     EQU 134             ;       Buffer error alert
ReadErr     EQU 135             ;       Read error alert
MemErr      EQU 136             ;       Memory error
NotText     EQU 137             ;       Not doing text
;
;       Dialogs
;
SaveIt      EQU  2              ;       Save changes to ... ?
DoSave      EQU  1              ;       1 => Yes, save it
;
;       File system equates
;
FFType      EQU 32              ;       Finder file type offset
FFAppl      EQU 36              ;       Finder application signature
MPType      EQU $504E5447       ;       PNTG MacPaint file
DAppl       EQU $444F4F44       ;       DOOD Doodle signature
DFSize      EQU 54              ;       Offset to data fork size in bytes
HdrSize     EQU 512             ;       MacPaint file header size
FileSize    EQU $4000           ;       16K file size
UnDoSize    EQU $2000           ;       8K for undo buffer
ScrSize     EQU $5460           ;       21.6K for screen image buffer
SWidth      EQU 72              ;       Width in bytes
SLength     EQU 300             ;       Length in bytes
FLength     EQU 720             ;       File length in rows

NullPat     EQU $B900           ;       Null pattern
LScroll     EQU 8               ;       # pixels in a line scroll
;
;       Register save/restore macros
;
        .MACRO  PUSHALL
        MOVEM.L D0-D7/A0-A4/A6,-(SP)
        .ENDM
;
        .MACRO  POPALL
        MOVEM.L (SP)+,D0-D7/A0-A4/A6
        .ENDM
;
;       Startup data area equates
;
ParmHdl EQU     16              ;       Offset to parameter handle
PMsg    EQU     0               ;       Parameter message
```

***Listing E.9** – The MacDoodle.equ file (continued)*

```
PCount  EQU     2               ;       File count
PVRef   EQU     4               ;       Volume reference number
PFType  EQU     6               ;       File type
PFVer   EQU     10              ;       File version
PFxxx   EQU     11              ;       Unused
PFName  EQU     12              ;       File count byte/name
;
;       Printing equates/xrefs.  These are used in linking with
;       PrLink.rel.  Normally, you would use "Include PrEqu.txt".
;
        XREF    PrOpen          ;       Print manager open
        XREF    PrClose         ;       Print manager close
        XREF    PrDrvrOpen      ;       Open the printer driver
        XREF    PrDrvrClose     ;       Close the printer driver
        XREF    PrCtlCall       ;       Printer control call
PrBits  EQU     4               ;       Bitmap print code
PrPaint EQU     1               ;       Paint mode
PrDCtl  EQU     7               ;       Device control
PrReset EQU     $10000          ;       Code for reset printer
PrPage  EQU     $20000          ;       Code for end of page
```

Listing E.9 *– The MacDoodle.equ file (continued)*

```
;
;       MacDoodleData.asm.  This file contains all global data items shared
;       between modules.
;
;
;       Initialized data:
;
        XDEF    Ptype                   ;
Ptype   DC.W    1                       ;       Type array for POPEN
        DC.L    $504E5447               ;       "MPNT" in ASCII
;
;       Uninitialized data:
;
        XDEF    Cursor
Cursor  DS.W    1                       ;       Cursor for event
        XDEF    PenSize                 ;       Pen size word
PenSize DS.W    1                       ;       0, 1, or 2
        XDEF    PenType                 ;       Pen type word
PenType DS.W    1                       ;       0, 1
        XDEF    DrwType                 ;       0-9
DrwType DS.W    1                       ;       Draw menu choice
        XDEF    Flags                   ;       Flags byte
Flags   DS.W    1                       ;       Use a word for alignment
        XDEF    FontSel                 ;       Font menu selection
FontSel DS.W    1                       ;       For UNCHECK macro
```

Listing E.10 *– The MacDoodleData.asm file*

```
        XDEF    FontNum              ;      Font number for text
FontNum DS.W    1                    ;      1-12
        XDEF    TextFlg              ;      Text flag word
TextFlg DS.W    1                    ;
        XDEF    TextSiz              ;      Text size
TextSiz DS.W    1                    ;                in points
        XDEF    TextSel              ;      Text size
TextSel DS.W    1                    ;               selection number
;
;       Dynamic buffer handles and other info
;
        XDEF    FileBuf              ;      Make
        XDEF    FBufLen              ;
        XDEF    FBufSiz              ;
        XDEF    ScrBuf               ;              all
        XDEF    SBufSiz              ;
        XDEF    UnDoBuf              ;                     accessible
        XDEF    UBufLen              ;
        XDEF    UBufSiz              ;
        XDEF    HdrBuf               ;
        XDEF    HBufLen              ;
FileBuf DS.L    1                    ;      Handle for file buffer
FBufLen DS.L    1                    ;      Data length in bytes
FBufSiz DS.L    1                    ;      Buffer size
ScrBuf  DS.L    1                    ;      Handle for screen buffer
SBufSiz DS.L    1                    ;      Length in bytes
UnDoBuf DS.L    1                    ;      Undo buffer handle
UBufLen DS.L    1                    ;      Data length in bytes
UBufSiz DS.L    1                    ;      Buffer size
HdrBuf  DS.L    1                    ;      File header buffer
HBufLen DS.L    1                    ;      Length of header buffer
;
;       Misc global variables
;
         XDEF   WPointer             ;      -> Window data structure
WPointer DS.L   1                    ;
         XDEF   FBStart              ;      -> First byte of first line
FBStart  DS.W   1                    ;         displayed in file buffer
         XDEF   FBEnd                ;      -> First byte of first line
FBEnd    DS.W   1                    ;         not displayed
         XDEF   SRow                 ;      Row number in screen buffer
SRow     DS.W   1                    ;      for scrolling purposes
         XDEF   SPoint               ;      Point for (0,0) on screen
SPoint   DS.L   1                    ;      in point format
         XDEF   SSize                ;      Screen size
SSize    DS.L   1                    ;      in point format
         XDEF   VolRef               ;      Volume reference number
VolRef   DS.W   1                    ;      (word length)
         XDEF   Fname                ;      File name
Fname:   DS.B   64                   ;                     buffer
         XDEF   InfoBuf              ;      File information
InfoBuf  DS.B   80                   ;                     buffer
```

Listing E.10 – *The MacDoodleData.asm file (continued)*

```
;
;       MacDoodleDraw.asm.  This file contains the procedures related to
;       the use of QuickDraw in the MacDoodle window.
;
;       Revision history:
;
;       <01>    Install hand-coded Pack and UnPack routines
;
        Include Maclib.Asm      ;       Include macro library
        Include MacDoodle.Equ   ;       Include definition file
        Include QuickEqu.D      ;       Include Quickdraw equates
        Include MacTraps.D      ;       and trap definitions
        XDEF    KeyDown         ;       Key-down event
        XDEF    CloseText       ;       Close out text processing
        XDEF    UpDate          ;       Macintosh update event
        XDEF    ReDraw          ;       Redraw window
        XDEF    EScreen         ;       Erase-screen routine
        XDEF    PrDraw          ;       Perform drawing for printer
        XDEF    ReSize          ;       Resize window
        XDEF    RePack          ;       Create single packed buffer
        XDEF    UnBuf           ;       Create 2 packed, 1 unpacked buffer(s)
        XDEF    DoScroll        ;       Line, page scroll handler
        XDEF    VScroll         ;       Vertical thumb scroll handler
        XDEF    HScroll         ;       Horizontal "      "      "
        XDEF    Mouse           ;       Mouse-down handler
        XDEF    DrawInit        ;       Initialize routine for draw
        XDEF    UnPack          ;       Unpack MacPaint file
        XDEF    Pack            ;       Pack MacPaint file
        XDEF    PackU           ;       Pack screen buffer to undo buffer
;
;       DrawInit routine.  This code is executed before we begin
;       handling events.
;
;       Entry conditions:
;
;               (none)
;
;       Exit conditions:
;
;               (none)
;
DrawInit                        ;       Here to initialize
        SetVMin #0              ;       Set vertical min to zero
        SETHMin #0              ;       Set horizontal min to zero
        CLR.L   SPoint(A5)      ;       Init point to (0,0)
        RTS                     ;       Quit
;
;       ReSize routine.  This procedure is called to change the size of the window.
;       Normally called as the result of a resize event from the event system.
;
;       Entry conditions:
;
```

Listing E.11 – The MacDoodleDraw.asm file

```
;               D0 = ReSize code (18 decimal)
;               D1 = New bottom-right point
;
;       Exit conditions:
;
;               (none)
;
ReSize  MOVE.L  SSize(A5),D2    ;       Fetch old size
        MOVE.W  #(SWidth*8)-1,D3;       Number of pixels in HScroll
        SUB.W   D1,D3           ;       Subtract screen width
        SetHMax D3              ;       Set new max
        SWAP    D1              ;       Vertical to low word
        SWAP    D2              ;          "      "  "   "
        MOVE.W  #FLength-1,D3   ;       D3 = # pixels in VScroll
        SUB.W   D1,D3           ;       Subtract screen length
        SetVMax D3              ;       Set new max
        MOVE.W  D1,D4           ;       Save swapped version
        SWAP    D1              ;       Restore to original
        CMP.W   D2,D4           ;       Compare new to old
        BEQ     @1              ;       Same size, just redraw screen
        BSR     RePack          ;       Pack screen to one buffer
        MOVE.L  D1,SSize(A5)    ;       Set size value
        BSR     UnBuf           ;       Unpack screen buffer
        RTS                     ;       Done
@1      MOVE.L  D1,SSize(A5)    ;       Save new size
        BRA     ReDraw          ;       Go do redraw
;
;       UpDate routine.  Calls ReDraw bracketed by BeginUpdate and
;       EndUpdate.
;
;       Entry conditions:
;
;               (none)
;
;       Exit conditions:
;
;               (none)
;
UpDate:                         ;
        MOVE.L  WPointer(A5),-(SP);     Push window pointer
        _BeginUpdate            ;       Signal update begin
        BSR     ReDraw          ;       Refresh screen
        MOVE.L  WPointer(A5),-(SP);     Push window pointer
        _EndUpdate              ;       Signal update end
        RTS                     ;       Back to event handler
;
;       ReDraw routine.  This code refreshes the screen GrafPort from
;       the screen buffer following any activity affecting the display:
;       scrolling, desk accessories, resizing the window, etc.
;
;       Entry conditions:
;
;               SSize(A5) gives window and bitmap size
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
;               Buffer is accurate
;
;       Exit conditions:
;
;               Window set to match buffer
;
ReDraw  PUSHALL                    ;       Save registers
        BSR     CopySetup          ;       Set up for CopyBits
        MOVE.L  PortAdr(A5),A0     ;       A0 -> Window's GrafPort
        PEA     SBitMap(A5)        ;       Source bitmap address
        PEA     PortBits(A0)       ;       Destination bitmap address
        PEA     SrcRect(A5)        ;       Source rectangle address
        PEA     TmpRect(A5)        ;       Destination rectangle address
        MOVE.W  #SrcCopy,-(SP)     ;       Blatant copy mode
        PEA     0                  ;       No mask region
        _CopyBits                  ;       Redraw entire window
        MOVE.L  ScrBuf(A5),A0      ;       A0 = Screen buffer handle
        _HUnLock                   ;       Unlock it
        POPALL                     ;       Restore registers
        RTS                        ;       Return to caller
;
;       Routine SaveScreen is called to save the current screen in
;       the screen buffer (the inverse of ReDraw).
;
;       Entry conditions:
;
;               (none)
;
;       Exit conditions:
;
;               Screen buffer updated
;
SaveScreen                         ;
        PUSHALL                    ;       Save caller's registers
        BSR     CopySetup          ;       Set up for CopyBits
        MOVE.L  PortAdr(A5),A0     ;       A0 -> Window's GrafPort
        PEA     PortBits(A0)       ;       Source bitmap address
        PEA     SBitMap(A5)        ;       Destination bitmap address
        PEA     TmpRect(A5)        ;       Source rectangle address
        PEA     SrcRect(A5)        ;       Destination rectangle address
        MOVE.W  #SrcCopy,-(SP)     ;       Blatant copy mode
        PEA     0                  ;       No mask region
        _CopyBits                  ;       Redraw entire window
        MOVE.L  ScrBuf(A5),A0      ;       A0 = Screen buffer handle
        _HUnLock                   ;       Unlock it
        POPALL
        RTS                        ;       Return to caller
;
;       CopySetup routine.  Calculates values for the screen bitmap and
;       source and destination rectangles for the screen and screen buffer.
;
;       Entry conditions:
;
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
;               (none)
;
;       Exit conditions:
;
;               No registers saved
;
CopySetup                       ;
;
;       Compute rectangle for screen
;
        MOVE.L  SSize(A5),D0    ;       Pick up screen size
        MOVE.L  SPoint(A5),D1   ;       and screen point
        CLR.L   TmpRect(A5)     ;       Window rectangle
        MOVE.L  D0,TmpRect+4(A5);               is (0,SSize)
;
;       (A bit harder) Compute rectangle for buffer.  The origin is
;       SFPoint(A5), and the limit is SFPoint(A5) + SSize(A5).  (You
;       must add the two halves separately.)
;
        MOVE.L  D1,SrcRect(A5)  ;       Buffer rectangle
        ADD.W   D0,D1           ;       is (SPoint,SPoint+SSize)
        SWAP    D1              ;       Now add high
        SWAP    D0              ;                       order
        ADD.W   D0,D1           ;                               words
        SWAP    D1              ;       Back to correct order
        MOVE.L  D1,SrcRect+4(A5);       Store in rectangle structure
;
;       Now formulate the buffer bitmap structure
;
        MOVE.L  ScrBuf(A5),A0   ;       A0 = Screen buffer handle
        _HLock                  ;       Lock it
        MOVE.L  (A0),A3         ;       A3 -> Screen buffer
        MOVE.L  A3,SBitMap(A5)  ;       Set bitmap address
        MOVE.W  #SWidth,SRBytes(A5);    Set rowBytes for screen buffer
        CLR.L   SBounds(A5)     ;       Origin is (0,0)
        MOVE.W  #SLength,Sbounds+4(A5);         Limit is (SLength,
        MOVE.W  #SWidth*8,SBounds+6(A5);                        SWidth*8)
        PEA     PortAdr(A5)     ;       Location for return value
        _GetPort                ;       Fetch the port address
        RTS                     ;       Return
;
;       Escreen routine.  Erases the current window area in the document.
;
;       Entry conditions:
;
;               (none)
;
;       Exit conditions:
;
;               Undo buffer updated
;
EScreen                         ;       Called from menu code
        PUSHALL                 ;       Save registers
```

Listing E.11 – The MacDoodleDraw.asm file (continued)

```
        JSR     PackU           ;       Save screen in undo buffer
        CLR.L   TmpRect(A5)     ;       Clear origin
        MOVE.L  SSize(A5),TmpRect+4(A5); Set screen size
        PEA     TmpRect(A5)     ;       Push rectangle address
        _EraseRect              ;       Erase it
        BSR     SaveScreen      ;       Now update screen buffer
        POPALL                  ;       Pop registers
        RTS                     ;       Return to menu code
;
;       PrDraw routine.  Does CopyBits to the printer port for printing
;       the document.
;
;       Entry conditions:
;
;               PrPort(A5) must contain printer port address
;
;       Exit conditions:
;
;               Document printed
;
PrDraw  PUSHALL                 ;       Save the registers
        MOVE.W  #PrDCtl,-(SP)   ;       Printer device control code
        MOVE.L  #PrReset,-(SP)  ;       Code for printer reset
        CLR.L   -(SP)           ;       Two null
        CLR.L   -(SP)           ;                 parameters
        JSR     PrCtlCall       ;       Do a reset before starting to print
        MOVE.W  SRow(A5),D7     ;       Remember current screen position
        CLR.L   D6              ;       D6 counts rows
@1      MOVE.L  D6,D0           ;       Set D0 for scroll op
        SWAP    D0              ;       To high word for VScroll
        BSR     VScroll         ;       Do the scroll
        MOVE.W  D6,D5           ;       Copy row number
        SUB.W   Srow(A5),D5     ;       Normalize to displayed screen
        MOVE.W  D5,TmpRect(A5)  ;       Set
        CLR.W   TmpRect+2(A5)   ;           up
        MOVE.W  SSize(A5),TmpRect+4(A5);       destination
        MOVE.W  #SWidth*8,TmpRect+6(A5);                      rectangle
        MOVE.L  ScrBuf(A5),A0   ;       Fetch
        _Hlock                  ;               screen
        MOVE.L  (A0),A3         ;                       address (locked)
        MOVE.L  A3,SBitMap(A5)  ;       Store in bitmap structure
        MOVE.W  #SWidth,SRBytes(A5);    Set rowBytes
        CLR.W   SBounds(A5)     ;       Origin is
        CLR.W   SBounds+2(A5)   ;                         (0,0)
        MOVE.W  SSize(A5),SBounds+4(A5); Destination
        SUB.W   D5,SBounds+4(A5);                      is
        MOVE.W  #SWidth*8,SBounds+6(A5);                  (SSize-D5.W,576)
        MOVE.W  #PrBits,-(SP)   ;       Specify bitmap print
        PEA     SBitMap(A5)     ;       Push source bitmap address
        PEA     TmpRect(A5)     ;       Destination rectangle address
        MOVE.L  #PrPaint,-(SP)  ;       Specify square resolution
        JSR     PrCtlCall
        MOVE.L  ScrBuf(A5),A0   ;       Unlock
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
        _HUnLock                ;               screen buffer
        ADD.W   SSize(A5),D6    ;       Bump row number
        CMP.L   #Flength,D6     ;       See if done
        BLT     @1              ;       GE => Finished
        MOVE.W  D7,D0           ;       Restore
        SWAP    D0              ;               vertical
        JSR     VScroll         ;                       position
        MOVE.W  #PrDCtl,-(SP)   ;       Device control code
        MOVE.L  #PrPage,-(SP)   ;       Indicate end of page
        CLR.L   -(SP)           ;       Two
        CLR.L   -(SP)           ;               nulls
        JSR     PrCtlCall       ;       Eject page
        POPALL                  ;       Page is printed
        RTS                     ;       Done
;
;       Scrolling event handlers.  Routine DoScroll is used for page
;       and line scrolls.
;       Entry conditions:
;
;               High word of D0 = Type:
;
;                       0 - Line scroll up
;                       1 - Line scroll down
;                       2 - Page scroll up
;                       3 - Page scroll down
;                       4 - Line scroll left
;                       5 - Line scroll right
;                       6 - Page scroll left
;                       7 - Page scroll right
;
;               D1 = Control handle
;
;       Exit conditions:
;
;               (none)
;
DoScroll                        ;
        SWAP    D0              ;       Type to low word
        MOVE.W  D0,ScrType(A5)  ;       Remember type
        TRACK   D1,XScroll      ;       Do the scroll
        RTS                     ;       Return
;
;       XScroll routine.  This routine is called repeatedly from the
;       run-time library (actually _TrackControl) to do scrolling.
;
;       Entry conditions (Pascal calling sequence):
;
;                (SP) = Control handle
;               4(SP) = Part code
;               6(SP) = Return address
;
;       Exit conditions:
;
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
;               Stack purged of all of the above
;
;       If the part code is nonzero, we actually do the scrolling.
;       If zero, we just ignore.  The variable ScrType tells us
;       what type of scroll to do.
;
XScroll                          ;
        MOVE.L  (SP)+,A0         ;       Fetch return address
        MOVE.W  (SP)+,D1         ;       and part code
        MOVE.L  (SP)+,D0         ;       and control handle
        MOVE.L  A0,-(SP)         ;       ReStack return address
        TST.W   D1               ;       Still in right spot?
        BNE     @1               ;       Yes, continue
        RTS                      ;       No, quit now
@1      PUSHALL                  ;       Save registers
        MOVE.W  ScrType(A5),D2   ;       Fetch scroll type
        MOVE.W  #LScroll,D3      ;       Move pixel count to D3
        MOVE.W  SSize(A5),D4     ;       D4 = Vertical page size
        MOVE.W  SRow(A5),D5      ;       D5 = Vertical position of (0,0)
        BTST    #2,D2            ;       Vertical or horizontal?
        BEQ     @2               ;       EQ => Vertical
        MOVE.W  SSize+2(A5),D4   ;       Correct page size
        MOVE.W  SPoint+2(A5),D5  ;       D5 = Horizontal position
@2      BTST    #1,D2            ;       Page or line scroll?
        BEQ     @3               ;       EQ => Line
        MOVE.W  D4,D3            ;       Correct pixel count for page
@3      BTST    #0,D2            ;       Left (up) or right (down)?
        BNE     @4               ;       EQ => Right (down)
        NEG.W   D3               ;       Negate # pixels
@4      ADD.W   D5,D3            ;       Make absolute
        MOVE.W  D3,D0            ;       Move to
        SWAP    D0               ;               D0 high word
        BTST    #2,D2            ;       Vertical or horizontal?
        BNE     xHScroll         ;       NE => Horizontal
        BRA     xVScroll         ;       Else, go do vertical
;
;       Thumb scroll routines.  Routines VScroll and HScroll
;       indicate a thumb scrolling request.
;
;       Entry conditions:
;
;               D0 (high word) = Thumb value
;               D1.L           = Control ID
;
;       Exit conditions:
;
;               (none)
;
VScroll PUSHALL                  ;       Save registers
xVScroll                         ;       Here to skip PUSHALL
        SWAP    D0               ;       Value to low half
        TST.W   D0               ;       Value positive?
        BGE     @1               ;       Yes, it's OK
```

Listing E.11 *– The MacDoodleDraw.asm file (continued)*

```
         CLR.W   D0               ;       No, force it
@1       MOVE.W  #FLength-1,D1    ;       D1 =
         SUB.W   SSize(A5),D1     ;               Maximum size
         CMP.W   D0,D1            ;       Check for > max
         BGE     @2               ;       OK, leave it
         MOVE.W  D1,D0            ;       Too big, force to max
@2       BSR     RePack           ;       Force back to single image
         MOVE.W  D0,SRow(A5)      ;       Set row number
         SetVVal D0               ;       Move the thumb
         BSR     UnBuf            ;       Unpack proper amount of screen buffer
         POPALL                   ;       Restore registers
         RTS                      ;       Return to caller
;
;        Equivalent routine for horizontal scroll
;
HScroll  PUSHALL                  ;       Save registers
xHScroll                          ;       Here to skip PUSHALL
         SWAP    D0               ;       Value to low-order word
         TST.W   D0               ;       Value OK?
         BGE     @1               ;       Yes, must be > 0
         CLR.W   D0               ;       No, make minimum
@1       MOVE.W  #SWidth*8,D1     ;       Fetch screen width
         SUB.W   SSize+2(A5),D1   ;       Calculate largest value
         CMP.W   D0,D1            ;       Check it
         BGE     @2               ;       GE => in range
         MOVE.W  D1,D0            ;       Else make maximum value
@2       SetHVal D0               ;       Set the value
         MOVE.W  D0,SPoint+2(A5)  ;       Set for ReDraw
         POPALL                   ;       Pop registers
         BRA     ReDraw           ;       Refresh screen
;
;        Routine Mouse is called when a mouse-down event occurs.
;
;        Entry conditions:
;
;               D1.L = Mouse point (local coords)
;
;        Exit conditions:
;
;               (Mouse up, correct thing drawn)
;
Mouse:                            ;
         TstFlag F_Text           ;       Text open?
         BEQ     @1               ;       No, continue
         JSR     CloseText        ;       Yes, close first
@1       MOVE.L  D1,Point1(A5)    ;       Save original point
         BSR     PackU            ;       Save screen in undo buffer
         PEA     OldPen(A5)       ;       Save
         _GetPenState             ;               pen state
         MOVE.W  PenSize(A5),D1   ;       Fetch pen size
         MOVE.W  D1,-(SP)         ;       Push
         MOVE.W  D1,-(SP)         ;               2 copies
         _PenSize                 ;       Set new pen size
         MOVE.L  Point1(A5),-(SP);        Push mouse point
         _MoveTo                  ;       Move pen to mouse point
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
        MOVE.W  DrwType(A5),D0  ;       Fetch draw routine index
        SUBQ.W  #1,D0           ;       Force zero relative
        LSL.W   #1,D0           ;       Multipy x 2 for word index
        MOVE.W  JBASE(D0.W),D0  ;       Fetch offset
        JSR     JBASE(D0.W)     ;       Call proper routine
        PEA     OldPen(A5)      ;       Now
        _SetPenState            ;               restore pen
        SetFlag F_Change        ;       Mark a change to file
        CMPI.W  #5,DrwType(A5)  ;       Just start text?
        BEQ     @2              ;       Yes, don't save screen
        JSR     SaveScreen      ;       Save screen in screen buffer
@2      RTS                     ;       Back for next event
Void:   StopAlert #ProgramErr   ;       Fatal error!
        EXIT                    ;       Back to Finder
;
;       Jump table for Draw routines
;
JBASE   JTAB    Lines,  FreeH,  Boxes,  Ovals
        JTAB    Text,   Void,   Eraser, Void
;
;       Lines routine.  Draw a straight line using the current
;       pen size.
;
Lines:                          ;       Line handler
        MOVE.W  #PatXor,-(SP)   ;       Set pen
        _PenMode                ;               to XOR for dragging
        MOVE.L  Pointl(A5),D3   ;       Save mouse point
        MOVE.L  D3,D4           ;       Need original point
@1:     PEA     Pointl(A5)      ;       Temp point
        _GetMouse               ;       Fetch mouse in local coords
        CMP.L   Pointl(A5),D3   ;       Check against old
        BEQ     @2              ;       Equal, don't update
        BSR     DrawLine        ;       Cancel old line
        MOVE.L  Pointl(A5),D3   ;       Save new point
        BSR     DrawLine        ;       Draw new line
@2:     CLR.W   -(SP)           ;       Room for result
        _StillDown              ;       Mouse still down?
        TST.B   (SP)+           ;       If NE,
        BNE     @1              ;               yes, still down
        MOVE.W  #PatCopy,-(SP)  ;       Change pen
        _PenMode                ;                   to copy mode
        BSR     DrawLine        ;       Draw final line
        RTS                     ;       Return to caller
;
;       DrawLine routine.  Draws a line from point in D4.L
;       to point in D3.L.
;
DrawLine                        ;       Draw a line
        MOVE.L  D4,-(SP)        ;       Reposition
        _Moveto                 ;                   pen
        MOVE.L  D3,-(SP)        ;       Cancel old line
        _Lineto                 ;       thusly
        RTS                     ;       Return to caller
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
;
;       Eraser routine.  Take a freehand swipe at existing
;       bitmap.  Same as a Sketch with a BIC pen mode.
;
Eraser:                         ;
        MOVE.W  #patBic,-(SP)   ;       Push desired mode
        _PenMode                ;       Set the pen mode
;
;       Fall through to FreeH
;
;       FreeH routine.  Draw a freehand line using the current
;       pen size.
;
FreeH:                          ;       Sketch handler
        MOVE.L  Pointl(A5),D3   ;       Save mouse point
@1:     PEA     Pointl(A5)      ;       Temp point
        _GetMouse               ;       Fetch mouse in local coords
        CMP.L   Pointl(A5),D3   ;       Check against old
        BEQ     @2              ;       Equal, don't update
        MOVE.L  Pointl(A5),D3   ;       Save new point
        MOVE.L  D3,-(SP)        ;       Push point
        _Lineto                 ;       Draw a line to new point
@2:     CLR.W   -(SP)           ;       Room for result
        _StillDown              ;       Mouse still down?
        TST.B   (SP)+           ;       If NE,
        BNE     @1              ;               yes, still down
        RTS                     ;       Return to caller
;
;       Routines Boxes and Ovals share common code to draw
;       stretchy boxes and ovals.
;
Boxes
        CLR.L   D7              ;       D7 is flag
        BRA     DoRect          ;       Now draw object
Ovals   MOVEQ.L #-1,D7          ;       Set all bits of D7
DoRect  MOVE.L  Pointl(A5),D1   ;       Fetch mouse point
        MOVE.L  D1,D2           ;       Copy mouse point
        LEA     NewRect(A5),A3  ;       A3 -> New rectangle
        LEA     OldRect(A5),A2  ;       A2 -> Old rectangle
        LEA     TmpRect(A5),A4  ;       A4 -> Temporary rectangle
        MOVEM.L D1/D2,(A2)      ;       Set old rect
        MOVEM.L D1/D2,(A3)      ;       and new rect to mouse point
        MOVE.W  #PatXor,-(SP)   ;       Set pen
        _PenMode                ;               to Xor mode
RectLoop                        ;       Here while mouse down
        CLR.W   -(SP)           ;       Room for result
        _StillDown              ;       See if
        TST.B   (SP)+           ;               still
        BEQ     DoFinal         ;                       down (EQ => No)
        PEA     BotRight(A3)    ;       Get
        _GetMouse               ;               mouse coords
        CLR.W   -(SP)           ;       Space for result
        MOVEM.L A2/A3,-(SP)     ;       Push A2 and A3
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
        _EqualRect                ;       Has mouse
        TST.B   (SP)+             ;                       moved?
        BNE     RectLoop          ;       No, continue polling
        MOVE.L  (A2),(A4)         ;       Set temp rect
        MOVE.L  4(A2),4(A4)       ;                       to old rect
        JSR     DoDraw            ;       XOR to restore screen contents
        MOVE.L  (A3),(A4)         ;       Set temp rect
        MOVE.L  4(A3),4(A4)       ;                       to new rect
        JSR     DoDraw            ;       Draw new object
        MOVE.L  (A3),(A2)         ;       Copy new rect
        MOVE.L  4(A3),4(A2)       ;                       to old rect
        BRA     RectLoop          ;       Repeat till mouse up
DoFinal                           ;       Here to make final
        MOVE.W  #PatCopy,-(SP)    ;       Set pen to copy mode
        _PenMode                  ;       thusly
;       JMP     DoDraw            ;       Draw last thing permanently
;
;       Here we drop into DoDraw, which will return to the dispatcher
;       at the front of this file.
;
DoDraw:                           ;       FrameRect(Oval) at (A4)
        MOVE.W  Bottom(A4),D0     ;       Fetch bottom word
        CMP.W   Top(A4),D0        ;       Compare with top
        BGE     @1                ;       If GE, don't swap
        MOVE.W  Top(A4),Bottom(A4);       Swap
        MOVE.W  D0,Top(A4)        ;               v coordinates
@1:                               ;       Here to compare h coordinates
        MOVE.W  Right(A4),D0      ;       Fetch right coord
        CMP.W   Left(A4),D0       ;       Compare with left
        BGE     @2                ;       If GE, don't swap
        MOVE.W  Left(A4),Right(A4);       Swap
        MOVE.W  D0,Left(A4)       ;               h coordinates
@2:     MOVE.L  A4,-(SP)          ;       Push rectangle address
        TST.W   D7                ;       Boxes?
        BNE     @3                ;       IF NE, then doing ovals
        _FrameRect                ;       Draw rectangle
        RTS                       ;       Return
@3:                               ;       Here if ovals
        _FrameOval                ;       Do the oval instead
        RTS                       ;       Return
;
;       Routine Text is called for a mouse-down event when Text is selected on
;       the Draw menu.
;
Text                              ;
        MOVE.W  #PatXor,-(SP)     ;       Set text
        _TextMode                 ;               mode to XOR
        MOVE.W  FontNum(A5),-(SP);        Set
        _TextFont                 ;               proper font
        MOVE.W  TextFlg(A5),-(SP);        Set text
        _TextFace                 ;               attributes
        MOVE.W  TextSiz(A5),-(SP);        Set text
        _TextSize                 ;               size
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
         CLR.W   BuffCnt(A5)      ;     Clear text buffer count
         LEA     TBuffer(A5),A0   ;     A0 -> Buffer
         MOVE.L  A0,BuffPtr(A5)   ;     Save it
         SetFlag F_Text           ;     Remember we're doing text
         MOVE.L  Point1(A5),Point2(A5); Set pen location
         BRA     DrawCurs         ;     Draw cursor
;
;        KeyDown routine.  Used to draw text on the screen following
;        a mouse event with Text selected on the Draw menu.
;
KeyDown                           ;
         TstFlag F_Text           ;     Doing text?
         BNE     Drawtext         ;     Yes, continue
         StopAlert #NotText       ;     No, give error
         RTS                      ;     and return
DrawText                          ;     Here to draw a character
         SetFlag F_Change         ;     Changed the buffer
         CMP.B   #BS,D1           ;     A backspace?
         BEQ     IsBS             ;     Yes, special
         CMP.B   #CR,D1           ;     Carriage return?
         BEQ     IsCR             ;     Yes, also special
         MOVE.L  BuffPtr(A5),A0   ;     Fetch buffer pointer
         ADD.W   #1,BuffCnt(A5)   ;     Increment character count
         MOVE.B  D1,(A0)+         ;     Save character
         MOVE.L  A0,BuffPtr(A5)   ;     Remember address
         BSR     DrawCurs         ;     Erase cursor
         MOVE.L  Point2(A5),-(SP);      Reset
         _MoveTo                  ;             pen location
         MOVE.W  D1,-(SP)         ;     Draw
         _DrawChar                ;             character
         PEA     Point2(A5)       ;     Get
         _GetPen                  ;             pen location
         BRA     DrawCurs         ;     Draw the cursor
IsBS:                             ;     Here if a backspace
         MOVE.W  BuffCnt(A5),D1   ;     Any characters in buffer?
         BEQ     CantBS           ;     No, can't backspace
         BSR     DrawCurs         ;     Draw (erase) cursor
         SUBQ.W  #1,D1            ;     Subtract 1 from count
         MOVE.W  D1,BuffCnt(A5)   ;     Store updated count
         MOVE.L  BuffPtr(A5),A0   ;     A0 -> Free area
         MOVE.B  -(A0),D3         ;     Character to D3
         MOVE.L  A0,BuffPtr(A5)   ;     Store pointer
         CLR.W   -(SP)            ;     Room for result
         MOVE.W  D3,-(SP)         ;     Find out how
         _CharWidth               ;          big the character is
         MOVE.W  (SP)+,D4         ;     Result to D4
         SUB.W   D4,Point2+2(A5)  ;     Decrement v coord by that much
         MOVE.L  Point2(A5),-(SP);      Back up
         _MoveTo                  ;             cursor
         MOVE.W  D3,-(SP)         ;     Redraw (erase)
         _DrawChar                ;                     character
         BSR     DrawCurs         ;     Draw cursor
CantBS   RTS                      ;     Done
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
;
;       CloseText routine.  Closes out text entry.
;
IsCR:                           ;       Also carriage return
CloseText:                      ;
        MOVEM.L D0-D7/A0-A6,-(SP);      Save registers
        BSR     DrawCurs        ;       Erase cursor character
        MOVE.W  BuffCnt(A5),D3  ;       Get character count
        MOVE.B  D3,TBufCnt(A5)  ;       Set byte
        MOVE.W  #PatCopy,-(SP)  ;       Set text
        _TextMode               ;               to copy mode
        MOVE.L  Point1(A5),-(SP);       Go back
        _MoveTo                 ;               to start position
        PEA     TBufCnt(A5)     ;       Push string address
        _DrawString             ;       Redraw in copy mode
        ClrFlag F_Text          ;       No longer in text mode
        BSR     SaveScreen      ;       Save screen in buffer now
        MOVEM.L (SP)+,D0-D7/A0-A6;      Restore registers
        RTS                     ;       Done
;
;       DrawCurs routine.  Draws a character for cursor.
;
DrawCurs:                       ;
        MOVE.L  Point2(A5),-(SP);       Reset
        _MoveTo                 ;               pen location
        MOVE.W  #CursChar,-(SP) ;       Delete
        _DrawChar               ;               cursor
        RTS                     ;       Return to caller
        IF      NEWPACK         ;<01>   Hand-coded pack/unpack
;<01>
;<01>   UnPack routine.  Unpacks a MacPaint file to a bitmap.
;<01>   File format is:
;<01>
;<01>           <positive count><"count" unique bytes>
;<01>           <negative count><repeated byte>
;<01>
;<01>   There are 72 bytes/line, 720 lines total. Counts are
;<01>   DBRA-adjusted in packed format.
;<01>
;<01>   Entry conditions:
;<01>
;<01>           A0 -> Source area (packed format)
;<01>           A1 -> Destination area
;<01>           A6 -> First byte beyond destination area
;<01>           D0 =  Number of rows to unpack
;<01>
;<01>   Exit conditions:
;<01>
;<01>           A0 -> First byte not unpacked
;<01>           A1 -> First byte not used in destination
;<01>
;<01>   This routine is rather time critical, so space is traded for
;<01>   speed where possible.
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
;<01>
UnPack  MOVEM.W D1-D3/A6,-(SP)  ;<01>   Stack registers used
        SUBQ.W  #1,D0           ;<01>   Adjust for DBRA
        SUB.L   #SWidth,A6      ;<01>   Max value for a row
URowLoop                        ;<01>   Unpack a single row
        MOVE.W  #SWidth,D2      ;<01>   D2 counts bytes
        CMPA.L  A1,A6           ;<01>   Check limit
        BLS     PMemErr         ;<01>   Can't do it
UItem                           ;<01>   Unpack a single item
        CLR.W   D1              ;<01>   D1 =
        MOVE.B  (A0)+,D1        ;<01>           Count
        BMI     URep            ;<01>   < 0 => Replicated value
        SUB.W   D1,D2           ;<01>   Adjust bytes remaining
        SUBQ.W  #1,D2           ;<01>   Fudge for DBRA
UCopy   MOVE.B  (A0)+,(A1)+     ;<01>   Do copy
        DBRA    D1,UCopy        ;<01>             count down
        BRA     UREnd           ;<01>   Go test for end of row
URep                            ;<01>   Here for repeated item
        NEG.B   D1              ;<01>   Negate for positive count
        BMI     UItem           ;<01>   Ignore 80H bytes
        SUB.W   D1,D2           ;<01>   Adjust bytes remaining
        SUBQ.W  #1,D2           ;<01>   Fudge for DBRA
        MOVE.B  (A0)+,D3        ;<01>   Load repeated value
UStore  MOVE.B  D3,(A1)+        ;<01>   Store a byte
        DBRA    D1,UStore       ;<01>   Count down
UREnd                           ;<01>   Here when item complete
        TST.W   D2              ;<01>   Any more bytes in row?
        BGT     UItem           ;<01>   Yes, get next item
        DBRA    D0,URowLoop     ;<01>   Count down rows
        MOVEM.W (SP)+,D1-D3/A6  ;<01>   Unstack registers
        RTS                     ;<01>   Return to caller
;<01>
;<01>   Pack routine.  This routine performs the inverse of UnPack.
;<01>
;<01>   Entry conditions:
;<01>
;<01>           A0 -> Source (raw bits)
;<01>           A1 -> Destination (packed)
;<01>           A6 -> Limit address
;<01>           D0 =  Number of rows to pack
;<01>
;<01>   Exit conditions:
;<01>
;<01>           A0 -> First byte not packed
;<01>           A1 -> First free byte in destination
;<01>
Pack    MOVEM.L D0-D3/A2/A6,-(SP);<01>  Stack registers
        SUBQ.W  #1,D0           ;<01>   Adjust # rows for DBRA
        SUBA.L  #SWidth+1,A6    ;<01>   Adjust limit
PRowLoop                        ;<01>   Top of loop for packing 1 row
        CMPA.L  A1,A6           ;<01>   Check limit
        BLS     PMemErr         ;<01>   Memory violation
        MOVE.W  #SWidth-1,D2    ;<01>   D2 counts bytes in row
```

Listing E.11 – The MacDoodleDraw.asm file (continued)

```
PItem   MOVE.L  A1,A2           ;<01>   A2 -> Count Byte
        ADDQ.L  #1,A1           ;<01>   A1 -> First data byte area
        CLR.W   D3              ;<01>   D3 counts characters in run
        MOVE.B  (A0)+,D1        ;<01>   D1 = First character in run
        SUBQ.W  #1,D2           ;<01>   Mark a character used
        CMP.B   (A0),D1         ;<01>   Check for run type
        BNE     PUnlike         ;<01>   Run of unlike bytes
PLike   ADDQ.W  #1,D3           ;<01>   Bump character count
        ADDQ.L  #1,A0           ;<01>   A0 -> Next character
        CMP.B   (A0),D1         ;<01>   Compare for equality
        DBNE    D2,PLike        ;<01>   Loop until unlike or count
        SUBQ.W  #1,D2           ;<01>   DBNE did not decrement if NE
        MOVE.B  D1,(A1)+        ;<01>   Store repeated character
        NEG.W   D3              ;<01>   Indicate repeated byte count
        BRA     PIEnd           ;<01>   Do end item stuff
PUnLike                         ;<01>   Here for string of unlike bytes
        ADDQ.W  #1,D3           ;<01>   Bump character count
        MOVE.B  D1,(A1)+        ;<01>   Store unlike byte
        MOVE.B  (A0)+,D1        ;<01>   Advance to next byte
        CMP.B   (A0),D1         ;<01>   See if still unequal
        DBEQ    D2,PUnLike      ;<01>   Loop till unequal or count
        SUBQ.W  #1,D2           ;<01>   DBRA did not decrement if NE
        BMI     PIEnd           ;<01>   End of row if MI
        TST.B   -(A0)           ;<01>   Back up
        ADD.W   #1,D2           ;<01>   Uncount character
        SUBQ.W  #1,D3           ;<01>   and decrement count
PIEnd   MOVE.B  D3,(A2)         ;<01>   Store count byte
        TST.W   D2              ;<01>   End of row yet?
        BPL     PItem           ;<01>   No, do another item
        DBRA    D0,PRowLoop     ;<01>   Do another row
        MOVEM.L (SP)+,D0-D3/A2/A6;<01>  Unstack registers
        RTS                     ;<01>   Return
        ELSE                    ;<01>   New Pack/UnPack stuff
;
;       UnPack routine.  This routine performs the inverse of Pack.
;
;       Entry conditions:
;
;               A0 -> Source (raw bits)
;               A1 -> Destination (packed)
;               A6 -> Limit address
;               D0 =  Number of rows to unpack
;
;       Exit conditions:
;
;               A0 -> First byte not unpacked
;               A1 -> First free byte in destination
;
UnPack  MOVEM.L D0-D3/A2/A6,-(SP);      Stack registers
        SUBQ.W  #1,D0           ;       Adjust # rows for DBRA
        SUBA.L  #SWidth,A6      ;       Adjust limit register
        MOVE.W  D0,D3           ;       Place in nonvolatile register
        MOVE.L  A0,Addr1(A5)    ;       Store
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
        MOVE.L  A1,Addr2(A5)    ;               addresses
@1      CMPA.L  Addr2(A5),A6    ;       Check limit
        BLS     PMemErr         ;       Too high, quit
        PEA     Addr1(A5)       ;       Push source address
        PEA     Addr2(A5)       ;       and destination
        MOVE.W  #SWidth,-(SP)   ;       Push byte count
        _UnPackBits             ;       Do the pack
        DBRA    D3,@1           ;       Loop till done
        MOVE.L  Addr1(A5),A0    ;       Restore
        MOVE.L  Addr2(A5),A1    ;       addresses
        MOVEM.L (SP)+,D0-D3/A2/A6;      Restore registers
        RTS                     ;       Back to caller
;
;       Pack routine.  This routine performs the inverse of UnPack.
;
;       Entry conditions:
;
;               A0 -> Source (raw bits)
;               A1 -> Destination (packed)
;               A6 -> Limit address
;               D0 =  Number of rows to pack
;
;       Exit conditions:
;
;               A0 -> First byte not packed
;               A1 -> First free byte in destination
;
Pack    MOVEM.L D0-D3/A2/A6,-(SP);      Stack registers
        SUBQ.W  #1,D0           ;       Adjust # rows for DBRA
        SUBA.L  #SWidth+1,A6    ;       Adjust limit
        MOVE.W  D0,D3           ;       Place in nonvolatile register
        MOVE.L  A0,Addr1(A5)    ;       Store
        MOVE.L  A1,Addr2(A5)    ;               addresses
@1      CMPA.L  Addr2(A5),A6    ;       Check limit
        BLS     PMemErr         ;       Too high, quit
        PEA     Addr1(A5)       ;       Push source address
        PEA     Addr2(A5)       ;       and destination
        MOVE.W  #SWidth,-(SP)   ;       Push byte count
        _PackBits               ;       Do the pack
        DBRA    D3,@1           ;       Loop till done
        MOVE.L  Addr1(A5),A0    ;       Restore
        MOVE.L  Addr2(A5),A1    ;       addresses
        MOVEM.L (SP)+,D0-D3/A2/A6;      Restore registers
        RTS                     ;       Back to caller
Addr1   DS.L    1               ;       Temp address
Addr2   DS.L    1               ;       Another "
        ENDIF                   ;<01>
PMemErr StopAlert #MemErr       ;       Out of memory
        EXIT                    ;       So quit
;
;       PackU routine.  This routine packs a screen buffer into the undo
;       buffer.
;
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
;       Entry conditions: (none)
;
;       Exit conditions:
;
;               D0 = Number of bytes unpacked
;
PackU   PUSHALL                 ;       Save caller's registers
        MOVE.L  UnDoBuf(A5),A0  ;       A0 -> Undo handle
        MOVE.L  A0,A3           ;       Save handle
        _HLock                  ;       Lock it down, too
        MOVE.L  (A0),A1         ;       A1 -> Undo buffer
        MOVE.L  A1,A4           ;       Save this
        MOVE.L  ScrBuf(A5),A0   ;       A0 -> Handle
        MOVE.L  A0,A2           ;       Save handle
        _HLock                  ;       Lock it down
        MOVE.L  (A0),A0         ;       A0 -> Unpacked data
        MOVE.W  SSize(A5),D0    ;       D0 = Count
        MOVE.L  A1,A6           ;       Calculate
        ADDA.L  UBufSiz(A5),A6  ;                   limit
        BSR     Pack            ;       Pack it
        SUB.L   A4,A1           ;       Compute byte count
        MOVE.L  A1,(SP)         ;       Put it in D0 (via POPALL)
        MOVE.L  A2,A0           ;       Unlock
        _HUnLock                ;               all
        MOVE.L  A3,A0           ;                   locked
        _HUnLock                ;                           buffers
        POPALL                  ;       Restore registers
        RTS                     ;       Return to caller
;
;       RePack routine.  Packs all from the screen into one contiguous buffer.
;
;       Entry conditions:
;
;               SSize(A5) must be correct
;
;       Exit conditions:
;
;               (None)
;
RePack  PUSHALL                 ;       Save all registers
        BSR     PackU           ;       Pack screen to UnDoBuf
        MOVE.L  UnDoBuf(A5),A0  ;       A0 ->
        MOVE.L  (A0),A0         ;               Undo buffer
        MOVE.L  FileBuf(A5),A1  ;       A1 ->
        MOVE.L  (A1),A1         ;               File buffer
        MOVE.L  A1,A2           ;       Save this
        CLR.L   D1              ;       Clear D1 to convert
        MOVE.W  FBEnd(A5),D1    ;       D1 = # bytes in bottom area
        SUB.L   FBufSiz(A5),D1  ;       Subtract buffer size
        NEG.L   D1              ;       D1 = Buffer offset
        SUB.W   D0,D1           ;       Compute new buffer offset
        LEA     0(A1,D1.W),A1   ;       A1 -> New buffer offset
        MOVE.L  A1,A4           ;       Save this
```

***Listing E.11** – The MacDoodleDraw.asm file (continued)*

```
        SUBQ.W  #1,D0            ;    Adjust D0 for DBRA now
        CMP.W   FBStart(A5),D1   ;    Overlap?
        BGE     @1               ;    No, it's OK
        StopAlert #BuffErr       ;    Out of memory,
        EXIT                     ;                   all is lost
@1      MOVE.B  (A0)+,(A1)+      ;    Move in undo buffer
        DBRA    D0,@1            ;    Until done
        SUB.W   FBStart(A5),D1   ;    Subtract start length
        ANDI.L  #$FFFF,D1        ;    Mask off upper part
        SUB.L   FBufSiz(A5),D1   ;    Compute
        NEG.L   D1               ;           end length
        MOVE.W  D1,FBEnd(A5)     ;    Store new count
        MOVE.W  FBStart(A5),D0   ;    D0 = # bytes to move
        BEQ     @3               ;    EQ => Top region empty
        LEA     0(A2,D0.W),A2    ;    A2 -> End of start region
        SUBQ.W  #1,D0            ;    DBRA-adjusted
        CLR.W   FBStart(A5)      ;    No bytes here now
@2      MOVE.B  -(A2),-(A4)      ;    Move in
        DBRA    D0,@2            ;           start of buffer
@3      POPALL                   ;    Restore registers
        RTS                      ;    Back to caller
;
;       UnBuf routine.  Unpacks a consolidated buffer (as when reading a file)
;       into the three separate regions:  buffer start (FBStart), screen buffer
;       (unpacked), and buffer end (FBEnd).
;
;       Entry conditions:
;
;               Packed data in end of buffer
;               SSize(A5) is correct
;               SRow(A5)  is correct
;
;       Exit conditions:
;
;               Buffer split into three pieces
;
UnBuf   PUSHALL                  ;    Save the registers
        MOVE.L  FileBuf(A5),A0   ;    A0 -> File buffer handle
        _HLock                   ;    Lock it
        MOVE.L  (A0),A1          ;    A1 -> File buffer
        MOVE.L  A1,A2            ;    Save for later
        CLR.L   D1               ;    Load
        MOVE.W  FBEnd(A5),D1     ;           byte count
        SUB.L   FBufSiz(A5),D1   ;    Compute
        NEG.L   D1               ;           new
        LEA     0(A1,D1.W),A0    ;                 offset
        MOVE.L  A0,A4            ;    Remember this
        MOVE.W  SRow(A5),D2      ;    D2 =
        MULS.W  #SWidth,D2       ;           # bytes
        BEQ     @5               ;    If EQ, do unpack
        CLR.L   D3               ;    D3 = Temp
@1      MOVE.B  (A0)+,D3         ;    Fetch count byte
        MOVE.B  D3,(A1)+         ;    Store in buffer top
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
         BMI     @3          ;       < 0 means repeated byte
         MOVE.W  D3,D4       ;       Copy count byte
@2       MOVE.B  (A0)+,(A1)+ ;       Move to
         DBRA    D4,@2       ;               buffer top
         BRA     @4          ;       Branch to end of loop
@3       NEG.B   D3          ;       Correct count
         BMI     @1          ;       Ignore hex 80 bytes
         MOVE.B  (A0)+,(A1)+ ;       Move data byte
@4       ADDQ.W  #1,D3       ;       Bump count (Un-DBRA Adjust)
         SUB.L   D3,D2       ;       Subtract item count
         BGT     @1          ;       > 0 means go get next
         BNE     Void        ;       A terrible error!
;
;        Now unpack file buffer to screen
;
@5       SUBA.L  A2,A1       ;       Compute
         MOVE.W  A1,FBStart(A5) ;            top byte count
         MOVE.L  A0,-(SP)    ;       Save A0 for a bit
         MOVE.L  ScrBuf(A5),A0 ;     A0 -> Screen buffer handle
         _HLock              ;       Lock it
         MOVE.L  (A0),A1     ;       A1 -> Screen buffer
         MOVE.L  (SP)+,A0    ;       Restore A0
         MOVE.W  SSize(A5),D0 ;      D0 = # rows to unpack
         MOVE.L  A1,A6       ;       Calculate
         ADDA.L  SBufSiz(A5),A6 ;              limit
         BSR     UnPack      ;       Unpack to screen buffer
         SUB.L   A4,A0       ;       Size of start + unpacked area
         MOVE.L  A0,D0       ;       Move to data register
         SUB.W   D0,FBEnd(A5) ;      Result is new size
         MOVE.L  ScrBuf(A5),A0 ;     Unlock
         _HUnLock            ;               all
         MOVE.L  FileBuf(A5),A0 ;                locked
         _HUnLock            ;                             handles
         BSR     PackU       ;       Make UnDoBuf correct now
         POPALL              ;       Pop the registers
         BRA     ReDraw      ;       Redraw the screen
;
;
;        Draw private data area
;
Point1:  DS.L    1           ;       Temp point
Point2:  DS.L    1           ;       Second temp point
TmpRect  DS.L    2           ;       Temporary rectangle
OldRect  DS.L    2           ;       Old rectangle for boxes, ovals
NewRect  DS.L    2           ;       New rectangle for boxes, ovals
OldPen:  DS.W    9           ;       Pen state structure
SBitMap  DS.L    1           ;       Bitmap structure for screen buffer
SRBytes  DS.W    1           ;               rowBytes word
SBounds  DS.W    4           ;               bounds rectangle
SrcRect  DS.W    4           ;       Source rectangle
PortAdr  DS.L    1           ;       GrafPort address
ScrType  DS.W    1           ;       Remember original type here
BuffCnt  DS.W    1           ;       # characters in buffer
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
BuffPtr DS.L    1              ;       -> Next free slot in buffer
TBufCnt DS.B    1              ;       Count byte for Draw string
TBuffer DS.B    TBuffSize      ;       Buffer itself
```

Listing E.11 – *The MacDoodleDraw.asm file (continued)*

```
;
;       MacDoodleMenu.asm.  This file contains the menu event handlers for the
;       MacDoodle program.
;
;       Revision history:
;
;       <01>    31-Aug-85       Install changes for Exercises 6.2 and 6.3
;
;       There are 7 menus, as follows:
;
;       Apple    File    Edit    Draw        PenSize     Font      Style
;       -----   -------  -----  ---------   ---------   -------   --------
; 1              New     UnDo   Lines         1 x 1     Chicago   Plain
; 2              Open    ----   Freehand      2 x 2     New York  Bold
; 3              Save    Cut    Boxes         3 x 3     Geneva    Italic
; 4              Save As Copy   Ovals         4 x 4     Monaco    Underline
; 5              ------- Paste  Text          5 x 5     Venice    Outline
; 6              Print          ------        6 x 6     London    Shadow
; 7              -------        Eraser        7 x 7     Athens    Compressed
; 8              Quit           ------        8 x 8     SanFran   Expanded
; 9                             Erase Screen  9 x 9     Toronto   ----------
;10                                          10 x 10    Cairo      9 Point
;11                                          11 x 11    LosAng    10 Point
;12                                          12 x 12    Seattle   12 Point
;13                                          13 x 13              14 Point
;14                                          14 x 14              18 Point
;15                                          15 x 15              20 Point
;16                                          16 x 16              24 Point
;17                                                               36 Point
;18                                                               72 Point
;
;       Note that the lines drawn between items occupy a menu choice number
;       that can never be selected.
;
PlainSel EQU    1              ;       Plain text selection
BoldSel  EQU    2              ;       First text attr selection
CompSel  EQU    7              ;       Compressed selection
ExpSel   EQU    8              ;       Last text attr selection
SizeBase EQU    10             ;       First size entry
;
;
        INCLUDE MACLIB.ASM     ;       Include macro library
        INCLUDE MacDoodle.Equ  ;       Include MacDoodle equates file
        XDEF    MenuInit       ;       Initializer routine
```

Listing E.12 – *The MacDoodleMenu.asm file*

```
        XDEF    CloseEvt        ;       Window-close event
        XDEF    MenuEvt         ;       Menu-choice event
        XREF    CloseText       ;       Text-close routine
        XREF    ReDraw          ;       Screen-refresh routine
        XREF    EScreen         ;       Screen-erase routine
        XREF    PrDraw          ;       Print CopyBits routine
        XREF    RePack          ;       Pack all of buffer together
        XREF    UnBuf           ;       Separate packed buffer into 3 pieces
        XREF    ReSize          ;       Resize-window routine
        XREF    UnPack          ;       Unpack buffer to bitmap routine
;
;       MenuInit routine.  This routine is called before we start taking events.
;       Select default Draw and PenSize menu choices.
;
        .TRAP   _NewHandle      $A122 ; Newhandle trap word
        .TRAP   _HLock          $A029 ; Lock dynamic area
        .TRAP   _HUnLock        $A02A ; Unlock dynamic area
        .TRAP   _GetFName       $A8FF ; Get font name trap
        .TRAP   _RealFont       $A902 ; Real font trap
MenuInit
        MOVE.L  #FileSize,D0    ;       File buffer size
        MOVE.L  D0,FBufSiz(A5)  ;       Remember for later
        _NewHandle              ;       Get buffer
        MOVE.L  A0,FileBuf(A5)  ;       Store handle
        TST.L   FileBuf(A5)     ;       Really got it?
        BEQ     MemError        ;       No, complain
        MOVE.L  #UnDoSize,D0    ;       Allocate
        MOVE.L  D0,UBufSiz(A5)  ;
        _NewHandle              ;                undo
        MOVE.L  A0,UnDoBuf(A5)  ;                       buffer
        TST.L   UnDoBuf(A5)     ;       Got it?
        BEQ     MemError        ;       No
        MOVE.L  #ScrSize,D0     ;       Get
        MOVE.L  D0,SBufSiz(A5)  ;               screen
        _NewHandle              ;                       buffer
        MOVE.L  A0,ScrBuf(A5)   ;       Set buffer handle
        TST.L   ScrBuf(A5)      ;       Got it?
        BEQ     MemError        ;       No
        MOVE.L  #HdrSize,D0     ;       Get
        MOVE.L  D0,HBufLen(A5)  ;               header
        _NewHandle              ;                       buffer
        MOVE.L  A0,HdrBuf(A5)   ;       Set buffer handle
        TST.L   HdrBuf(A5)      ;       Got it?
        BEQ     MemError        ;       No
;
;       Disable menu entries for all fonts not in the system file.
;
        LEA     Fonts,A3        ;       A3 -> Font table
        MOVEQ.L #1,D3           ;       D3 counts menu items
RFontloop                       ;       Top of loop
        MOVE.W  (A3)+,D0        ;       Fetch font ID
        BLT     RFDone          ;       LT => End of list
        MOVE.W  D0,-(SP)        ;       Push font ID
        PEA     FName(A5)       ;       -> Return name space
        _GetFName               ;       Get the font name
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
        TST.B   FName(A5)          ;     Any bytes in string?
        BNE     WasFont            ;     Yes, font must exist
        DISABLE #mFont,D3          ;     Disable the entry
WasFont ADDQ.L  #1,D3              ;     Bump entry number
        BRA     RfontLoop          ;     Loop till fonts exhausted
RFDone:                            ;     Here when all done
        CLR.B   FName(A5)          ;     Initialize file name to null
;
;       Initialize other menu data
;
        MOVE.W  #2,DrwType(A5)     ;     Set DrawType variable
        MOVE.W  #2,PenType(A5)     ;     Set pen type for cursor
        MOVE.W  #1,PenSize(A5)     ;     Set pen size
        CHECK   #mDraw,#2          ;     Check freehand
        CHECK   #mPen,#1           ;     Check single
        STYLE   #mStyle,#7,#mCmpress;    Change compressed style
        STYLE   #mStyle,#8,#mExpand;     Change expanded style
        BSR     Plain              ;     Make text plain
        MOVE.W  #SizeBase,D1       ;     Set size to minimum
        MOVE.W  D1,TextSel(A5)     ;     Set up so UNCHECK won't bomb
        BSR     TSize              ;     Go set size
        MOVE.W  #1,D1              ;     Check first font entry
        MOVE.W  D1,FontNum(A5)     ;     Set up so UNCHECK won't bomb
        BSR     SetFont            ;     Set it
        MOVE.W  #1,D1              ;     Go init cursor
        BSR     SetPen             ;     Choose single pixel pen
        BSR     New                ;     Clear out junk
;
;       Now process application parameters, if any
;
        MOVE.L  ParmHdl(A5),A0     ;     A4 ->
        _HLock                     ;<01>        Locked
        MOVE.L  (A0),A4            ;                    parameter area
        MOVE.W  PCount(A4),FCount(A5);<01> Any files there?
        BNE     @1                 ;     Yes, go process
        RTS                        ;     Done, back to event handler
@1      BSR     APOpen             ;<01> Open the file
        MOVE.L  ParmHdl(A5),A4     ;     Reload
        MOVE.L  (A4),A4            ;            parameter handle
        TST.W   PMsg(A4)           ;     Open or print
        BNE     @2                 ;<01> NE => Print
        RTS                        ;<01> Otherwise, we're done
@2      MOVE.L  A4,LastF(A5)       ;<01> Remember last file done
@3      BSR     xPrint             ;<01> Print file
        SUB.W   #1,FCount(A5)      ;<01> Decrement number of files
        BEQ     PDone              ;<01> EQ => Done
        MOVE.L  LastF(A5),A4       ;<01> Reload last file pointer
        LEA     PFName(A4),A0      ;<01> A0 -> Old name
        CLR.W   D0                 ;<01> Fetch
        MOVE.B  (A0),D0            ;<01>        count byte
        ADD.W   #2,D0              ;<01> Round
        BCLR    #0,D0              ;<01>        to word
        LEA     -4(A0,D0.W),A4     ;<01> Adjust A4
        BSR     APOpen             ;<01> Open the next one
        BRA     @3                 ;<01> and loop
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
PDone   EXIT                        ;       and quit
;
;       APOpen routine.  Opens a file from application parameter area.
;
APOpen  LEA     PFName(A4),A0       ;       A0 -> Finder-supplied name
        LEA     FName(A5),A1        ;       A1 -> File name area
        CLR.W   D0                  ;       D0 =
        MOVE.B  (A0),D0             ;               Count byte
@2      MOVE.B  (A0)+,(A1)+         ;       Move to
        DBRA    D0,@2               ;               expected area
        WTITLE  Fname(A5)           ;<01>   Display file name
        MOVE.W  PVref(A4),VolRef(A5);       Save volume reference #
        OPEN    FName(A5),DEFAULT,PVRef(A4); Open file
        MOVE.W  D0,D2               ;       Save file number
        BPL     @4                  ;       If OK
        StopAlert #OErr             ;       Tell user we couldn't do it
@3      EXIT                        ;       then die
@4      MOVE.W  VolRef(A5),D1       ;<01>   Set D1 to vol ref
        BSR     OpenOK              ;       Complete the open
        TstFlag F_File              ;       Did it go ok?
        BEQ     @3                  ;       EQ => No, alert already done
        RTS                         ;<01>   Back to caller
MemError                            ;       Here if out of memory
        StopAlert #MemErr           ;       Croak
        EXIT                        ;
FCount  DS.W    1                   ;<01>   Space for file count
LastF   DS.L    1                   ;<01>   -> Last file done
;
;       Close event.  This routine is called when the user clicks in the window
;       close box.  We treat this exactly like a New choice from the File
;       menu.  This deviates from standard Macintosh behavior in that we don't
;       close the window.
;
CloseEvt:                           ;       Called from the event loop
        MOVE.L  #$20001,D1          ;       Fake out menu handler
;
;       Menu event handler.  Entered with constant 10 (hex) in D0.W
;       and the menu number in D1.W.  The high word of D1 has the selection
;       number.  The menu number is in the range 2-n and the choice number
;       is in the range 1-n. (RTL handles menu #1, the Apple menu.)
;
MenuEvt:                            ;       Here for a menu choice
        TstFlag F_Text              ;       Text open?
        BEQ     @1                  ;       No, continue
        JSR     CloseText           ;       Yes, close first
@1:     MOVE.W  D1,D2               ;       Copy menu choice to D2
        SUBQ.W  #1,D2               ;       Force choice zero relative
        SWAP    D1                  ;       Menu number to D1.W
        SUBQ.W  #2,D1               ;       Force zero relative
        MOVE.W  D1,D3               ;       Copy to D3
        SWAP    D1                  ;       Choice back to D1.W
        MULU    #nMenu-1,D2         ;       Multiply D2 by # menus
        ADD.W   D2,D3               ;       Add in menu choice
        LSL.W   #1,D3               ;       Multiply by entry size (word)
        MOVE.W  JBASE(D3.W),D3      ;       Fetch offset word
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
        JMP     JBASE(D3.W)     ;       Go to routine
Void:   StopAlert #ProgramErr   ;       Here if a real problem
        EXIT                    ;       Fade away
;
;       Define menu jump table
;
JBASE:                          ;       Offset for jump
;
;               File     Edit     Draw     PenSize  Font      Style
;               ----     ----     ----     -------  ------    -------
        JTAB    New,     Undo,    Lines,   SetPen,  SetFont,  Plain    ; 1
        JTAB    xOpen,   Void,    FreeH,   SetPen,  SetFont,  TStyle   ; 2
        JTAB    Save,    Cut,     Boxes,   SetPen,  SetFont,  TStyle   ; 3
        JTAB    SaveAs,  Copy,    Ovals,   SetPen,  SetFont,  TStyle   ; 4
        JTAB    Void,    Paste,   Text,    SetPen,  SetFont,  TStyle   ; 5
        JTAB    xPrint,  Void,    Void,    SetPen,  SetFont,  TStyle   ; 6
        JTAB    Void,    Void,    Eraser,  SetPen,  SetFont,  TCmpress ; 7
        JTAB    xQuit,   Void,    Void,    SetPen,  SetFont,  TExpand  ; 8
        JTAB    Void,    Void,    EraseA,  SetPen,  SetFont,  Void     ; 9
        JTAB    Void,    Void,    Void,    SetPen,  SetFont,  TSize    ;10
        JTAB    Void,    Void,    Void,    SetPen,  SetFont,  TSize    ;11
        JTAB    Void,    Void,    Void,    SetPen,  SetFont,  TSize    ;12
        JTAB    Void,    Void,    Void,    SetPen,  Void,     TSize    ;13
        JTAB    Void,    Void,    Void,    SetPen,  Void,     TSize    ;14
        JTAB    Void,    Void,    Void,    SetPen,  Void,     TSize    ;15
        JTAB    Void,    Void,    Void,    SetPen,  Void,     TSize    ;16
        JTAB    Void,    Void,    Void,    Void,    Void,     TSize    ;17
        JTAB    Void,    Void,    Void,    Void,    Void,     TSize    ;18
;
;       New menu selection.  Check for modifications and ask the
;       user if he wants to save them.  Do a save if requested.
;       Then obliterate (zero) the buffers and display.
;
New:                            ;
        BSR     ChkSave         ;       Check for save first
        MOVE.L  HdrBuf(A5),A0   ;       A0 ->
        MOVE.L  (A0),A0         ;               header buffer
        MOVE.L  HBufLen(A5),D0  ;       D0 = Byte Count
        SUBQ.L  #1,D0           ;       Adjust for DBRA
@1:     CLR.B   (A0)+           ;       Clear out
        DBRA    D0,@1           ;                       header buffer
        MOVE.L  FileBuf(A5),A4  ;       A4 ->
        MOVE.L  (A4),A4         ;               File buffer
        MOVE.L  #FLength*2,D3   ;       Buffer length
        MOVE.L  D3,D4           ;       Copy to D4
        MOVE.W  D3,D0           ;       and to D0
        LSR.W   #1,D0           ;       Adjust D0 for word count
        SUBQ.W  #1,D0           ;       and DBRA
        SUB.L   FBufSiz(A5),D4  ;       Compute
        NEG.L   D4              ;               buffer
        LEA     0(A4,D4.L),A4   ;                       offset
        MOVE.L  A4,A0           ;       Copy to A0
@2:     MOVE.W  #NullPat,(A0)+  ;       Fill buffer with
        DBRA    D0,@2           ;       Compressed zero pattern
```

Listing E.12 – *The MacDoodleMenu.asm file (continued)*

```
        ClrFlag F_File          ;       No file name for buffer now
        WTitle  '(No File)'     ;       Set the window title
        BRA     DoUpdate        ;       Otherwise, it's just like a finished read
;
;       xOpen routine.  (The "x" is necessary to prevent assembler
;       confusion between the Open label and the OPEN macro.)
;
;       Open a new file and read it into the buffer.
;
xOpen                           ;
        BSR     ChkSave         ;       Check for unsaved changes
        POPEN   Ptype,DEFAULT,Fname(A5),DEFAULT;       open new file
        MOVE.W  D0,D2           ;       Save file reference number
        BPL     OpenOK          ;       If > 0, OK
        CMP.W   #Cancelled,D0   ;       Cancelled?
        BNE     @1              ;       No
        RTS                     ;       Yes, just return
@1      MOVE.L  #OErr,D0        ;       Indicate open error
        BRA     Bail            ;       Bail out
OpenOK  SCursor #Watch          ;       Indicate patience required
        MOVE.W  D1,VolRef(A5)   ;       Save volume reference number
        GETFINFO Fname(A5),InfoBuf(A5),VolRef(A5); Get file info
        LEA     InfoBuf(A5),A0  ;       A0 -> File Information
        MOVE.L  DFSize(A0),D4   ;       Copy file size to D4
        SUB.L   HBufLen(A5),D4  ;       Subtract file header size
        MOVE.L  HdrBuf(A5),A0   ;       A0 = Buffer handle
        MOVE.L  (A0),A4         ;       Dereference
        MOVE.L  HBufLen(A5),D3  ;       Byte count to D3
        BSR     IssueRd         ;       Go read it
        MOVE.L  D4,D3           ;       Main byte count to D3 now
        CMP.L   FBufSiz(A5),D3  ;       Enough room in buffer?
        BLT     BufOK           ;       Yes
        MOVE.L #MemErr,D0       ;       No, notify user
        BRA     OBail           ;       Bail out
BufOK                           ;
        MOVE.L  FileBuf(A5),A0  ;       A0 -> Buffer pointer
        MOVE.L  (A0),A4         ;       A4 -> Buffer
        SUB.L   FBufSiz(A5),D4  ;       Calculate
        NEG.L   D4              ;                    offset to bottom
        LEA     0(A4,D4.L),A4   ;       A4 -> File offset in buffer
        BSR     IssueRd         ;       Do the read
        CLOSE   D2              ;       Close the open file
        WTitle  FName(A5)       ;       Set the window title
        SetFlag F_File          ;       Now have a file in the buffer
DoUpdate:                       ;       Now get ready to update screen
        CLR.W   FBStart(A5)     ;       No bytes in buffer beginning
        CLR.W   SRow(A5)        ;       Screen buffer is row #0
        MOVE.W  D3,FBEnd(A5)    ;       Set up proper # of bytes
        JSR     UnBuf           ;       Convert to screen image
        ClrFlag F_Change        ;       No changes now
        MOVE.L  SSize(A5),D1    ;       Set size variable
        JMP     ReSize          ;       Reinit screen
;
;       IssueRd routine.  This routine issues a read into a dynamic
;       buffer.  Locks the buffer beforehand.
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
;
;       Entry conditions:
;
;               D2 = I/O reference number for file
;               A0 = Buffer handle
;               A4 ->Buffer address for transfer
;               D3 = Desired byte count
;
;       Exit conditions:
;
;               (none)
;
IssueRd                                 ;
        MOVE.L  A0,-(SP)                ;       Save handle
        _HLock                          ;       Lock it
        READ    D2,(A4),D3              ;       Read desired data
        MOVE.L  (SP)+,A0                ;       Restore handle address
        MOVE.L  D0,-(SP)                ;       Save returned byte count
        _HUnLock                        ;       Unlock handle
        CMP.L   (SP)+,D3                ;       Read OK?
        BNE     @1                      ;       No
        RTS                             ;       Yes, just return
@1      ADDQ.L  #4,SP                   ;       Pop off return address
        MOVE.L  #ReadErr,D0             ;       Set error code
        BRA     OBail                   ;       Bail out
;
;       ChkSave routine.  This code checks for modification of the
;       buffer and prompts the user to see if he wants to save
;       it.
;
ChkSave:                                ;
        TstFlag F_Change                ;       Any changes?
        BEQ     NoSave                  ;       No, just return
        Dialog  #Saveit                 ;       Do dialog
        CMP.W   #DoSave,D0              ;       Did user ask to save?
        BNE     NoSave                  ;       No, just quit
Save    TstFlag F_File                  ;       File in buffer?
        BEQ     SaveAs                  ;       No, need file name from user
        CREATE  Fname(A5),DEFAULT,VolRef(A5); Create the file
        MOVE.W  D0,D2                   ;       Save file ref number
        BMI     CFileErr                ;       If < 0, couldn't create
DoWrite SetFlag F_File                  ;       Now have a file in buffer
        SCursor #Watch                  ;       Set the cursor to the watch
        MOVE.L  HdrBuf(A5),A0           ;       Fetch buffer handle
        MOVE.L  (A0),A4                 ;       Dereference handle
        MOVE.L  HBufLen(A5),D3          ;       D3 = Desired byte count
        BSR     IssueWr                 ;       Go issue write
;
;       Now write out the packed data for the file.
;
HWriteOK                                ;       Here to write buffer
        JSR     RePack                  ;       Pack to single buffer
        MOVE.L  FileBuf(A5),A0          ;       A0 -> Handle
```

Listing E.12 – *The MacDoodleMenu.asm file (continued)*

```
        MOVE.L  (A0),A4         ;       A4 -> Buffer
        CLR.L   D3              ;       Clear D3 high word
        MOVE.W  FBEnd(A5),D3    ;       D3.L =  Ending byte count
        MOVE.L  D3,D4           ;       Copy buffer length
        SUB.L   FBufSiz(A5),D4  ;       Calculate
        NEG.L   D4              ;               offset to bottom
        LEA     0(A4,D4.L),A4   ;       A4 -> File offset in buffer
        BSR     IssueWr         ;       Do the write
        CLOSE   D2              ;       Close the file
        JSR     UnBuf           ;       Undo the repack
        ClrFlag F_Change        ;       Indicate file saved
        GETFINFO Fname(A5),InfoBuf(A5),VolRef(A5); Get file info
        LEA     InfoBuf(A5),A0  ;       A0 -> Buffer
        MOVE.L  #MPType,FFType(A0);     Set file type
        MOVE.L  #DAppl,FFAppl(A0);      Set file application
        SETFINFO Fname(A5),InfoBuf(A5),VolRef(A5); Update disk
NoSave  RTS                     ;       Done

;
;       IssueWr routine.  This routine issues a write from a dynamic
;       buffer.  Locks the buffer beforehand.
;
;       Entry conditions:
;
;               D2 = I/O reference for file
;               A0 = Buffer handle (for locking)
;               A4 ->Buffer address (for writing)
;               D3 = Byte count
;
;       Exit conditions:
;
;               (None)
;
IssueWr MOVE.L  A0,-(SP)        ;       Save the handle
        _HLock                  ;       Lock down the buffer
        _WRITE  D2,(A4),D3      ;       Do request
        MOVE.L  (SP)+,A0        ;       Restore handle address
        MOVE.L  D0,-(SP)        ;       and keep byte count
        _HUnLock                ;       Unlock it
        CMP.L   (SP)+,D3        ;       All written?
        BNE     @1              ;       No, bail out
        RTS                     ;       Return to caller
@1      ADDQ.L  #4,SP           ;       Pop return address
        MOVE.L  #Werr,D0        ;       Tell the user we didn't
        BRA     OBail           ;       close file and exit
;
;       SaveAs routine.  Same as Save, except that the file name is
;       typed by the user.
;
SaveAs  PCREATE 'Save As',Fname(A5),Fname(A5),DEFAULT
        MOVE.L  D0,D2           ;       Save file ref number
        BLT     CFileErr        ;       Error, quit
        MOVE.W  D1,VolRef(A5)   ;       Save volume reference number
```

Listing E.12 – *The MacDoodleMenu.asm file (continued)*

```
        WTITLE  Fname(A5)       ;       Set title to new name
        BRA     DoWrite         ;       Now go write the file
CFileErr                        ;       Can't create file
        MOVE.L  #Cerr,D0        ;       Set error code
        BRA     Bail            ;       Quit
;
;       Alert handling routines.  Branch to OBail if file is open,
;       Bail if not.
;
;       Entry conditions:
;
;               D0 = Alert template number
;               D2 = File reference number (OBail only)
;
;       Exit conditions:
;
;               (None)
;
OBail   CLOSE     D2            ;       Open bail out routine
Bail    SCursor   #Arrow        ;       Set the cursor to the arrow
        StopAlert D0            ;       Do the alert
        RTS                     ;       Return to caller
;
;       Print routine.  This code prints the file found in FileBuf on the
;       printer.  We use the generic code supplied with the MDS system here
;       (in file PRLink.rel).
;
;       We use the bitmap printing feature.
;
xPrint                          ;       Here when print selected
        SCursor #Watch          ;       Set cursor to watch
        JSR     PrDrvrOpen      ;       Open the printer driver
;
;       Call print draw routine (in MacDoodleDraw.asm)
;
        JSR     PrDraw          ;       Do print drawing
        JSR     PrDrvrClose     ;       Close the printer driver
        RTS                     ;       Return to caller
;
;       Quit routine.  This code checks for unsaved changes and exits to the
;       Finder.
;
xQuit:  BSR     ChkSave         ;       Check on unsaved changes
        EXIT                    ;       Back to the Finder
;
;       UnDo routine.  Replaces screen buffer with unpacked UnDoBuf.
;
;       Entry conditions:
;
;               (none)
;
;       Exit conditions:
;
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
;               (none)
Undo
        MOVE.L  UndoBuf(A5),A0  ;       A0 ->
        MOVE.L  (A0),A0         ;               Undo buffer
        MOVE.L  ScrBuf(A5),A1   ;       A1 ->
        MOVE.L  (A1),A1         ;               Screen buffer start
        MOVE.W  SSize(A5),D0    ;       D0 = number of rows to unpack
        MOVE.L  A1,A6           ;       Calculate
        ADDA.L  SBufSiz(A5),A6  ;               limit
        BSR     UnPack          ;       Unpack data to screen buffer
        JMP     ReDraw          ;       Now redraw screen
Cut
Copy
Paste
        RTS                     ;       Return to caller
;
;       Draw menu handlers.  These routines set the value of
;       global DrwType to the menu selection and the
;       value of PenType to the proper cursor.
;
Lines
FreeH
Boxes
Ovals
Text
Eraser
        UNCHECK #mDraw,DrwType(A5);     Uncheck old item
        CHECK   #mDraw,D1       ;       Check new item
        MOVE.W  D1,DrwType(A5)  ;       Save item number
        SUBQ.W  #1,D1           ;       Convert to zero-relative
        MOVE.B  PenTbl(D1),D0   ;       Fetch pen byte
        EXT.W   D0              ;       Out to word
        MOVE.W  D0,PenType(A5)  ;       Set pen type
        MOVE.B  CurTbl(D1),D0   ;       Fetch cursor
        BEQ     SetCursor       ;       None, compute it
        MOVE.W  D0,Cursor(A5)   ;       Set cursor
        RTS                     ;       and return
PenTbl  DC.B    2               ;       Lines
        DC.B    2               ;       FreeHand
        DC.B    0               ;       Boxes
        DC.B    0               ;       Ovals
        DC.B    0               ;       Text
        DC.B    0               ;       (Unused)
        DC.B    1               ;       Eraser
CurTbl  DC.B    0               ;       Lines
        DC.B    0               ;       FreeHand
        DC.B    Cross           ;       Boxes
        DC.B    Cross           ;       Ovals
        DC.B    IBeam           ;       Text
        DC.B    0               ;       (Unused)
        DC.B    0               ;       Eraser
;
;       Erase-screen routine.  Calls routine EScreen in
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
;       MacDoodleDraw to erase window area in document.
;
EraseA
        BSR     EScreen         ;       In draw code
        RTS                     ;       Back to event code


;
;       SetPen routine.  This routine sets the value of the
;       PenSize variable and alters the cursor.
;
SetPen  MOVE.W  Pensize(A5),D2  ;       Fetch old item
        UNCHECK #mPen,D2        ;       Uncheck old item
        CHECK   #mPen,D1        ;       Check new item
        MOVE.W  D1,PenSize(A5)  ;       Store pen size
        MOVE.W  DrwType(A5),D0  ;       Fetch draw type
        SUBQ.W  #1,D0           ;       Convert to zero-relative
        MOVE.B  CurTbl(D0),D0   ;       Need new cursor?
        BEQ     SetCursor       ;       EQ => Yes, do it
        RTS                     ;       Otherwise, return
SetCursor                       ;       Here to compute cursor
        MOVE.W  PenType(A5),D0  ;       Fetch pen type
        LSL     #4,D0           ;       Shift left 4
        MOVE.W  PenSize(A5),D3  ;       D3 =
        SUBQ.W  #1,D3           ;               Adjusted pen size
        ADD.W   D3,D0           ;       Cursor resource ID
        MOVE.W  D0,Cursor(A5)   ;       Set cursor
        RTS                     ;       Back for more
;
;       SetFont routine.  Handles a font menu selection.
;
SetFont:                        ;
        UNCHECK #mFont,FontSel(A5);     Uncheck old selection
        CHECK   #mFont,D1       ;       Check new selection
        MOVE.W  D1,FontSel(A5)  ;       Remember selection #
        SUBQ.W  #1,D1           ;       Convert to font number
        ADD.W   D1,D1           ;       Double for word index
        MOVE.W  Fonts(D1),FontNum(A5);  Set new font number
        LEA     Points,A3       ;       A3 -> Table of point sizes
        MOVE.W  #SizeBase,D4    ;       D4 = First size selection
FindPoint                       ;       Loop to find point size
        CLR.W   -(SP)           ;       Room for result
        MOVE.W  FontNum(A5),-(SP);      Push font number
        MOVE.W  (A3),-(SP)      ;       and size
        _RealFont               ;       Is it
        TST     (SP)+           ;               real or is it...?
        BNE     @1              ;       NE => Real, continue
        STYLE   #mStyle,D4,#0   ;       Clear style word
        BRA     @2              ;       Branch to end of loop code
@1      STYLE   #mStyle,D4,#mOutline;   Set outline mode
@2      ADDQ.W  #1,D4           ;       Bump menu selection
        TST     (A3)+           ;       Bump pointer
        BNE     FindPoint       ;       NE => Keep looking
```

Listing E.12 – *The MacDoodleMenu.asm file (continued)*

```
         RTS                       ;       If none real, leave alone
Fonts:   DC.W    Chicago,NewYork,Geneva,Monaco,Venice,London,Athens
         DC.W    SanFran,Toronto,Cairo,LosAng,Seattle,-1
;
;        Plain routine.  Sets Plain text mode.
;
Plain:   CLR.W   TextFlg(A5)       ;       Clear all attr bits
         MOVE.W  #mStyle,D2        ;       D2 = Menu number
         MOVE.W  #BoldSel,D3       ;       D3 = First style selection
PLoop:                             ;       Loop till done
         UNCHECK D2,D3             ;       Uncheck a bit
         ADDQ.W  #1,D3             ;       Bump D3
         CMP.W   #ExpSel,D3        ;       Last one?
         BLE     PLoop             ;       No, continue
         CHECK   #mStyle,#PlainSel;        Now check Plain
         RTS                       ;       and return
;
;        TStyle routine.  Add a style bit to TextFlg word.
;
TStyle:  UNCHECK #mStyle,#PlainSel;        Uncheck plain
         CHECK   #mStyle,D1        ;       Check whatever
         SUB.W   #BoldSel,D1       ;       Subtract base
         BSET.B  D1,TextFlg+1(A5);         Set the bit in memory
         RTS                       ;       Done
;
;        Compress and Expand are different.  Treat them as
;        mutually exclusive.
;
TCmpress                           ;       Compress selected
         BCLR.B  #Expand,TextFlg+1(A5);    Clear expanded bit
         UNCHECK #mStyle,#ExpSel ;         Uncheck it
         BRA     Tstyle            ;       Otherwise, its the same
TExpand                            ;       Expanded selected
         BCLR.B  #Cmpress,TextFlg+1(A5); Clear Compressed bit
         UNCHECK #mStyle,#CompSel;         Uncheck it
         BRA     Tstyle            ;       Go select it
;
;        TSize routine.  Set text size in points.
;
TSize:   UNCHECK #mStyle,TextSel(A5);      Uncheck previous
         CHECK   #mStyle,D1        ;       Check current
         MOVE.W  D1,TextSel(A5)    ;       Remember for next time
         SUB.W   #SizeBase,D1      ;       Subtract size base
         ADD.W   D1,D1             ;       Double for word index
         MOVE.W  Points(D1),TextSiz(A5); Remember size in points also
         RTS                       ;       Done
Points   DC.W    9,10,12,14,18,20,24,36,72; Point sizes
         DC.W    0                 ;       End of table flag
```

***Listing E.12** – The MacDoodleMenu.asm file (continued)*

```
;
;       MacDoodle RTL.  Configures the RTL code for use with MacDoodle.
;
TEXTRTL EQU     -1              ;       Not text RTL
CONTROL EQU      0              ;       Does have window controls
RESFILE EQU     -1              ;       No separate resource file
Include RTLcode.Asm             ;       Include RTL
        END
```

Listing E.13 – *The MacDoodleRTL.asm file*

```
*
* This is the resource file for the MacDoodle program
*

!MacDoodle

*
*       Bundle resource definition.  Specifies use of the
*       MacDoodle private icon to the Finder
*
Type BNDL
    ,128
DOOD 0
ICN#
0 128 1 129
FREF
0 128 1 129

Type FREF
    ,128
APPL 0          ;; Application maps to icon #0
    ,129
PNTG 1          ;; Document maps to icon #1

*
*       Version number resource
*
Type DOOD = GNRL
     ,0
.P
MacDoodle Program Version 1.0

*
*       MacDoodle icon definition
*

Type ICN# = GNRL
      ,128
.H
```

Listing E.14 – *The MacDoodle.R file*

```
00018000 00018000 00018000 7FFFFFFE
40000002 40000002 40200002 40500002
40500002 40500002 40508002 40500002
40500002 40708002 40C88002 41488402
42494802 444E3002 40000002 7FFFFFFE
00C18300 01818180 030180C0 06018060
0C018030 18018018 3001800C 60018006
C0018003 80018001 00000000 00000000
*
*       Icon mask
*
00018000 00018000 00018000 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
7FFFFFFE 7FFFFFFE 7FFFFFFE 7FFFFFFE
00FFFF00 01FFFF80 03FFFFC0 07FFFFE0
0FFFFFF0 1FFFFFF8 3FFFFFFC 7FFFFFFE
FFFFFFFF FFFFFFFF 00000000 00000000


*
* Document icon definition
*

Type ICN# = GNRL
     ,129
.H
1FFFF800 10000C00 10000A00 10000900
10000880 10000840 10000820 10000FF0
10000010 10018010 10018010 17FFFFD0
14000050 14100050 14280050 14280050
14280050 14284050 14280050 143C4050
14644050 14A4A450 15271850 14000050
17FFFFD0 10318C10 10618610 10C18310
11818190 130180D0 10000010 1FFFFFF0
*
*       Document icon mask
*
1FFFF800 1FFFFC00 1FFFFE00 1FFFFF00
1FFFFF80 1FFFFFC0 1FFFFFE0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0
1FFFFFF0 1FFFFFF0 1FFFFFF0 1FFFFFF0

*
*       Menu definitions
*

Type MENU
  ,1(4)
\14
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
  About MacDoodle ...
  (-


  ,2(4)
File
  New/N
  Open/R
  Save/W
  Save As ...
(-
  Print
(-
  Quit


  ,3(4)
Edit
  Undo
(-
  Cut/X
  Copy/C
  Paste/V


  ,4(4)
Draw
  Lines
  Freehand
  Boxes
  Ovals
  Text
(-
  Eraser
(-
  Erase Screen


  ,5(4)
PenSize
   1 x 1
   2 x 2
   3 x 3
   4 x 4
   5 x 5
   6 x 6
   7 x 7
   8 x 8
   9 x 9
  10 x 10
  11 x 11
  12 x 12
  13 x 13
  14 x 14
  15 x 15
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
   16 x 16


   ,6(4)
Font
   Chicago
   New York
   Geneva
   Monaco
   Venice
   London
   Athens
   San Francisco
   Toronto
   Cairo
   Los Angeles
   Seattle


   ,7(4)
Style
   Plain/P
   <BBold/B
   <IItalic/I
   <UUnderline/U
   <OOutline/O
   <SShadow/S
   Compressed
   Expanded
(-
    9 Point
   10 Point
   12 Point
   14 Point
   18 Point
   20 Point
   24 Point
   36 Point
   72 Point



Type DLOG
   ,1

100 100 190 450
Visible  NoGoAway
1
0
1


Type DITL
   ,1
3
```

Listing E.14 – *The MacDoodle.R file (continued)*

```

StaticText
15 20 36 350
This is the MacDoodle Program

StaticText
35 20 56 350
By Steve Williams

Button
60 260 80 320
OK



Type DLOG
  ,2

100 100 190 400
Visible  NoGoAway
1
0
2


Type DITL
  ,2
3

Button
60 175 80 225
Yes

Button
60 235 80 285
No

StaticText
15 20 36 325
Save changes before closing?



Type ALRT
   ,129
50 50 160 325
3
5555

Type DITL
   ,3
3

Button
85 205 105 265
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
QUIT

StaticText
20 60 40 225
MAJOR PROGRAM BUG

StaticText
50 20 70 225
This should "Never Happen"


Type ALRT
   ,130
50 50 160 325
4
5555

Type DITL
   ,4
3

Button
85 205 105 265
OK

StaticText
20 60 40 225
Write Error

StaticText
50 20 70 225
On Output File


Type ALRT
   ,131
50 50 160 325
5
5555

Type DITL
   ,5
2

Button
85 205 105 265
OK

StaticText
20 60 40 225
File Create Error


Type ALRT
   ,132
```

Listing E.14 *– The MacDoodle.R file (continued)*

```
50 50 160 325
6
5555

Type DITL
    ,6
2

Button
85 205 105 265
OK

StaticText
20 60 40 225
File Open Error


Type ALRT
    ,134
50 50 160 325
7
5555

Type DITL
    ,7
2

Button
85 205 105 265
OK

StaticText
20 60 40 225
Buffer Overflow Error

Type ALRT
    ,135
50 50 160 325
8
5555

Type DITL
    ,8
2

Button
85 205 105 265
OK

StaticText
20 60 40 225
File Read Error

Type ALRT
    ,136
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
50 50 160 325
9
5555

Type DITL
   ,9
2

Button
85 205 105 265
QUIT

StaticText
20 60 40 225
Insufficient Memory



Type ALRT
   ,137
50 50 160 325
10
5555

Type DITL
   ,10
2

Button
85 205 105 265
OK

StaticText
20 60 40 225
Please select TEXT First!

*
*       Main window definition
*

Type WIND
  ,1(4)

40 5 300 500
Visible GoAway
0
0

*
*       Scroll bar controls for main window
*

Type    CNTL
   ,256(4)
x
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
   0 480 245 495
   Visible
   16
   0
   0 720 0

Type    CNTL
   ,257(4)
x
   245 0 260 480
   Visible
   16
   0
   0 576 0


*
*       1 x 1 eraser cursor
*
Type CURS = GNRL        ;; Cursor type definition
    ,16(4)              ;; Resource ID
.H
*       16 words of cursor pattern
8000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
```

Listing E.14 – The MacDoodle.R file (continued)

```
0000
0000
0000
*       Hot spot point
0000
0000

*
*       2 x 2 eraser cursor
*

Type CURS = GNRL
     ,17
.H
*       16 words of cursor pattern
C000
C000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
*       3 x 3 eraser cursor
*
Type CURS = GNRL
     ,18
.H
*       16 words of cursor pattern
E000
A000
E000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
E000
E000
E000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       4 x 4 eraser cursor
*
Type CURS = GNRL         ;; Cursor type definition
     ,19                 ;; Resource ID
.H
*       16 words of cursor pattern
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
F000
9000
9000
F000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
F000
F000
F000
F000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       5 x 5 eraser cursor
*
Type CURS = GNRL
    ,20
.H
*       16 words of cursor pattern
F800
8800
8800
8800
F800
0000
0000
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
F800
F800
F800
F800
F800
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       6 x 6 eraser cursor
*

Type CURS = GNRL
    ,21
.H
*       16 words of cursor pattern
FC00
8400
8400
8400
8400
FC00
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
*       16 words of cursor mask
FC00
FC00
FC00
FC00
FC00
FC00
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000


*
*       7 x 7 eraser cursor
*

Type CURS = GNRL
        ,22
.H
*       16 words of cursor pattern
FE00
8200
8200
8200
8200
8200
FE00
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
FE00
FE00
FE00
FE00
FE00
FE00
FE00
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000



*
*       8 x 8 eraser cursor
*

Type CURS = GNRL
     ,23
.H
*       16 words of cursor pattern
FF00
8100
8100
8100
8100
8100
8100
FF00
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
FE00
FE00
FE00
FE00
FE00
FE00
FE00
0000
0000
0000
0000
0000
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
0000
0000
*       Hot spot point
0000
0000

*
*       9 x 9 point eraser cursor
*

Type CURS = GNRL        ;; Cursor type definition
     ,24                ;; Resource ID
.H
*       16 words of cursor pattern
FF80
8080
8080
8080
8080
8080
8080
8080
FF80
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
FF80
FF80
FF80
FF80
FF80
FF80
FF80
FF80
FF80
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
*       10 x 10 point eraser cursor
*
Type CURS = GNRL          ;; Cursor type definition
     ,25                  ;; Resource ID
.H
*       16 words of cursor pattern
FFC0
8040
8040
8040
8040
8040
8040
8040
8040
FFC0
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       11 x 11  point eraser cursor
*
Type CURS = GNRL          ;; Cursor type definition
     ,26                  ;; Resource ID
.H
*       16 words of cursor pattern
FFE0
```

Listing E.14 *– The MacDoodle.R file (continued)*

```
8020
8020
8020
8020
8020
8020
8020
8020
8020
FFE0
0000
0000
0000
0000
0000
*       16 words of cursor mask
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       12 x 12 point eraser cursor
*

Type CURS = GNRL          ;; Cursor type definition
    ,27                   ;; Resource ID
.H
*       16 words of cursor pattern
FFF0
8010
8010
8010
8010
8010
8010
8010
8010
8010
```

Listing E.14 – The MacDoodle.R file (continued)

```
8010
8010
FFF0
0000
0000
0000
0000
*       16 words of cursor mask
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       13 x 13 point eraser cursor
*

Type CURS = GNRL           ;; Cursor type definition
     ,28                   ;; Resource ID
.H
*       16 words of cursor pattern
FFF8
8008
8008
8008
8008
8008
8008
8008
8008
8008
8008
8008
FFF8
0000
0000
0000
*       16 words of cursor mask
```

Listing E.14 *– The MacDoodle.R file (continued)*

```
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
0000
0000
0000
*       Hot spot point
0000
0000

*
*       14 x 14 point eraser cursor
*

Type CURS = GNRL        ;; Cursor type definition
     ,29                ;; Resource ID
.H
*       16 words of cursor pattern
FFFC
8004
8004
8004
8004
8004
8004
8004
8004
8004
8004
8004
8004
FFFC
0000
0000
*       16 words of cursor mask
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
0000
0000
*       Hot spot point
0000
0000
*
*       15 X 15 point eraser cursor
*
Type CURS = GNRL          ;; Cursor type definition
     ,30                  ;; Resource ID
.H
*       16 words of cursor pattern
FFFE
8002
8002
8002
8002
8002
8002
8002
8002
8002
8002
8002
8002
8002
FFFE
0000
*       16 words of cursor mask
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
*       Hot spot point
0000
0000
*
*       16 X 16 point eraser cursor
*
Type CURS = GNRL        ;; Cursor type definition
     ,31                ;; Resource ID
.H
*       16 words of cursor pattern
FFFF
8001
8001
8001
8001
8001
8001
8001
8001
8001
8001
8001
8001
8001
8001
FFFF
*       16 words of cursor mask
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
*       Hot spot point
0000
0000
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
*
*       Single point pen cursor
*

Type CURS = GNRL          ;; Cursor type definition
     ,32                  ;; Resource ID
.H
*       16 words of cursor pattern
8000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       2 x 2 point pen cursor
*
*

```

Listing E.14 – *The MacDoodle.R file (continued)*

```
Type CURS = GNRL        ;; Cursor type definition
     ,33                ;; Resource ID
.H
*       16 words of cursor pattern
C000
C000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       3 x 3 pen cursor
*

Type CURS = GNRL
     ,34
.H
*       16 words of cursor pattern
E000
```

Listing E.14 *– The MacDoodle.R file (continued)*

```
E000
E000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000


*
*       4 x 4 pen cursor
*

Type CURS = GNRL          ;; Cursor type definition
     ,35                  ;; Resource ID
.H
*       16 words of cursor pattern
F000
F000
F000
F000
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       5 x 5 pen cursor
*
Type CURS = GNRL
      ,36
.H
*       16 words of cursor pattern
F800
F800
F800
F800
F800
0000
0000
0000
0000
0000
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
0000
0000
*        16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*        Hot spot point
0000
0000

*
*        6 x 6 pen cursor
*

Type CURS = GNRL
     ,37
.H
*        16 words of cursor pattern
FC00
FC00
FC00
FC00
FC00
FC00
0000
0000
0000
0000
0000
0000
0000
0000
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       7 x 7 pen cursor
*
Type CURS = GNRL
    ,38
.H
*       16 words of cursor pattern
FE00
FE00
FE00
FE00
FE00
FE00
FE00
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       8 x 8 pen cursor
*
Type CURS = GNRL
     ,39
.H
*       16 words of cursor pattern
FF00
FF00
FF00
FF00
FF00
FF00
FF00
FF00
0000
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       9 x 9 point pen cursor
*
Type CURS = GNRL        ;; Cursor type definition
     ,40                ;; Resource ID
.H
*       16 words of cursor pattern
FF80
FF80
FF80
FF80
FF80
FF80
FF80
FF80
FF80
0000
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       10 x 10 point pen cursor
*

Type CURS = GNRL          ;; Cursor type definition
    ,41                   ;; Resource ID
.H
*       16 words of cursor pattern
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
FFC0
0000
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
0000
0000
*
*       11 x 11  point pen cursor
*
Type CURS = GNRL        ;; Cursor type definition
     ,42                ;; Resource ID
.H
*       16 words of cursor pattern
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
FFE0
0000
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000
*
*       12 x 12 point pen cursor
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
*

Type CURS = GNRL        ;; Cursor type definition
     ,43                ;; Resource ID
.H
*       16 words of cursor pattern
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
FFF0
0000
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       13 x 13 point pen cursor
*

Type CURS = GNRL        ;; Cursor type definition
     ,44                ;; Resource ID
.H
```

***Listing E.14** – The MacDoodle.R file (continued)*

```
*       16 words of cursor pattern
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
FFF8
0000
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       14 x 14 point pen cursor
*

Type CURS = GNRL           ;; Cursor type definition
    ,45                    ;; Resource ID
.H
*       16 words of cursor pattern
FFFC
FFFC
FFFC
FFFC
```

Listing E.14 – *The MacDoodle.R file (continued)*

```
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
FFFC
0000
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000


*
*       15 X 15 point pen cursor
*

Type CURS = GNRL          ;; Cursor type definition
     ,46                  ;; Resource ID
.H
*       16 words of cursor pattern
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
```

Listing E.14 – The MacDoodle.R file (continued)

```
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
FFFE
0000
*       16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*       Hot spot point
0000
0000

*
*       16 X 16 point pen cursor
*

Type CURS = GNRL          ;; Cursor type definition
     ,47                  ;; Resource ID
.H
*       16 words of cursor pattern
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
FFFF
```

Listing E.14 *– The MacDoodle.R file (continued)*

```
FFFF
FFFF
FFFF
FFFF
FFFF
*        16 words of cursor mask
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
*        Hot spot point
0000
0000
```

Listing E.14 *– The MacDoodle.R file (continued)*

```
MacDoodle
MacDoodleDraw
MacDoodleMenu
MacDoodleData
MacDoodleRTL
PrLink
!XXSTART
/Bundle
/Type 'APPL' 'DOOD'
$
```

Listing E.15 *– The MacDoodle.link file*

```
Asm      MacDoodle.Asm   Exec    Edit
Asm      MacDoodleData.Asm       Exec    Edit
Asm      MacDoodleMenu.Asm       Exec    Edit
Asm      MacDoodleDraw.Asm       Exec    Edit
Asm      MacDoodleRTL.Asm        Exec    Edit
Link     MacDoodle.Link  Exec    Edit
Rmaker   MacDoodle.R     Exec    Edit
```

***Listing E.16** – The MacDoodle.job file*

CHAPTER 8

The run-time library has six files:

File Name	Listing	Purpose
RTL.asm	E.17	Default RTL definition file
RTLcode.asm	E.18	RTL code
RTLdata.asm	E.19	RTL data
RTL.R	E.20	Default RTL RMAKER input file
RTL.job	E.21	EXEC file to build default RTL
MACLIB.ASM	E.22	Macro library file

To run the programs in Chapters 4 and 6, you need all of these files. In addition, you must run the RTL.job file to generate the RTL.rel object file and RTL.RSRC resource file. To run a program that uses the run-time library, the MACLIB.ASM file must exist on the same disk as the source file during assembly, and the file RTL.RSRC must exist on some disk in the system.

```
;
;       Text RTL code
;
TEXTRTL EQU     0               ;       Text RTL
CONTROL EQU     -1              ;       Don't do controls
RESFILE EQU     0               ;       Has a separate resource file
Include RTLcode.Asm             ;       Include RTL
        END
```

***Listing E.17** – The RTL.asm file*

```
;
;       RTL source file for simplified Macintosh applications
;
;       Version 01.01   31-Aug-85
;
;       Revision history
;
;       <01>    Changes for Exercises in Chapter 8.
;               1. 1 call to XXPRINT from PRINT macro
;               2. _ExitToShell enhancement
;
;
;
Include MacTraps.D              ;       Traps definition file
Include ToolEquX.D              ;       Tool equates
Include FSEqu.D                 ;       File system equates
Include QuickEquX.D             ;       QuickDraw equates
Include SysEqux.D               ;       System equates
;
;       Resource file equates
;
MENURS  EQU     1               ;       First menu resource ID
DEFMENU EQU     3               ;       Three default menus
                                ;       -Apple (desk accessory)
                                ;       -File
                                ;       -Edit
                                ;
WINDOWR EQU     1               ;       Standard window resource #
ONTOP   EQU     -1              ;       Code for on top (GetNewWindow)
ScrollW EQU     15              ;       Scroll bar width (pixels)
TopW    EQU     19              ;       Top of window width (pixels)
VscrID  EQU     256             ;       Vertical scroll bar resource ID
HscrID  EQU     257             ;       Horizontal scroll bar resource ID
                                ;
APPLE   EQU     1               ;       Apple menu (desk accessories)
ABOUT   EQU     1               ;         Choice for About
                                ;
EDIT    EQU     3               ;       Edit menu (Cut/Paste)
UNDO    EQU     1               ;         Choice for Undo
CUT     EQU     2               ;         Choice for Cut
COPY    EQU     3               ;         Choice for Copy
PASTE   EQU     4               ;         Choice for Paste
;
;       ASCII characters
;
BS      EQU     8               ;       Backspace character
CR      EQU     $0D             ;       Carriage return (line terminator)
BL      EQU     $20             ;       Space (blank) character
;
;       Standard file system equates
```

***Listing E.18** – The RTLcode.asm file*

```
;
CANCEL  EQU     -1              ;       Error code for cancel
SFPut   EQU     1               ;       Put file routine code
SFGet   EQU     2               ;       Get file routine code
;
;       Event system masks
;
AllMask EQU     -1              ;       Get any event
KeyMask EQU     $28             ;       Keyboard input (key/autokey)
AbtMask EQU     $228            ;       Abort (Command-Period) event
;
;       Event system macros
;
        MACRO   NEXTEVT Evmask  =
        MOVE.W  {Evmask},D0     ;       Load mask register
        JSR     NextEvent       ;       Get next event
        |
;
;       Other event system equates
;
AbtEvt  EQU     9               ;       Abort event number
IBeam   EQU     1               ;       I-beam cursor #
ComKey  EQU     0               ;       Bit number for command key down
;
;       GETEVENT return codes
;
MenuChoice EQU  16              ;       Menu selection; D1 = parameters
CloseWin   EQU  17              ;       Close window code
GrowWin    EQU  18              ;       Grow window code; D1 = new size
ScrollWin  EQU  19              ;       Scroll window; D1 = control handle
TScrollV   EQU  20              ;       Thumb scroll;  D1 = control handle
TScrollH   EQU  21              ;       Thumb scroll;  D1 = control handle
;
;       Macros
;
        MACRO   SAVE    =       ;       Save registers
        MOVEM.L D0-D7/A0-A6,SaveRegs(A5)
        |
        MACRO   RETURN  =       ;       Return to caller with D0
        BRA     XRETURN         ;       Branch to exit routine
        |
        MACRO   RETURNALL =     ;       Return; restore all
        BRA     XRETURNALL      ;       Branch to exit routine
        |
        MACRO   RETURNP =       ;       Return D0/D1
        BRA     XRETURNP
        |

eofERR  EQU     -39             ;       File system EOF error

Include RTLdata.Asm             ;       Add data file

;
;
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
;       XXINIT routine.  This routine is called by the PROGRAM macro
;       to perform one-time setup functions.
;
;       Calling sequence:
;
;       JSR     XXINIT          ;       Call Init routine
;       DC.B    'Name'          ;       Application program name
;       DC.W    Menus           ;       Number of menus in resource file
;       DC.W    DEBUG           ;       Either $FF01 or $4E71
;
;
        XDEF    XXINIT          ;       For the linker
XXINIT:                         ;       Called by PROGRAM macro
;       DC.W    $FF01           ;       For debugging only!!
        PEA     -4(A5)          ;       QuickDraw scratch area
        _InitGraf               ;       Init QuickDraw
        _InitFonts              ;       Init font package
        _InitWindows            ;       Init windows package
        _InitMenus              ;       Init menu package
        PEA     DoExit          ;<01>   Restart procedure
        _InitDialogs            ;       Init dialog package
        _TEInit                 ;       Init Text Edit package
;
;       Open the separate resource file.  We try it twice: once with
;       the name supplied with the PROGRAM macro, and once with the
;       canned string RTL.RSRC.
;
        IF      RESFILE = 0     ;       Only if resource file separate
        MOVE.L  (SP),A2         ;       A2 -> Name string
        LEA     Scratch(A5),A0  ;       A0 -> Target count byte
        LEA     1(A0),A1        ;       A1 -> Destination
        BSR     MoveCnt         ;       Move in count-delimited string
        LEA     RSRC,A2         ;       A2 -> .RSRC string
        BSR     MoveCnt         ;       Move in this field
        ENDIF                   ;       Part 1 of resource conditional
        MOVE.L  (SP),A2         ;       Fetch string address again
        CLR.L   D7              ;       Load long
        MOVE.B  (A2),D7         ;                   unsigned byte
        MOVE.L  D7,D6           ;       Save this value
        ADDQ.L  #2,D7           ;       Adjust to
        BCLR.L  #0,D7           ;                    word boundary
        ADD.L   D7,A2           ;       A2 -> menu word
        IF      RESFILE = 0     ;       Start part 2 of resource conditional
        ADD.W   #RSRCLEN,D6     ;       Add in .RSRC length
        MOVE.B  D6,(A0)         ;       Update length count
        BSR     OpenRsrc        ;       Try to open file
        MOVE.W  D0,ResFD(A5)    ;       Save it for later
        BPL     ResOK           ;       If > 0, file opened OK
        LEA     Canned,A0       ;       Try default name
        BSR     OpenRsrc        ;       Attempt open
        MOVE.W  D0,ResFD(A5)    ;       Save it for later
        BMI     ResNFG          ;       Couldn't open, quit
ResOK:                          ;       Resource file open
        ENDIF                   ;       RESFILE = 0
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
;
;       For our next trick, we will set up the Macintosh standard menu
;       bar (File Edit ... etc.) as described in the resource file.
;
        MOVE.W  (A2)+,D6            ;       Fetch # of menus in resource file
        BNE     GotMenus            ;       NE => Default not chosen
        MOVE.W  #DEFMENU,D6         ;       Select default
GotMenus:                           ;
        MOVE.W  #MENURS,D7          ;       Menu resource number
MenuLoop:                           ;       Loop on menus
        CLR.L   -(SP)               ;       Clear space for menu handle
        MOVE.W  D7,-(SP)            ;       Push resource number
        _GetRMenu                   ;       Get menu handle
        CLR.W   -(SP)               ;       Insert after flag
        _InsertMenu                 ;       Insert current menu
        ADDQ.L  #1,D7               ;       Bump resource number
        CMP.W   D6,D7               ;       Done?
        BLE     MenuLoop            ;       Not yet, repeat until all inserted
;
;       Add desk accessories into first menu (Apple)
;
        CLR.L   -(SP)               ;       Room for menu handle result
        MOVE.L  #'MENU',-(SP)       ;       Indicate resource type
        MOVE.W  #MENURS,-(SP)       ;       Menu #1
        _GetResource                ;       Fetch menu handle
        MOVE.L  #'DRVR',-(SP)       ;       Accessory type
        _AddResMenu                 ;       Add to menu
        _DrawMenuBar                ;       Paint bar on screen
;
;       Now we create the standard window
;
        CLR.L   -(SP)               ;       Room for window handle
        MOVE.W  #WINDOWR,-(SP)      ;       Window resource number
        PEA     WArea(A5)           ;       -> Window space
        MOVE.L  #ONTOP,-(SP)        ;       Code for on top of others
        _GetNewWindow               ;       Create the window
        MOVE.L  (SP),A3             ;       A3 -> Window record
        MOVE.L  A3,WindowPtr(A5);           Save window record address
        _SetPort                    ;       Make window the Q/D current port
        MOVE.W  #monaco,-(SP)       ;       Set font to Monaco
        _TextFont                   ;
         MOVE.W  #9,-(SP)           ;       Use 9 point (80 cols on Mac)
        _TextSize                   ;
        MOVE.L  (SP)+,A0            ;       Get string address from stack
        BSR     XXWTITLE            ;       Set the window title
;
;       Now initialize scroll bars
;
        IF      CONTROL = 0         ;       If window controls
        CLR.L   -(SP)               ;       Space for result
        MOVE.W  #VscrID,-(SP)       ;       Push resource ID
        MOVE.L  A3,-(SP)            ;       Push window pointer
        _GetNewControl              ;       Add control to window
        MOVE.L  (SP)+,Vscroll(A5);          Save VScroll handle
```

Listing E.18 *– The RTLcode.asm file (continued)*

```
        CLR.L   -(SP)              ;       Space for result
        MOVE.W  #HscrID,-(SP)      ;       Push resource ID
        MOVE.L  A3,-(SP)           ;       Push window pointer
        _GetNewControl             ;       Add control to window
        MOVE.L  (SP)+,Hscroll(A5);         Save HScroll handle
        MOVE.L  A3,-(SP)           ;       Push window pointer
        _DrawGrowIcon              ;       Draw grow box
        ENDIF                      ;       CONTROL = 0
        BSR     SizeW              ;       Size the window
;
;       Create Text Edit record
;
    IF  TEXTRTL = 0                ;       If no text, don't create TE
        CLR.L   -(SP)              ;       Space for text handle result
        PEA     DestRect(A5)       ;       -> Destination rectangle area
        PEA     ViewRect(A5)       ;       -> View rectangle area
        _TENew                     ;       New text record
        MOVE.L  (SP),A3            ;       A3 -> -> text record
        MOVE.L  (A3),A3            ;       Dereference
        MOVE.B  #-1,teCROnly(A3);          Specify no wrap mode
        MOVE.L  (SP),THandle(A5) ;         Store text handle
        _TEActivate                ;       Activate text record
     ENDIF
;
;       Final initialization
;
        MOVE.L  #$0000FFFF,D0      ;       Mask for all events
        _FlushEvents               ;       Get rid of all events
        _InitCursor                ;       Make cursor the arrow
        CLR.W   xCursor(A5)        ;       Indicate last cursor set
        MOVE.L  (SP)+,EXITAddr(A5);        Save Finder's return address
        MOVE.L  SP,ORGSP(A5)       ;       and stack pointer contents
        MOVE.W  #IBeam,Cursor(A5);         Set I-beam cursor
        MOVE.L  WindowPtr(A5),A0;          Set return register
        MOVE.L  ViewRect+4(A5),D0;         Return window size, also
        JMP     (A2)               ;       Return to application

;
;       This routine is called when we couldn't open a resource file.
;       Deep prune yogurt time down south.
;
        IF      RESFILE = 0        ;       Only if resource file separate
ResNFG  MOVE.L  WMgrPort,-(SP)     ;       Push window manager GrafPort @
        _SetPort                   ;       Set it to our port
        CLR.L   ViewRect(A5)       ;       Init
        MOVE.L  #$2000200,ViewRect+4(A5);       emergency rect
        PEA     ViewRect(A5)       ;       to stack
        _EraseRect                 ;       Clears the screen
        MOVE.L  #$00100010,-(SP);          Position
        _MoveTo                    ;                 cursor
        PEA     'Cannot Find Resource File: RTL.RSRC'
        _DrawString                ;       Write message
        DC.W    -1                 ;       Bomb
        ENDIF                      ;       RESFILE = 0
```

Listing E.18 – The RTLcode.asm file (continued)

```
;
;       MoveCnt routine.  This routine moves a Pascal string to another
;       area.  The count byte is not moved.
;
MoveCnt:                            ;
        MOVE.L  D0,-(SP)            ;       Save D0
        CLR.L   D0                  ;       Clear out problem bits
        MOVE.B  (A2)+,D0            ;       Fetch count byte
        SUBQ.L  #1,D0               ;       Decrement for DBRA
MoveLoop:                           ;
        MOVE.B  (A2)+,(A1)+         ;       Move a byte
        DBRA    D0,MoveLoop         ;       Down count
        MOVE.L  (SP)+,D0            ;       Restore D0
        RTS                         ;       Return to caller
;
;       XXWTITLE routine. Called by WTITLE macro and by init code.
;       Sets the window title to the string pointed to by A0.
;
;       Entry conditions:
;
;               A0 -> New title string (Pascal format)
;
;       Exit conditions:
;
;               (none)
;
        XDEF    XXWTITLE            ;       For the linker
XXWTITLE                            ;
        SAVE                        ;       Save the registers
        MOVE.L  WindowPtr(A5),-(SP);       Push window address
        MOVE.L  A0,-(SP)            ;       and string address
        _SetWTitle                  ;       Set the title
        RETURNALL                   ;       Return to caller
;
;       Menu item Check and Uncheck routines.  Called by the CHECK and
;       UNCHECK macros.
;
;       Entry conditions:
;
;               D0 = Menu number (resource ID)
;               D1 = Item number
;
;       Exit conditions:
;
;               (none)
;
        XDEF    XXCHECK             ;       For the
        XDEF    XXUNCHECK           ;               linker
XXCHECK SAVE                        ;       Save registers
        MOVE.W  #CheckMark,D2       ;       Set mark character
        BRA     DoCheck             ;       Go do it
XXUNCHECK                           ;       Here to undo it
        SAVE                        ;       Save registers
        MOVE.W  #noMark,D2          ;       Set blank (null) mark
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
DoCheck MOVE.W  D1,D3           ;       Preserve item number
        BSR     DoRsrc          ;<01>   Get menu resource
        MOVE.W  D3,-(SP)        ;       Push item number
        MOVE.W  D2,-(SP)        ;       Push mark character
        _SetItmMark             ;       Mark it
        RETURN                  ;       and return
;
;       Menu item Enable and Disable routines.  Called from ENABLE and
;       DISABLE macros.
;
;       Entry conditions:
;
;               D0 = Menu number (Resource ID)
;               D1 = Item number
;
;       Exit conditions:
;
;               (none)
;
        XDEF    XXENABLE        ;       For the
        XDEF    XXDISABLE       ;               linker
XXENABLE                        ;       Enable entry point
        SAVE                    ;       Save registers
        MOVE.W  #1,D3           ;       Set flag register
        BRA     XXENDIS         ;       Branch around disable entry
XXDISABLE                       ;       Here to disable
        SAVE                    ;       Save the registers
        CLR.W   D3              ;       Clear flag register
XXENDIS MOVE.W  D1,D4           ;       Preserve item number
        BSR     DoRsrc          ;<01>   Go get resource
        MOVE.W  D4,-(SP)        ;       Push item number
        TST.W   D3              ;       Enable?
        BEQ     DoDisable       ;       No, disable
        _EnableItem             ;       Do the enable
        RETURN                  ;       Back to caller
DoDisable                       ;       Here to disable
        _DisableItem            ;       Disable it
        RETURN                  ;       Back to caller
;
;       Menu item Style routine.  Called by the STYLE macro.
;
;       Entry conditions:
;
;               D0 = Menu number (resource ID)
;               D1 = Item number
;               D2 = Style word
;
;       Exit conditions:
;
;               (none)
;
        XDEF    XXSTYLE         ;       For the linker
XXSTYLE                         ;
        SAVE                    ;       Save registers
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
        MOVE.W  D1,D3           ;       Preserve item number
        MOVE.W  D2,D4           ;       and style word
        BSR     DoRsrc          ;<01>   Go get resource
        MOVE.W  D3,-(SP)        ;       Push item number
        MOVE.W  D4,-(SP)        ;       Push style word
        _SetItmStyle            ;       Change it
        RETURN                  ;       and return
;
;       DoRsrc routine.  Returns menu handle on top of the stack
;       from resource ID in D0.W.
;
DoRsrc                          ;<01>   New with edit <01>
        MOVE.L  (SP),A3         ;<01>   Return address to A3
        MOVE.L  #'MENU',-(SP)   ;<01>   Specify resource type
        MOVE.W  D0,-(SP)        ;<01>   Convert
        _GetResource            ;<01>           ID
                                ;<01>               to handle (SP)
        JMP     (A3)            ;<01> Return to caller
;
;       RTL exit routines.  Entered from RETURN, RETURNALL, and RETURNP
;       macros.  Restore registers and RTS.
;
XRETURNALL:                     ;       Here to restore all regs
        MOVE.L  SaveRegs(A5),D0 ;       Restore D0
XRETURN:                        ;       Here to restore all save D0
        MOVE.L  SaveRegs+4(A5),D1;      Restore D1
XRETURNP:                       ;       Here to restore all save D0 and D1
        MOVEM.L SaveRegs+8(A5),D2-D7/A0-A6; Restore bulk of registers
        RTS                     ;       Return to caller
;
;       XXCURSOR routine.  Change cursor to a resource file cursor.
;
;       Entry conditions:
;
;               D0.W = Cursor code:
;                       0 => Arrow
;                       1 => I-beam (Text Edit cursor)
;                       2 => Plus (graphics)
;                       4 => Watch
;
;       Exit conditions:
;
;               (none)
;
;       This routine can be invoked either by the SCURSOR macro or
;       internally in RTL.  It does not use SAVE/RETURN.
;
        XDEF    XXCURSOR
XXCURSOR:
        CMP.W   xCursor(A5),D0  ;       Need to do this?
        BNE     @1              ;       Yes
        RTS                     ;       No
@1:     MOVEM.L D0-D2/A0-A2,-(SP);      Save registers
        MOVE.W  D0,xCursor(A5)  ;       Remember for next time
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
        TST.W   D0              ;       Arrow?
        BNE     @2              ;       No, do it differently
        _InitCursor             ;       Arrow done thusly
        BRA     CDone           ;       Finished
@2:                             ;       Must get from resource file
        CLR.L   -(SP)           ;       Space for cursor handle
        MOVE.W  D0,-(SP)        ;       Push cursor ID
        _GetCursor              ;       Fetch cursor handle
        MOVE.L  (SP)+,A0        ;       A0 -> Cursor pointer
        MOVE.L  (A0),-(SP)      ;       Push actual address
        _SetCursor              ;       Do the set
CDone:                          ;       Here with correct cursor
        MOVEM.L (SP)+,D0-D2/A0-A2;      Restore registers
        RTS                     ;       Return to caller
;
;       XXABORT routine. Checks for user keyboard abort.
;
;       Entry conditions:
;
;        (none)
;
;       Exit conditions
;
;       D0 = 0  If no abort
;       D0 = -1 If user abort desired
;
        XDEF    XXABORT
XXABORT:                        ;
        SAVE                    ;       Save registers
        NEXTEVT #AbtMask        ;       Get abort event, if any
        BEQ     NoEvent         ;       No event, return
        CMPI.W  #AbtEvt,What(A5);       Was an abort?
        BEQ     AbtEvent        ;       Yes, quit
        CMPI.B  #'.',Message+3(A5);     Check
        BNE     NoEvent         ;             for
        BTST    #CmmKey,Modify(A5);                Command-
        BEQ     NoEvent         ;                          Period
AbtEvent                        ;
        MOVEQ.L #-1,D0          ;       Set return code
        RETURN                  ;       Quit
NoEvent CLR.L   D0              ;       Set no event return code
        RETURN                  ;       Quit
;
;       NextEvent routine.  Code to perform Macintosh  GetNextEvent call.
;
;       Entry conditions:
;
;       D0 = Event mask desired
;
;       Exit conditions:
;
;       EventRecord(A5) filled in.
;       D0 = Event ID (=0 if no event present)
;
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
NextEvent:                        ;       Called internally only
            MOVE.L  D0,-(SP)      ;       Save D0 for a bit....
        IF          TEXTRTL = 0   ;       If text type RTL
            MOVE.L  THandle(A5),-(SP);    Push text record handle
            _TEIdle               ;       Call Text Edit idle proc
        ENDIF                     ;       TEXTRTL
            _SystemTask           ;       Let system tasks run
            MOVE.L  (SP)+,D0      ;       Recover mask
            CLR.W   -(SP)         ;       Make room for event ID
            MOVE.W  D0,-(SP)      ;       Push event mask
            PEA     EventRecord(A5) ;     Push event record address
            _GetNextEvent         ;       Fetch next Mac event
            MOVE.W  (SP)+,D0      ;       Event ID to D0
            RTS                   ;       Return
;
;           XXEVENT routine.  Called from GETEVENT macro.
;
;           Entry conditions:
;
;           D0 = Resource ID for cursor (while it's in the window)
;
;           Exit conditions:
;
;           D0 = Event code (0-15)
;           D1 = Event parameters
;
            XDEF    XXEVENT       ;       External
XXEVENT:                          ;
            SAVE                  ;       Save registers
            MOVE.W  D0,Cursor(A5) ;       Set default cursor
DoEvent     PEA     Point(A5)     ;       Push mouse point address
            _GetMouse             ;       Fetch mouse coordinates
            CLR.W   -(SP)         ;       Room for result
            MOVE.L  Point(A5),-(SP) ;     Push point
            PEA     DestRect(A5)  ;       Push rect address
            _PtInRect             ;       See if mouse in window
            TST.W   (SP)+         ;       Is the mouse in the window?
            BNE     ChangeCurs    ;       Yes, change the cursor
            CLR.L   D0            ;       Set to arrow (0)
            BRA     DoCurs        ;       Set the cursor
ChangeCurs:                       ;       Here if need to change cursor
            MOVE.W  Cursor(A5),D0 ;       Change to requested
DoCurs      BSR     XXCURSOR      ;       Change the cursor
            NEXTEVT #AllMask      ;       Get new event
            BEQ     NoEvent       ;       None there, try again
            MOVE.W  What(A5),D0   ;       D0 = Event type
            ADD.W   D0,D0         ;       Multiply by 2
            MOVE.W  Eventtab(D0),D0 ;     D0 = Offset
            JMP     Event(D0)     ;       Jump to event routine
Event:                            ;       Base point for offset table
;
;           Define a macro for event table entries
;
MACRO       ETAB    addr =        ;       Entry macro
```

Listing E.18 – The RTLcode.asm file (continued)

```
          DC.W    {addr}-XXBASE    ;      Generate an offset word
          |
;
;         Define the event table itself
;
EventTab                           ;      Offset table
XXBASE    SET     *                ;      Here is the base value
          ETAB    NullEvent        ;      0 => Null event, ignore
          ETAB    MouseDown        ;      1 => Mouse-down event
          ETAB    MouseUp          ;      2 => Mouse-up event
          ETAB    KeyDown          ;      3 => Key press (down)
          ETAB    KeyUp            ;      4 => Key release (up)
          ETAB    AutoKey          ;      5 => Autokey
          ETAB    Update           ;      6 => Update event
          ETAB    Disk             ;      7 => Disk
          ETAB    Activate         ;      8 => Activate
          ETAB    Abort            ;      9 => Abort
          ETAB    Network          ;      A => Network
          ETAB    Driver           ;      B => Driver
          ETAB    Appl             ;      C => Application #1
          ETAB    App2             ;      D => Application #2
          ETAB    App3             ;      E => Application #3
          ETAB    App4             ;      F => Application #4
;
;         End of event table
;
Activate                           ;      Activate window request
          MOVE.L  Message(A5),A0   ;      A0 = Window handle
          CMP.L   WindowPtr(A5),A0 ;      Is it ours?
          BNE     DoEvent          ;      No, get next event
          IF      CONTROL = 0      ;      Controls?
          BSR     IncControls      ;      Yes, include them
          ENDIF                    ;      CONTROL
;
;         (De)Activate the text window
;
          MOVE.W  Modify(A5),D0    ;      Activate or deactivate?
          BTST.L  #0,D0            ;      Bit 0 tells us
          BEQ     Deactivate       ;      Clear => Deactivate
          IF      TEXTRTL = 0      ;      Don't activate TE if not needed
          MOVE.L  THandle(A5),-(SP) ;     Push text handle
          _TEActivate              ;      Activate text window
          ENDIF                    ;
          MOVE.L  Message(A5),-(SP) ;     Push window pointer
          _SelectWindow            ;      Select this window as active
          MOVE.L  Message(A5),-(SP) ;     Push window pointer
          _SetPort                 ;      Make the Q/D active port
          IF      CONTROL = 0      ;      If window has controls
          MOVE.L  VScroll(A5),-(SP);      Show
          _ShowControl             ;              scroll
          MOVE.L  HScroll(A5),-(SP);                      bars
          _ShowControl             ;
          BRA     EndAct           ;      Do end stuff
          ELSE                     ;      CONTROL = 0
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         BRA     DoEvent          ;       Fetch next event
         ENDIF                    ;       CONTROL = 0
;
;        Here if deactivate specified
;
Deactivate:                       ;
         IF      TEXTRTL = 0      ;       Don't deactivate TE if no text
         MOVE.L  THandle(A5),-(SP) ;      Push text handle
         _TeDeActivate            ;       and zonk it
         ENDIF
         IF      CONTROL = 0      ;       If scroll bars
         MOVE.L  VScroll(A5),-(SP);       Hide
         _HideControl             ;               scroll
         MOVE.L  HScroll(A5),-(SP);                       bars
         _HideControl             ;
EndAct:  BSR     ExcControls      ;       Exclude controls
         ENDIF                    ;       CONTROL = 0
         BRA     Doevent          ;       Get next event
;
;        Event is an update request
;
Update:                           ;       Just update text part
         IF      TEXTRTL = 0      ;       If text only
         MOVE.L  WindowPtr(A5),-(SP) ;    Warn
         _BeginUpDate             ;               everyone first
         PEA     ViewRect         ;       Now repaint
         MOVE.L  THandle(A5),-(SP) ;      text window
         _TEUpdate                ;       thusly
         MOVE.L  WindowPtr(A5),-(SP) ;    Now signal end
         _EndUpdate               ;                       of update
         BRA     Doevent          ;       Done
         ELSE                     ;       Graphics mode
         BSR     IncControls      ;       Include controls in window
         MOVE.L  WindowPtr(A5),-(SP);     Push window pointer
         _DrawControls            ;       Draw the controls
         MOVE.L  WindowPtr(A5),-(SP);     Push window pointer
         _DrawGrowIcon            ;       Draw grow box
         BSR     ExcControls      ;       Take controls back out
         BRA     UserEvt          ;       Let user handle
         ENDIF                    ;
;
;        IncControls and ExcControls routines.  Call them to insert
;        and remove controls from the window valid region.
;
;        Improper ordering is allowed.
;
IncControls                       ;       Include controls
         MOVEM.L D0-D2/A0-A2,-(SP);       Preserve registers
         MOVEM.L DestRect(A5),D0/D1;      Pick up old rect
         MOVEM.L D0/D1,ViewRect(A5);      Reset to clipped
         ADD.W   #ScrollW,ViewRect+4(A5); Adjust
         ADD.W   #ScrollW,ViewRect+6(A5);         clip
         PEA     ViewRect(A5)     ;                       rectangle
         _ClipRect                ;
```

Listing E.18 – The RTLcode.asm file (continued)

```
          MOVEM.L (SP)+,D0-D2/A0-A2;      Restore registers
          RTS                     ;       Back to caller

ExcControls                       ;       Exclude
          MOVEM.L D0-D2/A0-A2,-(SP);      Preserve registers
          MOVEM.L DestRect(A5),D0/D1;     Pick up old rect
          MOVEM.L D0/D1,ViewRect(A5);     Reset to clipped
          PEA     ViewRect(A5)    ;       Push rectangle address
          _ClipRect               ;       Set to clipped
          MOVEM.L (SP)+,D0-D2/A0-A2;      Restore registers
          RTS                     ;       Back to caller

;
;         Event is a mouse down
;
MouseDown                         ;
          CLR     -(SP)           ;       Clear a space for result
          MOVE.L  Point(A5),-(SP) ;       Mouse at time of event
          PEA     WWindow(A5)     ;       Event window
          _FindWindow             ;       Find the window
          MOVE.W  (SP)+,D0        ;       Fetch index
          AND.W   #7,D0           ;       Force to range 0-7
          ADD.W   D0,D0           ;       Multiply by 2
          MOVE.W  WTable(D0),D0   ;       Fetch offset
          JMP     WTable(D0)      ;       Jump to appropriate routine
WTable:                           ;       Start dispatch table
XXBASE    SET     *               ;       Define macro base here
          ETAB    Doevent         ; 0     Mouse in desk area someplace
          ETAB    InMenu          ; 1     Mouse in menu area
          ETAB    SystemEvent     ; 2     Mouse in system window
          ETAB    Content         ; 3     Window content area
          ETAB    Drag            ; 4     In drag area
          ETAB    Grow            ; 5     In grow box
          ETAB    GoAway          ; 6     In close box
          ETAB    Doevent         ; 7     (Undefined)
;
;         Here if key down is indicated.  Handle Command-key entries, else
;         give key to user.
;
KeyDown
AutoKey
          MOVE.W  Message+2(A5),D1;       Key to D1.W
          BTST    #CmKey,Modify(A5);      Is Command key down?
          BEQ     UserEvt         ;       No, give key to user
          CLR.L   -(SP)           ;       Room for result
          MOVE.W  D1,-(SP)        ;       Push character
          _MenuKey                ;       See if a Command
          BRA     DoMenu          ;       Go do the menu
;
;         Here if mouse in the menu area
;
InMenu:                           ;
          CLR.L   -(SP)           ;       Clear space for return value
          MOVE.L  Point(A5),-(SP) ;       Push mouse point
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
        _MenuSelect            ;       Perform menu selection
DoMenu  CLR.W   -(SP)          ;       Unhighlight menus
        _HiLiteMenu            ;       This does all
        MOVE.L  (SP),D1        ;       Both to D1
        MOVE.W  (SP)+,D6       ;       Menu number    (from _MenuSelect)
        MOVE.W  (SP)+,D7       ;       Menu selection (from _MenuSelect)
        TST.L   D1             ;       Really a choice?
        BEQ     DoEvent        ;       No, just kidding
        MOVE.L  #MenuChoice,D0 ;       D0 = Menu choice return code
        CMP.W   #APPLE,D6      ;       Apple menu?
        BEQ     Inapple        ;       Yes, do this one ourselves
        CMP.W   #EDIT,D6       ;       No, how about the Edit menu?
        BEQ     InEdit         ;       Yes, handle this one too
        RETURNP                ;       Restore registers; return
Inapple:                       ;       Here if it's a choice we handle ....
        CMP.W   #ABOUT,D7      ;       Was it the About choice?
        BNE     DeskAcc        ;       No, must be a desk accessory
        MOVE.W  D7,D0          ;       Set parameter register
        BSR     XXDIALOG       ;       Do the dialog
        BRA     DoEvent        ;       and do another event
;
;       Here if a desk accessory
;
DeskAcc:                       ;
        CLR.L   -(SP)          ;       Clear return space
        MOVE.L  #'MENU',-(SP)  ;       Indicate menu resource
        MOVE.W  #APPLE,-(SP)   ;       Push menu ID
        _GetResource           ;       Get menu handle
        MOVE.W  D7,-(SP)       ;       Push menu selection
        PEA     Deskname(A5)   ;       -> Accessory name area
        _GetItem               ;       Fetch desk accessory name
        PEA     GPort(A5)      ;       Push longword address
        _GetPort
        CLR.W   -(SP)          ;       Space for handle
        PEA     DeskName(A5)   ;       -> Filled-in name area
        _OpenDeskAcc           ;       Open the accessory
        TST.W   (SP)+          ;       Discard handle
        MOVE.L  GPort(A5),-(SP) ;      Push old GrafPort address
        _SetPort               ;       Restore it
        BRA     Doevent        ;       And back for another event
;
;       Here if menu selection was in the Edit menu
;
InEdit:                        ;       Cut/Copy/Paste
        CLR.L   -(SP)          ;       Clear return area
        _FrontWindow           ;       Return front window pointer
        MOVE.L  (SP)+,A0       ;       Window handle to A0
        CMP.L   WindowPtr(A5),A0 ;     Our window?
        BNE     SystemEdit     ;       No, it's a system window
      IF        TEXTRTL = 0    ;       If TEXT RTL only
        CMP.W   #PASTE,D7      ;       Bigger than Paste?
        BLE     DoEdit         ;       No, it's OK
        RETURNP                ;       Restore; return
DoEdit: CMP.W   #COPY,D7       ;       Bigger than Copy value?
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
                BGE      DoCopy           ;        Yes, keep looking
;
;               Must be a Cut command
;
                MOVE.L   THandle(A5),-(SP) ;       Push text handle
                _TECut                    ;        Do the cut
                BRA      Doevent          ;        Back for more
DoCopy:         CMP.W    #PASTE,D7        ;        Is it a paste?
                BEQ      DoPaste          ;        Yes
                MOVE.L   THandle(A5),-(SP) ;       Push text handle
                _TECopy                   ;        Do the copy
                BRA      Doevent          ;        and back for more
DoPaste:                                  ;        Here for paste
                MOVE.L   Thandle(A5),-(SP) ;       Push text handle
                _TEPaste                  ;        Do it
                BRA      Doevent          ;        Get next event
            ELSE                          ;        Graphics mode
                MOVE.L   #MenuChoice,D0   ;        Set return code
                MOVE.W   D6,D1            ;        Copy choice words
                SWAP     D1               ;            -to-
                MOVE.W   D7,D1            ;        return register
                RETURNP                   ;        and return menu choice
            ENDIF
;
;               System window edit
;
SystemEdit:                               ;
                MOVE.W   D7,D0            ;        D0 = menu choice
                SUBQ.L   #1,D0            ;        Decrement
                CLR.W    -(SP)            ;        Space for result
                MOVE.W   D0,-(SP)         ;        Choice word
                _SysEdit                  ;        Do it
                MOVE.W   (SP)+,D0         ;        Result to D0
                BRA      Doevent          ;        Back for next event
;
;               Here if mouse in system window
;
SystemEvent:                              ;
                PEA      EventRecord(A5)  ;        -> Event not ours
                MOVE.L   WWindow(A5),-(SP) ;       Window handle
                _SystemClick              ;        Call O/S
                BRA      Doevent          ;        Back for another event
;
;               Here if mouse in content area of window
;
Content:                                  ;
                MOVE.L   WWindow(A5),A0   ;        A0 = window handle
                CMP.L    WindowPtr(A5),A0 ;        Ours?
                BNE      Doevent          ;        No, get next event
                MOVE.L   A0,-(SP)         ;        Stack window handle
                _SelectWindow             ;        Select window
                PEA      Point(A5)        ;        Stack point address
                _GlobalToLocal            ;        Convert to window coords
            IF           TEXTRTL = 0      ;        If text type RTL
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         MOVE.L  Point(A5),-(SP) ;       Push point in local coords
         CLR.W   -(SP)           ;       No extend flag
         MOVE.L  THandle(A5),-(SP) ;     Stack text handle
         _TEClick                ;       Do Text Edit click
         BRA     Doevent         ;       Get next event
       ELSE                      ;       Graphics mode
;
;        In graphics mode with controls, we must find out if the mouse is
;        in a control.  If not, (or if no controls) we return the point to
;        the user program.
;
         IF      CONTROL = 0     ;       Only if we have controls
         BSR     IncControls     ;       Include controls first
         CLR.W   -(SP)           ;       Room for result
         MOVE.L  Point(A5),-(SP) ;       Push point
         MOVE.L  WindowPtr(A5),-(SP);    Push window address
         PEA     CHandle(A5)     ;       Push control handle
         _FindControl            ;       Fetch the control
         MOVE.W  (SP)+,D0        ;       Get part code
         BEQ     NoControl       ;       None, give point to user
         MOVE.L  CHandle(A5),D1  ;       Fetch control handle
         CMP.W   #inThumb,D0     ;       Special case?
         BEQ     @2              ;       Yes, go handle
         SUB.W   #inUpButton,D0  ;       Force to range 0-3
         CMP.L   VScroll(A5),D1  ;       Was it?
         BEQ     @1              ;       Yes, give to user
         ORI.W   #4,D0           ;       Or in horizontal bit
@1       SWAP    D0              ;       Set return
         MOVE.W  #ScrollWin,D0   ;                       register
         BRA     @4              ;       Back to user
;
;        If in thumb, we can do the tracking for the user program.
;
@2       MOVE.L  D1,D4           ;       Preserve control ID
         CLR.W   -(SP)           ;       Room for result
         MOVE.L  D1,-(SP)        ;       Push control ID
         MOVE.L  Point(A5),-(SP) ;       Push original point
         CLR.L   -(SP)           ;       No action procedure
         _TrackControl           ;       Track it
         TST.W   (SP)+           ;       For real?
         BNE     @3              ;       Yes, process
         BSR     ExcControls     ;       Reset clip region
         BRA     DoEvent         ;       Get an event
@3                               ;
;
;        User changed the value.  Fetch new setting.
;
         CLR.W   -(SP)           ;       Room for result
         MOVE.L  D4,-(SP)        ;       Push control handle
         _GetCtlValue            ;       Fetch the value
         MOVE.W  (SP)+,D0        ;       To
         SWAP    D0              ;               return register
         MOVE.L  D4,D1           ;       Reset parameter register
         MOVE.W  #TScrollV,D0    ;       Assume vertical
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         CMP.L   VScroll(A5),D1   ;      Good assumption?
         BEQ     @4               ;      Yes
         MOVE.W  #TScrollH,D0     ;      No, fix it
@4       BSR     ExcControls      ;      Exclude controls
         RETURNP                  ;      Done, return to caller
NoControl                         ;      Here if user point
         BSR     ExcControls      ;      Exclude controls
         ENDIF                    ;      CONTROL = 0
;
;        Here if mouse is in the REAL content area.  Pass the event
;        back to the application.
;
         MOVE.W  What(A5),D0      ;      Set mouse-down return code
         MOVE.L  Point(A5),D1     ;      Return point in local coords
         RETURNP                  ;      Back to user
       ENDIF                      ;
;
;
;        XXTRACK routine.  Called from the TRACK macro in response to
;        a scroll page or scroll line event.  Assumes mouse still down.
;
;        Entry conditions:
;
;                D0 = Control handle
;                A0 ->User's scroll procedure
;
;        Exit conditions:
;
;                (None)
;
         XDEF    XXTRACK          ;      For the linker
XXTRACK  SAVE                     ;      Save the registers
         BSR     IncControls      ;      Clip to allow drawing controls
         CLR.W   -(SP)            ;      Room for result
         MOVE.L  D0,-(SP)         ;      Push control handle
         MOVE.L  Point(A5),-(SP)  ;      and original mouse point
         MOVE.L  A0,-(SP)         ;      and user's procedure
         _TrackControl            ;      Do the track
         ADDQ.L  #2,SP            ;      Pop return
         BSR     ExcControls      ;      Reclip for no controls
         RETURN                   ;      and return
;
;        Here if user wants to drag
;
Drag:                             ;
         MOVE.L  WWindow(A5),-(SP);      Push window handle
         MOVE.L  Point(A5),-(SP)  ;      Push mouse point
         PEA     DWBounds         ;      and bounds rectangle
         _DragWindow              ;      Do the drag
         BRA     Doevent          ;      and get a new event
;
;        Here if a grow request
;
Grow:                             ;
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
            IF      CONTROL = 0      ;      Only if controls in place
            CLR.L   -(SP)            ;      Room for result
            MOVE.L  WWindow(A5),-(SP);      Push window handle
            MOVE.L  Point(A5),-(SP)  ;      and mouse point
            PEA     GWBounds         ;      and bounds rectangle
            _GrowWindow              ;      Do the grow
            MOVE.L  (SP)+,D5         ;      D5 = New size
            BEQ     DoEvent          ;      If EQ, didn't change
;
;           First, generate an update for the old scroll bar region.
;
            PEA     ViewRect(A5)     ;      Convert ViewRect to
            _LocalToGlobal           ;              global coordinates
            MOVEM.L ViewRect(A5),D3/D4;     Fetch rectangle
            MOVEM.L ViewRect(A5),D0/D1;     twice
;
;           Invalidate VScroll bar
;
            MOVE.W  D1,D0            ;      Copy right to left
            ADD.W   #ScrollW,D1      ;      Adjust right for scroll bar width
            SWAP    D1               ;      Adjust
            SUB.W   #ScrollW,D1      ;              bottom
            SWAP    D1               ;                      for grow box
            MOVEM.L D0/D1,ViewRect(A5);     Now
            PEA     ViewRect(A5)     ;              make
            _InvalRect               ;                      invalid
;
;           Now, invalidate HScroll bar
;
            SWAP    D3               ;      Invert
            SWAP    D4               ;              rectangle
            MOVE.W  D4,D3            ;      Copy bottom to top
            ADD.W   #ScrollW,D4      ;      Adjust bottom for scroll width
            SWAP    D3               ;      Re-
            SWAP    D4               ;              swap
            ADD.W   #ScrollW,D4      ;      Adjust right for grow box
            MOVEM.L D3/D4,ViewRect(A5);     Now
            PEA     ViewRect(A5)     ;              make
            _InvalRect               ;                      invalid
;
;           Next, actually resize the window on the screen
;
            MOVE.L  WWindow(A5),-(SP);      Push window handle
            MOVE.L  D5,-(SP)         ;      Push size
            MOVE.W  #-1,-(SP)        ;      Push TRUE value
            _SizeWindow              ;      Update screen to new size
            BSR     SizeW            ;      Size the window
;
;           Erase the new window content area
;
            PEA     ViewRect(A5)     ;      Push view rectangle address
            _EraseRect               ;      Erase it
;
;           Here we move the scroll bars to their new location.  First,
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
;       hide both controls.
;
        MOVE.L  VScroll(A5),-(SP);      Hide
        _HideControl          ;                 VScroll bar
        MOVE.L  HScroll(A5),-(SP);      Hide
        _HideControl          ;                 HScroll bar
;
;       Next move and resize VScroll bar
;
        MOVE.L  VScroll(A5),-(SP);      Move V scroll
        MOVE.W  ViewRect+6(A5),-(SP);   "Right" - scroll width
        MOVE.W  ViewRect+0(A5),-(SP);   "Top"
        _MoveControl          ;         Move the scroll bar
        MOVE.L  VScroll(A5),-(SP);      Now resize VScroll bar
        MOVE.W  #ScrollW,-(SP)  ;       Width
        MOVE.W  ViewRect+4(A5),-(SP);   Length
        _SizeControl          ;         Do the size
;
;       Now move and resize HScroll bar
;
        MOVE.L  HScroll(A5),-(SP);      Move H scroll
        MOVE.W  ViewRect+2(A5),-(SP);   Left
        MOVE.W  ViewRect+4(A5),-(SP);   Bottom - scroll width
        _MoveControl          ;         Do the move
        MOVE.L  HScroll(A5),-(SP);      Push control handle
        MOVE.W  ViewRect+6(A5),-(SP);   Width
        MOVE.W  #ScrollW,-(SP)  ;       Length
        _SizeControl          ;         Size it
;
;       Now show both controls
;
        MOVE.L  VScroll(A5),-(SP);      Show
        _ShowControl          ;                 VScroll bar
        MOVE.L  HScroll(A5),-(SP);      Show
        _ShowControl          ;                 HScroll bar
;
;       Now pass the size information to the application.
;
        MOVE.L  ViewRect+4(A5),D1;      Pick up new size (clipped)
        MOVE.L  #GrowWin,D0     ;       Set return code
        RETURNP               ;         Give back to user
        ELSE                  ;         CONTROL = 0
        BRA     DoEvent       ;         No grows allowed
        ENDIF                 ;         CONTROL = 0
;
;       Determine window size
;
SizeW   MOVEM.L D0-D2/A0-A3,-(SP);      Save registers we use
        MOVE.L  WindowPtr(A5),A3;       A3 -> Window structure
        MOVE.L  contRgn(A3),A0  ;       A0 -> Content region (handle)
        MOVE.L  (A0),A0         ;       A0 -> Content region (pointer)
        MOVEM.L RgnBBox(A0),D0/D1;      D0,D1 = GrafPort rectangle
        IF      CONTROL = 0     ;       If doing control stuff
        SUB.W   #ScrollW,D1     ;       Subtract
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         SWAP    D1              ;            off
         SUB.W   #ScrollW,D1     ;                scroll
         SWAP    D1              ;                        region
         ENDIF                   ;     CONTROL = 0
         MOVEM.L D0/D1,DestRect(A5);   Set dest rect
         PEA     DestRect(A5)    ;     Convert
         _GlobalToLocal          ;             to local coords
         PEA     DestRect+4(A5)  ;     Both points
         _GlobalToLocal          ;                in rectangle
         MOVEM.L DestRect(A5),D0/D1;   Get normalized rectangle
         MOVEM.L D0/D1,ViewRect(A5);   Store as view rectangle
         PEA     ViewRect(A5)    ;     Set
         _ClipRect               ;             clip region
         MOVEM.L (SP)+,D0-D2/A0-A3;    Restore the registers
         RTS                     ;     Back to caller
;
;        Here if a GoAway request
;
GoAway:
         CLR.W   -(SP)           ;     Room for result
         MOVE.L  WWindow(A5),-(SP) ;   Push window handle
         MOVE.L  Point(A5),-(SP) ;     Push mouse point
         _TrackGoAway            ;     See if legitimate
         TST.W   (SP)+           ;     Pop result
         BEQ     Doevent         ;     User chickened out
         MOVE.L  #CloseWin,D0    ;     Set special return code
         RETURNP                 ;     Bail out

NullEvent
Disk
Abort
MouseUp
KeyUp
Network
Driver
Appl
App2
App3
App4
UserEvt  MOVE.W  Modify(A5),D0   ;     Modify word
         SWAP    D0              ;     to D0 high word
         MOVE.W  What(A5),D0     ;     Set return code
         MOVE.L  Message(A5),D1  ;     Message to D1.L
         RETURNP                 ;     Restore & return
;
;        Control value routines.  These are called from the following macros:
;
;        Routine         Macro         Purpose
;        --------        -----         -------
;        XXSETVMIN       SetVMin       Set upper limit on VScroll bar
;        XXSETVMAX       SetVMax       Set lower limit on VScroll bar
;        XXSETVVAL       SetVVal       Set thumb value on VScroll bar
;        XXSETHMIN       SetHMin       Set left limit on HScroll bar
;        XXSETHMAX       SetHMax       Set right limit on HScroll bar
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
;         XXSETHVAL       SetHVal          Set thumb value on HScroll bar
;
;         Entry conditions:
;
;                 D0.W = Value
;
;         Exit conditions:
;
;                 (none)
;
          XDEF    XXSETVMIN           ;        Make
          XDEF    XXSETVMAX           ;
          XDEF    XXSETVVAL           ;               all
          XDEF    XXSETHMIN           ;
          XDEF    XXSETHMAX           ;                       external
          XDEF    XXSETHVAL           ;
XXSETVMIN                             ;        Set upper limit
          SAVE                        ;        Save registers
          MOVE.L  VScroll(A5),-(SP)   ;        Push control handle
          BRA     SetMin              ;        Do the SetMin
XXSETHMIN                             ;        Set left limit
          SAVE                        ;        Save registers
          MOVE.L  HScroll(A5),-(SP)   ;        Push control handle
SetMin    MOVE.W  D0,-(SP)            ;        and value
          _SetMinCtl                  ;        Set it
          RETURNALL                   ;        Back to caller
                                      ;
XXSETVMAX                             ;        Set lower limit
          SAVE                        ;        Save registers
          MOVE.L  VScroll(A5),-(SP)   ;        Push control handle
          BRA     SetMax              ;        Do set max
XXSETHMAX                             ;        Set right limit
          SAVE                        ;        Save registers
          MOVE.L  HScroll(A5),-(SP)   ;        Push control handle
SetMax    MOVE.W  D0,-(SP)            ;        Push value
          _SetMaxCtl                  ;        Set it
          RETURNALL                   ;        Done
                                      ;
XXSETVVAL                             ;        Set vertical thumb
          SAVE                        ;        Save the registers
          MOVE.L  VScroll(A5),-(SP)   ;        Push control handle
          BRA     SetVal              ;        Do set val
XXSETHVAL                             ;        Set horizontal thumb
          SAVE                        ;        Save registers
          MOVE.L  HScroll(A5),-(SP)   ;        Push control handle
SetVal    MOVE.W  D0,-(SP)            ;        Push value
          _SetCtlValue                ;        Set value
          RETURNALL                   ;        Exit
;
;
;
;         XXDIALOG routine.  Handles DIALOG macro.  (Also used for
;         the "About ..." box in Apple menu.)
;
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         XDEF    XXDIALOG         ;       For the linker
XXDIALOG                          ;
         MOVEM.L D1/D6/A0-A1,-(SP);       Push volatile registers
         CLR.L   -(SP)            ;       Dialog pointer return area
         MOVE.W  D0,-(SP)         ;       Dialog box resource ID
         CLR.L   -(SP)            ;       No dialog storage
         MOVE.L  #ONTOP,-(SP)     ;       On top of all
         _GetNewDialog            ;       Create a new dialog box
         MOVE.L  (SP),D6          ;       Handle to D6
         _SetPort                 ;       Make it QuickDraw port
       IF        TEXTRTL = 0      ;       If text style RTL
         MOVE.L  THandle(A5),-(SP) ;      Deactivate
         _TEDeActivate            ;                   text window
       ENDIF                      ;       Text
         CLR.L   -(SP)            ;       No filter procedure
         PEA     ItemHit(A5)      ;       -> Item hit data
         _ModalDialog             ;       Wait for "OK" button
         MOVE.L  D6,-(SP)         ;       Push dialog box handle
         _DisposDialog            ;       and close it
         MOVE.W  ItemHit(A5),D0   ;       Load return register
         MOVE.L  WindowPtr(A5),-(SP);     Reset
         _SetPort                 ;               port address
         MOVEM.L (SP)+,D1/D6/A0-A1;       Restore registers
         RTS                      ;       Back to caller
;
;
;        XXPRINT routine.  Handles the PRINT macro.
;
;        Entry conditions:
;
;                A4 -> Pascal string to print
;
;        Exit conditions:
;
;                (None)
;
       IF        TEXTRTL = 0      ;       Junk whole routine if graphics
         XDEF    XXPRINT          ;       Define for linker
XXPRINT  SAVE                     ;       Save registers
         MOVE.W  D0,-(SP)         ;<01>   Place D0 on stack temporarily
         BSR     DoPrint          ;<01>   Print original string
         TST.W   (SP)+            ;<01>   Need carriage return?
         BEQ     @1               ;<01>   No, quit now
         LEA     XXCR,A4          ;<01>   A4 -> Carriage-return string
         BSR     DoPrint          ;<01>   Go print that
@1       RETURNALL                ;<01>   All done, go away
DoPrint  CLR.L   D6               ;       Clear a temporary
         MOVE.B  (A4)+,D6         ;       Fetch count byte
         MOVE.L  A4,A1            ;       A1 -> Text
         MOVE.L  D6,D0            ;       D0 = Count
         SUBQ.L  #1,D0            ;       Adjust for DBRA count
         BLT     TYPEDone         ;       Done if < 0
         CLR.L   D7               ;       Initialize counter
LookCR:  CMPI.B  #CR,(A1)+        ;       Is it a <CR>?
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         BNE      TYPEDBR          ;        No, don't bump count
         ADDQ.L   #1,D7            ;        Bump count
TYPEDBR  DBRA     D0,LookCR        ;        Repeat till count exhausted
         TST.W    D7               ;        Any line control chars?
         BEQ      DoType           ;        No, skip next
         MOVE.L   THandle(A5),A1   ;        A1 -> Text handle
         MOVE.L   (A1),A1          ;        A1 -> Actual text record
         CLR.L    D5               ;        Clear out D5 long
         MOVE.L   WindowPtr(A5),A0;         A0 -> Window record
         MOVE.W   PortRect+4(A0),D5;        D5 = Rectangle
         SUB.W    PortRect+0(A0),D5;                         size (vertical)
         DIVS     teLineHite(A1),D5;        D5 = # lines in dest rectangle
         SUB.W    teNLines(A1),D5  ;        How many already?
         BGT      GotSome          ;        Still some left at bottom
         CLR.L    D5               ;        Correct for none left
GotSome  SUB.W    D7,D5            ;        How many lines do we need?
         BGE      DoType           ;        None
         MOVE.W   D5,D0            ;        Copy to scratch register
         MULS.W   teLineHite(A1),D0;        Multiply to convert to pixels
         CLR.W    -(SP)            ;        No horizontal scroll
         MOVE.W   D0,-(SP)         ;        Push vertical scroll
         MOVE.L   THandle(A5),-(SP);        Push text handle
         _TEScroll                 ;        Scroll it
         MOVE.L   THandle(A5),A1   ;        A1 -> Text handle
         MOVE.L   (A1),A1          ;        A1 -> Actual text record
         MOVE.L   teViewRect(A1),teDestRect(A1);  Adjust vertical view for deletion
         MOVE.L   teViewRect+4(A1),teDestRect+4(A1);      "
         NEG.W    D5               ;        D5 = # lines to delete
         SUB.W    D5,teNlines(A1)  ;        Mark as deleted
         ADD.W    D5,D5            ;        Convert to index
         MOVE.W   teLines(A1,D5.W),D0;      First character to remain
         MOVE.L   teTextH(A1),A2   ;        A2 = Text handle
         MOVE.L   (A2),A2          ;        A2 -> Text start (destination)
         LEA      (A2,D0.W),A3     ;        A3 -> Source
         MOVE.W   teLength(A1),D1  ;        Count of characters to move
         SUB.W    D0,D1            ;        Fix for deletion
         EXT.L    D1               ;        Out to long
         MOVE.W   D1,D0            ;        Adjust
         SUBQ.W   #1,D0            ;              for DBRA
         MOVE.L   A2,A0            ;        Save address in A0
TypeDel  MOVE.B   (A3)+,(A0)+      ;        Move a byte
         DBRA     D0,TypeDel       ;        Until done
         MOVE.L   A2,-(SP)         ;        -> Text
         MOVE.L   D1,-(SP)         ;        = Character count
         MOVE.L   THandle(A5),-(SP);        -> -> Text record
         _TESetText                ;        "Replace" text in TE record
DoType:                            ;        Here to begin output
         MOVE.L   A4,-(SP)         ;        Push character pointer
         MOVE.L   D6,-(SP)         ;        Push length
         MOVE.L   THandle(A5),-(SP);        and text handle
         _TEInsert                 ;        Output the text
TYPEDone:                          ;        Here when done
         RETURNALL                 ;        Restore registers
       ENDIF                       ;        TEXTRTL
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
;
;
;       XXINPUT routine.  Called by INPUT macro.
;
;       Entry conditions:
;
;               A0 -> User buffer
;               D0 =  Count of bytes available
;
;       Exit conditions:
;
;               (none)
;
        IF      TEXTRTL = 0     ;       If text style RTL
        XDEF    XXINPUT
XXINPUT SAVE                    ;       Save user's registers
        CMP.W   #255,D0         ;       D0 > 1 byte?
        BLE     CountOK         ;       No, count is OK
        MOVE.W  #255,D0         ;       Use only 255
CountOK                         ;
        MOVE.W  D0,D4           ;       Preserve in nonvolatile reg
        MOVE.L  A0,A2           ;       Save buffer pointer
        MOVE.L  A2,A3           ;       in nonvolatile registers
        TST.B   (A3)+           ;       -> First character position
        CLR.W   D3              ;       = Character count so far...
InputLoop                       ;       Here to get next character
        NEXTEVT #KeyMask        ;       Get next keyboard event
        BEQ     InputLoop       ;       No, just ignore
        BTST    #ComKey,Modify(A5);     Command key down?
        BNE     InputLoop       ;       Yes, ignore
        MOVE.W  Message+2(A5),D0;       Character to D0
        CMP.B   #BS,D0          ;       Backspace?
        BLT     InputLoop       ;       Invalid character
        BNE     NotBS           ;       Check for control chars.
        TST.W   D3              ;       Any characters?
        BEQ     InputLoop       ;       No, invalid character
        SUBQ.W  #1,D3           ;       Decrement count
        TST.B   -(A3)           ;       Remove from buffer
        BRA     DoKey           ;       Do TEKey for screen update
NotBS:                          ;       Here to see if control character
        CMP.B   #CR,D0          ;       Was it carriage return?
        BEQ     IsCR            ;       Yes, store count and return
        CMP.B   #BL,D0          ;       Less than space?
        BLT     InputLoop       ;       Yes, error
        CMP.W   D3,D4           ;       Room in buffer?
        BLE     InputLoop       ;       No, don't do anything
        MOVE.B  D0,(A3)+        ;       Store in buffer
        ADDQ.L  #1,D3           ;       Increment count
DoKey:  MOVE.W  D0,-(SP)        ;       Now put on screen
        MOVE.L  THandle(A5),-(SP);
        _TEKey                  ;       and in TE buffer
        BRA     InputLoop       ;       Get next character
IsCR:                           ;       Here when CR typed
        MOVE.B  D3,(A2)         ;       Store the count byte
```

Listing E.18 *– The RTLcode.asm file (continued)*

```
        SUB.W   D3,D4           ;       How many bytes remain?
        BLE     PrintCR         ;       None, bail out
        SUB.W   #1,D4           ;       Adjust for DBRA
ZapBuf: CLR.B   (A3)+           ;       Clean out
        DBRA    D4,ZapBuf       ;               rest of buffer
PrintCR LEA     XXCR,A4         ;       A4 -> CR sequence
        JMP     DoPrint         ;       Print CR, then return
      ENDIF                     ;       TEXTRTL
;
;       XXOPENR routine.  Called by OPENR macro.
;
;       Entry conditions:
;
;               A0 -> filename
;
;       Exit conditions:
;
;               D0 = File handle or error code
;
;       This routine performs a volume search if the default open
;       fails.
;
        XDEF    XXOPENR
XXOPENR:                        ;       Entry point
        SAVE                    ;       Save the registers
        LEA     TRYRopen,A6     ;       A6 -> Open function
        CLR.L   Filebuf(A5)     ;       No special file buffer
        BRA     Gopen           ;       Go to general open function
;
;       This routine is called internally to do the OpenResFile trap, instead
;       of the generalized resource file open.
;
OpenRsrc:                       ;       Entry point
        SAVE                    ;       Save registers
        LEA     TRYRESOP,A6     ;       -> Open function
        CLR.L   Filebuf(A5)     ;       No special file buffer
        CLR.W   D1              ;       Default volume search
        BRA     Gopen           ;       Go do it
;
;       XXOPEND routine.  Called by OPEN macro.
;
;       Entry conditions:
;
;               A0 -> File name
;
;       Exit conditions:
;
;               D0 = File handle or error code
;               D1 = Volume reference number or zero
;
        XDEF    XXOPEND
XXOPEND SAVE                    ;       Entry point
        LEA     TRYDopen,A6     ;       A6 -> Open function
        MOVE.L  A1,Filebuf(A5)  ;       Save buffer address
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
;
;       General Open function. Called by resource and data open
;       front ends.
;
;       A6 -> Routine for open
;       A0 -> File name
;
Gopen:  MOVE.L  A0,A4           ;       Save file name pointer
        JSR     (A6)            ;       Try to open as is
        TST.W   D0              ;       Success?
        BGT     RQuit           ;       Yes, return
        TST.W   D1              ;       Volume specified?
        BNE     RQuit           ;       Yes, fail
        CLR.L   D6              ;       Load
        MOVE.B  (A4),D6         ;               length byte
        MOVE.L  VCBQHdr+2,A3    ;       A3 -> VCB entry
RVScan: TST.W   VCBDrvNum(A3)   ;       Is volume on line?
        BEQ     RNextV          ;       No, get next
        LEA     FScratch+1(A5),A1;      A1 -> Scratch area
        LEA     VCBVN(A3),A2    ;       A2 -> Volume name
        CLR.L   D7              ;       Fetch
        MOVE.B  (A2),D7         ;               count byte
        ADD.L   #1,D7           ;       Increment for ":" character
        ADD.W   D6,D7           ;       Compute total string length
        BSR     MoveCnt         ;       Move in volume name
        MOVE.B  #':',(A1)+      ;       Move in separator
        MOVE.L  A4,A2           ;       A2 -> File name
        BSR     MoveCnt         ;       Move file name in, too
        LEA     Fscratch(A5),A0 ;       A0 -> Composite name
        MOVE.B  D7,(A0)         ;       Plunk down length byte
        CLR.W   D1              ;       Default volume number
        JSR     (A6)            ;       Try to open that
        MOVE.W  VCBVREFNum(A3),D1;      Return volume ref number in D1
        TST.W   D0              ;       Success?
        BGT     RQuit           ;       Yes, stop
RNextV:                         ;       No, chain to next volume
        MOVE.L  (A3),A3         ;       A3 -> Next in chain
        MOVE.L  A3,D1           ;       Move to D reg
        BNE     RVScan          ;       Try again
        MOVE.L  #-1,D0          ;       Failed utterly
RQuit:  RETURNP                 ;       Restore registers; return
;
;       TRYRESOP routine.  Try to open resource fork of a file, for use
;       as the program's resource file.
;
TRYRESOP:                       ;       A0 -> File name
        CLR.W   -(SP)           ;       Clear out room
        MOVE.L  A0,-(SP)        ;       Push file name
        _OpenResFile            ;       Open
        MOVE.W  (SP)+,D0        ;       Fetch file handle/error code
        RTS                     ;       Return to caller
;
;       TRYDopen routine.  Try to open data fork of a file.
;
```

Listing E.18 – The RTLcode.asm file (continued)

```
TRYDopen:                               ;       A0 -> File name
        MOVE.L  A0,ioFileName+IOPB(A5); Put in file name
        LEA     IOPB(A5),A0     ;       A0 -> Parameter block
        CLR.L   ioCompletion(A0);       No completion routine
        MOVE.W  D1,ioVRefNum(A0);       Set volume ID
        CLR.W   ioFileType(A0)  ;       Default version
        CLR.B   ioPermssn(A0)   ;       Whatever we can get
        MOVE.L  Filebuf(A5),ioOwnBuf(A0); Special access buffer
        _Open                   ;       Try it
        LEA     IOPB(A5),A0     ;       A0 -> Parameter block again
        MOVE.W  ioResult(A0),D0 ;       Fetch error code
        BNE     OpenFail        ;       IF NE, back to caller
        MOVE.W  ioRefNum(A0),D0 ;       Get return parameter
OpenFail:                       ;       Common exit point
        EXT.L   D0              ;       Convert to long
        RTS                     ;       and back to caller
;
;       TRYRopen routine.  Try to open Resource fork of a file.
;
TRYRopen:                       ;       A0 -> File name
        MOVE.L  A0,ioFileName+IOPB(A5); Put in file name
        LEA     IOPB(A5),A0     ;       A0 -> Parameter block
        CLR.L   ioCompletion(A0);       No completion routine
        MOVE.W  D1,ioVRefNum(A0);       Set volume ID
        CLR.W   ioFileType(A0)  ;       Default version
        CLR.B   ioPermssn(A0)   ;       Whatever we can get
        MOVE.L  Filebuf(A5),ioOwnBuf(A0);       No special access buffer
        _OpenRF                 ;       Try it
        LEA     IOPB(A5),A0     ;       A0 -> Parameter block again
        MOVE.W  ioResult(A0),D0 ;       Fetch error code
        BNE     ROpenFail       ;       IF NE, back to caller
        MOVE.W  ioRefNum(A0),D0 ;       Get return parameter
ROpenFail:                      ;       Common exit point
        EXT.L   D0              ;       Convert to long
        RTS                     ;       and back to caller
;
;       XXPOPEN routine.  Opens a file using the Mac standard file
;       package.
;
;       Entry conditions:
;
;       A0 -> Extension string
;       A1 -> Buffer
;       A2 -> Return buffer for name
;       A3 -> Table of allowable file types
;
;       Exit conditions:
;
;       D0 = File reference number
;       D1 = Volume reference number
;
        XDEF    XXPOPEN         ;       For linker
XXPOPEN:                        ;       Entry point
        SAVE                    ;       Save caller's registers
```

Listing E.18 *– The RTLcode.asm file (continued)*

```
         MOVE.L  A0,ExtPtr(A5)    ;      Save the extension pointer
         MOVE.L  A1,Filebuf(A5)   ;      Place buffer in magic location
         MOVE.L  SfPoint,-(SP)    ;      Push point for box upper left
         CLR.L   -(SP)            ;      Push null prompt string
         PEA     Extfilter        ;      Use standard filter procedure
         MOVE.W  #-1,-(SP)        ;      Push default (no type filtering)
         MOVE.L  A3,D0            ;      Test for
         BEQ     NoTypes          ;               types there
         MOVE.W  (A3)+,(SP)       ;      Replace default with actual number
NoTypes:                          ;
         MOVE.L  A3,-(SP)         ;      Push type list address (or nil)
         CLR.L   -(SP)            ;      No dialog hook procedure either
         PEA     SFReply(A5)      ;      Push reply area pointer
         MOVE.W  #SFGet,-(SP)     ;      Push routine indicator
         _Pack3                   ;      Try the file get
         TST.B   Good(A5)         ;      Test return code
         BNE     NotCancelled     ;      NE => Not cancelled
         MOVE.L  #Cancel,D0       ;      Return
         BRA     XXPFAIL          ;      Failed, return
NotCancelled:                     ;      Here if OK
         MOVE.W  VolRef(A5),D1    ;      Set volume ID
         LEA     Fname(A5),A0     ;      A0 -> File name
         JSR     TryDOpen         ;      Try to open
         TST.W   D0               ;      Worked?
         BMI     XXPFAIL          ;      No, quit now
         MOVE.B  Fname(A5),(A2)+  ;      Move in count byte
         MOVE.L  A2,A1            ;      A1 -> Destination
         LEA     Fname(A5),A2     ;      A2 -> Source
         JSR     MoveCnt          ;      Move in name
         MOVE.W  ioVRefNum(A0),D1;       Set volume ref number
XXPFAIL:
         MOVE.L  WindowPtr(A5),-(SP);    Push GrafPort address
         _SetPort                 ;      Restore port
         RETURNP                  ;      Restore registers & return
;
;        The ExtFilter procedure is used to look at each file name
;        to see if it fits the display.
;
ExtFilter:                        ;      Filter file names
         TST.L   ExtPtr(A5)       ;      Any to filter
         BEQ     Success          ;      No, just bail out
         MOVE.L  ExtPtr(A5),A0    ;      A0 -> Extension string
         MOVE.L  4(A7),A1         ;      A1 -> IOPB from SF package
         MOVE.L  ioFileName(A1),A1;      A1 -> File name area
         CLR.L   D0               ;      Load extension
         MOVE.B  (A0)+,D0         ;                        count byte
         CLR.L   D1               ;      Load file name
         MOVE.B  (A1)+,D1         ;                        count byte
         SUB.W   D0,D1            ;      Adjust count
         BLT     Failure          ;      Lengths are incompatible
         LEA     0(A1,D1.W),A1    ;      Point A1 to comparison byte
         SUBQ.W  #1,D0            ;      Adjust for DBRA
         BLE     Failure          ;      If count <= 0, quit
ExtLoop:                          ;      Now compare byte by byte
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
        MOVE.B  (A1)+,D1        ;       Fetch character
        CMPI.B  #'a',D1         ;       Lower-
        BLT     DoCompare       ;               case?
        CMPI.B  #'z',D1         ;
        BGT     DoCompare       ;
        BCLR    #5,D1           ;       Convert to uppercase
DoCompare:                      ;       Here to compare
        CMP.B   (A0)+,D1        ;       Compare next
        BNE     Failure         ;       If NE, quit
        DBRA    D0,ExtLoop      ;       Repeat for count
Success:                        ;       Here if match
        MOVE.W  #0,8(SP)        ;       Load Boolean TRUE
        MOVE.L  (SP)+,(SP)      ;       Move return -> parameter
        RTS                     ;       Exit
Failure:                        ;       Here if no match
        MOVE.W  #-1,8(SP)       ;       Load Boolean FALSE
        MOVE.L  (SP)+,(SP)      ;       Move return -> parameter
        RTS                     ;       Exit
;
;       XXCREATE routine.  Creates a new file.
;
;       Entry conditions:
;
;       A0 -> File name
;
;       Exit conditions:
;
;       D0 = Error code/file handle
;
        XDEF    XXCREATE        ;       For the linker
XXCREATE:                       ;
        SAVE                    ;       Save caller's registers
        JSR     TryCreate       ;       Call internal routine
        RETURNP                 ;       Restore all but D0/D1

TryCreate:                      ;       Here if locally called
        MOVE.L  A0,ioFileName+IOPB(A5); Set file name
        MOVE.L  A1,A3           ;       Save A1
        LEA     IOPB(A5),A0     ;       A0 -> IOPB
        CLR.L   ioCompletion(A0);       No completion routine
        MOVE.W  D1,ioVRefNum(A0);       Specified volume
        CLR.W   ioFileType(A0)  ;       Default version
        CLR.B   ioPermssn(A0)   ;       Whatever we can get
        CLR.L   ioOwnBuf(A0)    ;       No special access buffer
        _Delete                 ;       Get rid of existing version
        _Create                 ;       Do it
        _GetFileInfo            ;       Get file information
        MOVE.L  #'TEXT',ioFlUsrWds(A0); Set file type
        MOVE.L  #'EDIT',ioFlUsrWds+4(A0); Set application
        _SetFileInfo            ;       Set the info
        MOVE.L  A3,ioOwnBuf(A0) ;       Set buffer address
        _Open
        MOVE.W  ioResult(A0),D0 ;       Get error code
        EXT.L   D0              ;       Extend to long
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
        BNE     CreateNFG        ;      Failed if nonzero
        MOVE.W  ioRefNum(A0),D0  ;      Get reference number
        MOVE.W  ioVRefNum(A1),D1;       Return volume reference number
CreateNFG:                       ;      Common exit point
        RTS                      ;      Return to caller
;
;
;       XXPCREATE routine.  This routine calls the standard file package
;       to create an output file.
;
;       Entry conditions:
;
;       A0 -> Prompt string
;       A1 -> 522 byte buffer for use by file system
;       A2 -> Name buffer return area (64 bytes)
;       A3 -> Default name
;
;       Exit conditions:
;
;       D0 = File reference number
;       D1 = Volume reference number
;
        XDEF    XXPCREATE
XXPCREATE:                       ;      Entry point
        SAVE                     ;      Save registers
        MOVE.L  A1,Filebuf(A5)   ;      Save buffer address
        MOVE.L  SFPoint,-(SP)    ;      Point for dialog box
        MOVE.L  A0,-(SP)         ;      Prompt string
        MOVE.L  A3,-(SP)         ;      Default name
        CLR.L   -(SP)            ;      No dialog hook procedure
        PEA     SFReply(A5)      ;      -> Reply area
        MOVE.W  #SFPut,-(SP)     ;      Routine selector
        _Pack3                   ;      Invoke package
        TST.B   Good(A5)         ;      Test return code
        BNE     NoCancel         ;      NE => Not cancelled
        MOVE.L  #Cancel,D0       ;      Return
        BRA     XXPFAIL          ;      Failed, return
NoCancel:                        ;      Here if OK
        MOVE.W  VolRef(A5),D1    ;      Set volume ID
        LEA     Fname(A5),A0     ;      A0 -> File name
        MOVE.L  Filebuf(A5),A1   ;      Restore A1
        JSR     TryCreate        ;      Try to create it
        TST.W   D0               ;      Worked?
        BMI     XXPFAIL          ;      No, quit now
        MOVE.B  Fname(A5),(A2)+  ;      Move in count byte
        MOVE.L  A2,A1            ;      A1 -> Destination
        LEA     Fname(A5),A2     ;      A2 -> Source
        JSR     MoveCnt          ;      Move in name
        MOVE.W  ioVRefNum(A0),D1;       Set volume ref number
        BRA     XXPFAIL          ;      Go exit

;
;       XXGETF routine.  Gets file info and returns to the user.
;
```

Listing E.18 *– The RTLcode.asm file (continued)*

```
;       Entry:
;
;               A0 -> File name
;               A1 -> Information buffer
;               D1 =  Volume reference number or 0 for scan
;
        XDEF    XXGETF
XXGETF:                           ;       Entry point
        SAVE                      ;       Save caller's registers
        LEA     TryGetF,A6        ;       Do volume scan for file
        MOVE.L  A1,FileBuf(A5)    ;       Save A1 in global area
        BRA     Gopen             ;       Do general open stuff
;
;       TryGetF routine.  Attempts to get file info for file.
;
TryGetF:                          ;
        MOVE.L  A0,IOPB+ioFileName(A5); Stuff file name
        LEA     IOPB(A5),A0       ;       A0 -> IOPB now
        MOVE.W  D1,ioVRefNum(A0);         Clear vol ref field
        CLR.W   ioFDirIndex(A0)   ;       Clear directory index field
        _GetFileInfo              ;       Try it
        MOVE.W  ioResult(A0),D0   ;       Error?
        BNE     GETNFG            ;       Yes, return error
        MOVE.W  #ioFQElSize-1,D0;         Load count
        LEA     IOPB(A5),A2       ;       A2 -> IOPB
        MOVE.L  FileBuf(A5),A1    ;       A1 -> User's buffer
GetMove:                          ;       Move it in
        MOVE.B  (A2)+,(A1)+       ;       Stuff a byte
        DBRA    D0,GetMove        ;       Until done
        MOVE.W  ioRefNum(A0),D0   ;       Load ref number
GETNFG: RTS                       ;       and bail out

;
;       SETF Routine.  Sets file info for named file.
;
;       Entry conditions:
;
;       A0 -> File name
;       A1 -> Buffer area
;
;       Exit conditions:
;
;       D0 = ioResult
;
;
        XDEF    XXSETF            ;       For linker
XXSETF: SAVE                      ;       Save registers
        LEA     IOPB(A5),A2       ;       A2 -> IOPB
        MOVE.W  #ioFQElSize-1,D0;         D0 = Count
SetMove:                          ;       Move parameters back
        MOVE.B  (A1)+,(A2)+       ;       Move a byte
        DBRA    D0,SetMove        ;       Loop till done
        MOVE.L  A0,IOPB+ioFileName(A5); Stuff file name
        LEA     IOPB(A5),A0       ;       A0 -> IOPB now
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
         MOVE.W  D1,ioVRefNum(A0);      Set volume reference number
         _SetFileInfo            ;      Try it
         MOVE.W  ioResult(A0),D0 ;      Get result code
         EXT.L   D0              ;      Extend to long
         RETURN                  ;      and bail out

;
;        XXREAD routine.  Reads from an open file.
;
;        Entry conditions:
;
;        D0 = File handle
;        D1 = Byte count
;        A0 -> Buffer
;        D2 = End-of-line character
;
;        Exit conditions:
;
;        D0 = # bytes actually read
;
         XDEF    XXREAD          ;      For the linker
XXREAD:  SAVE                    ;      Save caller's registers
         LEA     IOPB(A5),A1     ;      A1 -> Parameter block
         MOVE.W  D0,ioRefNum(A1) ;      Store file reference number
         MOVE.L  D1,ioByteCount(A1);    Store byte count
         MOVE.L  A0,ioBuffer(A1) ;      and buffer address
         MOVE.W  D2,ioPosMode(A1);      Specify sequential read
         MOVE.L  A1,A0           ;      A0 -> Parameter block
         _Read                   ;      Issue the read
         LEA     IOPB(A5),A0     ;      A0 -> Parameter block again
         MOVE.W  ioResult(A0),D0 ;      Fetch result code
         EXT.L   D0              ;      Extend to long
         BEQ     Noerr           ;      No error
         CMPI.L  #eofERR,D0      ;      End of file error?
         BNE     ReadDone        ;      No, return error
         TST.L   ioNumDone(A0)   ;      Any bytes read?
         BEQ     ReadDone        ;      No, return error
Noerr:   MOVE.L  ioNumDone(A0),D0;      Not (really) an error, return byte count
ReadDone:                        ;      Here when done
         RETURN                  ;      Restore the registers
;
;        XXWRITE routine.  Writes to an open file.
;
;        Entry conditions:
;
;        D0 = File handle
;        D1 = Byte count
;        A0 -> Buffer
;        D2 = End-of-line character
;
;        Exit conditions:
;
;        D0 = # bytes actually written/error count
;
```

***Listing E.18** – The RTLcode.asm file (continued)*

```
         XDEF    XXWRITE          ;      For the linker
XXWRITE  SAVE                     ;      Save caller's registers
         LEA     IOPB(A5),A1      ;      A1 -> Parameter block
         MOVE.W  D0,ioRefNum(A1)  ;      Store file reference number
         MOVE.L  D1,ioByteCount(A1);     Store byte count
         MOVE.L  A0,ioBuffer(A1)  ;      and buffer address
         CLR.W   ioPosMode(A1)    ;      Indicate sequential
         MOVE.L  A1,A0            ;      A0 -> Parameter block
         _Write                   ;      Issue the write
         LEA     IOPB(A5),A0      ;      A0 -> Parameter block again
         MOVE.W  ioResult(A0),D0  ;      Fetch result code
         EXT.L   D0               ;      Extend to long
         BNE     WriteDone        ;      NE => Error, done
         MOVE.L  ioNumDone(A0),D0;       No error, return actual byte count
WriteDone:                        ;      Here when done
         RETURN                   ;      Restore the registers
;
;        XXSEEK routine.  Called by SEEK macro.
;
;        Entry conditions:
;
;        D0  = I/O reference number
;        D1  = File offset
;        D2  = Sense word
;
         XDEF    XXSEEK
XXSEEK:  SAVE                     ;      Seek on an open file
         LEA     IOPB(A5),A0      ;      A1 -> Parameter block
         MOVE.W  D0,ioRefNum(A0)  ;      Store file reference number
         MOVE.L  D1,ioPosOffset(A0);     Store offset
         ANDI.W  #3,D2            ;      Insure proper range
         MOVE.B  SeekTab(D2),D2   ;      Translate to Macintosh code
         MOVE.W  D2,ioPosMode(A0);       Set proper field
         _SetFPos                 ;      Do the set
         LEA     IOPB(A5),A0      ;      A0 -> Parameter block
         MOVE.W  ioResult(A0),D0  ;      Fetch error code
         EXT.L   D0               ;      Out to long
         BMI     SeekDone         ;      < 0 means error
         _GetFPos                 ;      Fetch new pointer
         LEA     IOPB(A5),A0      ;      A0 -> Parameter block
         MOVE.L  ioPosOffset(A0),D0;     Load return value
SeekDone:                         ;      Here to return to caller
         RETURN                   ;      Restore caller's registers
SeekTab  DC.B    1,3,2            ;      Seek translation table
         .ALIGN  2                ;      Force word boundary

;
;        XXDELETE routine.  Deletes a file by name.
;
;        Entry conditions:
;
;        A0 -> File name string
;        D1 =  Volume reference number or zero for scan.
;
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
              XDEF    XXDELETE
     XXDELETE:                         ;
              SAVE                     ;       Save the registers
              LEA     TryDel,A6        ;       Do volume scan for file
              BRA     Gopen            ;       Do general open stuff
     ;
     ;        TryDel routine.  Attempts to delete a file.
     ;
     TryDel:                           ;
              MOVE.L  A0,IOPB+ioFileName(A5); Stuff file name
              LEA     IOPB(A5),A0      ;       A0 -> IOPB now
              MOVE.W  D1,ioVRefNum(A0);        Clear vol ref field
              CLR.W   ioFDirIndex(A0)  ;       Clear directory index field
              _Delete                  ;       Try it
              MOVE.W  ioResult(A0),D0  ;       Load return code
              EXT.L   D0               ;       Extend to longword
              RTS                      ;       and bail out
     ;
     ;        XXCLOSE routine.  Called by CLOSE macro.
     ;
     ;        Entry conditions:
     ;
     ;        D0.W = File number to close
     ;
              XDEF    XXCLOSE          ;       For linker
     XXCLOSE  SAVE                     ;       Save caller's registers
              LEA     IOPB(A5),A0      ;       A0 -> Parameter block
              MOVE.W  D0,ioRefNum(A0)  ;       Set reference number
              _Close                   ;       Close the file
              BSR     DoFlush          ;       Flush all volumes
              RETURNALL                ;       Return to caller
     ;
     ;        XXEXIT routine. Called by EXIT macro.
     ;
     ;        Entry conditions:
     ;
     ;        (None)
     ;
     ;        Exit conditions:
     ;
     ;        (Doesn't return)
     ;
              XDEF    XXEXIT           ;       For linker
     XXEXIT:  BSR     DoFlush          ;       Flush vols first
              MOVE.L  EXITAddr(A5),A0  ;       A0 -> Exit routine
              MOVE.L  ORGSP(A5),SP     ;       Take stack to original
              JMP     (A0)             ;       Quit
     ;
     ;        DoExit routine.  Called from dialog manager when BOMB dialog appears.
     ;
     DoExit                            ;<01>   New with edit <01>
              BSR     DoFlush          ;<01>   Flush disks
              _ExitToShell             ;<01>   Exit
     ;
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
;       DoFlush routine.  Flushes all volumes.
;
;
DoFlush MOVE.L  VCBQHdr+2,A3    ;       A3 -> VCB entry
EVScan: TST.W   VCBDrvNum(A3)   ;       Is volume online?
        BEQ     ENextV          ;       No, get next
        LEA     IOPB(A5),A0     ;       A0 -> I/O Parameter block
        MOVE.L  VCBVN(A3),ioVNPtr(A0);  Set volume name
        CLR.W   ioVDrvNum(A0)   ;       Specify no drive number
        _FlushVol               ;       Flush the volume
ENextV:                         ;       No, chain to next volume
        MOVE.L  (A3),A3         ;       A3 -> Next in chain
        MOVE.L  A3,D1           ;       Move to D reg
        BNE     EVScan          ;       Try again
        RTS                     ;       Back to caller
```

Listing E.18 – *The RTLcode.asm file (continued)*

```
;
;       RTL data area
;
;       Initialized data
;
           STRING_FORMAT 3      ;       All strings are Pascal strings
DWBounds:  DC.W  28,4,338,508   ;       DragWindow bounds rectangle
           IF    CONTROL = 0    ;       Only for controls
GWBounds:  DC.W  28,50,300,508  ;       GrowWindow bounds rectangle
           ENDIF                ;       CONTROL = 0
SFPoint:   DC.W  100,100        ;       Point for standard file window
           IF    RESFILE = 0    ;       Resource file separate
Canned:    DC.B  'RTL.RSRC'     ;       Default resource file name
RSRC:      DC.B  '.RSRC'        ;       Resource extension
RSRCLEN    EQU   *-RSRC-1       ;       Adjusted byte count
           ENDIF                ;       RESFILE = 0
           XDEF  XXCR           ;       Newline sequence for TYPE macro
XXCR       DC.B  1,13           ;       Carriage return
          .ALIGN 2              ;       Align to next word
;
;       Uninitialized data
;
EXITaddr:  DS.L  1              ;       Exit address to Finder
ORGSP:     DS.L  1              ;       Stack pointer from Finder
FileBuf:   DS.L  1              ;       File buffer address for open
Scratch:   DS.B  32             ;       32-byte resource name scratch area
Fscratch:  DS.B  32             ;       32-byte file name scratch area
           IF    RESFILE = 0    ;       Only if separate resource file
ResFD:     DS.W  1              ;       Resource file descriptor
           ENDIF                ;       RESFILE = 0
```

Listing E.19 – *The RTLdata.asm file*

```
WindowPtr DS.L  1               ;       Returned window pointer
ViewRect: DS.W  4               ;       Text record's view rectangle
DestRect: DS.W  4               ;       Text record's destination rectangle
          IF    TEXTRTL = 0     ;       If text RTL
THandle:  DS.L  1               ;       Text record handle
          ENDIF                 ;       TEXTRTL
GPort:  DS.L    1               ;       Current GrafPort
WArea:    DS.W  WindowSize      ;       Window storage area
ItemHit:  DS.W  1               ;       Modal dialog result
DeskName: DS.W  16              ;       Desk accessory name
Cursor:   DS.W  1               ;       Cursor code for window
xCursor:  DS.W  1               ;       Last cursor code for window

;
;       Event record for get next event
;
EventRecord DS.W        0       ; 0
What:       DS.W        1       ; 2     Event type
Message:    DS.L        1       ; 4     Event message
When:       DS.L        1       ; 8     Event time
Point:      DS.L        1       ;12     Mouse position
Modify:     DS.W        1       ;14     Modifiers

WWindow:    DS.L        1       ;       Input to FindWindow
SaveRegs:   DS.L        15      ;       Register save area
;
;       I/O parameter block for use with file I/O
;
IOPB:       DS.W        0       ;       Global tag
            DS.B        ioFQELSize;     Define area
;
;
;       Reply area for standard file package
;
SFReply:    DS.W        0       ;       Structure tag
Good:       DS.B        1       ;       Boolean TRUE => not cancelled by user
Copy:       DS.B        1       ;       Currently unused
fType:      DS.L        1       ;       O/S file type
VolRef:     DS.W        1       ;       Volume reference number
version:    DS.W        1       ;       File version number
Fname:      DS.B        64      ;       Returned file name
ExtPtr:     DS.L        1       ;       -> User-specified file extension string
;
;       Memory area for window controls
;
VScroll:    DS.L        1       ;       Handle for vertical scroll bar
HScroll:    DS.L        1       ;       Handle for horizontal scroll bar
CHandle:    DS.L        1       ;       Handle for  FindControl
```

***Listing E.19** – The RTLdata.asm file (continued)*

```
*
* This is the resource file for the Mac run-time library
*

RTL.RSRC



Type MENU
  ,1
\14
 RTL Programs
 (-

  ,2
File
  Quit

  ,3
Edit
  Cut/X
  Copy/C
  Paste/V


Type DLOG
  ,1

100 100 190 500
Visible  NoGoAway
1
0
1


Type DITL
  ,1
3

StaticText
15 20 36 400
This Program uses the special Mac Run Time Library

StaticText
35 20 56 400
By Steve Williams

Button
60 260 80 320
OK


Type WIND
```

Listing E.20 – *The RTL.R file*

```
   ,1

40 5 300 500
Visible NoGoAway
0
0


*
*       Scroll bar controls for main window
*

Type    CNTL
   ,256(4)
x
   40 485 300 500
   Visible
   16
   0
   0 720 0

Type    CNTL
   ,257(4)
x
   285 5 300 500
   Visible
   16
   0
   0 576 0
```

Listing E.20 *– The RTL.R file (continued)*

```
Asm     RTL.Asm Exec    Edit
Rmaker  RTL.R   Exec    Edit
```

Listing E.21 *– The RTL.job file*

```
;
;       Macro library file for simplified Macintosh applications
;
;       Version 01.01   31-Aug-85
;
```

Listing E.22 *– The MACLIB.ASM file*

```
;       Revision history:
;
;       <01>    31-Aug-85       Revision for Exercise 8.2
;
;
;       PROGRAM macro.  This macro is used to define an application.
;       It must be the first code-generating statement in the program.
;
;       Usage:
;
;       PROGRAM name,debug,menus
;
;       Where name is the applications name (used to find the
;       resource file, for instance).
;
;       Debug is either DEBUGON or DEBUGOFF, to make the program's
;       first instruction an illegal instruction ($FF01) or a NOP.
;       This can later be patched.
;
;       Menus is the number of menus in the main menu bar, or the
;       DEFAULT token.
;
        STRING_FORMAT 3         ; Set all strings to Pascal standard
        .MACRO  PROGRAM         ; name,menus,debug =
        XDEF    XXSTART         ; Entry point name
        XREF    XXINIT          ; Initializer
XXSTART JSR     XXINIT          ; Call init routine
        IF      '%1' <> ''      ; Name specified
        DC.B    '%1'            ; Yes
        ELSE                    ;
        DC.B    'No Name'       ; Default
        ENDIF
        .ALIGN  2               ; Force word alignment
        IF      '%3' <> ''      ; Menus?
        DC.W    %3              ; Do specified number
        ELSE                    ; Not specified
        DC.W    0               ; Use default
        ENDIF
        IF      '%2' <> ''      ; Debug specified?
        DC.W    %2              ; Yes
        ELSE                    ; Not specified
        DC.W    $4E71           ; Use NOP instruction
        ENDIF
        .ENDM
DEFAULT  EQU    0               ; Default parameter
DEBUGON  EQU    $FF01           ; Debug option
DEBUGOFF EQU    $4E71           ; No debug option
;
;
;
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
;
;       PRINT macro.  Prints a Pascal string on the main window.
;
;       Usage:
;
;       PRINT    'String'[,crflag]
;
;            -or-
;
;       PRINT    address[,crflag]
;
;       The second form allows use of a resource file, or an arbitrary
;       addressing mode.  "crflag", if nonblank, specifies no newline.
;
        .MACRO  PRINT              ;string,crflag =
        XREF    XXPRINT            ;Define for linker
        MOVEM.L D0/A4,-(SP)        ;<01> Save D0 + A4
        LEA     %1,A4              ;A4 -> Output string
        IF      '%2' <> ''         ;Is carriage return desired?
        CLR.W   D0                 ;<01> For no carriage return
        ELSE                       ;<01> If not blank
        MOVE.W  #-1,D0             ;<01> For carriage return
        ENDIF
        JSR     XXPRINT            ;Call output string
        MOVEM.L (SP)+,D0/A4        ;<01> Restore D0 + A4
        .ENDM
;
;       INPUT macro.  Accepts a line of keyboard input.
;
;       INPUT    address[,length]
;
;       Returns a Pascal format string at "address".  Length is the number of
;       bytes available (including count byte).
;
        .MACRO  INPUT              ;address,length =
        XREF    XXINPUT            ;Define for linker
        MOVEM.L A0/D0,-(SP)        ;Save parameter registers
        LEA     %1,A0              ;-> Return buffer
        IF      '%2' = ''          ;Not specified?
        MOVEQ.L #80,D0             ;Default to 80
        ELSE                       ;Use specified
        MOVE.L  %2,D0              ;
        ENDIF                      ;
        JSR     XXINPUT            ;Call RTL routine
        MOVEM.L (SP)+,A0/D0        ;Restore registers
        .ENDM


;
;       ENDPROGRAM macro.  This macro must be the last line in the source file.
;
```

***Listing E.22** – The MACLIB.ASM file (continued)*

```
;        Usage:
;
;        ENDPROGRAM
;
         MACRO   ENDPROGRAM =
         STRING_FORMAT 1
         |
;
;        OPEN macro.  This macro opens the data fork of a file.  The file handle is
;        returned in D0.W.
;
;        Usage:
;
;        OPEN    filename,filebuff,volref
;
;        Filename is the address of a Pascal string containing the name of the file
;        to be opened. Filebuff is the address of a 522-byte buffer to be used by
;        the file system.  This can be 0 if only one file is open at a time.
;
;        Volref is the volume reference number, or blank for a volume scan.
;
         .MACRO  OPEN            ; filename,buffer,volref =
         MOVEM.L D1/A0/A1,-(SP)  ; Save registers
         LEA     %1,A0           ; A0 -> File name
         LEA     %2,A1           ; A1 -> Buffer area
         IF      '%3' = ''
         CLR.W   D1              ; Do vol scan
         ELSE                    ; Don't
         MOVE.W  %3,D1           ;
         ENDIF
         XREF    XXOPEND         ; RTL routine name
         JSR     XXOPEND         ; Call RTL
         MOVEM.L (SP)+,D1/A0-A1  ; Restore registers
         .ENDM
;
;        OPENR macro.  This macro opens the resource fork of a file.
;        The file handle is returned in D0.W.
;
;        Usage:
;
;        OPENR   filename,buffer,volref
;
;        Filename is the address of a Pascal string containing the name of
;        the file to be opened. Buffer is the address of a 522-byte buffer to
;        be used by the file system.
         .MACRO  OPENR           ; filename,buffer,volref =
         MOVEM.L D1/A0-A1,-(SP)  ; Save input registers
         LEA     %1,A0           ; A0 -> File name
         LEA     %2,A1           ; A1 -> Buffer area
         IF      '%3' = ''
         CLR.W   D1              ; Do vol scan
         ELSE                    ; Don't
         MOVE.W  %3,D1   ;
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
        ENDIF
        XREF    XXOPENR         ; RTL routine name
        JSR     XXOPENR         ; Call RTL
        MOVEM.L (SP)+,D1/A0-A1  ; Restore input registers
        .ENDM
;
;       POPEN macro.  This macro opens the data fork of a file using
;       the Macintosh standard file name prompt.
;
;       Usage:
;
;       POPEN    types,ext,namebuf,buffer
;
;       Types is the address of a type array:
;
;               DC.W    #Types
;               DC.L    Type1
;               DC.L    Type2
;               <Etc.>
;
;       Namebuf is the address of a 63-byte array to receive the file name
;       selected by the user.  Ext is the file extension.  Zero
;       means all extensions.  Buffer is the optional 522-byte buffer.
;
        .MACRO  POPEN           ; types,ext,namebuf,buffer     =
        MOVEM.L A0-A3,-(SP)     ; Save registers
        LEA     %1,A3           ; A3 -> Type array
        LEA     %2,A0           ; A0 -> Extension string
        LEA     %3,A2           ; A2 -> Buffer for name
        LEA     %4,A1           ; A1 -> Buffer
        XREF    XXPOPEN         ; For the linker
        JSR     XXPOPEN         ; Call RTL routine
        MOVEM.L (SP)+,A0-A3     ; Restore registers
        .ENDM
Cancelled EQU   -1              ; Code for cancelled by user
;
;       CREATE macro.  This macro creates a new file.  The file is left open for
;       access to the data fork.  The file handle is returned in D0.W.
;
        .MACRO  CREATE          ; filename,buffer,volref =
        MOVEM.L A0-A1,-(SP)     ; Save registers
        LEA     %1,A0           ; A0 -> File name
        IF      '%2' <> ''      ; Buffer specified?
        LEA     %2,A1           ; A1 -> Buffer
        ELSE                    ; No buffer
        SUBA.L  A1,A1           ; Clear A1
        ENDIF                   ;
        IF      '%3' = ''       ; VolRef specified?
          CLR.W D1              ; No, use zero
        ELSE                    ; Specified,
          IF    '%3' = 'DEFAULT'; Take the default?
           CLR.W        D1      ; Do vol scan
          ELSE                  ; Don't
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
                MOVE.W      %3,D1   ; Use specified
              ENDIF                 ; = DEFAULT
            ENDIF                   ; Specified
            XREF    XXCREATE        ; RTL routine name
            JSR     XXCREATE        ; Do the create
            MOVEM.L (SP)+,A0-A1     ; Restore the registers
            .ENDM
;           PCREATE macro.  This macro creates a new file, using the Macintosh prompt
;           feature.
;
;           Usage:
;
;           PCREATE prompt,defname,namebuf,buffer
;
;           Prompt is the prompt string.
;
;           Defname is the default name string.
;
;           Namebuf is the buffer for the returned file name.
;
;           Buffer is the optional 522-byte buffer.
;
            MACRO   PCREATE prompt,defname,namebuf,buffer   =
            MOVEM.L A0-A3,-(SP)     ; Save registers
            LEA     {prompt},A0     ; A0 -> Prompt string
            LEA     {defname},A3    ; A3 -> Default Name
            LEA     {namebuf},A2    ; A2 -> Name buffer area
            LEA     {buffer},A1     ; A1 -> 522-byte buffer
            XREF    XXPCREATE       ; For the linker
            JSR     XXPCREATE       ; Call RTL routine
            MOVEM.L (SP)+,A0-A3     ; Restore registers
            |
;
;           SEEK macro.  This macro moves the file pointer on an open file.
;
;           Usage:
;
;           SEEK    ioref,offset,sense
;
;           Ioref is the I/O reference number, offset is the offset longword, and
;           sense is a word parameter giving the meaning of offset.
;
            MACRO   SEEK    ioref,offset,sense =
            MOVEM.L D1-D2,-(SP)     ; Save input registers
            MOVE.W  {ioref},D0      ; Load ioref #
            MOVE.L  {offset},D1     ; Load offset
            MOVE.W  {sense},D2      ; and sense word
            XREF    XXSEEK          ; For the linker
            JSR     XXSEEK          ; Call RTL routine
            MOVEM.L (SP)+,D1-D2     ; Restore input registers
            |
;
;           DELETE macro.  Deletes a file by name.
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
;
;        Usage:
;        DELETE   Filename[,volref]
;
;        Filename is the address of a file-name string.  Volref is
;        a volume reference number.
;
         .MACRO   DELETE
         MOVEM.L  A0/D1,-(SP)      ; Save input registers
         LEA      %1,A0            ; Load file name address
         IF       '%2' <> ''       ; Is name null?
         MOVE.W   %2,D1            ; Load volume reference
         ELSE                      ; Not null,
         CLR.W    D1               ;              make it zero
         ENDIF                     ;
         XREF     XXDELETE         ; For the linker
         JSR      XXDELETE         ; Call RTL routine
         MOVEM.L  (SP)+,A0/D1      ; Restore registers
         .ENDM
;
;
;        GETFINFO macro.  This macro gets the file information block for a named file.
;
;        Usage:
;
;        GETFINFO         filename,filebuff,volref
;
;        Filename is the address of a Pascal string containing the name of the file to
;        be queried. Filebuff is the address of an 80-byte buffer to receive the file info.
;
         .MACRO   GETFINFO         ; filename,buffer,volref =
         MOVEM.L  A0/A1/D1,-(SP)   ; Save registers
         LEA      %1,A0            ; A0 -> File name
         LEA      %2,A1            ; A1 -> Buffer area
         IF       '%3' <> ''       ; Volref specified?
           IF     '%3' = 'DEFAULT'
             CLR.W       D1        ; Do vol scan
           ELSE                    ; Don't
             MOVE.W %3,D1          ; Do specific vol
           ENDIF                   ; DEFAULT
         ELSE                      ;
           CLR.W D1                ; Not specified
         ENDIF                     ; Specified
         XREF     XXGETF           ; RTL routine name
         JSR      XXGETF           ; Call RTL
         MOVEM.L  (SP)+,A0/A1/D1   ; Restore registers
         .ENDM
;
;        SETFINFO macro.  This macro sets the file information block for a named file.
;
;        Usage:
;
;        SETFINFO         filename,filebuff,volref
```

Listing E.22 – The MACLIB.ASM file (continued)

```
;
;       Filename is the address of a Pascal string containing the name of the file to
;       be set. Filebuff is the address of an 80-byte buffer containing the file info.
;
        .MACRO  SETFINFO        ; filename,buffer,volref =
        MOVEM.L A0/A1/D1,-(SP)  ; Save registers
        LEA     %1,A0           ; A0 -> File name
        LEA     %2,A1           ; A1 -> Buffer area
        IF      '%3' <> ''      ; Volref specified?
          IF    '%3' = 'DEFAULT'
            CLR.W       D1      ; Do vol scan
          ELSE                  ; Don't
            MOVE.W      %3,D1   ; Do specific vol
          ENDIF                 ; DEFAULT
        ELSE                    ;
          CLR.W D1              ; Not specified
        ENDIF                   ; Specified
        XREF    XXSETF          ; RTL routine name
        JSR     XXSETF          ; Call RTL
        MOVEM.L (SP)+,A0/A1/D1  ; Restore registers
        .ENDM
;
;       READ macro.  Reads data from an open file.
;
;       Usage:
;
;       READ    filenum,buffer,count
;
;       Filenum is a word, buffer is an address, and count is a longword.
;
        MACRO   READ    filenum,buffer,count =
        MOVEM.L D1-D2/A0,-(SP)  ; Save registers
        MOVE.W  {filenum},D0    ; Load file number
        LEA     {buffer},A0     ; A0 -> Buffer address
        MOVE.L  {count},D1      ; D1 = Byte count
        CLR.W   D2              ; Indicate no newline
        XREF    XXREAD          ; Define RTL routine name
        JSR     XXREAD          ; Call RTL
        MOVEM.L (SP)+,D1-D2/A0  ; Restore registers
        |
;
;       READLN macro.  Reads a line of data from an open file.
;
;       Usage:
;
;       READLN  filenum,buffer,count
;
;       Filenum is a word, buffer is an address, and count is a longword.
;
        MACRO   READLN  filenum,buffer,count =
        MOVEM.L D1-D2/A0,-(SP)  ; Save registers
        MOVE.W  {filenum},D0    ; Load file number
        LEA     {buffer},A0     ; A0 -> Buffer address
```

***Listing E.22** – The MACLIB.ASM file (continued)*

```
        MOVE.L  {count},D1       ; D1 = Byte count
        MOVE.W  #3456,D2         ; CR terminator + flag
        XREF    XXREAD           ; Define RTL routine name
        JSR     XXREAD           ; Call RTL
        MOVEM.L (SP)+,D1-D2/A0   ; Restore registers
        |
;
;       WRITE macro.  Writes data to an open file.
;
;       Usage:
;
;       WRITE   filenum,buffer,count
;
;       Filenum is a word, buffer is an address, and count is a longword.
;
        MACRO   WRITE   filenum,buffer,count =
        MOVEM.L D1-D2/A0,-(SP)   ; Save registers
        MOVE.W  {filenum},D0     ; Load file number
        LEA     {buffer},A0      ; A0 -> Buffer address
        MOVE.L  {count},D1       ; D1 = Byte count
        XREF    XXWRITE          ; Define RTL routine name
        JSR     XXWRITE          ; Call RTL
        MOVEM.L (SP)+,D1-D2/A0   ; Restore registers
        |
;
;       CLOSE macro.  Closes an open file.
;
;       Usage:
;
;       CLOSE   filenum
;
        MACRO   CLOSE   filenum =
        MOVE.L  D0,-(SP)         ; Save D0
        MOVE.W  {filenum},D0     ; D0 = File number to close
        XREF    XXCLOSE          ; Define RTL routine name
        JSR     XXCLOSE          ; Call RTL
        MOVE.L  (SP)+,D0         ; Restore D0
        |
;
;       EXIT macro.  Flushes volumes, returns to Finder.
;
;       USAGE:
;
;       EXIT
;
        MACRO   EXIT    =
        XREF    XXEXIT           ; For linker
        JMP     XXEXIT           ; Call RTL exit routine
        |
;
;       GETEVENT macro.  This macro returns in D0.W the event number
;       of an event obtained from the system's event queue.
;
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
        .MACRO  GETEVENT        ;       cursor =
        IF      '%1' <> ''      ;       specified?
          MOVE.W %1,D0          ;       Yes, use specified
        ELSE                    ;       Wasn't
          MOVE.W #1,D0          ;       Use I-beam cursor
        ENDIF                   ;       Cursor
        XREF    XXEVENT         ;       For the linker
        JSR     XXEVENT         ;       Call RTL routine
        .ENDM
;
;       QUIT macro.  This macro waits for a Quit selection from the File menu.
;
        MACRO   QUIT            =
@1:     GETEVENT                ;       Get next event
        CMP.W   #16,D0          ;       Menu choice
        BNE     @1              ;       No, get next event
        CMP.L   #$20001,D1      ;       File menu/Quit?
        BNE     @1              ;       No, get next
        EXIT                    ;       Now quit
        |
;
;       CHKABORT macro.  This macro checks for a Command-Period abort
;       indicator typed by the user.
;
        MACRO   CHKABORT        =
        XREF    XXABORT         ;       For the linker
        JSR     XXABORT         ;       Call abort check routine
        |
;
;       Alert macro.  Calls Macintosh ALERT facility.
;
;       Usage:
;
;       Alert   #ID[,filterproc]
;
;       #ID is a resource ID, and filterproc is an event filter
;       address.  Returns # of the item hit in D0.W.
;
        .MACRO  Alert
        MOVEM.L D1-D2/A0-A1,-(SP) ;     Save registers
        CLR.W   -(SP)           ;       Room for result
        MOVE.W  %1,-(SP)        ;       Push resource ID
        IF      '%2' <> ''      ;       If proc specified
        PEA     %2              ;       Load pointer
        ELSE                    ;       No proc
        CLR.L   -(SP)           ;       Push 0 long
        ENDIF                   ;
        DC.W    $A985           ;       _Alert trap
        MOVE.W  (SP)+,D0        ;       Result to D0
        MOVEM.L (SP)+,D1-D2/A0-A1 ;     Unsave registers
        .ENDM
;
;       StopAlert macro.  Calls Macintosh StopAlert facility.
;
;       Usage:
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
;
;       StopAlert       #id[,filterproc]
;
;       #id is a resource ID, and filterproc is an event filter
;       address.  Returns # of the item hit in D0.W.
;
        .MACRO  StopAlert
        MOVEM.L D1-D2/A0-A1,-(SP);      Save registers
        CLR.W   -(SP)           ;       Room for result
        MOVE.W  %1,-(SP)        ;       Push resource ID
        IF      '%2' <> ''      ;       If proc specified
        PEA     %2              ;       Load pointer
        ELSE                    ;       No proc
        CLR.L   -(SP)           ;       Push 0 long
        ENDIF                   ;
        DC.W    $A986           ;       _StopAlert trap
        MOVE.W  (SP)+,D0        ;       Result to D0
        MOVEM.L (SP)+,D1-D2/A0-A1;      Unsave registers
        .ENDM

;
;       NoteAlert macro.  Calls Macintosh NoteAlert facility.
;
;       Usage:
;
;       NoteAlert       #id[,filterproc]
;
;       #id is a resource ID, and filterproc is an event filter
;       address.  Returns # of the item hit in D0.W.
;
        .MACRO  NoteAlert
        MOVEM.L D1-D2/A0-A1,-(SP);      Save registers
        CLR.W   -(SP)           ;       Room for result
        MOVE.W  %1,-(SP)        ;       Push resource ID
        IF      '%2' <> ''      ;       If proc specified
        PEA     %2              ;       Load pointer
        ELSE                    ;       No proc
        CLR.L   -(SP)           ;       Push 0 long
        ENDIF                   ;
        DC.W    $A987           ;       _NoteAlert trap
        MOVE.W  (SP)+,D0        ;       Result to D0
        MOVEM.L (SP)+,D1-D2/A0-A1;      Unsave registers
        .ENDM
;
;       CautionAlert macro.  Calls Macintosh CautionAlert facility.
;
;       Usage:
;
;       CautionAlert    #id[,filterproc]
;
;       #id is a resource ID, and filterproc is an event filter
;       address.  Returns # of the item hit in D0.W.
;
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
        .MACRO  CautionAlert
        MOVEM.L D1-D2/A0-A1,-(SP);      Save registers
        CLR.W   -(SP)           ;       Room for result
        MOVE.W  %1,-(SP)        ;       Push resource ID
        IF      '%2' <> ''      ;       If proc specified
        PEA     %2              ;       Load pointer
        ELSE                    ;       No proc
        CLR.L   -(SP)           ;       Push 0 long
        ENDIF                   ;
        DC.W    $A988           ;       _CautionAlert trap
        MOVE.W  (SP)+,D0        ;       Result to D0
        MOVEM.L (SP)+,D1-D2/A0-A1;      Unsave registers
        .ENDM
;
;       SCURSOR macro.  Change the cursor to one of the standard
;       values.
;
;       SCURSOR #ID
;
;       #ID is 0-5 (cursor ID)
;
        .MACRO  SCURSOR
        MOVE.L  D0,-(SP)        ;       Save register
        MOVE.W  %1,D0           ;       Set ID
        XREF    XXCURSOR        ;       RTL routine
        JSR     XXCURSOR        ;       Call RTL
        MOVE.L  (SP)+,D0        ;       Restore register
        .ENDM
;
;       DIALOG macro.  Performs a Macintosh _ModalDialog function.
;
;       Usage:
;
;       DIALOG  #ID
;
;       #ID is the resource ID of the dialog template in the resource file.
;       D0 has the item number selected by the user.
;
        .MACRO  DIALOG
        MOVE.W  %1,D0           ;       Load ID
        XREF    XXDIALOG        ;       For the linker
        JSR     XXDIALOG        ;       Call RTL routine
        .ENDM
;
;       CHECK macro.  Checks an item in a menu.
;
;       Usage:
;
;               CHECK   menu,item
;
;       Where menu is the menu resource number and item
;       is the item number.
;
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
        .MACRO   CHECK
        MOVEM.L  D0-D1,-(SP)      ;       Push registers
        MOVE.W   %1,D0            ;       Menu to D0
        MOVE.W   %2,D1            ;       Item to D1
        XREF     XXCHECK          ;       For the linker
        JSR      XXCHECK          ;       Call RTL
        MOVEM.L  (SP)+,D0-D1      ;       Restore registers
        .ENDM
;
;       UNCHECK macro.  Unchecks an item in a menu.
;
;       Usage:
;
;                UNCHECK menu,item
;
;       Where menu is the menu resource number and item
;       is the item number.
;
        .MACRO   UNCHECK
        MOVEM.L  D0-D1,-(SP)      ;       Push registers
        MOVE.W   %1,D0            ;       Menu to D0
        MOVE.W   %2,D1            ;       Item to D1
        XREF     XXUNCHECK        ;       For the linker
        JSR      XXUNCHECK        ;       Call RTL
        MOVEM.L  (SP)+,D0-D1      ;       Restore registers
        .ENDM
;
;       ENABLE macro.  Enables an item in a menu.
;
;       Usage:
;
;                ENABLE  menu,item
;
;       Where menu is the menu resource number and item
;       is the item number.
;
        .MACRO   ENABLE
        MOVEM.L  D0-D1,-(SP)      ;       Push registers
        MOVE.W   %1,D0            ;       Menu to D0
        MOVE.W   %2,D1            ;       Item to D1
        XREF     XXENABLE         ;       For the linker
        JSR      XXENABLE         ;       Call RTL
        MOVEM.L  (SP)+,D0-D1      ;       Restore registers
        .ENDM
;
;       DISABLE macro.  Disables an item in a menu.
;
;       Usage:
;
;                DISABLE menu,item
;
;       Where menu is the menu resource number and item
;       is the item number.
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
;
        .MACRO  DISABLE
        MOVEM.L D0-D1,-(SP)     ;       Push registers
        MOVE.W  %1,D0           ;       Menu to D0
        MOVE.W  %2,D1           ;       Item to D1
        XREF    XXDISABLE       ;       For the linker
        JSR     XXDISABLE       ;       Call RTL
        MOVEM.L (SP)+,D0-D1     ;       Restore registers
        .ENDM
;
;       STYLE macro.  Sets the style word for a menu item.
;
;       Usage:
;
;               STYLE   menu,item,style
;
;       Where menu is the menu resource number and item
;       is the item number.  Style is a QuickDraw text style
;       word.
;
        .MACRO  STYLE
        MOVEM.L D0-D2,-(SP)     ;       Push registers
        MOVE.W  %1,D0           ;       Menu to D0
        MOVE.W  %2,D1           ;       Item to D1
        MOVE.W  %3,D2           ;       Style word to D2
        XREF    XXSTYLE         ;       For the linker
        JSR     XXSTYLE         ;       Call RTL
        MOVEM.L (SP)+,D0-D2     ;       Restore registers
        .ENDM
;
;       TRACK macro.  Performs call to external routine that does
;       a _TrackControl.  Should be called from a routine in response
;       to a scroll line or page event.
;
;       Usage:
;
;       TRACK   handle,proc
;
;       Where handle is a control handle and proc is the scroll
;       procedure.
;
        .MACRO  TRACK
        MOVEM.L D0/A0,-(SP)     ;       Save the registers
        MOVE.L  %1,D0           ;       D0 gets handle
        LEA     %2,A0           ;       A0 gets procedure
        XREF    XXTRACK         ;       For the linker
        JSR     XXTRACK         ;       Call RTL routine
        MOVEM.L (SP)+,D0/A0     ;       Restore the registers
        .ENDM
;
;       SETVMIN macro.  Sets the upper limit on a window's
;       vertical scroll bar.
;
```

Listing E.22 – The MACLIB.ASM file (continued)

```
;       Calling sequence:
;
;       SETVMIN value
;
;       Where value is the desired value.
;
;
;
        .MACRO   SETVMIN         ;
        MOVEM.L  D0,-(SP)        ;       Push registers
        MOVE.W   %1,D0           ;       D0 = Value
        XREF     XXSETVMIN       ;       For the linker
        JSR      XXSETVMIN       ;       Call RTL routine
        MOVEM.L  (SP)+,D0        ;       Pop registers
        .ENDM
;
;       SETVMAX Macro.  Sets the lower limit on a window's
;       vertical scroll bar.
;
;       Calling sequence:
;
;       SETVMAX value
;
;       Where value is the desired value.
;
;
;
        .MACRO   SETVMAX         ;
        MOVEM.L  D0,-(SP)        ;       Push registers
        MOVE.W   %1,D0           ;       D0 = Value
        XREF     XXSETVMAX       ;       For the linker
        JSR      XXSETVMAX       ;       Call RTL routine
        MOVEM.L  (SP)+,D0        ;       Pop registers
        .ENDM
;
;       SETVVAL macro.  Sets the thumb value on a window's
;       vertical scroll bar.
;
;       Calling sequence:
;
;       SETVVAL value
;
;       Where value is the desired value.
;
;
;
        .MACRO   SETVVAL         ;
        MOVEM.L  D0,-(SP)        ;       Push registers
        MOVE.W   %1,D0           ;       D0 = Value
        XREF     XXSETVVAL       ;       For the linker
        JSR      XXSETVVAL       ;       Call RTL routine
        MOVEM.L  (SP)+,D0        ;       Pop registers
        .ENDM
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
;
;       SETHMIN macro.  Sets the left limit on a window's
;       horizontal scroll bar.
;
;       Calling sequence:
;
;       SETHMIN value
;
;       Where value is the desired value.
;
;
;
        .MACRO   SETHMIN        ;
        MOVEM.L DO,-(SP)        ;       Push registers
        MOVE.W  %1,DO           ;       DO = Value
        XREF    XXSETHMIN       ;       For the linker
        JSR     XXSETHMIN       ;       Call RTL routine
        MOVEM.L (SP)+,DO        ;       Pop registers
        .ENDM
;
;       SETHMAX macro.  Sets the right limit on a window's
;       horizontal scroll bar.
;
;       Calling sequence:
;
;       SETHMAX value
;
;       Where value is the desired value.
;
;
;
        .MACRO   SETHMAX        ;
        MOVEM.L DO,-(SP)        ;       Push registers
        MOVE.W  %1,DO           ;       DO = Value
        XREF    XXSETHMAX       ;       For the linker
        JSR     XXSETHMAX       ;       Call RTL routine
        MOVEM.L (SP)+,DO        ;       Pop registers
        .ENDM
;
;       SETHVAL macro.  Sets the thumb value on a window's
;       horizontal scroll bar.
;
;       Calling sequence:
;
;       SETHVAL value
;
;       Where value is the desired value.
;
;
;
        .MACRO   SETHVAL        ;
        MOVEM.L DO,-(SP)        ;       Push registers
        MOVE.W  %1,DO           ;       DO = Value
```

Listing E.22 – *The MACLIB.ASM file (continued)*

```
        XREF    XXSETHVAL       ;       For the linker
        JSR     XXSETHVAL       ;       Call RTL routine
        MOVEM.L (SP)+,D0        ;       Pop registers
        .ENDM
;
;       WTitle macro.  Sets the title bar to a specified string.
;
;       Usage:
;
;       WTitle  string
;
;       Where string is the address of the new string (Pascal format).
;
        .MACRO  WTitle
        MOVE.L  A0,-(SP)        ;       Save A0
        LEA     %1,A0           ;       A0 -> New string
        XREF    XXWTITLE        ;       Call
        JSR     XXWTITLE        ;               RTL routine
        MOVE.L  (SP)+,A0        ;       Pop A0
        .ENDM
```

Listing E.22 *– The MACLIB.ASM file (continued)*

INDEX

Selections from The SYBEX Library

Introduction to Computers

OVERCOMING COMPUTER FEAR

by Jeff Berner

112 pp., illustr., Ref. 0-145

This easy-going introduction to computers helps you separate the facts from the myths.

INTRODUCTION TO WORD PROCESSING

by Hal Glatzer

205 pp., 140 illustr., Ref. 0-076

Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

PARENTS, KIDS, AND COMPUTERS

by Lynne Alper and Meg Holmberg

145 pp., illustr., Ref. 0-151

This book answers your questions about the educational possibilities of home computers.

PROTECTING YOUR COMPUTER

by Rodnay Zaks

214 pp., 100 illustr., Ref. 0-239

The correct way to handle and care for all elements of a computer system, including what to do when something doesn't work.

YOUR FIRST COMPUTER

by Rodnay Zaks

258 pp., 150 illustr., Ref. 0-142

The most popular introduction to small computers and their peripherals: what they do and how to buy one.

THE SYBEX PERSONAL COMPUTER DICTIONARY

120 pp., Ref. 0-199

All the definitions and acronyms of microcomputer jargon defined in a handy pocket-sized edition. Includes translations of the most popular terms into ten languages.

Computer Books for Kids

MONICA THE COMPUTER MOUSE

by Donna Bearden, illustrated by Brad W. Foster

64 pp., illustr., Hardcover, Ref. 0-214

Lavishly illustrated in color, this book tells the story of Monica the mouse, as she travels around to learn about several different kinds of computers and the jobs they can do. For ages 5–8.

POWER UP! KIDS' GUIDE TO THE APPLE IIe® /IIc™

by Marty DeJonghe and Caroline Earhart

200 pp., illustr., Ref. 0-212

Colorful illustrations and a friendly robot highlight this guide to the Apple IIe/IIc for kids 8–11.

BANK STREET WRITING WITH YOUR APPLE®

by Stanley Schatt, Ph.D. and Jane Abrams Schatt, M.A.

150 pp., illustr., Ref. 0-189

These engaging exercises show children aged 10–13 how to use Bank Street Writer for fun, profit, and school work.

POWER UP! KIDS' GUIDE TO THE COMMODORE 64™

by Marty DeJonghe and Caroline Earhart

192 pp., illustr., Ref. 0-188

Colorful illustrations and a friendly robot highlight this guide to the Commodore 64 for kids 8–11.

Humor

COMPUTER CRAZY

by Daniel Le Noury

100 pp., illustr., Ref. 0-173

No matter how you feel about computers, these cartoons will have you laughing about them.

MOTHER GOOSE YOUR COMPUTER: A GROWNUP'S GARDEN OF SILICON SATIRE

by Paul Panish and Anna Belle Panish, Illustrated by Terry Small

96 pp., illustr., Ref. 0-198

This richly illustrated hardcover book uses parodies of familiar Mother Goose rhymes to satirize the world of high technology.

CONFESSIONS OF AN INFOMANIAC

by Elizabeth M. Ferrarini

215 pp., Ref. 0-186

This is one woman's tongue-in-cheek revelations of her pursuit of men, money, and machines. Learn about the many shopping services, information banks, and electronic dating bulletin boards available by computer.

Special Interest

THE COLLEGE STUDENT'S PERSONAL COMPUTER HANDBOOK

by Bryan Pfaffenberger

210 pp., illustr., Ref. 0-170

This friendly guide will aid students in selecting a computer system for college study, managing information in a college course, and writing research papers.

COMPUTER POWER FOR YOUR LAW OFFICE

by Daniel Remer

142 pp., Ref. 0-109

How to use computers to reach peak productivity in your law office, simply and inexpensively.

CELESTIAL BASIC

by Eric Burgess

300 pp., 65 illustr., Ref. 0-087

A collection of BASIC programs that rapidly complete the chores of typical astronomical computations. It's like having a planetarium in your own home! Displays apparent movement of stars, planets and meteor showers.

COMPUTER POWER FOR YOUR ACCOUNTING FIRM

by James Morgan, C.P.A.

250 pp., illustr., Ref. 0-164

This book is a convenient source of information about computerizing your accounting office, with an emphasis on hardware and software options.

PERSONAL COMPUTERS AND SPECIAL NEEDS

by Frank G. Bowe

175 pp., illustr., Ref. 0-193

Learn how people are overcoming problems with hearing, vision, mobility, and learning, through the use of computer technology.

ESPIONAGE IN THE SILICON VALLEY

by John D. Halamka

200 pp., illustr., Ref. 0-225

Discover the behind-the-scenes stories of famous high-tech spy cases you've seen in the headlines.

ASTROLOGY ON YOUR PERSONAL COMPUTER

by Hank Friedman

225 pp., illustr., Ref. 0-226

An invaluable aid for astrologers who want to streamline their calculation and data management chores with the right combination of hardware and software.

Computer Specific

Apple II—Macintosh

THE PRO-DOS HANDBOOK

by Timothy Rice/Karen Rice

225 pp., illustr., Ref. 0-230

All Pro-DOS users, from beginning to advanced, will find this book packed with vital information. The book covers the basics, and then addresses itself to the Apple II user who needs to interface with Pro-DOS when programming in BASIC. Learn how Pro-DOS uses memory, and how it handles text files, binary files, graphics, and sound. Includes a chapter on machine language programming.

THE MACINTOSH™ TOOLBOX

by Huxham, Burnard, and Takatsuka

300 pp., illustr., Ref. 0-249

This tutorial on the advanced features of the Macintosh toolbox is an ideal companion to The Macintosh BASIC Handbook.

THE MACINTOSH™ BASIC HANDBOOK

by Thomas Blackadar/Jonathan Kamin

800 pp., illustr., Ref. 0-257

This desk-side reference book for the Macintosh programmer covers the BASIC statements and toolbox commands, organized like a dictionary.

PROGRAMMING THE MACINTOSH™ IN ASSEMBLY LANGUAGE

by Steve Williams

400 pp., illustr., Ref. 0-263

Information, examples, and guidelines for programming the 68000 microprocessor are given, including details of its entire instruction set.

THE EASY GUIDE TO YOUR APPLE II®

by Joseph Kascmer

147 pp., illustr., Ref. 0-122

A friendly introduction to the Apple II, II plus, and the IIe.

BASIC EXERCISES FOR THE APPLE®

by J.P. Lamoitier

250 pp., 90 illustr., Ref. 0-084

Teaches Applesoft BASIC through actual practice, using graduated exercises drawn from everyday applications.

THE APPLE II® BASIC HANDBOOK

by Douglas Hergert

250 pp., illustr., Ref. 0-115

A complete listing with descriptions and instructive examples of each of the Apple II BASIC keywords and functions. A handy reference guide, organized like a dictionary.

APPLE II® BASIC PROGRAMS IN MINUTES

by Stanley R. Trost

150 pp., illustr., Ref. 0-121

A collection of ready-to-run programs for financial calculations, investment analysis, record keeping, and many more home and office applications. These programs can be entered on your Apple II plus or IIe in minutes!

YOUR FIRST APPLE II® PROGRAM

by Rodnay Zaks

182 pp., illustr., Ref. 0-136

This fully illustrated, easy-to-use introduction to Applesoft BASIC programming will have the reader programming in a matter of hours.

THE APPLE® CONNECTION

by James W. Coffron

264 pp., 120 illustr., Ref. 0-085

Teaches elementary interfacing and BASIC programming of the Apple for connection to external devices and household appliances.

THE APPLE IIc™: A PRACTICAL GUIDE

by Thomas Blackadar

175 pp., illustr., Ref. 0-241

Learn all you need to know about the Apple IIc! This jargon-free companion gives you a guided tour of Apple's new machine.

THE BEST OF EDUCATIONAL SOFTWARE FOR APPLE II® COMPUTERS
by Gary G. Bitter, Ph.D. and Kay Gore
300 pp., Ref. 0-206
Here is a handy guide for parents and an invaluable reference for educators who must make decisions about software purchases.

YOUR SECOND APPLE II® PROGRAM
by Gary Lippman
250 pp., illustr., Ref. 0-208
The many colorful illustrations in this book make it a delight for children and fun for adults who are mastering programming on any of the Apple II line of computers, including the new IIc.

THE MACINTOSH™: A PRACTICAL GUIDE
by Joseph Caggiano
280 pp., illustr., Ref. 0-216
This easy-to-read guide takes you all the way from set-up to more advanced activities such as using Macwrite, Macpaint, and Multiplan.

MACINTOSH™ FOR COLLEGE STUDENTS
by Bryan Pfaffenberger
250 pp., illustr., Ref. 0-227
Find out how to give yourself an edge in the race to get papers in on time and prepare for exams. This book covers everything you need to know about how to use the Macintosh for college study.

Atari

YOUR FIRST ATARI® PROGRAM
by Rodnay Zaks
182 pp., illustr., Ref. 0-130
This fully illustrated, easy-to-use introduction to ATARI BASIC programming will have the reader programming in a matter of hours.

BASIC EXERCISES FOR THE ATARI®
by J.P. Lamoitier
251 pp., illustr., Ref. 0-101
Teaches ATARI BASIC through actual practice, using graduated exercises drawn from everyday applications.

THE EASY GUIDE TO YOUR ATARI® 600XL/800XL
by Thomas Blackadar
175 pp., illustr., Ref. 0-125
This jargon-free companion will help you get started on the right foot with your new 600XL or 800XL ATARI computer.

ATARI® BASIC PROGRAMS IN MINUTES
by Stanley R. Trost
170 pp., illustr., Ref. 0-143
You can use this practical set of programs without any prior knowledge of BASIC! Application examples are taken from a wide variety of fields, including business, home management, and real estate.

YOUR SECOND ATARI® PROGRAM
by Gary Lippman
250 pp., illustr., Ref. 0-232
The many colorful illustrations in this book make it a delight for children and fun for adults who are mastering BASIC programming on the ATARI 400, 800, or XL series computers.

Coleco

WORD PROCESSING WITH YOUR COLECO ADAM™
by Carole Jelen Alden
140 pp., illustr., Ref. 0-182
This is an in-depth tutorial covering the word processing system of the Adam.

THE EASY GUIDE TO YOUR COLECO ADAM™
by Thomas Blackadar
175 pp., illustr., Ref. 0-181
This quick reference guide shows you how to get started on your Coleco Adam using a minimum of technical jargon.

Commodore 64/VIC-20

THE BEST OF COMMODORE 64™ SOFTWARE
by Thomas Blackadar
150 pp., illustr., Ref. 0-194

Save yourself time and frustration with this buyer's guide to Commodore 64 software. Find the best game, music, education, and home management programs on the market today.

THE BEST OF VIC-20™ SOFTWARE
by Thomas Blackadar
150 pp., illustr., Ref. 0-139
Save yourself time and frustration with this buyer's guide to VIC-20 software. Find the best game, music, education, and home management programs on the market today.

THE COMMODORE 64™/VIC-20™ BASIC HANDBOOK
by Douglas Hergert
144 pp., illustr., Ref. 0-116
A complete listing with descriptions and instructive examples of each of the Commodore 64 BASIC keywords and functions. A handy reference guide, organized like a dictionary.

THE EASY GUIDE TO YOUR COMMODORE 64™
by Joseph Kascmer
126 pp., illustr., Ref. 0-126
A friendly introduction to the Commodore 64.

YOUR FIRST VIC-20™ PROGRAM
by Rodnay Zaks
182 pp., illustr., Ref. 0-129
This fully illustrated, easy-to-use introduction to VIC-20 BASIC programming will have the reader programming in a matter of hours.

THE VIC-20™ CONNECTION
by James W. Coffron
260 pp., 120 illustr., Ref. 0-128
Teaches elementary interfacing and BASIC programming of the VIC-20 for connection to external devices and household appliances.

YOUR FIRST COMMODORE 64™ PROGRAM
by Rodnay Zaks
182 pp., illustr., Ref. 0-172
You can learn to write simple programs without any prior knowledge of mathematics or computers! Guided by colorful illustrations and step-by-step instructions, you'll be constructing programs within an hour or two.

COMMODORE 64™ BASIC PROGRAMS IN MINUTES
by Stanley R. Trost
170 pp., illustr., Ref. 0-154
Here is a practical set of programs for business, finance, real estate, data analysis, record keeping, and educational applications.

GRAPHICS GUIDE TO THE COMMODORE 64™
by Charles Platt
261 pp., illustr., Ref. 0-138
This easy-to-understand book will appeal to anyone who wants to master the Commodore 64's powerful graphics features.

COMMODORE 64™ FREE SOFTWARE
by Gary Phillips
300 pp., Ref. 0-201
Find out what "free software" is all about and how to find the specific programs you need.

YOUR SECOND COMMODORE 64™ PROGRAM
by Gary Lippman
240 pp., illustr., Ref. 0-152
A sequel to *Your First Commodore 64 Program*, this book follows the same patient, detailed approach and brings you to the next level of programming skill.

THE COMMODORE 64™ CONNECTION
by James W. Coffron
250 pp., illustr., Ref. 0-192
Learn to control lights, electricity, burglar alarm systems, and other non-computer devices with your Commodore 64.

PARENTS, KIDS, AND THE COMMODORE 64™
by Lynne Alper and Meg Holmberg
110 pp., illustr., Ref. 0-234

This book answers parents' questions about the educational possibilities of the Commodore 64.

THE BEST OF EDUCATIONAL SOFTWARE FOR THE COMMODORE 64
by Gary G. Bitter, Ph.D. and Kay Gore
250 pp., Ref. 0-223
Here is a handy guide for parents and an indispensable reference for educators who must make decisions about software purchases for the Commodore 64.

CP/M Systems

THE CP/M® HANDBOOK
by Rodnay Zaks
320 pp., 100 illustr., Ref 0-048
An indispensable reference and guide to CP/M—the most widely-used operating system for small computers.

MASTERING CP/M®
by Alan R. Miller
398 pp., illustr., Ref. 0-068
For advanced CP/M users or systems programmers who want maximum use of the CP/M operating system . . . takes up where our *CP/M Handbook* leaves off.

THE BEST OF CP/M® SOFTWARE
by John D. Halamka
250 pp., Ref. 0-100
This book reviews tried-and-tested, commercially available software for your CP/M system.

THE CP/M PLUS™ HANDBOOK
by Alan R. Miller
250 pp., illustr., Ref. 0-158
This guide is easy for beginners to understand, yet contains valuable information for advanced users of CP/M Plus (Version 3).

IBM PC and Compatibles

THE ABC'S OF THE IBM® PC
by Joan Lasselle and Carol Ramsay
143 pp., illustr., Ref. 0-102
This book will take you through the first crucial steps in learning to use the IBM PC.

THE BEST OF IBM® PC SOFTWARE
by Stanley R. Trost
351 pp., Ref. 0-104
Separates the wheat from the chaff in the world of IBM PC software. Tells you what to expect from the best available IBM PC programs.

THE IBM® PC-DOS HANDBOOK
by Richard Allen King
296 pp., Ref. 0-103
Explains the PC disk operating system. Get the most out of your PC by adapting its capabilities to your specific needs.

BUSINESS GRAPHICS FOR THE IBM® PC
by Nelson Ford
259 pp., illustr., Ref. 0-124
Ready-to-run programs for creating line graphs, multiple bar graphs, pie charts, and more. An ideal way to use your PC's business capabilities!

THE IBM® PC CONNECTION
by James W. Coffron
264 pp., illustr., Ref. 0-127
Teaches elementary interfacing and BASIC programming of the IBM PC for connection to external devices and household appliances.

BASIC EXERCISES FOR THE IBM® PERSONAL COMPUTER
by J.P. Lamoitier
252 pp., 90 illustr., Ref. 0-088
Teaches IBM BASIC through actual practice, using graduated exercises drawn from everyday applications.

USEFUL BASIC PROGRAMS FOR THE IBM® PC
by Stanley R. Trost
144 pp., illustr., Ref. 0-111
This collection of programs takes full advantage of the interactive capabilities of your IBM Personal Computer. Financial calculations, investment analysis, record keeping, and math practice—made easier on your IBM PC.

YOUR FIRST IBM® PC PROGRAM

by Rodnay Zaks

182 pp., illustr., Ref. 0-171

This well-illustrated book makes programming easy for children and adults.

DATA FILE PROGRAMMING ON YOUR IBM® PC

by Alan Simpson

219 pp., illustr., Ref. 0-146

This book provides instructions and examples for managing data files in BASIC. Programming design and development are extensively discussed.

SELECTING THE RIGHT DATA BASE SOFTWARE FOR THE IBM® PC

SELECTING THE RIGHT WORD PROCESSING SOFTWARE FOR THE IBM® PC

SELECTING THE RIGHT SPREADSHEET SOFTWARE FOR THE IBM® PC

by Kathleen McHugh and Veronica Corchado

100 pp., illustr., Ref. 0-174, 0-177, 0-178

This series on selecting the right business software offers the busy professional concise, informative reviews of the best available software packages.

THE MS™-DOS HANDBOOK

by Richard Allen King

320 pp., illustr., Ref. 0-185

The differences between the various versions and manufacturer's implementations of MS-DOS are covered in a clear, straightforward manner. Tables, maps, and numerous examples make this the most complete book on MS-DOS available.

ESSENTIAL PC-DOS

by Myril and Susan Shaw

300 pp., illustr., Ref. 0-176

Whether you work with the IBM PC, XT, PC*jr*, or the portable PC, this book will be invaluable both for learning PC-DOS and for later reference.

IBM PCjr

IBM® PC*jr*™ BASIC PROGRAMS IN MINUTES

by Stanley R. Trost

175 pp., illustr., Ref. 0-205

Here is a practical set of BASIC programs for business, financial, real estate, data analysis, record keeping, and educational applications, ready to enter on your PC*jr*.

THE COMPLETE GUIDE TO YOUR IBM® PC*jr*™

by Douglas Herbert

625 pp., illustr., Ref. 0-179

Learn to master the new hardware and DOS features that IBM has introduced with the PC*jr*. A fold-out reference poster is included.

THE EASY GUIDE TO YOUR IBM® PC*jr*™

by Thomas Blackadar

175 pp., illustr., Ref. 0-217

This jargon-free companion is designed to give you a practical working knowledge of your machine—no prior knowledge of computers or programming is needed.

BASIC EXERCISES FOR THE IBM® PC*jr*™

by J.P. Lamoitier

250 pp., illustr., Ref. 0-218

PC*jr* BASIC is easy when you learn by doing! The graduated exercises in this book were chosen for their educational value and application to a variety of fields.

TI 99/4A

THE BEST OF TI 99/4A™ CARTRIDGES

by Thomas Blackadar

150 pp., illustr., Ref. 0-137

Save yourself time and frustration when buying TI 99/4A software. This buyer's guide gives an overview of the best available programs, with information on how to set up the computer to run them.

YOUR FIRST TI 99/4A™ PROGRAM
by Rodnay Zaks
182 pp., illustr., Ref. 0-157
Colorfully illustrated, this book concentrates on the essentials of programming in a clear, entertaining fashion.

Timex

YOUR TIMEX/SINCLAIR 1000® AND ZX81™
by Douglas Hergert
159 pp., illustr., Ref. 0-099
This book explains the set-up, operation, and capabilities of the Timex/Sinclair 1000 and ZX81. Covers how to interface peripheral devices and introduces BASIC programming.

THE TIMEX/SINCLAIR 1000® BASIC HANDBOOK
by Douglas Hergert
170 pp., illustr., Ref. 0-113
This complete alphabetical listing with explanations and examples of each word in the T/S 1000 BASIC vocabulary will allow you quick, error-free programming of your T/S 1000.

TIMEX/SINCLAIR 1000® BASIC PROGRAMS IN MINUTES
by Stanley R. Trost
150 pp., illustr., Ref. 0-119
A collection of ready-to-run programs for financial calculations, investment analysis, record keeping, and many more home and office applications. These programs can be entered on your T/S 1000 in minutes!

MORE USES FOR YOUR TIMEX/SINCLAIR 1000® Astronomy on Your Computer
by Eric Burgess and Howard J. Burgess
176 pp., illustr., Ref. 0-112
Ready-to-run programs that turn your TV into a planetarium.

TRS-80

THE RADIO SHACK® NOTEBOOK COMPUTER
by Orson Kellogg
118 pp., illustr., Ref. 0-150
Whether you already have the Radio Shack Model 100 notebook computer or are interested in buying one, this book will clearly explain what it can do for you.

YOUR COLOR COMPUTER
by Doug Mosher
350 pp., illustr., Ref. 0-097
Patience and humor guide the reader through purchasing, setting up, programming, and using the Radio Shack TRS-80 Color Computer. A complete introduction.

Software Specific

Spreadsheets

VISICALC® FOR SCIENCE AND ENGINEERING
by Stanley R. Trost and Charles Pomernacki
203 pp., illustr., Ref. 0-096
More than 50 programs for solving technical problems in science and engineering. Applications range from math and statistics to electrical and electronic engineering.

MASTERING VISICALC®
by Douglas Hergert
217 pp., 140 illustr., Ref. 0-090
Explains how to use the VisiCalc "electronic spreadsheet" functions and provides examples of each. Makes using this powerful program simple.

DOING BUSINESS WITH MULTIPLAN™
by Richard Allen King and Stanley R. Trost
250 pp., illustr., Ref. 0-148
This book will show you how using Multiplan can be nearly as easy as learning

to use a pocket calculator. It presents a collection of templates for business applications.

DOING BUSINESS WITH VISICALC®

by Stanley R. Trost

260 pp., illustr., Ref. 0-086

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

DOING BUSINESS WITH SUPERCALC™

by Stanley R. Trost

248 pp., illustr., Ref. 0-095

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

MULTIPLAN™ ON THE COMMODORE 64™

by Richard Allen King

260 pp., illustr., Ref. 0-231

This clear, straighforward guide will give you a firm grasp on Multiplan's functions, as well as provide a collection of useful template programs.

Word Processing

INTRODUCTION TO WORDSTAR®

by Arthur Naiman

202 pp., 30 illustr., Ref. 0-134

Makes it easy to learn WordStar, a powerful word processing program for personal computers.

THE COMPLETE GUIDE TO MULTIMATE™

by Carol Holcomb Dreger

250 pp., illustr., Ref. 0-229

A concise introduction to the many practical applications of this powerful word processing program.

PRACTICAL WORDSTAR® USES

by Julie Anne Arca

303 pp., illustr., Ref. 0-107

Pick your most time-consuming office tasks and this book will show you how to streamline them with WordStar.

THE THINKTANK™ BOOK

by Jonathan Kamin

200 pp., illustr., Ref. 0-224

Learn how the ThinkTank program can help you organize your thoughts, plans, and activities.

Data Base Management Systems

UNDERSTANDING dBASE III™

by Alan Simpson

250 pp., illustr., Ref. 0-267

For experienced dBASE II programmers, data base and program design are covered in detail; with many examples and illustrations.

UNDERSTANDING dBASE II™

by Alan Simpson

260 pp., illustr., Ref. 0-147

Learn programming techniques for mailing label systems, bookkeeping, and data management, as well as ways to interface dBASE II with other software systems.

ADVANCED TECHNIQUES in dBASE II™

by Alan Simpson

250 pp., illustr., Ref. 0-228

If you are an experienced dBASE II programmer and would like to begin customizing your own programs, this book is for you. It is a well-structured tutorial that offers programming techniques applicable to a wide variety of situations. Data base and program design are covered in detail, and the many examples and illustrations clarify the text.

Integrated Software

MASTERING SYMPHONY™

by Douglas Cobb

763 pp., illustr., Ref. 0-244

This bestselling book provides all the information you will need to put Symphony to work for you right away. Packed with practical models for the business user.

SYMPHONY™ ENCORE: PROGRAM NOTES

by Dick Andersen

325 pp., illustr., Ref. 0-247

Organized as a reference tool, this book gives shortcuts for using Symphony commands and functions, with troubleshooting advice.

JAZZ ON THE MACINTOSH™

by Joseph Caggiano and Michael McCarthy

400 pp., illustr., Ref. 0-265

The complete tutorial on the ins and outs of the season's hottest software, with tips on integrating its functions into efficient business projects.

MASTERING FRAMEWORK™

by Doug Hergert

450 pp., illustr., Ref. 0-248

This tutorial guides the beginning user through all the functions and features of this integrated software package, geared to the business environment.

ADVANCED TECHNIQUES IN FRAMEWORK™

by Alan Simpson

250 pp., illustr., Ref. 0-267

In order to begin customizing your own models with Framework, you'll need a thorough knowledge of Fred programming languages, and this book provides this information in a complete, well-organized form.

ADVANCED BUSINESS MODELS WITH 1-2-3™

by Stanley R. Trost

250 pp., illustr., Ref. 0-159

If you are a business professional using the 1-2-3 software package, you will find the spreadsheet and graphics models provided in this book easy to use "as is" in everyday business situations.

THE ABC'S OF 1-2-3™

by Chris Gilbert and Laurie Williams

225 pp., illustr., Ref. 0-168

For those new to the LOTUS 1-2-3 program, this book offers step-by-step instructions in mastering its spreadsheet, data base, and graphing capabilities.

MASTERING APPLEWORKS™

by Elna Tymes

250 pp., illustr., Ref. 0-240

Here is a business-oriented introduction to AppleWorks, the new integrated software package from Apple. No experience with computers is assumed.

Languages

BASIC

YOUR FIRST BASIC PROGRAM

by Rodnay Zaks

182 pp., illustr. in color, Ref. 0-092

A "how-to-program" book for the first time computer user, aged 8 to 88.

FIFTY BASIC EXERCISES

by J. P. Lamoitier

232 pp., 90 illustr., Ref. 0-056

Teaches BASIC through actual practice, using graduated exercises drawn from everyday applications. Programs written in Microsoft BASIC.

BASIC FOR BUSINESS

by Douglas Hergert

224 pp., 15 illustr., Ref. 0-080

A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully-explained accounting programs, and shows you how to write your own.

EXECUTIVE PLANNING WITH BASIC

by X. T. Bui

196 pp., 19 illustr., Ref. 0-083

An important collection of business management decision models in BASIC, including inventory management (EOQ), critical path analysis and PERT, financial ratio analysis, portfolio management, and much more.

BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS
by Alan R. Miller
318 pp., 120 illustr., Ref. 0-073
This book from the "Programs for Scientists and Engineers" series provides a library of problem-solving programs while developing the reader's proficiency in BASIC.

Pascal

INTRODUCTION TO PASCAL (Including UCSD Pascal™)
by Rodnay Zaks
420 pp., 130 illustr., Ref. 0-066
A step-by-step introduction for anyone who wants to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

THE PASCAL HANDBOOK
by Jacques Tiberghien
486 pp., 270 illustr., Ref. 0-053
A dictionary of the Pascal language, defining every reserved word, operator, procedure, and function found in all major versions of Pascal.

APPLE® PASCAL GAMES
by Douglas Hergert and Joseph T. Kalash
372 pp., 40 illustr., Ref. 0-074
A collection of the most popular computer games in Pascal, challenging the reader not only to play but to investigate how games are implemented on the computer.

PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS
by Alan R. Miller
374 pp., 120 illustr., Ref. 0-058
A comprehensive collection of frequently used algorithms for scientific and technical applications, programmed in Pascal. Includes programs for curve-fitting, integrals, statistical techniques, and more.

DOING BUSINESS WITH PASCAL
by Richard Hergert and Douglas Hergert
371 pp., illustr., Ref. 0-091
Practical tips for using Pascal programming in business. Covers design considerations, language extensions, and applications examples.

Other Languages

FORTRAN PROGRAMS FOR SCIENTISTS AND ENGINEERS
by Alan R. Miller
280 pp., 120 illustr., Ref. 0-082
This book from the "Programs for Scientists and Engineers" series provides a library of problem-solving programs while developing the reader's proficiency in FORTRAN.

A MICROPROGRAMMED APL IMPLEMENTATION
by Rodnay Zaks
350 pp., Ref. 0-005
An expert-level text presenting the complete conceptual analysis and design of an APL interpreter, and actual listing of the microcode.

FIFTY PASCAL PROGRAMS
by Bruce H. Hunter
338 pp., illustr., Ref. 0-110
More than just a collection of useful programs! Structured programming techniques are emphasized and concepts such as data type creation and array manipulation are clearly illustrated.

Technical

Assembly Language

PROGRAMMING THE 6502
by Rodnay Zaks
386 pp., 160 illustr., Ref. 0-135
Assembly language programming for the 6502, from basic concepts to advanced data structures.

6502 APPLICATIONS
by Rodnay Zaks
278 pp., 200 illustr., Ref. 0-015
Real-life application techniques: the input/output book for the 6502.

MacDoodle and Related Software

Available on Disk

If you'd like to use the programs in this book but don't want to type them in yourself, you can send for a disk containing the source files for every program in the book. The disk contains all programs listed in Appendix E of this book. To obtain this disk, complete the order form and return it along with $19.95, plus $2.50 to cover postage and handling. California residents add 6 percent sales tax. Please allow 6-8 weeks for delivery.

MacDoodle Source Disk
P.O. Box 3262
Monterey, CA 93940

Name____________________________________

Address__________________________________

City/State/ZIP_____________________________

Enclosed is my check or money order.
(Make check payable to *MacDoodle Source Disk.*)

SYBEX is not affiliated with MacDoodle Source Disk *and assumes no responsibility for any defect in the disk or program.*

Programming the Macintosh in Assembly Language

PROGRAMMING THE MACINTOSH IN ASSEMBLY LANGUAGE is a complete tutorial and reference guide for programming the 68000 in the Macintosh environment. Programmers at all levels—from intermediate Macintosh users to professional applications developers—will find this an indispensible programming handbook, covering

- Basic assembly language concepts
- Complete 68000 architecture and instruction set
- The Macintosh Toolbox—including memory management, QuickDraw, windows, alerts, and dialogs
- Advanced programming concepts—including data structures, sorting, searching, and recursion
- Linking assembly language routines with programs in BASIC, C, and Pascal

Sequential exercises take you from writing your first simple program through a complete 4,000-line model Macintosh application. The author provides a macro and run-time library to let you write sophisticated Macintosh programs while mastering advanced features at your own pace. Detailed appendices include

- Complete macro and run-time library source listings
- Source listings for all sample programs
- Macintosh character set
- 68000 quick reference table

ABOUT THE AUTHOR

Steve Williams is the manager of the Monterey branch of Control Technology, Inc. Prior to that, he was the Research & Development Project Manager at Digital Research, and he managed the operating system development team for the TRS-80 Model 16, which uses the 68000 processor. He has a B.S. in electrical engineering and a Master's degree in computer science from the University of Tennessee. He is the author of the best-selling *Programming the 68000* from SYBEX.

ISBN 0-89588-263-9