HAYDEN Macintosh Library

Programming the 68000

Macintosh Assembly Language

Edwin Rosenzweig
& Harland Harrison

Programming the 68000

Macintosh™ Assembly Language

Programming the 68000

Macintosh™ Assembly Language

Edwin Rosenzweig
and Harland Harrison

Hayden Book Company
A DIVISION OF HAYDEN PUBLISHING COMPANY, INC.
HASBROUCK HEIGHTS, NEW JERSEY

Acquisitions Editor: Bill Grout
Production Editor: Ronnie Groff
Cover design: Jim Bernard
Cover photo: Lou Odor/George Baquero
Composition: Elizabeth Typesetting Company
Printed and bound by: Command Web Offset, Inc.

Library of Congress Cataloging-in-Publication Data

Rosenzweig, E.J. (Edwin J.)
Programming the 68000.

Includes index.
1. Motorola 68000 (Microprocessor)–Programming. 2. Assembler language (Computer program language) I. Harrison, Harland. II. Title.
QA76.8.M6895R67 1986 005.265 86-3085
ISBN 0-8104-6310-5

Printed in the United States of America

1	2	3	4	5	6	7	8	9	Printing
86	87	88	89	90	91	92	93	94	Year

Preface

This book assumes that you have some knowledge of programming. We expect you to know how to code a simple BASIC or Pascal program. If you know how to program in assembler, so much the better.

However, we also realize that you may never have programmed in assembler before, don't know what a linker is, etc. Topics such as binary and hexadecimal are covered in appendices so the less advanced programmer can go there for additional knowledge about assembly language programming. If you know another assembler language such as the 8080, 6502, 8088, Z80, or even 360/370, then you will find this book that much easier—you can skip most of the appendices.

There are two essential processes involved in reading this book. One is the process of learning about 68000 assembly language. The other process is learning your way around the Macintosh operating system—its "guts" which make all the graphics, menus, and windows possible. Assembler is the gateway to the heart of the Macintosh.

If you program in Pascal or BASIC and want to keep programming in those languages, a knowledge of assembler is very useful—especially in debugging. You will find that certain routines have to be made fast and compact—your knowledge of assembler will be invaluable then.

However, the main reason that any assembler language programmer gives for wanting to program in assembler is the feeling of POWER. You can actually get down into the insides of the machine and make it perform to its maximum capability. With the sophistication of 68000 assembler and the Macintosh ROMs, your coding won't be that much slower than coding in a higher-level language and your debugging will probably be faster.

There is also a certain FUN in programming in assembler which you don't get when programming in a higher-level language. We suppose it is

the same thrill a mechanic gets from tuning up a fine race car or any mad tinkerer gets from a new invention.

We look forward to helping people learn to program in 68000 assembler on the Macintosh without having to struggle as we did. Of course, we didn't have this book to help us!

This book was written just the way Pterodactyl Software, the company of which we are both part, is run. Ed Rosenzweig wrote the less technical chapters while Harland Harrison wrote the more technical, detailed chapters and the sample program.

EDWIN ROSENZWEIG AND HARLAND HARRISON
(Fairfax and Belmont, California)

Acknowledgments

We would like to thank the following people:

Mike McGrath, Bill Grout, and the people at Hayden—for all their help.

April Post—for helping with the illustrations.

Kirk Austin and Bruce Southwick—for proving, by reading the manuscript and immediately starting to write code, that one could learn Macintosh assembler and the 68000 through this book.

Reina Harrison—for patience and understanding.

Lori Sweet—for patience while an author struggled.

Organization of this Book

This book is written so that people who want to learn assembly language on the Macintosh computer can learn at their own pace. If you know a higher level language such as Pascal, BASIC, or C but have never programmed in assembler then you should read all the chapters in sequence with detours to the appendices where appropriate. If you have programmed in assembler before, you can skim the first chapter, skip the appendix on binary and hexadecimal, and start your in-depth reading with Chapter 2, The Addressing Modes of the 68000. You an also skim the sections on assemblers and linkers in Chapter 6. The section on 68000 hardware is very interesting (it explains why all addresses must start on even boundaries, for example) but it, too, can be skimmed.

Summary of the Chapters

The first chapter, Introduction to Assembly Language, shows how assembly language relates to higher level languages, what a linker does, how the Macintosh editor and assemblers work, what the format of an assembler file is, what addresses are, and some information about the registers and stack of the 68000. This chapter gets you up to speed with the tools and concepts necessary to start learning 68000 assembler. People not previously exposed to assembly language may want to read Appendix A, The Binary Numbering System, at this time.

The second chapter, The Addressing Modes of the 68000, describes all the various ways of finding where the data is located. When using the 68000 there are twelve different ways you can address data including modes as powerful as Program Counter relative with index and displacement. This chapter is one of the building blocks of the rest of the book; it must be understood completely before continuing.

The third chapter, The 68000 Instruction Set, describes every 68000 op code (short for operation code) and related operands. The op codes are arranged by frequency of use. As a reference, Appendix B contains timing information and complete formats. Appendix B is an alphabetical listing. Appendix C contains effects on condition codes for each 68000 instruction.

The fourth chapter, Sample Programs, contains a series of simple subprograms so that you can see concrete examples of how 68000 assembler is put together into workable programs. This chapter shows how to compare two strings, how to optimize assembler, how to convert from hexadecimal to decimal, how to write a key stroke handler, how to do simple arithmetic operations.

The fifth chapter, Overview of the 68000 Hardware, describes how the 68000 looks to a programmer. Here such questions as why instructions

must start on an even byte and what a bus line does are answered. This information is helpful to the programmer who wants to better understand the idiosyncrasies of the 68000 microprocessor.

The sixth chapter, Macintosh Tools, describes the assembler, linker, and resource compiler in detail. Here you will learn the steps necessary to create working Macintosh programs once you have assembled a program.

The seventh chapter, The Mac Environment, describes how windows, menus, and dialog boxes are handled, what events are, Update and Activate events, resource files, memory and jump tables.

The eighth chapter, Macintosh ROM Calls, explains how Macintosh ROM calls are made via traps. How to make a ROM call in assembler is illustrated. Heaps and handles are discussed, sample QuickDraw ROM calls are described.

The ninth chapter, SimpleCalc, has a complete sample program and an explanation of its internals. SimpleCalc is a simple integer spreadsheet program written in MDS assembler. Many of the procedures you will need to program a Macintosh application are shown in this example.

The tenth chapter, Some Advanced Subroutines Not in SimpleCalc, shows sample assembler code to make ToolBox calls for such functions as using the Memory Manager, creating variable text in a dialog box, cursor handling, dragging selections with the mouse, changing menu items, printing, and using the SANE floating point package.

Contents

CHAPTER

Introduction to Assembly Language

Who Will Find This Book Helpful

This book is for people who have had some experience with programming in Assembler, BASIC, Pascal, C, or some other higher level language. Experienced Assembler programmers will find interesting tips and examples that can quickly get them going in the world of Macintosh assembly language programming.

Assembler Used in This Book

Although assemblers differ, even when they're for the same chip, once you know one assembler you are 80% of the way to knowing the next assembler you encounter. The assembler we will be using throughout this book is the one sold by Apple Computer for the Macintosh computer—the 68000 Mac Development System (or MDS).

Why Should You Use Assembly Language?

Higher level language programs often have certain functions they must perform that require great processing speed, more compact code, or interaction with portions of the Macintosh otherwise inaccessible. Often, portions of these programs are written in assembler and called by the higher level language to perform such functions. For example, routines to sort data are usually written in assembler—whenever a sort is needed the higher level program calls the sort routine written in assembler.

Disk access routines and data bases are often written in assembler as well. So are word processors and games programs. Game programs which

run as fast and do as much as the computer can possibly handle make for good competition—they are therefore nearly always written in assembler. Higher level language compilers, interpreters, and programming tools in general are often written completely in assembler since they must be as fast and compact as possible. Finally, programs involving either graphics or music must often be written in assembler, at least in part, in order to have enough speed to create useful effects.

Relationship of Higher Level Languages to Assembler

Many people who use higher level languages are not aware of the true relationship between the language they use and the computer. There is a lot that goes on between the BASIC or Pascal (or perhaps C or Forth) program and the actual hardware of the 68000.

The Macintosh (or any computer) only understands what is called machine language, an encoding of instructions to the 68000 computer chip to tell it what to do. Interpreters (programs that allow you to run BASIC or some kinds of Pascal) read each character then figure out what has to be done. (For example, when an interpreter finds "I" then "F" then a space it realizes that this is an IF statement, and it goes to the machine language routine that was written to handle the IF statement). So although the user of an interpreter often doesn't realize it, the computer doesn't really understand BASIC or Pascal, it only understands 68000 machine language. When you use an interpreter you are actually using a sophisticated 68000 machine language program that someone wrote. So an interpreter is a program that reads a user's program and can do whatever is necessary to make the user's program work the way that it needs.

A compiler (often used together with something called a linker) is another way of making higher level programs work. Compilers are more difficult to use than interpreters, in general, since rather than just running the program you must compile, then link, then resource compile. When using a compiler you edit the program using an editor or word processor. As soon as the program is the way you want it, you compile it by running a program called a compiler and telling it which text file you want the compiler to use.

Then the compiler reads through the entire text file from beginning to end and translates the program into machine language (in our case, 68000 machine language). Some compilers generate the input to a linker program rather than generating a program that you can run immediately. In that case you must run the linker program before you have a program that you can use. Compilers often take a minute or two to compile a program of a few thousand lines.

In other words, when using an interpreter your program stays in text file format and is run as a text file. When using a compiler your program is

BASIC
In the following code, the BASIC is represented by lines starting with a semicolon. Each line of BASIC is followed by its equivalent assembly language; one line of BASIC translates into many lines of assembler. Notice that there is no code generated for a REMark line.

```
;     IF Item% < 11 THEN 810    ' Wait for OK
 cmpi.w #$000B,ITEM.$(A5)
 bge P$068
 jmp L$0810
P$068
;    REM Screen print the dialog box
;    IF Item% = 11 THEN LCOPY WINDOW
 cmpi.w #$000B,ITEM.$(A5)
 bne P$071
 n$p P$072
 move.l $0200+ifn$ptr(A5),file$num(A5)
 moveq #$02,D0
 jsr pr$btl
P$072
P$071
;    OPEN "Mail List" FOR APPEND AS 1
 jsr lit$str
 DC.B 'Mail List'
 DC.B 0
 .align 2
 move.w #$0010,file$mode(A5)
 move.w #$0001,file$num(A5)
 move.w #$0080,D1
 move.l (A7)+,D0
 jsr open$file
;    FOR I%=2 TO 6
 move.w #$0002,-(A7)
 move.w #$0006,-(A7)
 move.w #$0001,-(A7)
 clr.l D2
P$075 lea  I.$(A5),A1
 jsr r$4int
 bra P$076
P$077
```

Figure 1-1 Assembly Language Equivalents for Lines of Short BASIC/Pascal Programs

PASCAL
The Pascal is represented by lines starting with a semicolon and is followed by its equivalent assembly language. Only part of this Pascal procedure is shown, enough so that you can see how Pascal code also generates multiple lines of assembler.

```
;PROCEDURE DrawBrick(pt1, pt2: Point3D);
;{ draws a 3D brick with shaded faces.
;  only shades correctly in one direction.
;}
;VAR        tempRgn: RgnHandle;

           xdef       drawbrick
drawbrick
           link       A6,#-28
           movem.l    A4/D7,-(SP)
;BEGIN
           movea.l    12(A6),A4
           move.l     (A4),-12(A6)
           move.l     4(A4),-8(A6)
           move.l     8(A4),-4(A6)
           movea.l    8(A6),A4
           move.l     (A4),-24(A6)
           move.l     4(A4),-20(A6)
           move.l     8(A4),-16(A6)
   ;tempRgn := NewRgn;
           clr.l      -(SP)
           dc.w       $A8D8       ;
           move.l     (SP)+,D7
   ;OpenRgn;
           move.l     D7,-28(A6)
           dc.w       $A8DA
   ;MoveTo3D(pt1.X, pt1.Y, pt1.Z); { front face, y=y1 }
           move.l     -12(A6),-(SP)
           move.l     -8(A6),-(SP)
           move.l     -4(A6),-(SP)
           jsr.w      moveto3d
   ;LineTo3D(pt1.X, pt1.Y, pt2.Z);
           move.l     -12(A6),-(SP)
           move.l     -8(A6),-(SP)
           move.l     -16(A6),-(SP)
           jsr.w      lineto3d
; {more code to procedure follows, but is not shown here}
```

Figure 1-1 Assembly Language Equivalents for Lines of Short BASIC/Pascal Programs *(continued)*

translated by a relatively slow process into a machine language program which is then run. Once a program has been compiled it may be run as many times as you want without recompiling; an interpreted program must be re-interpreted each time it is run.

Compilers are used by professional programmers and people who need programs that run faster than an interpreter would allow. Games

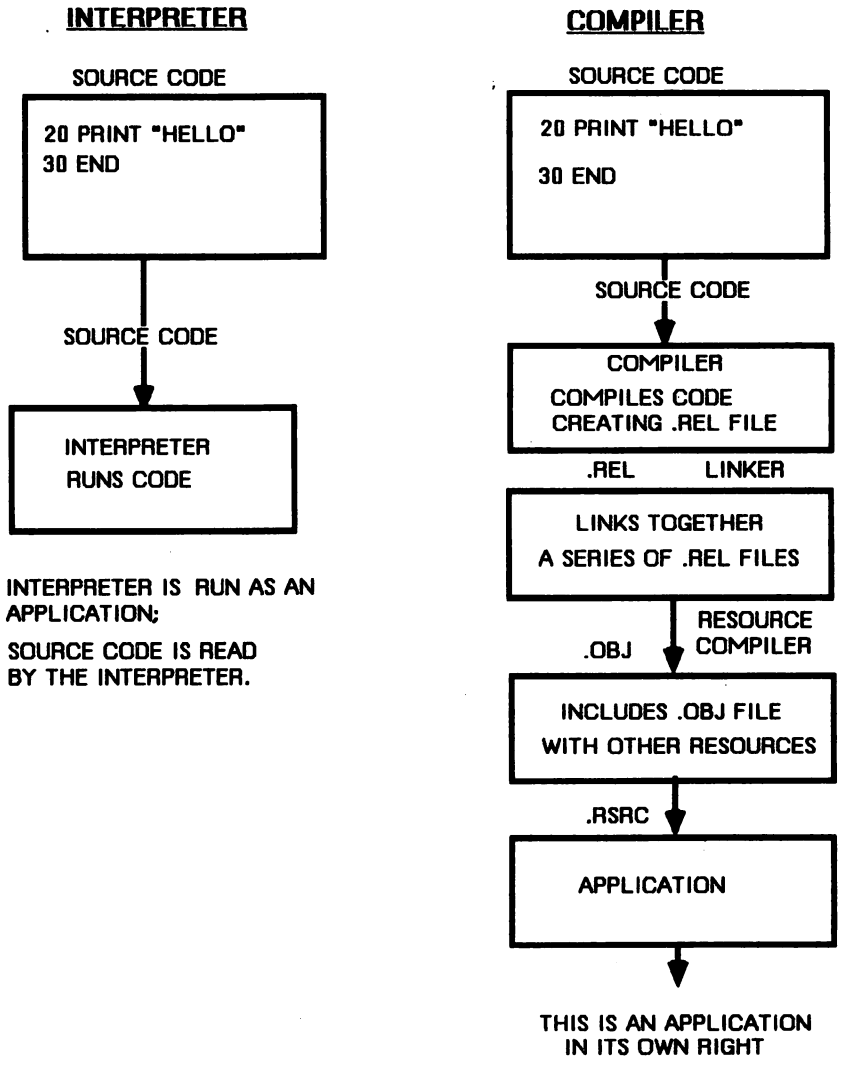


Figure 1-2 Interpreter versus Compiler (*Pascal code reprinted courtesy of TML Systems*)

programmers and many graphics programmers can't use compilers since they generate machine code that is too big and slow—the machine code generated by a compiler is nowhere near as "tight" as that written by an assembly language programmer. People who really need to push their computer to the limit use assembler.

The advantages of an assembler include those of a compiler: the ability to create an application which you can start by double clicking (with its own icon), greater speed than an interpreter, ease of manufacture, privacy of source code, and greater ease of use for the application. Assembler programs also include the capacity for greater speed and flexibility. Unfortunately, it usually takes longer to program the same function in assembler than in a compiled language.

Before going on, let's clarify some terms. *Code* is just a file containing a computer program. The *source code* is a program written in a higher level language such as BASIC, Pascal, or even assembler. The *object code* is the output of a compiler or assembler. *Executable code* is a file that is ready to run on the Macintosh—you can double click in its icon (or the icon of a file that it has created) and it will run. Oftentimes you must run the object code through a linker before you have an executable module. On the Macintosh you usually have a resource compiler phase following the linker phase (more on resources and resource compilers later). The "$" symbol in front of a number means the number is in hexadecimal, a "%" preceding a number means it's binary.

So the phases you run through in compiling a Macintosh program are: edit your source program, compile the program, link it, and resource compile it. Test it on the Macintosh and cycle back to editing, compiling, linking, and resource compiling until you have a working program. When using an assembler you will usually edit, *assemble*, link, and resource compile.

A linker takes a whole series of programs that you have assembled and merges them all together into one big program. This way, if you have a system that would take two and a half hours to assemble as one big mass (yes, there are such systems) you would instead break the system down into twenty small programs that each take only a minute or two to compile and link them all together with the linker. Since the linker only takes a minute or two itself, you have reduced the time for making change and testing again from two or three hours to making the same change and testing in three or four minutes.

In programming, as in other disciplines, it really pays to be able to break a large task into smaller, simpler tasks that each work independently. That way you can break a large task into smaller tasks that you can do yourself or you can work on with other people as part of a team. Each member of the team can develop a small, self-contained part of the large system. A linker is very helpful in such a breakdown of tasks.

When you learn assembly on the Macintosh you will be using a linker, since the output of the assembler that you will be using generates a linker file. These files are often called "rel" files, short for *relative* files. This is because the linker generates programs that can be relocated in memory.

How to Program in 68000 on a Mac

In order to program in 68000 you have to know how to use an assembler. We will describe the 68000 development system assembler/debugger package (also called MDS, short for Mac Development System) from Apple Computer for the Macintosh. Since many developers also use the Lisa (Macintosh XL) system to develop programs for the Macintosh using the Pascal Workshop's assembler and the Software Supplement, we will also describe these packages. The detailed description of this development system will come in Chapter 6, Macintosh Tools, which comes after the basics of 68000 assembler have been learned.

We might mention at this point that although the assembler can be run with one microfloppy disk drive, it works best with two disk drives. Of course, if you can afford it, a hard disk is very helpful in any professional system. Also, 512K or more of RAM memory in your Macintosh is also essential to using the 68000 Mac Development system.

The MDS actually consists of a whole series of "tools" (software programs) that you can use to create assembly language programs that use the full power of the Macintosh. When you are programming in assembler not only do you need an *assembler* program to turn your assembly language text file into machine language that can actually be executed, but you need a *debugger* program that can be used to find what is going wrong when your program doesn't work. Usually other programs such as *linkers* which can link together separate assembler programs into one big assembler program are necessary also. On the Macintosh, however, one other program which is unique to the Macintosh environment is also necessary—a *resource compiler*. We will describe the resource compiler later, after we describe the assembler.

An assembler has to have an *editor* so you can text edit your program before sending it into the assembler for translation into machine language. The editor that comes with the Macintosh assembler is relatively easy to use. It is made for programming in assembler with its indentations and large programs. There is a menu that allows you to instantly go to any other part of the MDS, such as the EXEC files which execute canned series of commands. Using EXEC files, you don't have to repeatedly type in the same sequences of commands to assemble and link a program for example.

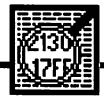
If you have ever compiled a Basic or Pascal program you are probably familiar with the compiler (assembler) and the linker. You have probably

never used the resource compiler since that is a concept unique to the Macintosh. We'll introduce the editor and enough about the assembler to get you started; detailed descriptions of the linker, resource compiler, and EXEC files will be presented in Chapter 6.

The Editor

If you have worked with MacWrite you are well on the way to understanding the editor that comes with the Macintosh assembler. In fact, if you want, you could edit your programs with MacWrite, turn them into text files by saving them under the "text only" option, and then use them in the assembler without ever having to use the Macintosh assembler editor. We will assume that you have used MacWrite and understand already the concepts of cutting, copying, pasting, and opening files—we will therefore only describe the differences between the assembler editor and MacWrite. You enter the editor just the way you do any Macintosh program, by double clicking in the EDIT icon.

The editor supplied with the assembler is geared to fast writing and printing of assembler programs, and eliminates some of the fancier features of MacWrite. Also, MDS is what is called an "integrated" system—this means that you can easily and quickly switch from any one part, such as the editor, to any other part, such as the assembler, quickly and easily. In an integrated system the files created by any one part can be used by the other parts of the system.



Discussions and Comparisons

Menu options show some of the differences between MacWrite and the assembler editor. In the File menu of the assembler editor there are two Open commands—the first Open command uses a dialog box to select a text file to open. There is an Align command in the Edit menu which allows you to line up the starting columns of a whole series of text lines to that of the first text line. Also, you can shift a whole series of selected lines right or left one space by selecting Move Right and Move Left from this same Edit menu or by using the command key options ⌘ for these commands. The Show Invisible option on the Format menu allows you to see all the special characters which control such things as tabbing, spacing, and carriage return.

The assembler editor provides these special tabbing and alignment operations because the text file which is used as input to an assembler must follow certain rules. When writing in assembler you must have the labels in column 1, the op codes in column 10 or so (depending on how long your labels are—the opcodes must not be in column 1, though), the operands lined up in another column, and finally the comments lined up around column 30 or 40.

The assembler editor's main difference from MacWrite is the Transfer menu which allows you to exit the edit program and go to the assembler (ASM), linker (LINK), executive function (EXEC), and resource compiler (RMAKER). The usual sequence of events is: *edit, assemble, link, resource compile* (assemble, link, and resource compile are often run under control of an EXEC file), and *run* the program using debug.

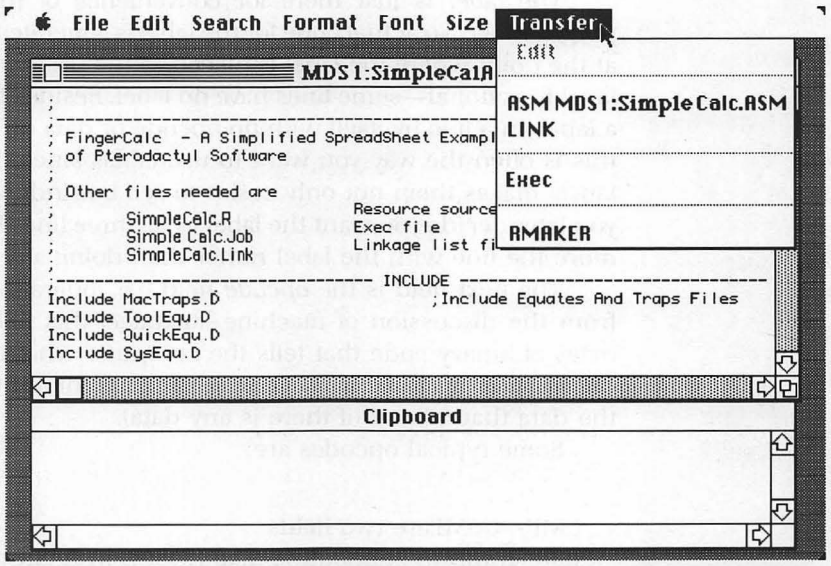


Figure 1-3 Assembler Editor's Transfer Menu

The Assembler

Nearly all assemblers that you use have certain areas or *fields* where they expect information to be placed. If a word is typed in the first column on most assemblers, it is considered to be a label. This label is then used in JMP (JuMP) instructions—like GOTO in BASIC/Pascal—or JSR (Jump Sub-Routine)—instructions like GOSUB in BASIC or a Procedure in Pascal. For example:

```
;label    opcode data
         JSR MyLabel
         JMP Nextlbl
MyLabel  MOVE.L A0,D0
         RTS
         etc.
```

In the fourth line of the above example, MyLabel is the label used by the JSR statement two lines above it. An equivalent program in BASIC wouldn't have a label—you would GOSUB a line number instead. However, some of the more advanced BASICs are now using labels similar to the lower level assembly language. In Pascal, you would just give the name of the procedure, MyLabel, in another place in the program—the procedure would be performed and control would return to the next statement after the procedure.

The *label* is just there for convenience of the assembly language programmer. No actual code for the label is generated in machine language at the point where the label is placed in the program. As you can see, the label is optional—some lines have no label. Besides this, you can also have a label on a line by itself with no opcode or data on the same line. In fact, this is often the way you want to use labels since this way of implanting labels makes them not only easier to see but independent of the code. If you later decide you want the label to be three lines further down, you just move the line with the label rather than doing any editing within a line.

The next field is the *opcode* field (i.e., operating code). As you know from the discussion of machine language, this field is turned into two bytes of binary code that tells the computer what to do next. Part of the information in those two bytes tells the machine the format to expect in the data that follows (if there is any data).

Some typical opcodes are:

CMP—CoMPare two fields
JSR—Jump to SubRoutine (like GOSUB in BASIC)
JMP—JuMP (like GOTO in a higher level language)
ADD—ADD two numbers
RTS—ReTurn from Subroutine (like RETURN in BASIC)
MOVE—MOVE the first field's data to the second field
SUB—SUBtract the first number from the second number
BNE—Branch Not Equal (example: if a prior CMP doesn't come out equal, GOTO...)
ASL—Arithmetic Shift Left (shift all the bits left by a count)
MULU—MULtiply Unsigned (multiply two 2-byte numbers giving a 4-byte number)

Sometimes the opcode field will end in a ".B" or ".W" or ".L". These opcode suffixes tell the 68000 that it will have to operate on either a byte, a

word (two bytes), or a long word (four bytes). So MOVE.B A0,D0 will move the rightmost (low order) byte of A0 to the rightmost byte of D0, MOVE.W A0,D0 will move the rightmost two bytes of address register A0 to the rightmost two bytes of data register D0, and, lastly, MOVE.L A0,D0 will move the whole of register A0 (all four bytes) to register D0.

The third field is the *data* field. This field contains whatever information the op code needs in the correct form. Sometimes the opcode needs no data—then there is no data following the opcode. Oftentimes the data will be a pair of registers. The most often used instruction a 68000 programmer invokes is the MOVE instruction. The MOVE instruction contains the source of the data in the leftmost part of the operand, a comma, and the destination to the right. So MOVE A0,A1 will move data from address register zero to address register one.

Sometimes there is a semicolon after the data which indicates the start of a *comment*. This is like REM in BASIC or "(*" "*)" placed at the start and end of a comment in Pascal. The rest of the line following the semicolon is ignored by the assembler. In assembly language it is imperative that you comment nearly every line. You can get away without comments in a higher level language, and still be able to follow the code, much more easily than in assembler. Using meaningful names as labels and variable names also helps document your program.

Now you see how the code and labels work. But there is more to assembler than code—there is also data. Often you will want to create space for variables. Assembler isn't like BASIC or Pascal where you declare a variable and it magically exists. Sometimes you need to create a table of constant values which are needed by your program. Data is handled differently in assembler than in a higher level language.

The primary ways of creating data are with the DS (define storage) and DC (define constant) commands. If you want to create a variable in assembler you use the DS command. For detailed description of these and other assembler directives, see Chapter 6, Macintosh Tools.

Besides assembly language's unique way of handling variables there are other assembler features that you usually don't find in higher level languages. There is the ability to include other text files containing code and data definitions in your program (some languages such as Pascal and COBOL do give you this capacity—BASIC often doesn't). You can also use sophisticated text editing commands called *macros* inside your assembler program. When you combine these macros with the ability to include or delete certain code with *conditional assembly* commands you can customize your programs so that when you change one or more switches, different versions of your program are automatically created. A *switch* is a data area with a label that can be set either zero or non-zero. Most often, you can set a switch so that one version of your program is created for debugging and a different version is created for production (i.e., for users of your program).

Getting Closer to the Hardware

When you are programming in assembler you are closer to the actual circuitry of the computer than you are with any other language. Before actually getting into assembler there are some things about the computer that you must know.

When you program in a higher level language there are various data types such as strings, integers, real numbers, etc. In general, you only need to know the decimal system. However, down at the hardware level the binary system is used. You need to know not only binary but the hexadecimal mode of counting before you can do anything in assembly language. If you don't know binary and hexadecimal, see Appendix A at this time; throughout this book we will assume you know how to count in binary and hexadecimal.

Addresses

Every single byte of memory in the computer, whether RAM or ROM, has an address. Addresses are just numbers, really. You can look at a given address by giving a number inside an assembler instruction or by typing it into a debugging program. Let's say you want to look at memory address number $0000F3C2 or memory address $00010C32. It is through this address that

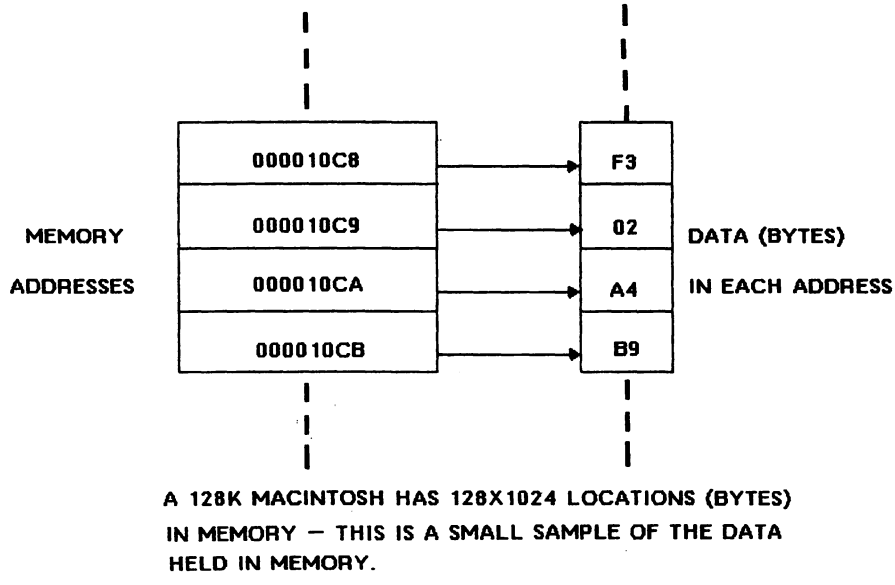


Figure 1-4 Picture of Addresses

you may read what is in that byte of memory or insert a new value there. In higher level languages memory is taken care of automatically—you rarely need to consider actual addresses when programming. In assembly language you are constantly thinking in terms of actual locations in memory, particularly when debugging.

When using the 68000, you will notice that nearly all addresses are four byte addresses (8 hex digits) although the highest byte is unused. Therefore you may have a maximum of 256 times 256 times 256 or 16 million bytes of memory in a 68000 machine.

Internals of the 68000 Microprocessor

Within most microcomputer central processing units are special memory locations called *registers*. There are three main types of registers in the 68000: data, address, and status. When you use these special memory locations your program code takes up less space and goes faster. This is because these memory locations are within the 68000 microprocessor chip.

In the 68000 chip there are not only registers which contain data, as in the 6502 microprocessor chip, but also registers which contain addresses. If you want to write code which deals with one memory location the first time through the code but a different memory location the next then you must use these address registers.

Each register in the 68000 contains four bytes. There are eight registers for data, designated D0, D1, D2,...,D6,D7 (the D stands for Data, obviously) and eight registers for addresses, labeled A0, A1,..., A5, A6, A7.

Typically, you load the data registers with up to 8 different variables that are used the most in your program at that time. You fill the address registers with up to 7 different starting points for blocks of data that can periodically change their location in memory or starting points of code that could be different each time the program is run.

The last address register, A7, is special—it holds the address of a *stack*. A stack is a series of bytes of data that are organized as though they have been "stacked" one on top of the other. The two most common operations are where you place a new element on the stack, called pushing on the stack, or take an element off the stack, called popping the stack. The A7 or seventh address register points to the top element on the stack. Typically, it is also referred to as "SP," short for Stack Pointer, in assembler programs.

On the 68000 the stack grows downward and the stack therefore extends upward in memory from the place the stack pointer points. So the element on the stack with the lowest addresss is on the top of the stack. When you place a new element on the stack, the A7 register is decreased by the number of bytes you place on the stack. That way, it points to the first byte of the new element that you have placed on the stack. When you

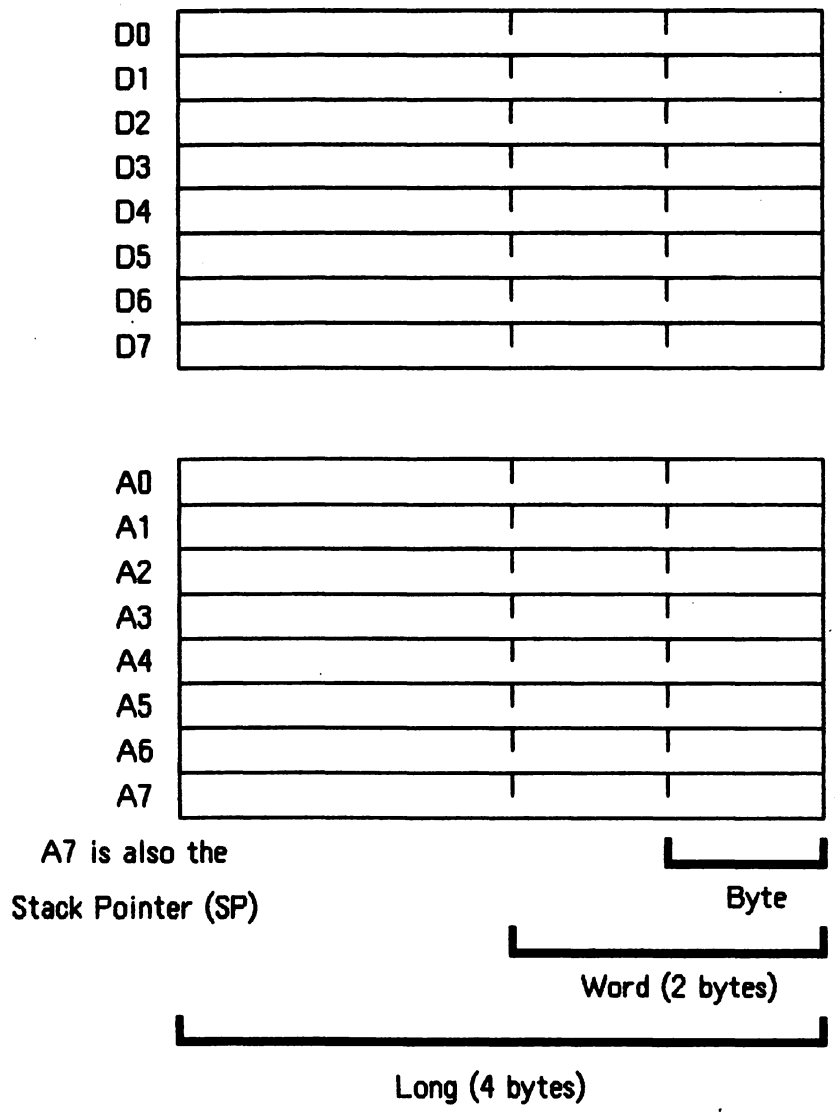
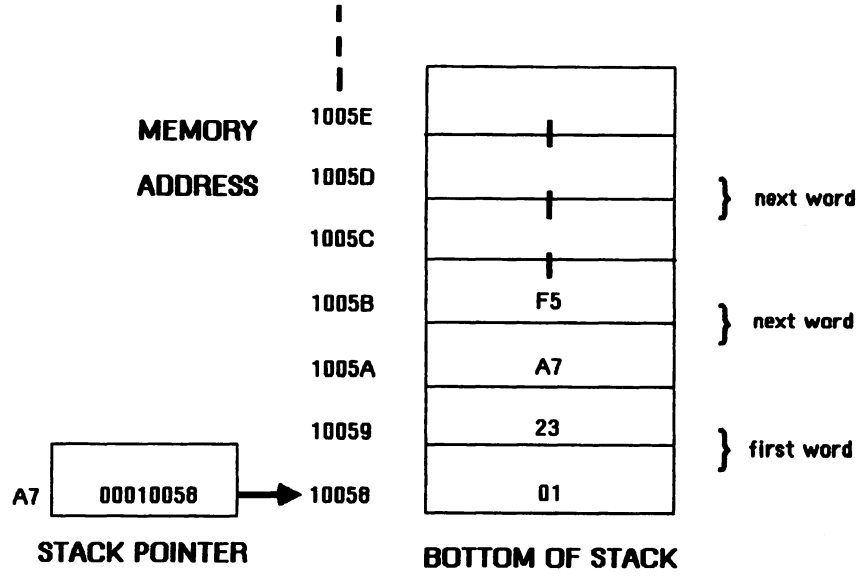


Figure 1-5 Data and Address Registers

take an element off the stack, the stack pointer is increased by the number of bytes you have taken off the stack.

There is one minor technicality about the stack we might mention now. If you place only one byte on the stack, the stack pointer is decreased

by two and the byte is placed where the stack pointer is now pointing. When you take one byte off the stack the reverse happens—the byte is pulled off the stack and the stack pointer is increased by two. This is because the stack pointer must always point to an even address (we will explain later in Chapter 5, Hardware). This is the only exception to the rule of the stack pointer increasing by the number of bytes pulled off the stack or decreasing by the number of bytes pushed on the stack.



NOTE: FIRST WORD ON STACK : 0123

NEXT WORD (DEEPER) ON STACK :A7F5

Figure 1-6 Picture of Stack with A7 Register

Stacks are used to keep track of hierarchies of operations or data. Here's a human analogy. Suppose a busy programmer is dealing with one person on a problem when another person with a different problem suddenly comes into his cubicle. The newcomer has a more urgent problem that must be solved right away. The programmer memorizes where he is in the first problem (pushes data on the stack) and deals with the second problem. Once that problem is resolved, he returns to the first problem where it was interrupted. Recalling where he left off on the prior

problem, the programmer "pops the data off the stack." This means that he begins working with the problem just as it was when he left off. From his memory (stack) he recalls all the significant details that he memorized earlier.

With the Macintosh, the stack that A7 points to is used for subroutine calls—the equivalent in 68000 assembler of a GOSUB in BASIC or a Procedure in a block-structured language like Pascal or C, or a mental bookmark for our problem-solving programmer. The stack is used to save the address of the current instruction in the program when the 68000 leaves off to start a subroutine. That way when you encounter the instruction that says "return from the subroutine" you pull the address off the stack and start the program again from the retrieved address of the next instruction.

This brings up another area of memory within the 68000 CPU—the *Program Counter*. The Program Counter keeps track of where you are in the current machine language program. Inside this 4-byte area is kept the address of the next instruction to be executed. It is updated automatically as each instruction is executed so that it always points at the next instruction.

When a JSR (Jump to SubRoutine instruction) is executed, the address of the next location in memory is pushed onto the stack pointed to by register A7. When a RTS (ReTurn from Subroutine instruction) is performed, that address is popped off the stack and placed in the program counter. Then the next instruction that is executed is the one following the original JSR that called this subroutine. This is the same way that BASIC or Pascal interpreters/compilers work. They just don't tell you about the stack.

The stack is also used to pass data to subroutines and for *interrupts*. Interrupts are an advanced topic which we will discuss in Chapter 3, 68000 Instruction Set, and Chapter 5, Hardware. To whet your appetite, interrupts occur when something outside the computer affects one of the inputs to the computer (such as an operator at the keyboard). An interrupt allows a machine language program to stop, remember what it was doing when it stopped, call another machine language program that was written to handle, for example, a keypress, and then return to what it was doing after handling the interrupt. Interrupts use the stack to keep track of what was happening inside the original interrupted program when it was stopped (the information is pushed on the stack). When the interrupt program is done, then the information, like a bookmark you place in a book when you are interrupted in your reading, is popped from the stack and the original program goes on its merry way. Obviously, part of the information that is pushed on the stack is the program counter telling where the original program was when it was interrupted.

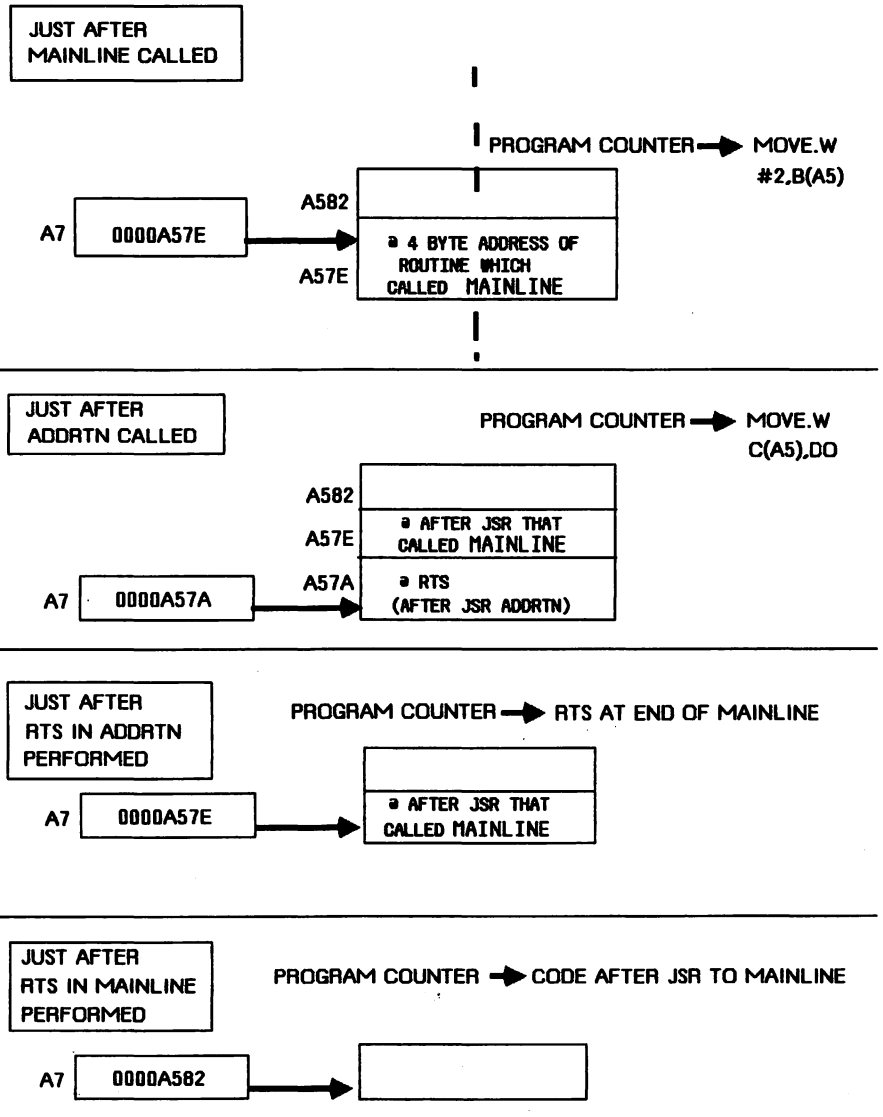


Figure 1-7 JSR-RTS Effect on Stack

Discussions and Comparisons

Here is an example of some assembler code that does a subroutine to add two numbers and return. For comparison, we first show equivalent code in BASIC and Pascal.

BASIC code:

```
10 AMT1=2: AMT2=3
20 GOSUB 40
30 END
40 SUM=AMT1+AMT2
50 RETURN
```

Pascal code:

```
Program Testit;
var Sum, Amt1, Amt2: integer;
Procedure AddRtn; (*this is the subroutine*)
begin
 Sum:=Amt1 + Amt2
end;
begin (* mainline *)
 Amt1 :=2;
 Amt2 :=3;
 AddRtn (* this is the "subroutine" call *)
end;
```

Here is the equivalent code in 68000 Assembler:

The semicolon indicates the start of a comment and the ".ds" means Define Storage. For example, ".ds 2" means set aside 2 bytes of storage.

```
                                    ;Here is the equivalent code
Sum       .ds 2                     ;set aside 2 bytes of storage for the
                                    ;variable sum
Amt1      .ds 2                     ;set aside 2 bytes of storage for the
                                    ;variable Amt1
Amt2      .ds 2                     ;set aside 2 bytes of storage for the
                                    ;variable Amt2
MAINLINE
          MOVE.W   #2,Amt1(A5)      ;move the constant 2 to Amt1
          MOVE.W   #3,Amt2(A5)      ;move the constant 3 to Amt2
          JSR      ADDRTN           ;Jump to SubRoutine called
                                    ;ADDRTN (just like a GOSUB)
```

```
                                    ;push address of following
                                    ;instruction on stack when do above
                                    ;JSR
        RTS                         ;ReTurn from Subroutine (just like a
                                    ;RETURN)
ADDRTN
        MOVE.W  Amt1(A5),D0         ;move what is in the 2 bytes of Amt1
                                    ;to D0
                                    ;(.W stands for word = 2 bytes)
        ADD.W   Amt2(A5),D0         ;add what is in Amt2 to D0
                                    ;(2 rightmost bytes of Data Register)
        MOVE.W  D0,Sum(A5)          ;move what is in the 2 bytes of D0 to
                                    ;Sum
        RTS                         ;ReTurn from Subroutine (pop stack
                                    ;and use address)
```

This assembler program, MAINLINE, illustrates a subroutine calling another subroutine. We can tell that MAINLINE is a subroutine itself since the fourth line of MAINLINE is an RTS (ReTurn from Subroutine). Just before MAINLINE was entered, the program which called it had placed upon the stack (as the top element) the address of the next instruction it wanted to execute.

The first two lines move the numbers two and three into the places in memory that ADDRTN is expecting them. Then ADDRTN is called with a JSR; the address of the RTS following the line JSR ADDRTN is placed upon the stack. Now the stack has two return addresses on it, the place where MAINLINE must return and the place where ADDRTN must return. The addition is performed in ADDRTN, the address where ADDRTN must return is pulled off the stack and placed in the program counter, the RTS at the end of MAINLINE is performed and the address where MAINLINE must return is then pulled off the stack and placed in the program counter. This shows how nested subroutine calls handle the stack.

Condition Code Portion of Status Register

Another important memory location internal to the 68000 chip is the *status register*. This register consists of a series of one-bit flags that record the results of operations performed by prior machine language instructions. The condition code portion of the status register is used to communicate between two or more 68000 instructions to create the same effect as one statement in a higher level language. The flags in the condition code portion of the 68000 status register are: Negative (N), Zero (Z), Overflow (O),

Carry (C), and Extend (E). For example, the zero flag in the status register can be used to combine the compare instruction (CMP) with a branch not equal instruction (BNE) to create the same effect as "IF A <> B THEN GOTO..." in a higher level language.

In general, a binary one in a flag means "true" while a binary zero means false. When the result of a compare instruction is equal, the zero bit is set to 1; when the result of a compare instruction is unequal, the zero bit is set to 0. A flag is said to be "on" if it is 1, and "off" if it is zero. So "true," 1, and "on" are used interchangeably when talking about flag settings; "false," 0, or "off" are used interchangeably for the opposite setting.

A compare instruction subtracts one operand from the other. If the result of the subtraction is zero then the instruction sets the zero flag to 1 saying "it is true that the result of this last operation was zero." If the result was not zero (the two numbers compared are unequal) the compare instruction sets the zero flag to 0.

Discussions and Comparisons

In higher level languages you would write something like the following:

BASIC code:

```
10 IF A <> B THEN 40
```

Pascal code:

```
if a=b then
 begin
 end
 else
 line40; (* if a <> b then the procedure line40 is performed *)
```

There is no one statement in assembler that causes the program to go somewhere else if a condition is true. Instead the above code in assembler would consist of two statements:

```
MOVE B(A5),D0
CMP A(A5),D0    ;compare what is in memory address A to what is in memory address B
BNE LINE40      ;and branch to the label LINE40 if the zero flag is off (it is a zero)
```

In this case the compare instruction simply sets the zero flag either on or off based on whether the result of subtracting A from B is equal to zero. Then the BNE instruction takes action—it branches based on whether the zero flag is off. This branch instruction resets the program counter to the address of LINE40—BASIC programmers call it a GOTO.

Machine Language

The 68000 microprocessor only really understands something called machine language. Machine language is pure binary code in the computer's memory. The 68000 reads through this binary machine code sequentially in memory and performs the actions that are indicated.

Let us look at the rather hectic but repetitious lifestyle of a 68000 Central Processing Unit. The 68000 first looks at the place where its program counter is pointing. There it sees 2 bytes of binary. These bytes are called an *op code* (short for "operation code"). For example, a CMP or BNE.

After reading an op code, the 68000 usually goes to the next place in memory (it adds two to the program counter). If the op code says, "expect data there" then the machine reads the data. If the op code says, "I have no data following me" then the 68000 gets ready to read another op code and do what it says.

This is all the 68000 Central Processing Unit chip does all day. It reads an op code, looks for the data (if there), and performs an action. Then the program counter is bumped past the data and it reads another op code, and so on. Sometimes a JMP (JuMP instruction) changes the program counter and it reads the next op code from a place in memory that is not sequential. Of course the 68000 does these operations millions of times a second.

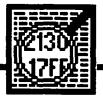
Sometimes the 68000 CPU encounters an instruction which is not in its repertoire. This definitely is a change of pace for the 68000. The 68000 creates its own interruption of the flow of the program and goes to a place where the Apple computer people have a program that displays a window with a picture of a bomb and an ID = 02 (The IDs are a code telling the cause of the blowup—they are low integers). If you look up what an ID of 2 means it says, "bus error" but really it usually means that the 68000 was asked to perform something it does not know how to read.

Although you need to know hexadecimal and binary in order to understand the data in memory, you will rarely need to know the format of op codes and operands in 68000 machine language. You can see them as represented in assembler mnemonics, however, when you look at any debugger's disassembly listing of machine code (more on disassemblers later in the section on debuggers in Chapter 6—disassembly is only one function of a debugger). A disassembly is where machine code in memory is turned into the assembler mnemonics such as "CMP."

Somewhere back in the birth pangs of the 68000 chip some computer programmer sat down and wrote an assembler. An assembler program reads a text file with such codes as "CMP A(A5),D0" and turns it into a binary file with op codes and encoded data following it. Branches to labels are turned into the op code for the branch and an offset from the current location to that of the label.

Between assemblers and disassemblers you will probably never need to know machine language. You will only have to know assembler language. Each line of assembler usually corresponds to one op code (this is represented by the "mnemonic" such as CMP) and the data that may follow that op code (in the above example, "A(A5),D0").

It is much easier to remember the three letters CMP mean compare than the binary value 1011 010 001 001 011, which means CMP.W (with data expected in the form A3,D2). The first 4 bits of this two-byte op code signals that this is a CMP instruction; the next 3 bits, which represent the number 2 in binary, specify that the register to receive the value is data register D2. Then come 3 bits which signal that 2 bytes (a word) are to be compared. The 3 bits which follow indicate that the pattern of the data that is the source for the compare is an address register, and the final 3 bits say that it is address register 3 (A3).



Discussions and Comparisons

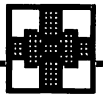
Binary code:

1011 010 001 001 011

Assembler code:

CMP.W A3 D2
Meaning: compare 2 bytes of A3 to D2

As you can imagine, almost any competent programmer would much rather type CMP.W A3,D2 then 1011 0100 0100 1011 or (in hexadecimal) B44B.



Summary

The next chapter, 68000 Addressing Modes, introduces you to the syntax of 68000 assembler. Once you read Chapters 2 and 3 (68000 Instruction Set) you will know the complete syntax of 68000 assembly language.

CHAPTER

The Addressing Modes of the 68000

One of the real differences of assembler language from higher level language is the assortment of ways that you can access data. There are 12 ways of accessing data in the 68000, called "addressing modes." Most of the real power of assembler comes from understanding all the addressing modes. Therefore, to keep from getting lost, you really must understand these modes before going on to another chapter. We will go through these addressing modes in order from simplest to the most complex. But first let's look at an addressing mode that doesn't address any data at all.

Inherent Mode

Inherent mode is a pretentious way of saying there is no data to address. Inherent mode does not count as a true addressing mode. An RTS, which means ReTurn from Subroutine, stands by itself and has no operand; i.e., it is in "inherent mode." RTS automatically knows where to return since that location is always on the top four bytes of the stack. It should be inherently obvious that RTE (ReTurn from Exception) and RTR (ReTurn and Restore condition codes) both use inherent mode. Most inherent mode operations pull data from a known place (such as the stack) or go to a place based on a known pointer (usually a place in memory agreed in advance.) Some inherent mode operations such as NOP (NO OPeration) don't need any data since they take up space and time and do nothing.

Example

Here's an example where an RTS inherent mode op code is used:

```
        JSR  ADD1       ;like a GOSUB or Procedure in Pascal, do the subroutine at ADD1
        JMP  ENDIT      ;jump to another place outside this chunk of code
                        ;the ADD1 subroutine
ADD1    ADD  #1,D0      ;add 1 to register D0
        RTS             ;return from where you came (like RETURN in BASIC)
```

In the above example JSR ADD1 jumps to the location ADD1 and puts the location of JMP ENDIT on the stack. The routine adds 1 to register D0 and then pulls the location after the JSR ADD1 (i.e. the location of JMP ENDIT) off the stack and returns there. Then it takes the JuMP to ENDIT and continues on its way. RTS is an inherent mode operation—it knows exactly what to do and needs nothing further on the same line (in contrast to the other op codes in this short program segment.)

Example

Here's an example where the NOP code is used:

```
        MOVE #20,D0     ;cycle around loop 20 times
TIMELOOP NOP            ;do nothing for a little while
        NOP             ;do nothing for a little while
        SUB  #1,D0      ;count down from 20
        BNE  TIMELOOP   ;so long as the count isn't zero, jump to timeloop
```

NOPs are often used in loops to literally waste time. For example, timing loops such as the one above are used to slow down the beats of an oscillator thus making a musical tone deeper or to slow down the response to the mouse so that a person has time to respond. NOPs act like politicians at banquets taking up time prior to the meal with speeches that lack content; they don't need any operand to supply them with substance.

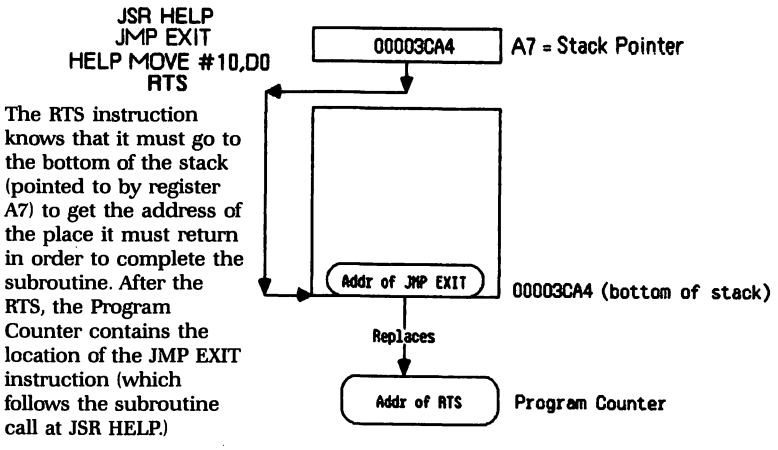


Figure 2-1 Inherent Addressing Mode

1. Immediate Mode

With *immediate mode* the data is immediately available for the operation. Usually you see immediate mode data with a "#" symbol in front of it. For example:

```
MOVE #1,D0
```

This moves the constant, 1, into register D0. The source operand, 1, is in immediate mode. Another way of thinking of immediate mode is that the data immediately follows the op code as part of the program itself. Immediate data is constant data. The above statement would be like saying D0 = 1 in a higher level language.

Immediate mode is always in the source of an operand, never in the destination, since you can't operate on a constant.

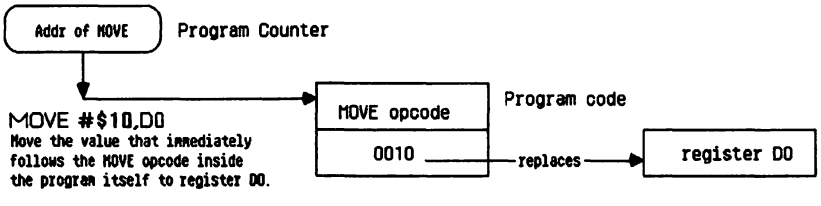


Figure 2-2 Immediate Addressing Mode

2. Data Register Direct Mode

As you remember, the data registers are called D0, D1 ..., D7. The secret to coding fast, tight code on the 68000 is to keep the most often used data in these registers. Data in these registers is not only the easiest to access, it is also the fastest.

Data register direct mode means simply that you name the source or destination operand to be a data register. The most common form is something like the following:

```
MOVE.L D0,D7
```

Here both source and destination are in data register direct mode. This instruction moves all 32 bits of D0 to D7. In the example for immediate mode where we had MOVE #1,D0 the destination was in data register direct mode.

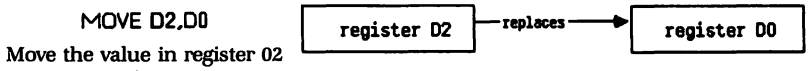


Figure 2-3 Data Register Direct Mode

3. Address Register Direct Mode

Address register direct mode is the same action as data register direct mode except that the address registers A0 through A7 are used. By placing an address into a register you can create a pointer to some data in a table. For example, suppose you know a table has two byte long elements, an index of an element of the table is in the D0 register, and the table's start is in the A0 register. You want the address of a particular element from the table to be in the A1 register. Here is the code to do it:

```
MOVE.L  D0,D1   ;use D1 as a temporary work register
ASL.L   #1,D1   ;multiply D1 by 2 by shifting left one position (bit)
                ;ASL means Arithmetic Shift Left, #1 means one bit
MOVE.L  A0,A1   ;now the table's start location is in A1
ADD.L   D1,A1   ;finally, the location of the required element is in A1
```

The first line has both source and destination as data register direct. The second line has the source as immediate and the destination as data register direct. The third line is an example of both source and destination being address register direct mode. Finally, the fourth line has the source as data register direct and the destination as address register direct.

When you think about it, you usually just think "MOVE register D1 to register A1." It is only when you encounter an assembler error that you look up the tables of permissible source and destinations to find that the appropriate mode is impossible (in Appendix B in this book). For example, you must MOVEA rather than MOVE anytime the destination is an address. So line three could read MOVEA.L A0,A1. However, all 68000 assemblers automatically substitute MOVEA for MOVE if the destination is an address register. That is why we placed that technically incorrect statement in line two—it will work in your assembler.

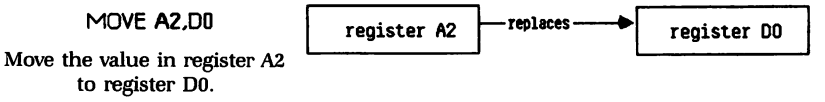


Figure 2-4 Address Register Direct Mode

4. Absolute Short Mode

This mode uses the two bytes of immediate data following the instruction's opcode as an address. It can only be used for a fixed address in memory. You may not use this mode of addressing very often since the Macintosh often reallocates memory during processing, putting absolute addresses off target. The advantage of using absolute short mode is that it takes less time to access data and less coding space than using absolute long addressing since only 2 byte rather than 4 bytes addresses are used.

Form: <*constant*> (where *constant* is less than 32K or greater than 16 Meg minus 32K)

But the 68000 has 4-byte, not 2-byte, addresses you say? Well, this gets a mite tricky. If the address is between $0 and $7FFF (the address is in the first 32K of absolute memory) then the address is just where you specify it in the first 32K of memory. However, if the address is between $8000 and $FFFF the address will be in the last 32K at the top of the 16 Megabyte address range of the 68000. Technically, the 2 bytes are said to be "sign extended." There aren't too many computers that come with 16 Megabytes as standard—it may be a while before they upgrade the Macintosh to 16 Megabytes!

Since the Macintosh pages most application program memory in and out, you usually will not be using absolute addressing at all, much less memory in the first 32K. By "paging in and out" we mean that portions of both the program and data are constantly being moved in memory and even erased from memory to be re-read from the disk when they are needed. When a part of the program or data is read from the disk that is

called "paging in" and when part of the program or data is (optionally) changed on the disk and then erased from memory that is called "paging out." This process must be kept in mind constantly by programmers accustomed to smaller microcomputers where programs and data could stay in fixed locations in memory—hardly anything should be fixed in memory on the Macintosh.

Example:

```
MYRTN  equ $1234 N ;the subroutine MYRTN is located at $1234, a location below $7FFF
       JSR MYRTN   ;where MYRTN is in the first 32K
```

In this example, MYRTN is a subroutine fixed in memory at address $1234. The "equ," short for equate, defines MYRTN as a constant, $1234. When JSR MYRTN is encountered by the assembler, it generates a Jump to SubRoutine at location $1234 using short addressing since the address is less than 32K.

Another example:

```
LOWMEM equ $4567          ;a location below 7FFF
       MOVE D0,LOWMEM ;move data register D0 to a place in the first 7FFF of memory
```

In this case, LOWMEM is used to find data rather than code. The data in D0 is moved to a location in low memory; the destination LOWMEM uses short addressing mode.

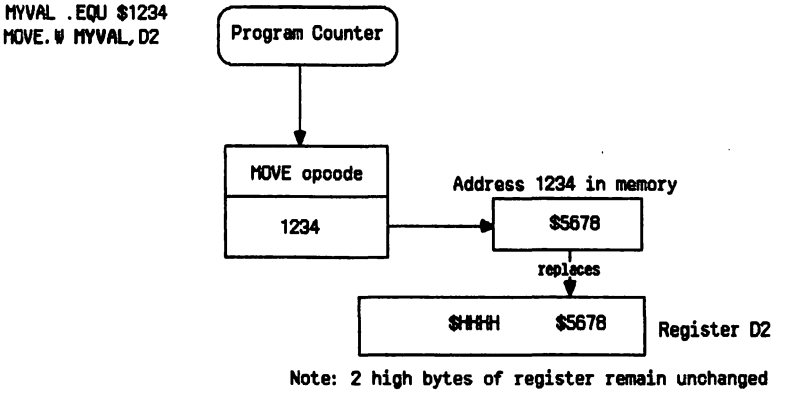


Figure 2-5 Absolute Short Addressing Mode

5. Absolute Long Mode

More often used than absolute short mode, *absolute long mode* addresses memory with full 4 byte locations. Again, using absolute addressing of memory is very unusual on the Macintosh since all memory is dynamic — it is in constant movement. You usually put the starting address of your data's location into an address register and then find a particular portion of data as an offset from that. More on this later.

Form: <*constant*> (where *constant* does not fall in the short addressing range)

Examples:

```
MYRTN   equ $11345
        JSR MYRTN
```

This example is absolute long addressing mode since MYRTN is a constant ($11345) and is greater than $FFFF.

Another example:

```
HIMEM   equ $91067
        MOVE D0,HIMEM
```

In this case data is being moved to a fixed location location in memory ($91067) causing this to be absolute long addressing mode.

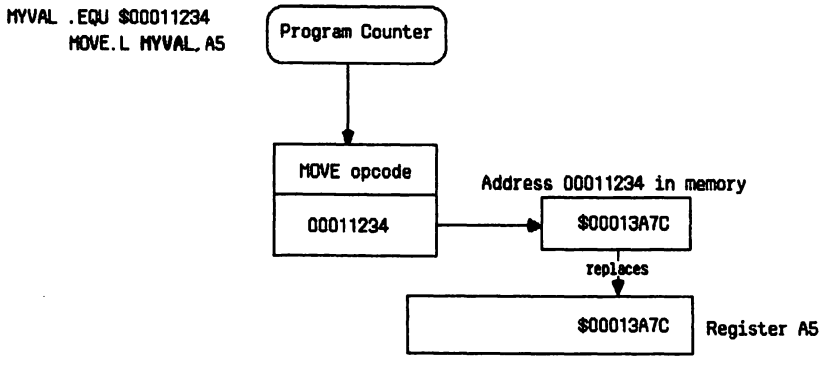


Figure 2-6 Absolute Long Addressing Mode

6. Addressing Register Indirect Mode

Addressing register indirect mode is the first of many indirect addressing modes. Indirect addressing means that rather than getting information from a constant place in memory, you calculate where your data is, put the result in an address register, and operate on the data at that location.

Indirect addressing is the first real difference from the usual way of dealing with variables in higher level languages. Only C and some versions of Pascal that use pointers provide ways of accessing data similar to this mode.

Indirect addressing is represented by parentheses. Parentheses say "use the address inside us to get to the location you want."

Form: (*An*)

Let's review some code we looked at before to calculate an address:

```
MOVE.L D0,D1   ;use D1 as a temporary work register
ASL.L  #1,D1   ;multiply D1 by 2 by shifting left one position
MOVE.L A0,A1   ;now the table's start location is in A1
ADD.L  D1,A1   ;finally, the location of the required element is in A1
```

and let's add another statement to actually retrieve the data at that location:

```
MOVE (A1),D2 ;now the required element is in register D2
```

This statement takes the two bytes pointed at by A1 and places them in register D2. The reason MOVE operates on two bytes is that the default is move a word (two bytes). Therefore the above statement is equivalent to:

```
MOVE.W (A1),D2 ;now the required element is in register D2
```

Typical uses of indirect addressing are for setting pointers to data, subscripting, linking together chunks of data, tables of subroutines (roughly equivalent to the ON ... GOSUB or ON...GOTO in BASIC or the CASE statements of Pascal or C.) All of these uses involve calculating addresses because the absolute address you will use is not known when the program is first written.

7. Register Indirect with Displacement Mode

The *register indirect with displacement mode* of addressing is just like register indirect only with a slight twist: you may add or subtract a constant from the calculated address. This constant is called the displacement. The most typical use is in a series of fixed-length records with each field in the record having a fixed offset from the start. The address register in this case usually points to the start of the record.

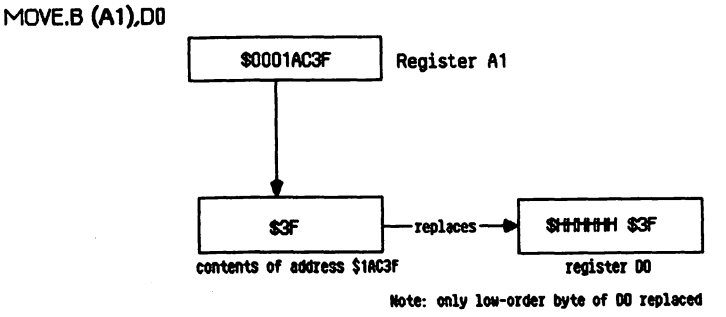


Figure 2-7 Address Register Indirect Mode

Form: *displacement*(An)

Calculate the start of the record in memory and place the result into an address register. Then, use the fixed offset from the start of the record for the displacement. It is usually best to use labels rather than numbers for the offsets so that the program is easier to read. The offset can be in the range plus or minus 32K. That is, a 2-byte two's complement number that is sign-extended.

Here is an example of an employee record with seven fields of different lengths:

```
                ;the following definitions tell how far each field is from the start of the
                ;record
emplynum    equ 0   ;the employee number (labels must be 8 or less bytes...)
emplynam    equ 4   ;the employee's name (up to 40 characters allowed)
title       equ 44  ;the job title of the employee (up to 16 characters allowed)
salary      equ 60  ;what the salary is in cents/month (4 byte hex number)
sickleav    equ 64  ;count of number of sick leave sub records (how many illnesses)
sickrec     equ 66  ;start of sick leave records, 2 bytes per record (up to 7 records)
empleng     equ 80  ;length of record
```

This record consists of bytes 0 through 3 being the employee number, bytes 4 through 43 being the employee name, bytes 44 through 59 being the title of the employee, bytes 60 through 63 being the salary of the employee in cents, and bytes 64 and 65 being the count of the number of sick leaves taken by the employee. Bytes 66 and 67 are the zeroth time the employee took sick leave, bytes 68 and 69 are for the first time, bytes 70 and 71 are for the second, etc. (In assembler you often don't start with the first entry but with the "zeroth" entry for convenience of indexing.)

Assume that there is a series of these records. Let's say we wanted to look at the employee number, employee name, and salary for an employee whose number is in register D0. Assume the start of the series of records is in address register A0. The code would look like this:

```
                                        ;code to find the D0th employee
                                        ;
                                        ;start address of D0th record=start of
                                        ;all records + D0 * length of record
FINDEMPL  MOVE.W  #empleng, D1          ;move employee length to D1
          MULU    D0,D1                 ;multiply employee index in D0 by length
                                        ;in D1
          MOVE.L  A0,A1                 ;put start of table into A1
          ADD.L   D1,A1                 ;now A1 holds the start address for the
                                        ;D0th record
          MOVE.L  emplynum(A1),D2       ;D2 receives the employee number (Long =
                                        ;4 bytes)
          LEA     emplynam(A1),A2       ;now A2 has the address of the employee
                                        ;name
          MOVE.L  salary(A1),D3         ;now the salary in cents is in D3
```

The LEA (Load Effective Address) instruction calculates the address that the source field is pointing to and puts that address into the destination address register. Think about that for a second. The LEA instruction doesn't affect the data that is at the location—it is used to find the address of that data.

The address of the employee name has been moved to A2. Usually the above code would be preparation for coding a loop to move the employee name to the receiving area. The loop to move the employee name to an area pointed to by A3 would be:

```
MOVELOOP  MOVE    #title-emplynam,D2    ;move the length of employee name
                                        ;to D2
MOVER     MOVE.B  (A2), (A3)
                                        ;move the byte pointed to by A2 to the
                                        ;place pointed by A3
          ADDQ    #1,A2                 ;add 1 to register A2
          ADDQ    #1,A3                 ;add 1 to register A3
          SUBQ    #1,D2                 ;subtract 1 from D2
          BNE     MOVER                 ;branch if D2 is not zero
```

The MOVER subroutine shows how to move the employee name, pointed by A2, to another location, pointed by A3. One byte at a time is moved by the MOVE.B instruction from the current location of A2 to the current location of A3; the address in A2 is incremented to point to the next byte of the string to move and A3 is incremented to point to the next location where the byte is to be moved. By using different addressing modes along with the DBRA instruction, both of which you will learn later, you can reduce this loop to two instructions!

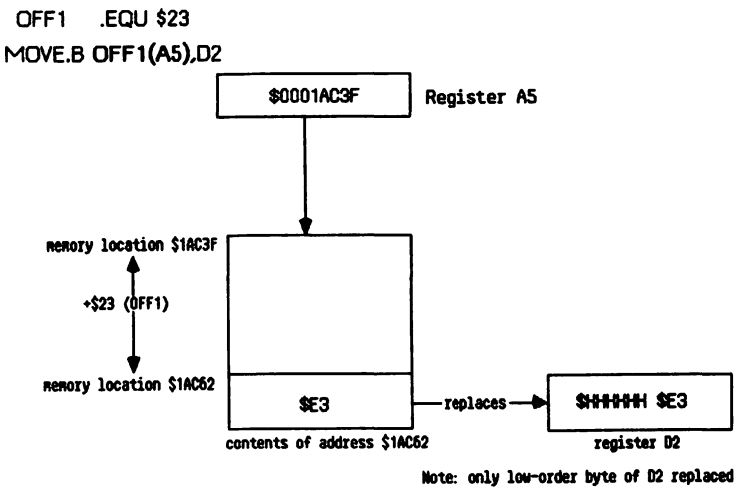


Figure 2-8 Address Register Indirect with Displacement Mode

8. Register Indirect with Index and Displacement Mode

Now that you understand register indirect with displacement let's throw in one more variable, an index.

Form: *displacement*(An,Dn) **or** *displacement*(An,Am)

The address used in the sum of the address register, the data or address (index) register, and the displacement. Since the displacement is one byte it can only range from −128 to +127. Therefore you cannot use this mode on record segments that are longer than 128 bytes.

Typically, this mode is used for an array of records. The example which we used above for register indirect with displacement could have been done using this mode as follows:

```
                                        ;code to find the D0th employee
                                        ;
                                        ;start address of D0th record = start of
                                        ;all records + D0 * length of a record
FINDEMPL  MOVE.W   #empleng,D1          ;move employee length to D1
          MULU     D0,D1                ;multiply employee index in D0 by length
                                        ;in D1
          MOVE.L   A0,A1                ;put start of table into A1
          MOVE.L   emplynum(A1,D1),D2   ;D2 receives the employee number
          LEA      emplynam(A1,D1),A2   ;now A2 has the address of the employee
                                        ;name
          MOVE.L   salary(A1,D1),D3     ;now the salary in cents is in D3
```

The last three lines of FINDEMPL show examples of the *register indirect with index and displacement mode* as it is used to index into a multi-record table. The first line adds the location of the start of the table, A1, to the offset of the pertinent record within that table, D1, and to that is added the offset of the employee number within the record. The other two lines are parallel in construction.

Notice that the statement ADD D1,A1 from the example in the Register Indirect with Displacement mode section has been removed since the addition is automatically done by this addressing mode. Carefully compare this series of code with the series of code for the register indirect with displacement mode. We could have changed the code to move the employee name so that it reads as follows:

```
MOVELOOP    MOVE     #title-emplynam,D2        ;move the length of employee name to
                                               ;D2
MOVER       MOVE.B   -1(A2,D2), -1(A3,D2)      ;so the BNE works use -1
                                               ;move the byte pointed to by A2 indexed
                                               ;by D2 to the place pointed to by A3
                                               ;indexed
            SUBQ     #1,D2                     ;subtract 1 from D2
            BNE      MOVER                     ;branch if D2 is not zero
```

This code would move the data starting with the last byte of the employee name and going forward through the name to the first byte. The index for the bytes goes from 40 down to 1 but we really want the index to go from 39 to 0. Therefore we use a displacement of −1 to compensate.

Sometimes you want to have a mode that consists of an address register indexed by a data register with no displacement. There is no such mode. Fortunately, all you have to do is use a zero displacement with this mode:

0(A*n*,D*m*) where *n,m* are numbers between 0 and 7

Be careful, the index defaults to a word length. If you have a value outside the range +/−32K inside D*m* then you must enter

displacement(An,Dm.L)

to make sure that all 4 bytes of D*m* are added into the final value.

Rarely will you have an index that is an address register. About the only time to do this is when you have run out of data registers and you have a spare address register. Even then, it probably isn't a good idea. We mention this possibility for completeness since the 68000 allows you to do this.

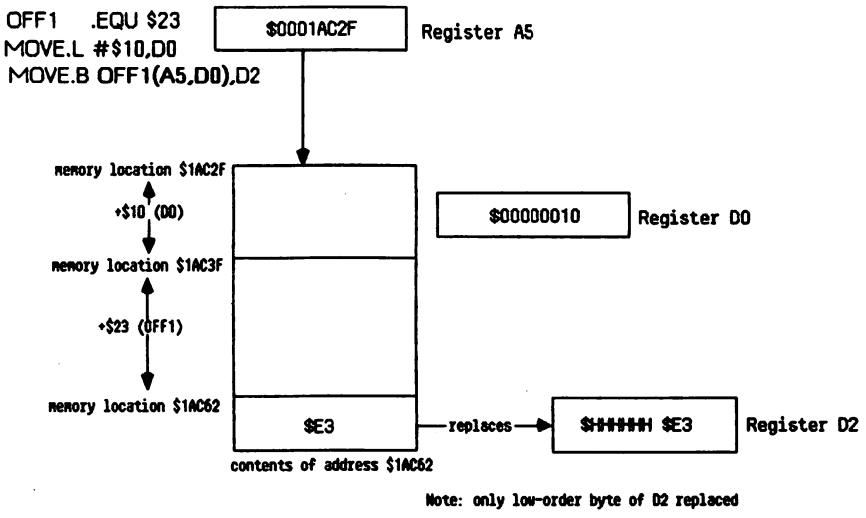


Figure 2-9 Address Register Indirect with Index and Displacement Mode

9. Postincrement Register Indirect Mode

Postincrement register indirect mode is exactly like the address register indirect mode except the address register is updated to point to the next element after the operation is done. The main use of this mode is when you have a sequence of data in memory and you need to operate on each element in sequence. This mode lets you avoid using extra add or increment instructions which are necessary in some other assembly languages to get to the next element.

If the instruction is a byte instruction, the address register is updated by adding one after the instruction is done. If the instruction is a word instruction, the address register is updated by adding two after the instruction is done. The address register is updated by four after a long instruction.

Form: (An)+ where the "+" following is a mnemonic for post increment

Doing our MOVELOOP once again in yet another way:

```
MOVELOOP   MOVE     #title-emplynam,D2    ;move the length of employee name to
                                          ;D2
MOVER      MOVE.B   (A2)+,(A3)+           ;move what is at A2 to A3 and add 1
                                          ;when done
           SUBQ     #1,D2                 ;subtract 1 from D2
           BNE,     MOVER                 ;branch if D2 is not zero
```

This loop moves a byte at a time from the address pointed by A2 to the location pointed by A3. The complete MOVELOOP subroutine moves the employee name, pointed at by A2, to an area pointed at by A3. Since MOVE.B is a byte instruction, one is added to A2 and one is added to A3 immediately after the MOVE.B instruction is performed. Caution: One consequence of doing the loop this way is that the registers A2 and A3 have their original values destroyed.

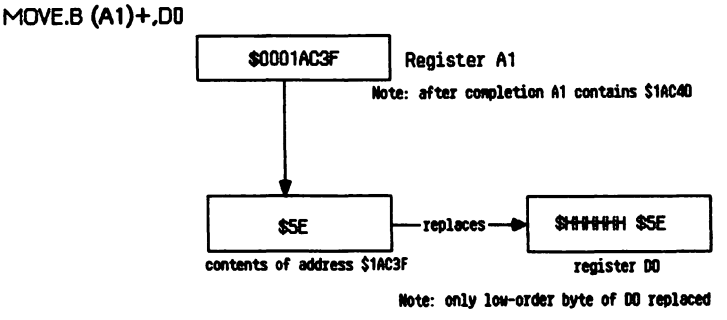


Figure 2-10 Address Register Indirect with Postincrement Mode

10. Predecrement Register Indirect Mode

The *predecrement register indirect mode* is the complement of the postincrement register indirect mode. Where postincrement *adds* to an address register after the instruction is executed, predecrement *subtracts* from the address register before the instruction is executed. Outside of this difference, all the other rules are the same.

The amount decremented is one for a byte instruction, two for a word instruction, and four for a long instruction. Instead of having a plus after the address register in parenthesis there is a minus before them.

Form: −(*An*)

Once again, we could have done our loop to move the employee name as follows:

```
MOVELOOP MOVE.L  #title-emplynam,D2    ;move the length of employee name to
                                       ;D2
         ADD.L   D2,A2                 ;point at the end of the source string
         ADD.L   D2,A3                 ;point at the end of the destination string
MOVER    MOVE.B  -(A2),-(A3)           ;move what is at A2 to A3 after
                                       ;subtracting 1
         SUBQ    #1,D2                 ;subtract 1 from D2
         BNE     MOVER                 ;branch if D2 is not zero
```

However, this way of moving bytes would be rare unless you were moving overlapping fields to the right. Usually you would use the post-increment mode.

Pushing onto the stack is usually done with the predecrement mode using A7, the stack pointer. Pulling things off of the stack is usually done with the postincrement mode. For example, if you wanted to push 4 bytes onto the stack:

```
MOVE.L MYVAR,-(A7)
```

or if you wanted to pull 4 bytes off the stack:

```
MOVE.L (A7)+,MYVAR
```

As you might have gathered, the stack's end location is pointed to by A7 and the stack goes from high memory to low memory. It is a good idea to draw a picture of the stack and watch what each of these instructions does:

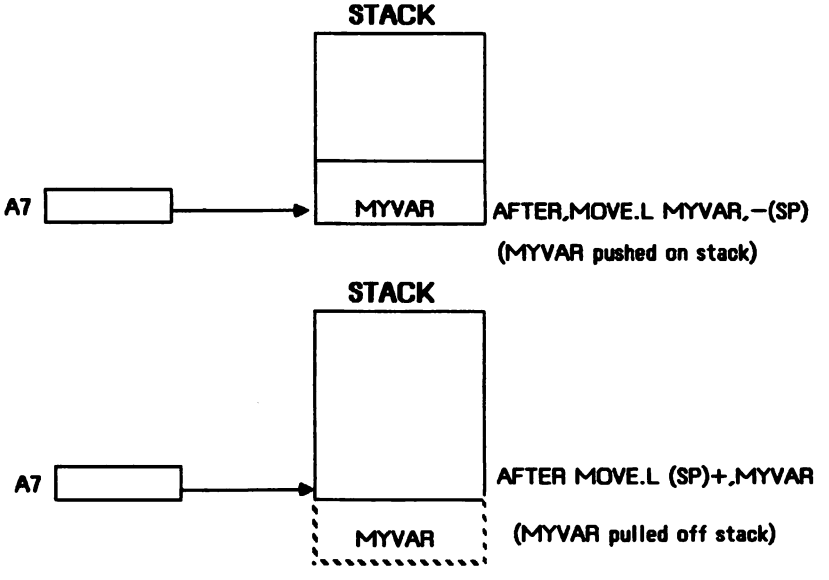


Figure 2-11 Picture of Stack Being Pushed, Popped

11. Program Counter Relative with Displacement Mode

The 68000 assembler is built so code can be moved anywhere in memory and the addressing of code and data still works. The *program counter relative with displacement mode* of addressing was included in the instruction set to help accomplish this. The idea is that your data can be

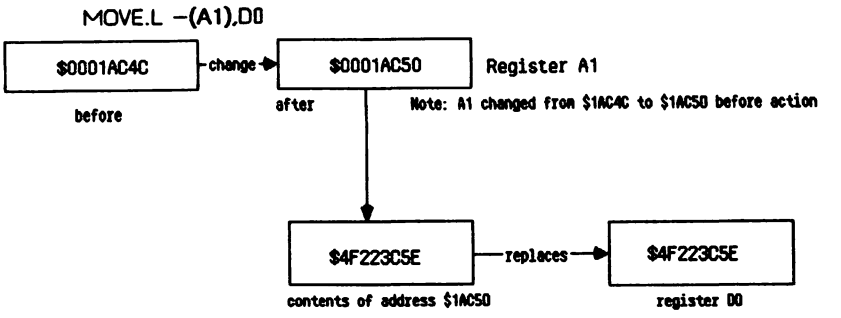


Figure 2-12 Address Register Indirect with Predecrement Mode

part of the program. When the program is moved to a new memory location, the references to your data are also changed.

Form: *displacement* (PC)
or simply
<label>
in assembler, where *<label>* refers to a location inside the program.

This mode really says "the data is *n* bytes backward/forward from where I am now," where the *n* bytes is the displacement, the backward/forward is the sign of the displacement, and the "where I am now" is the program counter of the data (not the op code). The program counter is used as if it were an address register for indirection. Obviously if the code is moved elsewhere, the program counter will reflect the new location and the data will still be the same displacement relative to where the program counter is then.

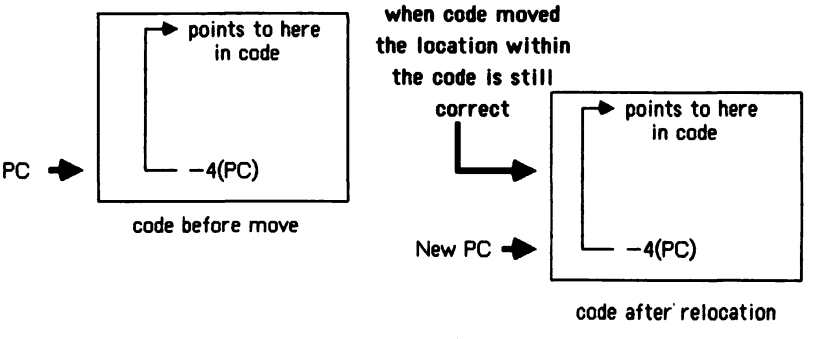


Figure 2-13 Program Counter Relative Still Operates When Code Is Moved to New Location

The important point is that all of this is transparent to you since the assembler figures out that the data is part of your code and generates this mode automatically. Let's say you have the following chunk of code:

```
MYVAR    DC 10 ;reserve 10 bytes of storage
YOURVAR DC 4   ;reserve 4 bytes of storage
;
;program segment
;
PROGSEG MOVE MYVAR,DO
;this will generate MOVE -16(PC),DO since MYVAR is 16 bytes back of the
;present loc.
        MOVE THISVAR,D1
;this generates MOVE +4(PC),D1 since THISVAR is 4 bytes forward in the code
        RTS
THISVAR DS 2
```

As you can see, you simply code your variables as if they were normal absolute-style addresses and the assembler notices that they are part of the code and turns them into Program Counter ("PC" for short) relative addressing.

Since the Macintosh can move your program anywhere in memory at just about any time, this is the main form of addressing you would use for constants. The other form of addressing that you would use, which is even more independent, is using data on the stack.

You can think of this mode as a special form of register indirect with displacement since we just use PC instead of *An*. However, in practice, this mode is used for constants in the program while the register indirect with displacement mode is usually used for external data structures.

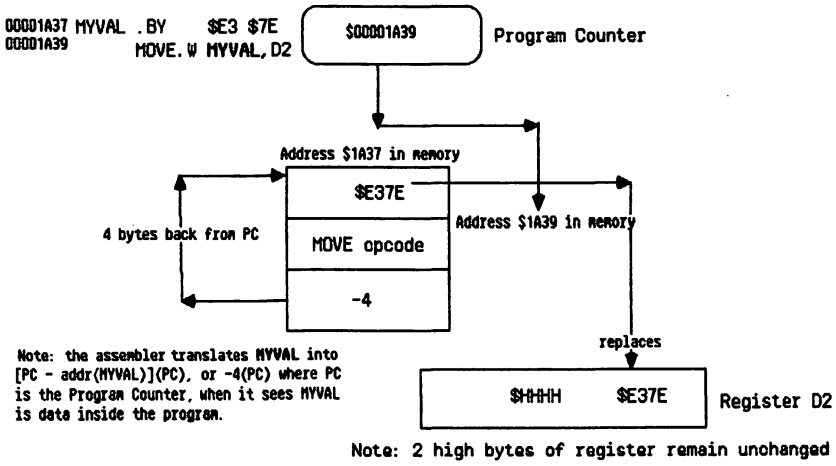


Figure 2-14 Program Counter Relative with Displacement Mode

12. Program Counter Relative with Index and Displacement Mode

Program counter relative with index and displacement mode is a special case of register indirect with index and displacement.
Form: *displacement*(PC,D*n*) or *displacement*(PC,A*n*)
although it is usually written as:
label(D*n*) or *label*(A*n*)
when you are in your assembler.

This form is exactly like program counter relative with displacement, only an address or data register is added in as well. You would use this mode of addressing if you had a constant table imbedded in your program. Then, the index register would give the offset from the start of the table to the start of the record you wanted. The displacement would give the offset within the record of that field.

Let's use the employee record from above as an example:

```
emplynum equ 0  ;the employee number (labels must be 8 or less bytes...)
emplynam equ 4  ;the employee's name (up to 40 characters allowed)
title    equ 44 ;the job title of the employee (up to 16 characters allowed)
salary   equ 60 ;what the salary is in cents/month (4 byte hex number)
sickleav equ 64 ;count of number of sick leave sub records (how many illnesses)
sickrec  equ 66 ;start of sick leave records, 2 bytes per record (up to 7 records)
empleng  equ 80 ;length of record
;
emptable dcb.b empleng*20              ;reserve enough room for 20 records
                                       ;inside the program
empnm    dc  'John Jones'
emptti   dc  'janitor  '
                                       ;
                                       ;insert data for the second employee

CALC2ND  MOVE   #empleng,D0
         MULU   #2,D0                  ;multiply 2 times employee record length
                                       ;(80)
PUTDATA  LEA    emptable + emplynum (D0),A0
         MOVE.L #011750,emplynum, (A0) ;use D0 to index to the second employee
                                       ;record
         MOVE   #39,D1                 ;count of characters in employee name
                                       ;- 1
         LEA    emplynam(D0),A0        ;get address of employee name in table
         LEA    empnm,A1               ;get address of 'John Jones'
```

```
EMPNLOOP MOVE.B. (A1)+,(A0)+        ;move a byte and add to addresses
         DBRA    D1,EMPNLOOP        ;keep looping and decrementing D1 until
                                    ;D1 < 0
                                    ;the above loop shows how a whole
                                    ;string can be moved in 2 instructions
                                    ;DBRA means Decrement and BRAnch—
                                    ;more on this later
                                    ;and so forth for the rest of the table...
```

The above program shows how this addressing mode is used. The employee number and employee name, which is inside the program itself, are addressed by adding the offset of emplynum, PC, and D0 (which contains the offset from the start of emptable to the second record). This mode has its most typical use when you have a short table interspersed with code.

Warning

When your first learn 68000 assembler, the most common use of this form is in syntax errors which the assembler lets slide through without a warning! Usually, you allocate area for a record by using equates. However, if you accidentally use define constant, then this mode is generated and the program counter is added into your address, unbeknownst to you. You think you are using register indirect with displacement rather than this mode. This problem disappears after you firmly understand the different ways to set up record areas.

The Effective Address

We've completed our discussion of addressing modes, but we should explain about the effective address since understanding this concept is necessary before you can learn more about 68000 assembler. The effective address is the *actual physical address* that an addressing mode accesses. You may think of this address as the location where "effects" take place if it helps you to remember this. Sometimes, this address is the physical address that the data is coming from as well as the physical address that the data is going to.

Let's say you wanted the effective address for $56(A0). If A0 contains $00001200, the effective address of $56(A0) is $00001200 plus $56 or

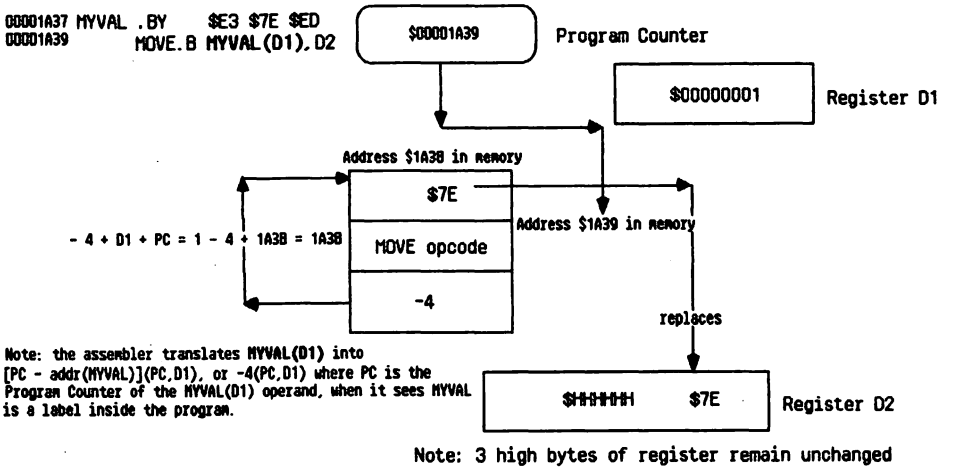


Figure 2-15 Program Counter Relative with Index & Displacement Mode

$00001256. You can use the LEA instruction to calculate an effective address and place the result of the calculation in an address register. Please note that all the other statements in 68000 deal with the *data* at the effective address while the LEA instruction moves the *value of the effective address itself* to the destination. If you had entered MOVE.L $56(A0),A1 and location $1256 through $1259 held $0001B3AC then A1 would contain $0001B3AC while, in contrast, LEA $56(A0),A1 places $1256 into A1. If you understand this distinction it will help you immeasurably in understanding 68000 assembler.

Example

Here is another example. If A3 holds $00050000 and D1 holds $230 then the effective address of −1(A3,D1) is $00050000 plus 230 plus −1 so the effective address is $0005022F. The instruction LEA −1(A3,D1),A4 would result in A4 holding $5022F.

It might interest you to know that there is also a PEA instruction which pushes the 4-byte effective address onto the stack. Using the above example:

```
PEA  -1(A3,D1)
```

would push $0005022F onto the stack and update the stack pointer accordingly.

Summary

We've now introduced you to the important addressing modes used when programming the 68000 in assembly language. Although these names may seem complicated, their functions allow you to access data in many different ways. Some addressing modes you'll use more than others, but you'll want to be familiar with all of them so that you fully understand what your programming options are. Next, let's take a look at the 68000 instruction set, the real building blocks of assembly language.

CHAPTER

The 68000 Instruction Set

In this chapter we describe the 68000 instruction set in detail. The instruction set represents the building blocks from which all assembly language programs are made. It is assumed throughout this chapter that you understand the addressing modes well. If you are unsure, go back to Chapter 2 and review them.

Fortunately, to start programming you really only need to know a few of the instructions intimately. The instructions are listed in this chapter in the order of their importance as determined by frequency of use. If you learn the first 25 instructions we mention here you are in a position to write adequate code—in fact 98% of the code you write will use only those first 25 instructions.

It is a good idea to learn all the op codes in the next section, Most Frequently Used Instructions, and at least have a nodding acquaintance with the rest of the instructions in this chapter. You can skim the System Control Operations, going over them just enough to understand when to use them and where to find them when you need them.

Most Frequently Used Instructions

You will use the following instructions whenever you code in 68000 assembler. You must understand completely what each of them does. The instructions are ordered according to frequency of use and, hence, the importance of each command. The most important op codes come first.

MOVE

MOVE is probably the most often used instruction in 68000 assembler. This one powerful instruction can get data from any place and move it any

place else. Data can be located in an area in memory, a data or address register, the condition codes (part of the status register), the stack, the status register, or the user stack pointer.

You can move one, two, or four bytes at a time by appending a ".B," ".W," or ".L" to a MOVE. That is, MOVE.B means move a byte, MOVE.W means move a word (2 bytes), and MOVE.L means move a long word (4 bytes). All addressing modes can be used to specify the source of the data, and all except for address register direct, program counter relative, program counter indexed, and immediate mode are allowed in specifying the destination of a MOVE instruction. You can use address register direct in the destination if you do a MOVEA. Fortunately, the assembler is smart enough to look at the destination and change a MOVE to a MOVEA automatically.

Some examples of MOVE follow:

```
MOVE A0,D0
```

Since no appendix is put on the opcode, ".W" or a 2-byte move is assumed. Please be careful—many bugs are caused when you mean to move all 4 bytes of an address or data register but forget to append the ".L" and hence only 2 bytes get moved!

```
MOVE.L D0,D1
```

Move all four bytes of data register D0 to data register D1.

```
MOVE.B (A0)+,(A1)+
```

This construct is often used when a series of bytes (often a string) are being moved from one area of memory to another. Two pointers, registers A0 and A1, are used to point into the source and destination string respectively. One byte is moved and then the two pointers are each updated to point to the next character in the string. This construct is most often used with a DB*cc*. Here *cc* stands for Condition Code; for example EQ is equal (DBEQ), GT is greater than (DBGT).) This construct is also often used with a Decrement and Branch instruction to form a loop as follows:

```
LOOP MOVE.B (A0)+,(A1)+   ;move a byte from where A0 points to where A1 points
     DBEQ D0,LOOP         ;keep looping for D0 bytes or until a zero byte is hit
```

This LOOP moves D0 plus one bytes from the location pointed by A0 to the location pointed by A1.

```
MOVE.L D3,-(A7)      ;this would push the 4 bytes of D3 on the stack
MOVE.L (A7)+,D3      ;and this would pop them off the stack and place
                     ;them back in D3
```

These two instructions show how to use MOVE to push or pop data from the stack.

```
MOVE.W #$2A4E,(A0)+
```

This would place the two bytes 2A and 4E into the location pointed by A0 and the byte after that. When this instruction is done A0 would point to the byte right after the 4E above.

The condition codes are set by the MOVE instruction. We used this fact implicitly in the loop code above. Unfortunately, the MOVEA instruction *does not* set the condition codes. So the following sequence will not work the way you expect:

```
MOVE.L D0,A2
BEQ    LABEL  ;BEQ stands for Branch on EQual, in other words branch if Zero Condition
              ;Code is set
```

Whether the branch to label is taken is completely independent of whether D0 contains zero! This is because the MOVE.L is translated to MOVEA.L and the condition codes are not set. For future reference, the following will do what you want:

```
MOVE.L D0,A2
TST.L  D0
BEQ    LABEL
```

Since you will rarely use the MOVE to set condition codes, status register, or user stack pointer, we will simply give the syntax here without examples:

MOVE to condition codes:

MOVE *source*,CCR ;*source* is any addressing mode except A*n*

MOVE to status register:

MOVE *source*,SR ;*source* is any addressing mode except A*n*

MOVE from status register:

MOVE SR,*dest*. ; *dest* is any addressing mode except A*n*, immediate, or program counter relative since these addressing modes may not be modified.

MOVE to/from the user stack pointer:

MOVE USP,A*n*, or MOVEA*n*,USP ;the user stack pointer must go to or come from an address register.

On the Macintosh the system stack pointer is the one used, the user stack pointer is not used for anything! So USP has no effect. Do not use USP with the Macintosh assembler.

MOVEA we already know. This is where the destination is an address register. The source may be a word, ".W," or a long, ".L," but not a byte, ".B."

So MOVEA.W or MOVEA.L are legal but MOVEA.B is illegal. If the word form is used the word is extended to 4 bytes before it is moved—the high two bytes of the destination are always clobbered. (Definition of "clobber"; to inadvertantly overlap a portion of memory or a register, sometimes creating a bug thereby.)

There is a MOVEP instruction that is used for data going to a port using memory mapped I/O to an 8-bit device. Since there are some 8-bit devices in the Macintosh, some system code may use these instructions—you will rarely have to. The form of a MOVEP is:

```
MOVEP Dm,d(An)
MOVEP d(Am),Dn
```

The four bytes in the data register are transferred to alternate bytes in memory with the high order byte being transferred first. This way all the bytes go out on either the high or low order bytes of the memory mapped data bus depending on whether the address is even or odd—high if even, low if odd.

Example:

```
MOVEP D0,0(A2)
```

where D0 holds $C809F74E and A2 holds $00013F44. Then $C8 would be moved to address $13F44, $09 would be moved to $13F46, $F7 would be moved to $13F48, and finally $4E would be moved to $13F4A.

An associated instruction is the MOVEM instruction. Although this instruction looks like a MOVE instruction, and it is used to move data, you will mainly use this instruction in one very specific circumstance. MOVEM is usually used to save the "environment" of a higher level routine while calling a lower level routine. The environment preserved is a list of registers.

Here is a typical situation that you will encounter when you are programming in assembler. You are about to write a subroutine. Every register is in use by the routines which will call this subroutine (or perhaps you have no idea which registers will be in use by the calling routines—any register you use could upset the higher level routines). You must therefore save off every register you plan to use upon entry to the routine and restore them when you leave. If your subroutine is large you could find yourself coding five or ten statements to push registers on the stack coming in and five or ten going out which pop registers off the stack. This

is clumsy. Even clumsier is reading each subroutine to see which registers it uses (and having your program go wrong when you miss one) or using only some of the registers available to you.

The answer to this probelm is the MOVEM instruction. When you write a subroutine you can push all the data and address registers you are using on the stack, and then restore them when you leave. By using this procedure you can use any register you want (making your subroutine more efficient) and yet not worry whether you will clobber what is in a register which another routine at a higher level is using.

The MOVEM instruction lists all the registers you want moved separated by slashes ('/'). To save a range of registers, such as D0 through D4, you would indicate the starting and ending registers with a dash ('-') in between—for D0 through D4 this would be D0-D4.

Let's say you wanted to use address registers A0,A1, and A2 and data registers D2 and D3. You would place the following instructions at the start and end of the code for a subroutine:

```
myroutin
        MOVEM.L A0-A2/D2/D3,-(A7) ;push the contents of the registers on the stack
                                  ;don't forget to make it movem.L or only 2
                                  ;bytes of the addresses will be saved!
        (more code)               ;lots of useful code
        MOVEM.L(A7)+,A0-A2/D2/D3
                                  ;restore all the registers from the stack
        RTS
```

Picture of MOVEM.L (SP)+,A0–A2/D2/D3
and MOVEM.L A0–A2/D2/D3,–(SP)
since the order of movement to the stack is D0–D7,A0–A7
and the order of movement from the stack is the reverse.

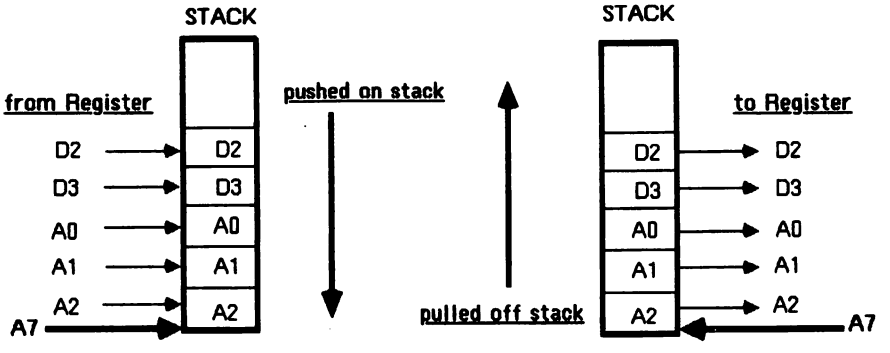


Figure 3-1 Move Multiple

Finally, there is a special hi-speed move if you want to move a constant value that is between 0 and $FF (0 and 255 in decimal) to a data register—the MOVEQ instruction. Again, the assembler is smart enough to change your MOVE to MOVEQ in appropriate circumstances. Although only the bottom 8 bits are actually contained in the instruction, all 32 bits are changed (so this instruction is always long). The number is sign extended. Here are some examples of MOVEQ.

```
MOVEQ   #1,D3       ;00000001 is moved to data register D3
MOVEQ   #'A',D0     ;00000041 is moved to data register D0 ('A' in ASCII is $41)
MOVEQ   #$FE,D2     ;FFFFFFFE is moved to D2 (sign extension since $FE is negative)
```

Usually you would not use MOVEQ #0,D*n* since there is an instruction, the CLR.L D*n* instruction, which clears a register to zero. MOVEQ #0,D0 and CLR.L D0 do the same thing, therefore.

Once you understand the addressing modes and understand the MOVE instruction you are free to move data wherever you want inside your Macintosh.

Branch Instructions (B*cc*)

There are fourteen different branch instructions in 68000 assembler. By combining a TST (TeST) or CMP (CoMPare) instruction, which sets the condition codes, with a branch you can do the equivalent of an IF instruction in a higher level language.

The branches operate based on a combination of the carry flag, the zero flag, the negative flag, and the overflow flag. Before you can understand the branch instructions you must understand these four flags. These flags are generated as follows by other instructions:

Set means the bit flag is 1, *cleared* means the bit flag is 0.

N (negative): If the result has the highest bit set (1) then the negative flag is set—if the result has the highest bit cleared (0) then the negative flag is cleared. Any negative number results in a two's complement number with the high bit set, hence the reason for calling this flag the negative flag.

Two examples of MOVE instructions setting the negative flag:

```
MOVEQ #$83,D0
```

Since #$83 has the high bit on, the negative flag is set. Note that with sign extension, $FFFFFF83 is moved to D0.

```
MOVE.L #-1,D3
```

Since −1 is the same as $FFFFFFFF in two's complement form, the negative flag is set after this instruction.

Z (zero): If the result is zero this flag is set—cleared if not zero. This flag is often tested after a subtraction of one number from another. If the result is zero the two numbers are equal, otherwise the two numbers are unequal.

Example:

```
CMP.W #$23,D0
```

CMP.W subtracts $0023 from the low 2 bytes in D0 and sets the condition codes accordingly. If D0 contains $0023 then the zero flag is set since the result of the subtraction is zero. If D0 contains something else, then the zero flag is cleared. SUB.W #$23,D0 where SUB is the subtract op code would set the zero flag in exactly the same way.

C (carry): If you remember your addition and subtraction exercises in school you will understand this flag. A carry is generated in decimal addition when a column sums to more than nine, and a borrow is necessary if the number subtracted is greater than the number from which you are subtracting in the same column. With the computer, if a carry is generated when two numbers are added together or a borrow is necessary when two numbers are subtracted, the carry flag is set. If no carry or borrow occurs, this flag is cleared. The only difference is that here the addition and subtraction is in hexadecimal and the carry is from byte to byte, word to word, or long word to long word. The carry is only used in unsigned arithmetic since a carry resulting from the addition of two signed numbers is an error (see the overflow flag described later). The carry flag is mostly used for decision making on the 68000; the extend flag (described later) is used for arithmetic carry. For this reason, the carry flag is somewhat of a misnomer on the 68000.

Example:

```
ADD.B #$23,D5
```

With this line of code any number greater than $DC in the low byte of D5 will create a carry. Conceptually, the addition of a number greater than $DC to $23 results in a number greater than or equal to $100—i.e., a 1 is carried to the next byte. The carry is out of the byte in an ADD.B, out of the word in an ADD.W, and out of the register (4 bytes) in an ADD.L.

The carry flag is also used by the compare instructions to signal a carry in the subtraction done by those instructions.

X (eXtend): In the 68000 there is also a special carry flag called the eXtend or X flag. The extend flag is always set the same way the carry flag is set—however, sometimes the carry flag is modified in circumstances such as MOVE and CMP when the extend flag is not modified. Since the extend flag is only used in arithmetic operations we describe it in more detail in

that section. As mentioned before, in arithmetic on the 68000 the extend flag is used for carrying, not the carry flag.

V (overflow): This is the hardest to understand of the flags. This occurs when there is an arithmetic overflow. This means that you have gotten an answer that doesn't make sense in terms of signed arithmetic or that data is lost in a division. When you have an overflow the arithmetic result is too large for the receiving field and therefore information is lost and the data is misrepresented.

Example:

```
ADD.B #$80,D0
```

Suppose D0 is also $80. $80 is the same as decimal −128. When you add −128 to −128 in normal arithmetic the result is −256; in hexadecimal when you add $80 to $80 the result is $100 which means zero with a carry of one. Therefore, in this case the result is zero, not −256! The overflow flag is set to indicate that you can't represent the answer in one byte.

```
ADD.B #$41,D0
```

Suppose D0 holds $50. In decimal this would be like adding 65 and 80, the result is 145. However, in this case the result is $91 which is −111! The overflow flag is set to indicate that the answer we expected, 145, can't be represented in one byte which can only represent values between −128 and +127.

Another way of expressing overflow is to see that in signed arithmetic the sign comes out "wrong." In the first example we added two negative numbers and came out with a positive number. In the second example we added two positive numbers and came out with a negative number. We have overflowed the ability to represent this number within the confines of a signed number having the same number of bytes as the two signed input numbers.

Division operations also can result in overflow resulting in the overflow flag being set. In this case the overflow is more straight forward, the result of dividing a 2-byte number into a 4-byte number cannot be held in two bytes.

With this understanding of the flags you can now go forward to understanding how branches work.

1) BCC—Branch on Carry Clear. If the carry flag is clear, this branch is taken. There are two places where this instruction is typically used. One is in an addition/subtraction of many digits—the other is after a CoMPare instruction (CMP). In a multidigit addition this instruction would be used to skip an addition of one to the next digit representing a carry. After a CMP instruction this branch is taken if the destination (the second operand) is

greater than or equal to the source (the first operand) when both operands are seen as unsigned numbers.

For those interested in grisly details, let's look at an example. Suppose you subtract $20 from D0 resulting in a borrow if $20 is greater than D0—a borrow is represented by carry set in 68000 assembler. So if D0 is greater than or equal to $20, there is no borrow and the carry is clear.

```
CMP.B #$20,D0    ;set the flags as if you had subtracted $20 from D0.
BCC LABEL1
```

If D0 contains $20 through $FF, the BCC LABEL1 branch is taken and you will be at LABEL1 in the code next. If D0 is $0 through $1F, you will be positioned at the next instruction after the BCC instead.

Important Note: In everything you do in 68000 assembler the source is the leftmost instruction and the destination is the rightmost instruction. When you SUBtract two numbers you subtract the number to the left from the number to the right. However, when you CoMPare and BRAnch, you are comparing the number to the RIGHT (the supposed destination) to the number to the LEFT (the supposed source). Hence CMP #$10,D3 followed by BLE MYLABEL, where BLE is the mnemonic for Branch Less then or Equal, means branch if the operand to the RIGHT (D3) is less than or equal to the operand to the left (#$10)! Many references don't mention this and it came as a bit of a shock when one of us debugged his first program.

2) BCS—Branch on Carry Set. This is the reverse of BCC; if the carry flag is set the branch is taken. Since the carry set indicates a carry in addition (or a borrow in subtraction) this instruction is sometimes used for arithmetic operations. The more usual situation is that you are comparing two operands using the compare op code (CMP) and you want the second operand to be less than the first operand when both are viewed as unsigned numbers.

Example:

```
CMP.W    $1234,D5      ;compare $1234 to the low-order
                       ;(rightmost) 2 bytes of D5
BCS LABEL1             ;if this word in D5 is<$1234 then
                       ;branch to LABEL1.
```

3) BEQ—Branch on Equal to Zero. This instruction means branch if the zero flag is set. However, many think of it as branch if the preceding

operation resulted in a zero. After a compare instruction (CMP) this means that the result of subtracting the two numbers was zero. Hence, branch if these two operands are equal.

```
        CMP.L    D0,D1          ;"compare" all 4 bytes of D0 and D1
        BEQ      D0EQUD1        ;branch if D0 is exactly equal to D1


LOOPDELP CMPM.B  (A0)+,(A1)+    ;compare a byte from where A0 is
                                ;pointing to where A1
                                ;is pointing—bump both pointers by 1
                                ;afterwards.
        BEQ      LOOPDELP       ;if the bytes you just compared were the
                                ;same, keep on trucking (branch back to
                                ;LOOPDELP).
```

The above chunk of code will compare two strings and exit when they reach a point where one is not equal to the other.

4) BNE—Branch Not Equal. This instruction means if the zero flag is cleared, this branch is taken. When the zero flag is cleared, the result of a prior operation was not equal to zero. Typically used in loops such as the following.

```
LOOPY   MOVE.B   (A2)+,(A0)+    ;move a byte from address A2 to
                                ;address A0
                                ;bump both pointers after the move.
        BNE      LOOPY          ;if the byte moved was not a zero,
                                ;branch back to
                                ;LOOPY (the MOVE command sets the
                                ;zero flag).
```

This code is used to move a string. The string is terminated by a byte of zero.

5) BHI—Branch on HIgh. If the carry flag is clear and the zero flag is clear, the branch is taken. This instruction is a compounding of branch on carry clear and branch not equal. In other words, when both operands are viewed as unsigned numbers, the second operand is strictly greater than the first operand (by strictly greater again we mean that equal doesn't count).

```
        MOVE.W   #-4,D0         ;move-4 ($FFFC) to the low 2 bytes of
                                ;register D0
LOOPER  ADD.W    #4,D0          ;add 1 to the low 2 bytes of D0
```

```
        MOVE.L   10(A0,D0.W),5(A3,D0.W)   ;move 4 bytes from location A0+10
                                          ;indexed by D0
                                          ;to location A3+5 also indexed by D0
        CMP.W    #40,D0                   ;set the flags as though you had
                                          ;subtracted $40 from D0
        BHI      EXIT                     ;if D0 > $40 then branch to EXIT in
                                          ;code
        BRA      LOOPER                   ;if D0 <= $40 gets here and then
                                          ;always branches back to LOOPER
EXIT    etc.                              ;a label to exit the routine
```

In the above code BHI EXIT checks to see if the loop needs to be exited when D0 becomes strictly greater than $40. The condition codes which determine this are set by the CMP.W which precede the BHI EXIT instruction.

6) BLS—Branch on Less or Same. With this instruction if the carry is set or the zero is set, this branch is taken. Put more simply, if the second operand is less than or the same as the first operand when viewed as an unsigned number, take the branch. This instruction is the opposite of BHI.

We could code the above example better by using BLS:

```
        CLR.W    D0                       ;move zero to the low 2 bytes of register
                                          ;D0
LOOPER  MOVE.L   10(A0,D0.W),5(A3,D0.W)   ;move 4 bytes from location A0+10
                                          ;indexed by D0
                                          ;to location A3+5 also indexed by D0
        ADD.W.   #4,D0                    ;add 4 to the low 2 bytes of D0
        CMP.W    #$40,D0                  ;set the flags as though
                                          ;you had subtracted $40 from D0
        BLS      LOOPER                   ;if D0 <= $40 then
                                          ;branches back to LOOPER
EXIT    etc.                              ;a label used when the routine exits
```

In this simplified routine, the BLS to LOOPER keeps taking place while D0 <= $40. When Do > $40 the branch is not taken back to LOOPER and the routine exits by continuing to the next statement, labeled EXIT.

7) BGT—Branch Greater Than. This instruction branches if the second operand is greater than the first operand, where both operands are viewed as signed numbers. Remember that when using unsigned numbers, the 2-byte numbers run from $0 to $FFFF (65535 in decimal) while in signed numbers they run from $8000 (−32768) through $FFFF (−1) and thence

through $0000 (0) and then up to $7FFF (+32767). When using the BGT therefore $FFFF is less than $0000 which in turn is less than $12A6. In contrast BHI performs the same function, but for unsigned numbers.

A typical use of BGT would be to compare two numbers in a floating point routine:

```
            MOVE.B   EXPON1(A5),D0   ;move 2 signed exponents to D0 and D1,
                                     ;respectively
            MOVE.B   EXPON2(A5),D1   ;each exponent takes up one byte (-128
                                     ;thru +127)
EXPONCMP    CMP.B    D0,D1           ;compare the exponents
            BGT      XPN2GRTR        ;if exponent 2 is bigger, then branch to
                                     ;XPN2GRTR
```

8) BGE—Branch Greater Than or Equal. Same as BGT but the branch is also taken if the two values are equal.

9) BLT—Branch Less Than. If the second operand is less than the first operand when both are viewed as signed numbers then this branch is taken. See BGT above.

10) BLE—Branch Less Than or Equal. If the second operand is less than or equal to the first operand when both are seen as unsigned numbers, the branch is taken.

11) BMI—Branch on MInus. This branch is taken if the condition code for negative is set. Typically, the condition code of negative is set by a MOVE or a TST instruction.

```
            TST.W D0
            BMI DONEG    ;go to a routine to handle negative
                         ;numbers
                         ;drops through to here to handle a
                         ;positive number
```

12) BPL—Branch on Plus. If the number is greater than or equal to zero when viewed as a signed number, the negative flag is cleared; this branch is taken. The opposite of BMI. Below is an example of a function that turns a 4-byte number in D0 into its absolute value:

```
            TST.L    D0      ;see if D0 contains a positive number
            BPL      ISPOS   ;continue if it is already positive
            NEG.L    D0      ;D0 is turned into its two's complement
                             ;(made positive)
ISPOS                        ;continue, now D0 contains the absolute
                             ;value of what was there
```

The TST.L instruction tests the full four bytes of register D0 and sets the condition codes. If the negative condition code is cleared (meaning D0 is a positive number) then the BPL ISPOS branch is taken.

13) BVS—Branch if the oVerflow Flag is Set. If overflow has occurred in an arithmetic operation (see the start of this section on branching where the overflow flag is described) then take this branch.

14) BVC—Branch on oVerflow Clear. Branch if the overflow flag is clear. If an overflow in an arithmetic operation has not occurred, take this branch.

This completes the section on branching. The following table describes the exact combination of condition codes under which each branch is taken—it is included for reference only. We leave it as an exercise for you to verify that these combinations of condition code settings really result in the behavior you would expect.

Branch Instruction—Condition under Which Branch Is Taken
BCC—carry clear
BCS—carry set
BEQ—zero set
BNE—zero clear
BVS—overflow set
BVC—overflow clear
BMI—negative set
BPL—negative clear
BLS—carry set or zero set
BHI—carry clear and zero clear
BLT—negative set and overflow clear or negative clear and overflow set
BGE—negative set and overflow set or negative clear and overflow clear
BGT—negative set and overflow set and zero clear or negative clear and overflow clear and zero clear
BLE—zero set or negative set and overflow clear or negative clear and overflow set

Decrement and Branch (DB*cc*)

To simplify the coding of loops, the staple of computer programmers, the decrement and branch instruction is included in 68000 assembler. This loop works very much like a FOR-NEXT loop in BASIC, a FOR loop in Pascal or C, or a DO loop in FORTRAN. However, unlike these loops in higher level languages as normally used, the decrement and branch instruction starts at a number greater than zero and keeps counting down and looping until it either goes negative or has a certain condition met. In this way it would be roughly similar to the following Pascal code:

```
D0 :=constant;
repeat
   (* assorted code *)
   if not condition then
   D0 := D0 - 1;
   until (D0 = $FFFF) or (condition);
```

The form of the instruction is:

DB*cc* D*n*,<*label*> ;*cc* is any of the usual condition codes for branching

In the above code, condition can be any of the conditions which were described in the branch instructions. For example, DBEQ is decrement and branch until equal, and DBLE is decrement and branch until less than or equal. In addition there are two other conditions, False and True. Usually you don't say DBF (Decrement and Branch on False) but instead use DBRA which is read decrement and branch. This last instruction keeps branching until the D*n* register goes to −1 ($FFFF). These instructions are always .W or word. For this reason you can only have a loop that goes up to 32768 times around. Fortunately, this is sufficient most of the time. The following code searches through a string until it finds a space:

```
          CLR.W     D0              ;make D0's low word equal zero
          MOVE.B    (A0)+,D0        ;move the first byte of string (length) to
                                    ;D0
          BEQ       STRGZERO        ;if length of string is zero, skip
          SUBQ.W    #1,D0           ;subtract 1 from D0 so will correctly
                                    ;count and exit after processing last byte
                                    ;of string
FINDSPAC  CMP.B     #' ',(A0)+      ;A0 points to the next byte in the string
          DBEQ      D0,FINDSPAC     ;keep looping until the string exhausted
                                    ;when D0 = -1 or until a space is
                                    ;discovered
          BEQ       SPCFOUND        ;if the equal flag is set a space has been
                                    ;found
STRGZERO                            ;string empty, continue
                                    ;if execution of the code gets here then
                                    ;no space in string or string empty
          ...                       ;code to handle this case
SPCFOUND  SUB.L     #1,A0           ;now A0 points to the byte with the
                                    ;space
          ...                       ;code to handle this case
```

Another example which moves 13 bytes of the string pointed to by A0 to the string pointed to by A1 follows:

```
        MOVE.W  #12,D1          ;one less than 13 since DBRA stops at
                                ;-1, not 0
MOVER   MOVE.B  (A0)+,(A1)+     ;move a byte and point at the next byte
                                ;in source and destination strings
        DBRA    D1,MOVER        ;keep looping until D1 is -1 (at which
                                ;point 13 bytes moved)
```

Compare Instructions

We have already seen many instances of the CMP instruction in the examples of branching in the previous section. This is because CMP and branching go hand in hand.

CMP subtracts the source operand (the first operand) from the destination operand (the second operand) and sets the condition codes accordingly. The source operand can be any of the usual addressing modes, the destination operand must be a data register, D*n*. The destination operand is *not* affected by the subtraction—in this way it is different from the subtract operation, SUB, which does alter the destination. Here is how each of the flags are set:

The **Negative flag** is set if the result is negative, otherwise it is cleared (>=0).

The **Zero flag** is set if the result is zero and cleared if the result is non-zero.

The **Carry flag** is set if a borrow is generated and cleared if there is no borrow generated. A borrow is generated if the first operand is greater than the second number when both are viewed as unsigned numbers.

The **Overflow flag** is set if an overflow is generated and cleared if none is generated.

The **eXtend flag** is not affected.

One of the most common sequences in 68000 assembler is the following:

```
CMP.B # '+',(A0)    ;compare to see if a byte is a plus symbol
BNE   NOTAPLUS      ;branch if it is not a plus
```

After a CMP, Branch Not Equal (BNE) is taken if the source and destination are different. Actually, the instruction generated above would be a CMPI or CoMPare Immediate. CMPI is used if the source operand is immediate data (signified by a '#' symbol) and the destination operand is any of the normal addressing modes. CMPA is another compare instruction used when the destination operand is an address register rather than a data register, but it is otherwise similar to the CMP instruction. CMPA cannot be used when you want to compare a byte—CMPA.B is an illegal instruction. Fortunately, 68000 assemblers automatically change a CMP

into a CMPI or CMPA when these are appropriate so just type CMP in these cases. The condition codes are set the same by these three CMP instructions.

There is one more compare instruction, CMPM. CMPM stands for CoMPare Memory. The form of CMPM is:

CMPM (A*n*)+,(A*m*)+

It can operate on bytes, words, or long words. This compare instruction is used to compare two long series of bytes; it is often used with a DB*cc* instruction.

For example, to compare two strings to find whether they are equal the following subroutine could be used:

```
                                    ;the strings are pointed to by A2 and A4,
                                    ;each string has a length byte at the
                                    ;start of the string
STRGCMPR  CLR.W     D0              ;set the low order word of D0 to zero
          MOVE.B    (A2)+,D0        ;move the length byte of the first string
                                    ;to D0
                                    ;and move A2 to point to the first string
                                    ;byte
          CMP.B     (A4)+,D0        ;compare the lengths of the two strings
                                    ;and move A4 to point to the first string
                                    ;byte
          BNE       EXIT            ;if lengths different, exit with zero flag
                                    ;cleared
          BRA       COMPSTR2        ;let DBNE subtract 1 at start to turn
                                    ;length to index
                                    ;if lengths are both zero, then strings
                                    ;are equal
COMPSTRG  CMPM.B    (A2)+,(A4)+     ;compare the two strings pointed to by
                                    ;A2 and A4
COMPSTR2  DBNE      D0,COMPSTRG     ;keep branching until a point of
                                    ;difference found
                                    ;or either string is exhausted.
                                    ;if strings same, then zero flag set upon
                                    ;coming here,
                                    ;if strings different, DBNE stopped
                                    ;looping when Not Equal (zero flag clear)
EXIT                RTS             ;exit with the zero flag indicating
                                    ;same/different
```

Upon return from this subroutine the zero flag could be checked; if it is set the two string were equal, if cleared the two strings were unequal.

Assuming the lengths are equal, the point of difference between the two strings is one byte back of where A2 and A4 are pointing into their respective strings.

You might carefully study the above routine before going on. This is a complete function using as building blocks the various statements we have been describing such as CMP, MOVE, B*cc*, and DB*cc*. String compare routines are very common in assembly language programming.

Effective Address (LEA, PEA)

In the chapter on addressing modes we saw many examples of LEA, Load Effective Address, and PEA, Push Effective Address. We also learned how to calculate the effective address. The form of LEA is:

```
LEA <ea>,An
```

where <*ea*> is any of the following types of addressing modes:

(A*n*), d(A*n*), d(A*n*,D*n* or A*n*), Abs, d(PC), d(PC,D*n* or A*n*)

Notice that these are the modes of addressing that point to specific places in memory, oftentimes indirectly. The place that these addressing modes point to is loaded into the destination operand An, an address register. The LEA and PEA only deal with long words since addresses are all 4 bytes.

The LEA instruction makes it very easy for the programmer to get an address. You address the location in which you are interested and then say "get me the address, machine!". Since so much of the programming in 68000 assembler is with data that is being relocated, these LEA and PEA instructions allow you to pin down where the data actually is located in physical memory.

If you want to get tricky, the LEA instruction can be used as a quick way to add together the values in an address register, a data register, and a constant using one instruction. This is sometimes the most straightforward way to do such an addition. As an example LEA −1(A3,D2),A0 would add the contents of A3 to the contents of D2 and then add −1 (our constant displacement) with the result going into A0.

Assembly language doesn't have subscripts and arrays like higher level languages. Instead you can use the indexed indirect with displacement addressing mode to form 2-dimensional arrays. You could then find the location of a particular element in the array by using the LEA instruction.

Suppose you have an array with 2-byte integers, 10 rows by 20 columns, and you want to find the value at (3,7). Let's say A0 points to the start of the array. The following code will find the location of (3,7) and then subtract 1 from this element using the LEA address. (Notice the use of a multiplication opcode, MULU, that multiplies unsigned numbers.)

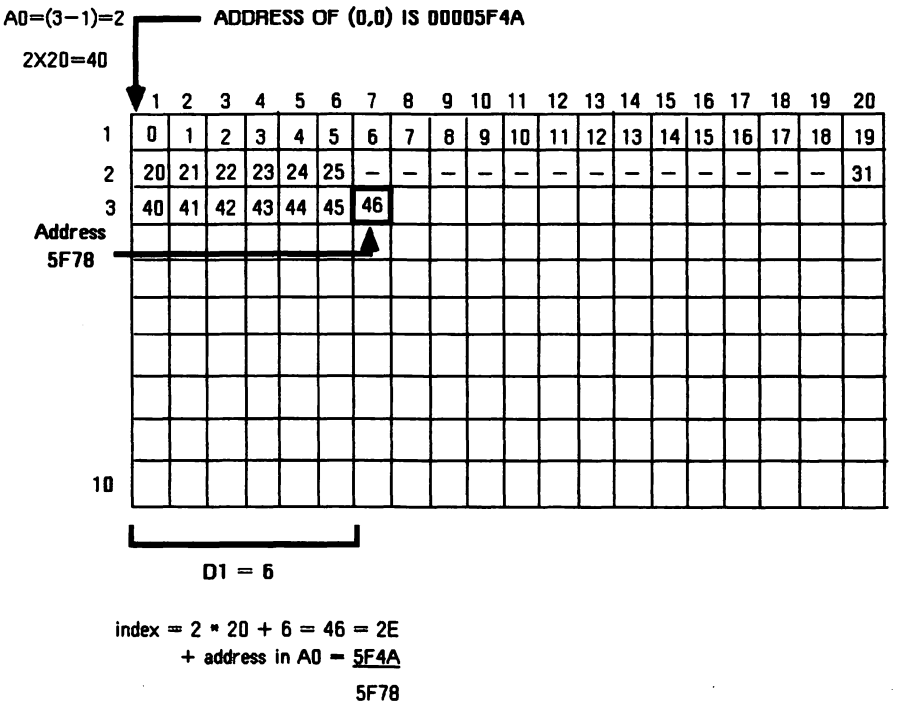


Figure 3-2 Picture of Effective Address Calculation for Element (3,7) in an Array

```
length    EQU 20
          MOVE.L #2,D0        ;D0 and D1 contain the x and y coordinates (each −1)
          MOVE.L #6,D1
FIND3.7   MOVE.L #length, D2  ;move the length of a row to D2
          MULU D0,D2          ;now D2 has the offset of the start of row 3
                              ;MULU means MULtiply Unsigned (D2 is multiplied by D0
                              ;and the result placed in D2)
          ADD.L D1,D2         ;now D2 has the offset of the start of element (3,7)
          LEA 0(A0,D2),A1     ;now A1 has the address of this element
          SUB.W #1,(A1)       ;subtract 1 from this element
```

The PEA, or Push Effective Address instruction, is used most frequently to pass the location of structures to subroutines via the users' stack. The address of an array could be passed on the stack using a PEA, as an example. When you are using the Macintosh ROM routines they require the address of the variables to be passed on the stack. Here is an example of a call to the floating point package using PEA (the floating point package processes decimal numbers and has square root, exponents, etc. These are just like the functions on a pocket calculator.):

```
PEA EXTFLOAT(A5)
FSQRTX
```

Now lets look at each of these two lines in detail:

```
PEA EXTFLOAT(A5)
```

Here EXTFLOAT is an extended floating point number after the PEA instruction, A7 is 4 less—now A7 points to the address of EXTFLOAT(A5) which has been pushed on the stack.

```
FSQRTX
```

This is a *macro*. A macro gets turned by the assembler into many lines of code (in this case three lines). FSQRTX expects the 4-byte address of an extended number to be on the stack—the address points to a 10-byte long extended precision floating point number. When the routine has done its work, the square root of what was originally in EXTFLOAT(A5) is now placed there.

PEA is functionally equivalent to the following:

```
LEA EXTFLOAT(A5),A0   ;load the effective address of EXTFLOAT off A5 into A0
MOVE.L A0,-(A7)       ;and push all 4 bytes of A0 on the stack
```

Of course, in the above code, A0 is destroyed by this maneuver. Also twice as many lines of code are used.

The Remaining Data Movement Operations

We have already looked at the MOVE, MOVEM, MOVEQ, LEA and PEA instructions. Some less frequently used, but still useful instructions, are the EXG, SWAP, LINK, and UNLK instructions.

EXG Instruction

The EXG instruction can be used to exchange the data in two registers—all 4 bytes are always exchanged. The two registers can be either both data registers, both address registers, or a data and an address register. In the third case, where there is an exchange of information between a data and an address register, the data register must be in the source operand (the

first operand) and the address register must be in the destination operand (the second operand). Let's look at an example of each of the three kinds of exchanges.

In this example data in data registers are exchanged:

```
   MOVE.L #1234,D0
   MOVE.L #5678,D1
   EXG D0,D1      ;notice that EXG D1,D0 would have the same effect
;now D0 contains 5678 and D1 contains 1234
```

In this example data in the address registers are exchanged:

```
LABEL1 DC.W $12D3
LABEL2 DC.W $A5F8
       LEA  LABEL1,A4
       LEA  LABEL2,A3
       EXG  A3,A4
;now A3 points to LABEL1 and A4 points to LABEL2
```

Finally, in this example the contents of a data register and an address register are exchanged:

```
EXG D0,A4 ;remember, EXG A4,D0 would be illegal
```

Typically, you use EXG when you want to operate on a value in a particular register in a mode that is illegal for that register (for example, operate on the last byte of an address). You exchange the register with another register that allows you to correctly perform the operation you desire and then exchange the data back into the original register.

Another, more general, reason to use EXG is if a subroutine or function requires a particular register to have data in it when the data, at that time, is in a different register. You would use EXG before and after calling the function.

In the following example DATACHG uses data in D0 and changes it—however, the data you want to operate upon is in D3. You also know that DATACHG does not effect register D3.

```
EXG D0,D3      ;put the data in D3 into D0, and vice
               ;versa
JSR DATACHG    ;the data that was in D3 is now changed
EXG D0,D3      ;reverse the data back
               ;(now D3 has been changed and D0 is
               ;the same as before)
```

SWAP Instruction

The SWAP instruction operates only on a data register. The high two bytes of the register are exchanged with the low two bytes of the register.

Example:

```
MOVE.L #$12A59D4C,D0   ;D0 has $12A59D4C in it
SWAP D0                ;now D0 has $9D4C12A5 in it
```

Typically, you use the SWAP instruction when you are dealing with two 2-byte words—often, two integers. You SWAP, manipulate the first integer (which was in the high two bytes), SWAP, and manipulate the second of the two integers (which was in the low two bytes).

LINK and UNLK

These two instructions go together. If you know a structured language, like Pascal, PL/I, or C, you will find it easier to understand what the LINK and UNLK instructions do. LINK and UNLK (UNLinK) allow you to set up variables for a subroutine (called a PROCEDURE in Pascal) on the stack and then to throw them away when the subroutine is finished.

LINK sets up the variables, and UNLK throws them away, undoing the work that LINK has done. The form of LINK is:

LINK *An,#<displacement>* where *<displacement>* is a constant number of bytes (invariably negative, for reasons we will describe later).

You use LINK to set aside space on the stack for local variables. For example, here's a line of code that sets aside 20 bytes on the stack:

```
LINK A3,#-20   ;set aside 20 bytes on the stack
```

A local variable is one that is only used inside the subroutine; nothing outside the subroutine needs know about it. In general, LINK is used for scratch space on the stack that is used by a subroutine.

When a LINK is performed, three things are done:

1. The original contents of the register you name are pushed on the stack. In other words, the original contents are saved there so that they can be restored when you leave the subroutine.
2. The stack pointer (which is now pointing to the contents of the register you just pushed) is put into the register you named. Thus you can use the register to address your new data space. Notice that you will have to use negative displacements to address your data since it is arrayed below where your register is pointing.
3. The displacement you specify is *added* to the stack pointer. Since the stack goes downward, you want to add a negative number so that space is created on the stack.

The UNLK instruction undoes what the LINK instruction does:

1. It restores the stack pointer from the register you name. Now the stack pointer has undone the displacement you added to it and is pointing at the old value of the register.

2. It pops the old value of the register you name off the stack. Now the stack pointer is pointing at the place it was prior to the LINK.

The form of UNLK is:

UNLK *An*

For example, UNLK A3 restores the original stack pointer and value of A3 that was changed using the LINK A3, #−20 instruction above.

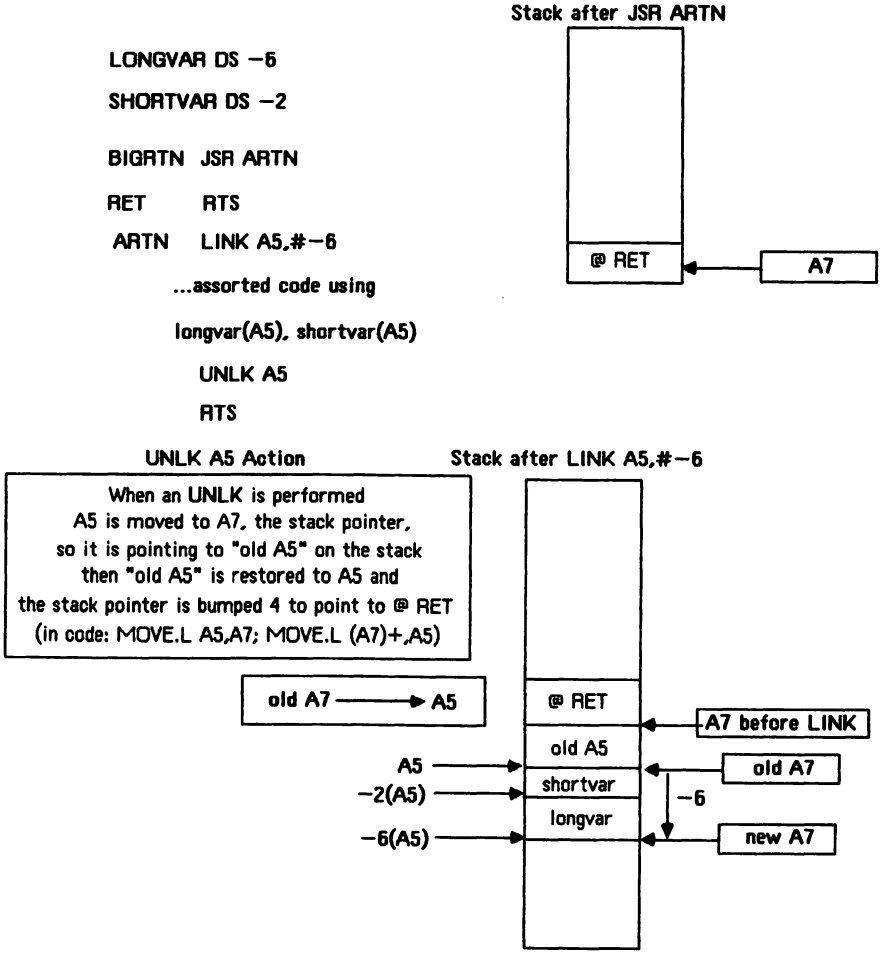


Figure 3-3 LINK/UNLK Code

Now the register is back to where it was before you entered the subroutine and the stack pointer is also restored.

Let's look at some sample code to see the LINK and UNLK instructions in action. This is a subroutine that concatenates STRING1 to the end of STRING2, and places the result in STRING2. It assumes that the length of the strings are in the first byte and that neither string is longer than 80 bytes (and the total is <= 80 bytes).

```
CONCATEN  PEA STRING1(A5)          ;push the address of a string on the
                                   ;stack
          PEA STRING2(A5)          ;push the address of another string on
                                   ;the stack
          JSR CONCAT               ;jump to a subroutine to concatenate the
                                   ;two strings
                                   ;now the concatenated string is in
                                   ;STRING2(A5), zero flag clear if error,
                                   ;set if OK
          RTS                      ;return to the higher level routine
                                   ;subroutine CONCAT that concatenates the
                                   ;two strings
                                   ;assumes the addresses of the strings
                                   ;are on the stack
                                   ;destroys registers A0-A2
STRG1     EQU 8                    ;relative location of string1 above A2
STRG2     EQU 4                    ;relative location of string2 above A2
TOTLEN    EQU -2                   ;length of the concatenated string
ERRCODE   EQU -4                   ;error if concatenated length > 80 is
                                   ;-1, else 0
CONCAT    MOVE.L A7,A2             ;save the location of the addresses of the
                                   ;strings
          LINK A6,#-4              ;reserve 4 bytes on the stack
          MOVE.L STRG1(A2),A0      ;get the address of string1 into A0
          MOVE.L. STRG2(A2),A1     ;get the address of string2 into A1
          CLR.W TOTLEN(A6)         ;zero out the total length variable on the
                                   ;stack
          MOVE.B (A0),TOTLEN(A6)   ;move the length of string1 to the total
                                   ;length
          MOVE.B (A1), D0          ;ADD requires a data register
          ADD.B D0,TOTLEN(A6)      ;and add the length of string2 to the
                                   ;total length
          BCS ERROR                ;if overflowed 255 ($FF) then obviously
                                   ;> 80
          CMP.B #80,TOTLEN(A6)     ;compare total length to 80
```

```
        BHI ERROR                   ;if > 80 then error
                                    ;here a routine, ADDSTRG, assuming that
                                    ;strings have been pointed to by A0,A1,
                                    ;concatenates them.
        JSR ADDSTRG
        MOVE.W #0,ERRCODE(A6)       ;if execution gets here,
                                    ;the lengths were ok so no error
        MOVE.B TOTLEN(A6),A1
        BRA NOERR
ERROR   MOVE.W #-1,ERRCODE(A6)      ;if execution gets here,
                                    ;the lengths added to more than 80
NOERR   TST.W
        ERRCODE(A6)                 ;sets the zero flag based on error
        UNLK A6                     ;restore A6, stack
        RTS                         ;and return from subroutine
ADDSTRG CLR.W   D0                  ;ADDSTRG subroutine
        MOVEM.L D0/A1/A2,-(A7)
        MOVE.B  (A1)+,D0
        LEA     0(A1,D0),A1
        MOVE.B  (A0)+,D0
        BRA     @20
@10     MOVE.B  (A0)+,(A1)+
@20     DBRA    D0,@10
        MOVEM.L (A7)+,D0/A1/A2
        RTS
```

You don't have to understand everything that is happening in the above routine. Notice, however, how the LINK and UNLK statements work, how space for TOTLEN and ERRCODE is created on the stack, how TOTLEN and ERRCODE are addressed, and how UNLK clears the space created for these two variables off the stack.

By using A6 (or some other address register) as your LINK variable you could create a whole series of subroutines which call other subroutines. All the subroutines would use the same address register to access their temporary variables. Each time you entered a subroutine the address register would be used to point to your temporary variables, each time you returned,the address register would be restored as it was before calling the subroutines. Meanwhile, each time you entered a subroutine, another layer of variables would be put onto the stack and each time you exited this layer would be removed from the stack. If you were three layers deep in subroutines you would have three layers of variables on the stack.

LINK and UNLK are very careful instructions if you want to avoid permanently setting aside space for variables that are only used for temporary results in a subroutine. However, if you needed to look at those variables in either a higher or even a lower subroutine, you would not use

this method of creating space for variables. You would have to pass the address of variables in the higher subroutine to lower subroutines if they wanted to use those variables, either through PEA on the stack or through a global variable set aside for the purpose. You could never use the variables set aside by LINK in a higher level routine since the stack space is freed by the UNLK command and could be clobbered at any time by an interrupt (for example). By using offsets, which are unstable with any change in code, you can access the LINK variables in a higher routine; although you may see this in code generated by a compiler, you should not do this in your own assembler code.

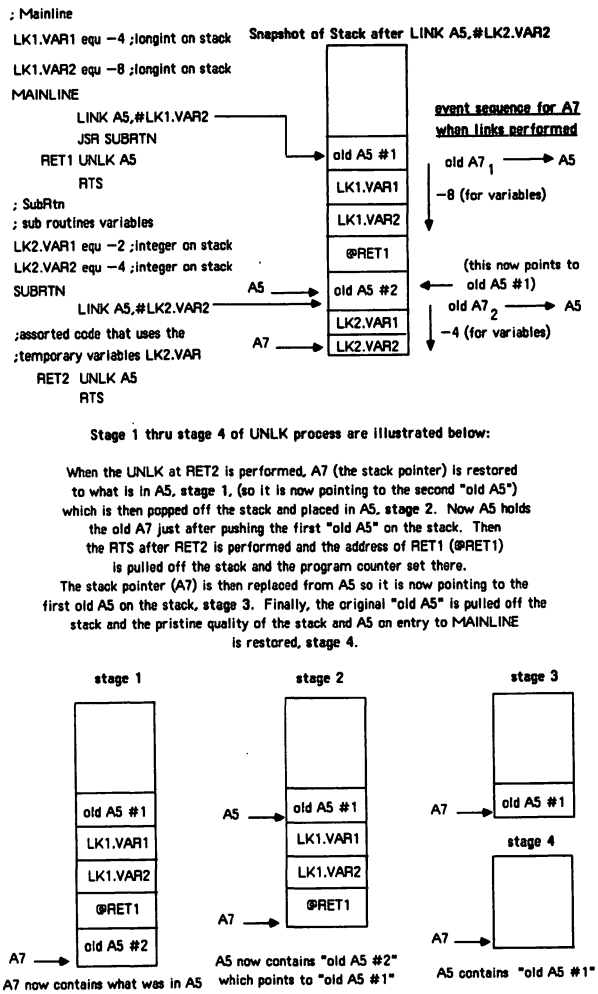


Figure 3-4 Nested LINK/UNLK

The Rest of the Program Control Operations

Program control instructions allow you to change your location within the program. In assembler, this means that they change the program counter so it points to a place other than the next instruction. We have already seen many of these instructions. The branch instructions, B*cc* and BRA, are program control instructions—so is DB*cc*. We have already seen JSR and RTS many times. The remaining instructions are S*cc*, Set byte Conditionally, BSR, Branch SubRoutine, JMP, JuMP, and RTR, ReTurn and Restore condition codes.

Let's take a closer look at the JSR-RTS pair. If you GOSUB a subroutine and RETURN in BASIC or call a Procedure in Pascal, you magically return to the next statement after the GOSUB or the Procedure. You have to understand how a subroutine works before you can properly do assembler programs.

The JSR instruction pushes the program counter of the instruction following itself onto the stack. Then it resets the program counter to the address in the operand. Usually, this address is a program label. When you have a whole series of subroutines you want to access, you may use a form of indirect addressing.

The form most often used is:

```
ALABEL ...          ;a label somewhere in the code
       RTS          ;if called here from the JSR below, the address on the stack is pulled
                    ;and placed into the program counter.
       ...
       JSR ALABEL   ;the address of NEXTPLAC is pushed onto the stack
NEXTPLAC ...
```

Here is another common form for a series of subroutines, which operates like a CASE statement in a block structured language. In this example, if you call CASE with D0 holding the number of the subroutine you want it selects the subroutine and executes it:

```
CASE1 LEA TABLE,A0   ;get address of TABLE into A0
      MULU #4,D0     ;D0 now holds number of subroutine times 4 (a JMP is 4 bytes)
      JSR -4(A0,D0)  ;jump to one of three subroutines based on D0 (D0=1,2, or 3)
                     ;the above code jumps to the code at TABLE+4*D0-4
                     ;(ie TABLE if 1, TABLE+4 if 2, etc)
      RTS
SUB1  (subroutine code for SUB1)
      RTS
SUB2  (subroutine code for SUB2)
      RTS
```

```
SUB3  (subroutine code for SUB3)
      RTS
                    ; a "jump table"
TABLE JMP SUB1      ;comes here if D0 holds 1
      JMP SUB2      ;comes here if D0 holds 2
      JMP SUB3      ;comes here if D0 holds 3
```

An example of how to code a three way branch using a more complex JSR is illustrated above. You calculate the position of the jump to the appropriate subroutine, then JSR to the location of the jump. The jump goes to the appropriate subroutine and executes it. That subroutine returns to the RTS which ends the CASE subroutine. There are more elegant ways of coding this using PC relative mode which are covered in Chapter 4, sample Programs.

An RTS, ReTurn from Subroutine, pops the address off the stack and places it into the program counter. If you have data passed on the stack that you want to pull off, you would pull off the return address using the MOVE instruction instead of using an RTS so you could pull the data off the stack while in the subroutine.

You will see the following code in 68000 assembler fairly often:

```
      JSR MYRTN
      ...
MYRTN MOVE.L (A7)+,A0 ;pull return address off stack
      MOVE.L (A7)+,D1 ;pull some data off the stack which was below the return addr.
      ...
      JMP (A0)        ;jump to return address (acts like an RTS)
```

An understanding of the stack is crucial to understanding 68000 assembler (or any assembler for that matter). Let's draw pictures of the stack when the following code is executed:

In the following code, we've numbered the lines to make references to them easier.

```
0       JSR HIGHRTN
1  AFTHIGH ...
           ...
2  HIGHRTN   MOVE.L D0,A1  ;busy work
3            JSR LOWRTN    ;call a sub-subroutine
4  AFTLOW    MOVE.L D1,A2  ;busy work
5            RTS           ;return from higher level subroutine
6  LOWRTN    MOVE.L A0,D2  ;busy work
7            RTS           ;return from lower level routine
```

Let us step through the program above and take "snapshots" of the stack at each point. In line 1 above the address after JSR HIGHRTN (@

means "address of") is pushed onto the stack. The stack now looks like this with the address of AFTHIGH on the stack:

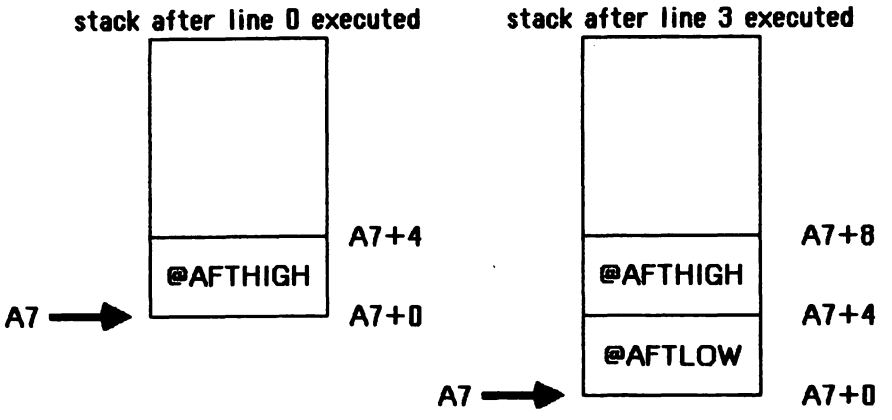


Figure 3-5 The Effect of the Above Code on Stack

In line 3 the address after JSR LOWRTN is pushed on the stack; the stack has the address of AFTLOW highest on the stack and the address of AFTHIGH next on the stack.

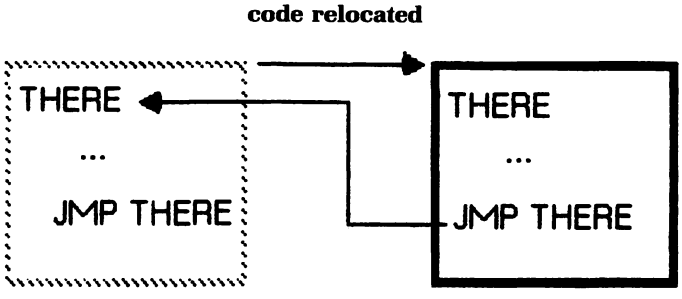
In line 7 the address of AFTLOW is pulled off the stack and placed in the program counter—the stack looks the same as after line 1 had just been executed. In line 5 the address of AFTHIGH is pulled off the stack and placed in the program counter.

Go over the above sequence of code and stack diagrams until you really understand how a subroutine works. It is very important. As an exercise, try diagramming a subroutine that in turn calls two other subroutines in sequence; then try a three level deep subroutine call. When you have one subroutine inside another they are called "nested" subroutines.

What is the difference between branch and jump? Branch takes your current address and adds an amount (which may be negative) to find your new address. If the code is placed anywhere in memory, a branch will still work. A branch is relative to where the program counter is pointing. A jump, by contrast, goes to a specific address in memory. If the code is moved somewhere else in memory the jump will no longer go to the right place, it will go where the address was when the code was created, not where it is now. A jump refers to a specific location in memory.

In the same way that a jump, JMP, is related to a branch, BRA, a jump subroutine, JSR, is related to a branch subroutine, BSR. JSR will only work on absolute addresses, BSR will work on relative addresses. The return address pushed on the stack is the same, so RTS is used for both.

In general, you use BSR to use a subroutine within the same segment of code and use JSR to access a subroutine in a fixed location. On the Macintosh you would only use JSR on a computed address. In general, since code can be relocated at any instant on a Macintosh, you should use BRA and BSR rather than JMP and JSR.



When originally coded, THERE was a certain location. Unfortunately, the code has since been relocated by the Macintosh memory manager so there is no THERE there! (Instead you code with relative intersegment branches or JMPs through the memory manager jump table.)

Figure 3-6 Why You Can't Have Absolute JuMPs in Macintosh Code

Let's complete the program control instructions by looking into RTR and S*cc*. An RTR first pulls the condition codes off the stack, then does a normal RTS. So an RTR is an RTS plus. You, the programmer, must push the condition codes on the stack so they are there when the RTR expects them to be. To do this you use the following code:

```
MOVE SR,-(A7)
```

which moves a word of data from the status register to the stack. This is usually done immediately upon entering the subroutine. Note that the RTR only restores the condition codes which are 5 bits of the status register, however, the whole word of the status register is pulled from the stack.

A subroutine that uses the following:

```
MYSUB MOVE SR,-(A7)
      ...
      RTR
```

returns with the exact same condition codes that you had before calling the subroutine.

Typically the code that uses this type of subroutine would look like this:

```
CLR.W AVALUE(A5)        ;set AVALUE(A5) to zero
AVA TST.W AVALUE(A5)    ;see if AVALUE(A5) is zero (sets zero flag in condition codes)
BSR MYSUB               ;branch to above subroutine which preserves condition codes
BEQ AVA                 ;branch to AVA and loop until AVALUE is not zero
```

In the above example, it is assumed that AVALUE(A5) is used as a flag and set non-zero by MYSUB when it is one cycle away from completing. Usually you do not need the condition codes to be the same as when the subroutine was entered, so you can use RTS to save time and space.

Finally, let's look at the *Scc* instruction. This instruction sets a byte to all binary ones if the condition is true ($FF) or all zeros if the condition is false ($00). You can think of this instruction as the set-a-flag-based-on-a-condition instruction.

For example, the following code:

```
TST.B MARRIED(A5)
SEQ FLAG(A5)
```

sets FLAG(A5) to $FF if MARRIED(A5) is zero and FLAG(A5) to zero if MARRIED(A5) is not zero.

The equivalent code in Pascal would be:

```
if married = 0 then flag := true else flag := false;
```

or in BASIC:

```
10 IF MARRIED = 0 THEN FLAG = 1 ELSE FLAG = 0
```

where 1 signifies true and 0 signifies false. The *Scc* instruction assumes you adopt a convention of $FF is true and $00 is false inside a byte.

Arithmetic Operations in the 68000

You can do both binary and decimal arithmetic using the 68000 chip. Binary arithmetic is limited to byte, word, or long arithmetic (in other words very short, short or long integers). Unlike the binary arithmetic, the decimal arithmetic is limited to a single byte at a time. Each byte of the decimal arithmetic operand contains two decimal digits. In decimal mode therefore you can add two digits at a time with carry. As with all arithmetic operations, the carry is contained in both the extend bit flag and the carry bit flag when it is generated.

If you want floating point calculations (like a calculator with decimal point and exponent) you will have to use Apple's Standard Apple Numeric

Environment, whose acronym is SANE. SANE is a package of subroutines that perform all the functions of a calculator, some functions include: the ability to operate on 4-byte, 8-byte, and 10-byte floating point numbers; addition, subtraction, multiplication, division, square root, exponentiation, and trigonometric functions. How to use SANE is described in Chapter 10, Advanced Subroutines Not in SimpleCalc.

The Extend Flag

The extend bit is a flag in the condition codes. When it is set it is always the same as the carry flag since the carry flag is always set at the same time. Sometimes the carry flag is set and the extend flag is not, however. For example, when you do a compare (CMP) the extend flag is *not* set. The extend flag is set by add, subtract, negate, and some shift instructions.

The extend bit is set by a carry in an add instruction or a borrow in a subtract instruction—the same as the carry flag. Many other processors have only a carry flag and no extend flag. This results in the carry being disrupted by a comparison or MOVE type instruction. Oftentimes you want to do MOVEs in between additions or subtractions, especially when you add one long number to another and place the result in a third location. Sometimes you want to do a CMP and branch to end the loop by doing an add. Because the carry information is also in a separate extend bit you can MOVE or CMP in the middle of a multi-byte add or subtract without having it disrupt the arithmetic operation when the MOVE or CMP overlays the carry bit.

Addition

There are a series of additions which one can perform: ADD, or ADD binary, ADDI, ADD Immediate, ADDQ for ADD Quick, ADDA for ADD Address, ADDX for ADD eXtended, ABCD for add decimal with extend (mnemonically, Add Binary Coded Decimal). Although, each is appropriate in certain circumstances, all add the source to the destination and place the result in the destination.

ADD, ADDI, and ADDQ are the most commonly used addition instructions. ADD adds two numbers together. The numbers can be a byte long, a word long (2 bytes), or long word (4 bytes). Both the carry bit and the extend bit are set when there is a carry out of the addition. Either the source or the destination operand must be a data register. The source operand can be any of the addressing modes if the destination is a data register. The destination operand cannot be an address, PC relative, or immediate if the source is an address register.

Here are two examples of ADD instructions that use word and byte sizes and various addressing modes:

```
ADD.W  myvalue(A3,D0),D4
ADD.B  (A0)+,D2
```

ADDI is used when you have immediate data in the source to add to the appropriate addressing mode in the destination. Only addressing modes which reference data that can be changed are allowed in the destination. Hence immediate, both PC relative modes, and address mode (see ADDA) are not allowed.

Examples:

```
ADD.B #$2E,(A0)+
ADDI.L #$A327E9A4,D2
```

ADDQ can be viewed as a special hi-speed form of ADDI. ADDQ can have address mode, An, as a destination also, but only word and long length. If address mode is the destination, the condition codes are not affected—specifically there is no carry or zero flag set. If you remember, MOVEQ can be any value that can fit in one byte—0 to 255 ($00 – $FF)—but ADDQ can *only* be used for values between 1 and 8. ADDQ is usually used as an increment instruction.

Examples:

```
ADDQ.L #1,D1      ;increment D1
ADDQ.B #2,(A3)+ ;add 2 to the byte A3 points to, then add 1 to A3
```

ADDA is just like ADD only the destination is an address register. However, as with all 68000 instructions that have address registers as destinations the only lengths allowed are word and long and the condition codes are not affected. Invariably, you will want the long length but the default is word. This can create a very subtle bug in your code which will only appear when you cross a 32K boundary! So don't forget to put ADDA*.L* when you are using this instruction. All assemblers turn ADD to ADDA if the destination is an address register.

Example:

```
ADDA.L #12,A3 ;add 12 to address register 3
```

ADDX means add extended. This adds the source to the destination, just like ADD, but then *also* adds 1 if the extend flag is set, and puts the result into the destination. In other words, ADDX is used to add up numbers that are larger than 4 bytes long by adding up a 1-, 2-, or 4-byte "chunk" at a time and carrying the result into the next "chunk". This is just like adding a pair of multi-digit numbers together in normal decimal with carry.

Here is a sample program to add two 8-byte long numbers together. The numbers are in memory. The first number is pointed to by A0, the second number is pointed to by A1, the result is to go where A2 points. The fourth line illustrates ADDX.

```
MOVE.L  (A1),(A2)+
MOVE.L  4(A1),(A2)+
```

```
ADDQ.L   #4,A0
MOVE.L   (A0),D0
ADD.L    D0,-(A2)
ADDX.L   -(A0),-(A2)  ;add the second 4 bytes (with carry if it exists)
```

The last type of add involves binary coded decimal. Here, ABCD is not just the first four letters of the alphabet but it also stands for Add Binary Coded Decimal. Before describing this instruction we will take a detour and describe binary coded decimal.

Binary Coded Decimal

Binary coded decimal means that each decimal number is placed into hexadecimal one nybble at a time—hence there are two decimal numbers in each byte since there are two nybbles in each byte. In other words, you can read the decimal numbers straight from a hexadecimal print out of memory. For example, 1234 in decimal becomes 12 in the first byte and 34 in a second byte of hexadecimal. The number 789 becomes 07 in one byte and 89 in the succeeding byte.

In normal binary addition carry occurs when you exceed 255 or $FF. In BCD the carry occurs when you exceed 99!

For example:

```
73 + 82 = 55 with a carry in BCD.
```

The add opcode, ABCD only works on one byte at a time. All three numbers in the above addition each fit in one byte. If you wanted to add two 10-digit numbers (each taking up 5 bytes) you would do it this way:

```
  1374025609   or   13   74 02   56 09
+ 2193517832         21   93 51   78 32
  3567543441         34+C 67 53+C 34 41
```

where +C means add the carry generated from the preceding pair of digits.

Getting back to the ABCD instruction, we note that there are two forms the operands can take:

ABCD D*n*,D*m*
ABCD -(A*n*),-(A*m*)

For example:

```
ABCD D0,D2
ABCD -(A3),-(A1)
```

You can add a byte of BCD in the rightmost byte of D*n* to the rightmost byte of D*m* with the result winding up in the rightmost byte of D*m*. If the result exceeds 99 a carry and an extend bit is generated.

The zero flag in ABCD is *not* set if the result is zero (however, it is cleared if the result is non-zero). If you are interested in knowing whether the result of a multi-byte decimal addition is zero, the zero flag must be set before starting. That way if any pair of digits is non-zero they will clear this flag and if no pair of digits clear it then the whole multi-digit number must be zero.

Now let's code something that will perform the 10-digit add that you saw above. This code has the first 10-digit number pointed to by A0, the second by A1; the result overlays the second operand.

```
TENDGADD ADDQ.L #5,A0        ;point at the last byte + 1 of the source 10-digit number
         ADDQ.L #5,A1        ;point at the last byte + 1 of the destination 10-digit number
         MOVEQ.W #4,D0       ;use D0 to count in the DBRA below (5 times through loop)
         MOVE #$04,CCR       ;clear the carry and extend flags, set the zero flag
TENDLOOP ABCD -(A0),-(A1)    ;add a BCD byte from where A0 points to where A1 points
         DBRA D0,TENDLOOP    ;decrement D0 and branch until D0 is -1 ($FFFF)
         RTS                 ;return from subroutine, A0 & A1 now point to start of
                             ;respective numbers; the zero flag reflects a zero result.
```

TENDLOOP starts with A0 and A1 pointing to the last byte plus one of the decimal numbers to be added. Then the ABCD instruction subtracts one from A0 and A1 and does a decimal add of the bytes to which it is now pointing with the result replacing the latter byte.

Subtraction

The SUB, or SUBtract binary, SUBI SUBtract Immediate, SUBQ, SUBtract Quick, SUBA, SUBtract Address, SUBX, SUBtract eXtended, and SBCD, SUBtract decimal with extend instructions (Subtract Binary Coded Decimal) handles the operands exactly like the equivalent add instructions. The carry and extend flags are set whenever a borrow is necessary just as they are set in add when a carry is generated.

Multiplication

There are two forms of multiply: MULS, Signed MULtiply, and MULU, or Unsigned MULtiply. In both cases the source operand can be anything except an address register; the destination must be a data register.

Forms: MULS *source*,Dn
MULU *source*,Dn

These two instructions take the low 2 bytes of the source and multiply them by the low 2 bytes of the destination data register and put the 4-byte result into the destination data register. The high 2 bytes of the destination data register are ignored.

The MULS, multiply signed, instruction treats the two mutipliers as signed numbers and comes out with a signed result. The MULU treats the

two multipliers and the result as unsigned. Therefore if both multipliers are between 0 and 32767 ($7FFF) the result is the same when using either MULS or MULU.

If D0 holds −2 ($FFFE) and D1 holds 10 ($000A) then the results of the following instructions are:

```
MULS D0,D1   ;after this D1 holds $FFFFFFEC (−20)
MULU D0,D1   ;after this D1 holds $0009FFEC (655340)
             ;or 10 ($000A) times 65534 ($FFFE)
```

Division

The division instructions are similar to the multiplication instructions. There are two forms just like multiply: DIVS, divide signed; DIVU, divide unsigned.

Both divide the 4-byte destination data register contents by the low 2 bytes of the source operand. The 2-byte quotient goes into the low 2 bytes of the destination register and a 2-byte remainder goes into the high 2 bytes of the destination data register. The sign of the remainder is the same as the dividend (unless the remainder is zero, obviously).

Division by zero causes a trap: on the Macintosh this results in a bomb screen with ID = 04. The ID number in the lower righthand corner of the bomb screen indicates what sort of fatal error; 04 means division by zero.

Overflow means that the quotient can't contain the result. A signed divide (DIVS D0,D1) with D0 containing $7FFF (32767), D1 containing $3FFFFFFF (1073741823), for example, should result in a quotient of $8001 (32769). Unfortunately, you can't have any positive number larger than 32767 as a signed number within a 2-byte result. Thus the quotient in $8001 results in an overflow and the overflow flag is set. An overflow with signed or unsigned numbers usually happens when you have a small divisor and a large dividend:

For example $20000000 divided by $0001 results in an overflow because $20000000 can't be held in 2 bytes (4 hex digits).

Let's look at some examples of these two types of divide that are typically used. Let's look at DIVU or divide unsigned first. If A0 points to the number $1E07 (7687) in memory and D2 contains $1A2D24A6 then:

```
DIVU (A0),D2
```

The above results in D2 containing $0079DF2B where $0079 in the high two bytes of D2 is the remainder and $DF2B in the low two bytes of D2 is the quotient. If you want to get at the remainder you would use the SWAP instruction we mentioned in the data movement instructions to get the remainder into the low-order word.

Now let's set up an example for DIVS, or divide signed. Suppose A3 contains the address $0001124A, D2 contains $00000002, MYOFF is a

constant equal to 4, location in memory $00011250 (the effective address of MYOFF(A3,D2)) contains $3A and location $00011251 contains $04 (that is, the word pointed to contains $3A04 or 14852 in decimal form). Finally, D5 contains the dividend of $ED0732A0 (−318295392).

```
DIVS MYOFF(A3,D2),D5   ;the result is $F77CAC49 in D5
```

Now that all that has been said the result is $AC49 (−21431) and remainder of $F77C (−2180) which are in the low 2 bytes of D5 and the high two bytes, respectively.

Other Miscellaneous Arithmetic Operations

You have just reviewed the 4 major arithmetic operations: add, subtract, multiply, and divide. In binary arithmetic there are other operations which are necessary such as EXT, EXTend, CLR, to CLeaR out an operand, NEG and NEGX, which NEGate an operand, and TST, to TeST an operand and set the condition codes appropriately. All of these miscellaneous operations work on a single operand. Since TAS, Test And Set, which would normally be associated with this group of instructions, is usually used for system work, we will discuss it in the System Control section to come later in this chapter.

EXT—sign extend

Often in arithmetic you are given a number that is byte sized which has to be used in arithmetic on words or you are given a word which you would like to be a long operand. EXT gives you the ability to make a 1-byte into a 2-byte form (.W) or a 2-byte into a 4-byte form (.L).

To make this conversion, the upper byte(s) of the smaller size number must be filled with binary ones if the number is negative and binary zeros if the number is positive. Put another way, the high bit is replicated throughout the upper byte(s). EXT has an operand which is a data register.

Example:

```
D2 contains $2307E2D3
EXT.W D2              ;now the result is $2307FFD3
                      D1 contains $4A7CA20A
EXT.L D1              ;now the result is $FFFFA20A
                      D4 contains $A45320E8
EXT.L D4              ;D4 now contains $000020E8
                      ;D0 contains $A45320E8
EXT.W D0              ;D0 has $A453FFE8
```

Once you've completed the conversion you can now perform arithmetic combining a 1-byte number with a 2-byte number or a 2-byte

number with a 4-byte number. If you want to arithmetically combine a 1-byte number with a 4-byte number you must perform a word and then a long extend to turn the 1-byte number into a 4-byte number prior to the arithmetic operation.

CLR—clear

CLR CLeaRs a byte, word, or long to zeros. The operand can be any of the usual addressing modes except PC relative or immediate. As usual, address registers can be only word or long.

Example:

```
CLR.L (A2)+ ;zeros the 4 bytes that A2 points at, adds 4 to A2 afterward
operates the same as MOVE.L #0, (A2)+ only faster and in less space
```

NEG, NEGX—negate

With the NEG and NEGX instructions the operand is subtracted from zero and the result of this replaces what was in the operand. The addressing modes (address register direct, PC relative, and immediate) are not allowed. The negative of the number replaces the number. NEG just subtracts the number from zero while NEGX also subtracts one if the extend bit is set. NEGX is used where you want to negate a number that stretches over more than 4 bytes. For example, if D0 holds $13450002, the instruction NEG.W D0 changes the contents of D0 to $1345FFFE. If D1 holds $3F7010A4 and the extend bit is set, the instruction NEGX.L D1 results in 0—$3F7010A4—1 or $C08FEF5B in D1.

TST—test

TST or TeST looks at a value in memory or a register using the usual addressing modes (address registers, PC relative, and immediate are not allowed, of course) and compare that value to zero. No data is changed, but condition codes are changed, just as with CMP. You can test data that is a byte, word, and long lengths. You usually use TST on a value in a register, on the stack, or in memory that you are using as a flag, and, based on whether it is zero or negative, the program takes certain actions. Invariably TST is followed by some form of conditional branch. For example, if A0 points to the value $0345, the instruction TST.W (A0) results in the N(eg), Z(ero), oVerflow, and C(arry) flags being cleared, but the eXtend bit flag is not changed.

Logical (Bitwise) Operations

The 68000 processor supports four logical operations: AND, OR, EOR, and NOT. Since these operations are very useful we will go into some detail in

describing them. These "bitwise" operations work on every bit in an operand or between pairs of bits in two different operands. Essentially each bit is seen as a "truth" value—1 means "true" and 0 means "false." The meaning of AND, OR, EOR, and NOT can be understood from seeing how true or false statement pairs act when they are connected by these conjunctions.

AND

Suppose we had two statements such as "1 + 1 makes 2" which is true and "the moon is made of green cheese" which is false. Then we join the two statements with an AND so:

1 + 1 makes 2 AND the moon is made of green cheese.

This statement is false. So a true statement AND a false statement make a false statement. If 1 represents a true statement and 0 represents a false statement we could say "1 AND 0 = 0." Actually we could notice that the result of two statements with an AND between them is only true if both statements are true. Let us summarize how AND works:

```
0 AND 0 = 0 (a false statement AND a false statement results in a false statement)
0 AND 1 = 0 (a false statement AND a true statement results in a false statement)
1 AND 0 = 0 (a true statement AND a false statement results in a false statement)
1 AND 1 = 1 (a true statement AND a true statement results in a true statement)
```

Now we are coming closer to understanding what the instruction AND.W D0,D1 means. The AND operation works on each pair of bits in two operands using the four equations above to arrive at a resulting bit.

So if D0 holds %1011010101111011 (the "%" symbol means that a binary number follows just as "$" means a hexadecimal number follows) in its last 2 bytes and D1 holds %0010110110110010 then the result of ANDing these two together is:

```
    1011010101111011
AND 0010110110110010
    0010010100110010
```

The rule is simple—if a column has two ones in it the result is one, otherwise the result is zero. This is the same rule as we saw above in the four equations about truth values.

You may be wondering, "well, so what? What is this useful for?" It turns out to be very useful when you want to isolate a part of a register or one bit. Then you have one of the two operands be a pattern of 1s (which are bits you are interested in) and 0s (which are bits you are not interested in). For example, if you are only interested in whether the second bit of a spot in memory is zero you would AND it with a pattern that had all zeros except for a 1 in the place of the second bit. Such a pattern is called a *mask*. This is

because just as a mask only allows certain parts of your face to be seen and masks off the rest so a bit pattern together with an AND instruction can mask off bits you aren't interested in. After you did the AND you would BEQ. The branch would be taken if the flag was zero and not taken if the flag was one.

Example

IF D1 holds %00101110, and we are interested in the second bit from the right, a one, we use D0 to hold %00000100 as a mask for the second bit from the right. The instruction AND.B D0,D1 results in %00000100 or a non-zero result in D1. You can follow this with BEQ BITFALSE to branch if bit two is false.

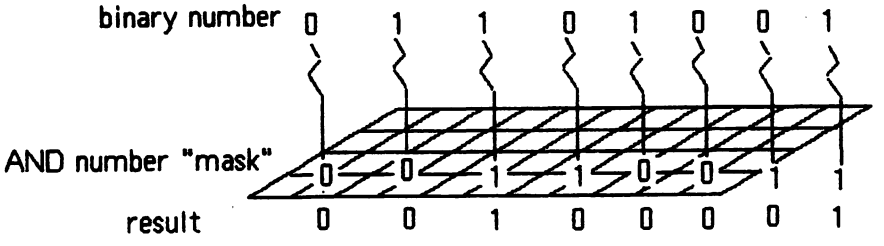


Figure 3-7 Mask as Sieve

OR

AND is an instruction used to turn selected bits off—to mask out bits. In contrast, OR is used to turn selected bits on and leave the other bits alone.

The truth equations for OR are as follows:

If either statement in a pair are true then the resulting statement with an OR in it is true. Only if both statements are false is an OR false.

0 OR 0 = 0 (a false statement OR a false statement results in a false statement)
0 OR 1 = 1 (a false statement OR a true statement results in a true statement)
1 OR 0 = 1 (a true statement OR a false statement results in a true statement)
1 OR 1 = 1 (a true statement OR a true statement results in a true statement)

An example of 0 OR 1 = 1 is: "The moon is made of green cheese OR 1 + 1 is 2" is a true statement.

Let's look at the same example as above, only with the OR statement. If D1 holds %00101010 and we are interested in the second bit from the right, we use D0 to hold %00000100 to make sure the second bit from the right is on. Then the instruction OR.B D0,D1 results in %00101110 in D1.

OR is usually used to turn on individual bits which are being used as flags.

EOR

Similar to OR is EOR, Exclusive OR. Exclusive OR is true if one or the other of the two statements are true but is false if both statements are true.

```
0 EOR 0 = 0 (a false statement EOR a false statement results in a false statement)
0 EOR 1 = 1 (a false statement EOR a true statement results in a true statement)
1 EOR 0 = 1 (a true statement EOR a false statement results in a true statement)
1 EOR 1 = 0 (a true statement EOR a true statement results in a false statement)
```

Notice that only the last equation is different from OR. EOR is used to "flip bits"—to turn a 1 bit to a 0 bit or vice versa for selected bits. Those bits you wish to flip should have a 1 in the mask; those you wish to leave alone should have a zero in them. Oftentimes you will want to flip all the bits in a byte, word, or long—this is called the one's complement. Then simply EOR with $FFFFFFFF ($F is hexadecimal for all ones).

To flip the bits on the lowest and third lowest bits, use the mask %00000101 as follows. If D3 contains %10011011 in its rightmost byte, performing an EOR with mask %00000101 results in D3 having %10011110.

The EOR instruction would look as follows:

```
EOR.B #%101,D3 ;now D3 has %10011110 in its rightmost byte
```

Notice how the lowest bit which had a 1 in it now has a zero and the third lowest bit which had a zero now has a one. All other bits are the same. EOR is used to change true to false and false to true—this is also called "flipping the bits."

NOT

The last of the bitwise operands is NOT. NOT operates on only one operand. It flips all of its bits. Like EOR with $FFFFFFFF it too creates a one's complement. For example, if A0 points to an area in memory that has %01110011 in it, the instruction NOT.B (A0) changes this area in memory to %10001100.

A final helpful hint on working with bitwise operands. In actual practice you don't use constants down in the code. You equate the various masks you are going to use with names. Then when you want to mask off the braces flag which is in the fourth bit you code the following:

```
BRACES equ $08 ;%00001000, a bit flag for use by this program to signal braces
        ... ;"..." means assorted other code
        AND.B #BRACES,D4 ;mask off the braces bit flag which is in D4
        BNE BRACHERE     ;branch if braces
```

This makes for much more readable code. Also you have less chance of making a mistake on setting up your mask.

Bit Manipulation Operations

In 68000 assembler you are given some powerful instructions which allow you to deal with individual bits. Since bits are often used for flags, the usual purpose of these instructions is to do some fancy bit twiddling on your flags. The instructions are: BTST for Bit TeST, BSET for Bit test and SET, BCLR for Bit test and CLeaR, and BCHG for Bit test and CHanGe.

BTST

BTST, Bit TeST, tests a bit and sets the zero flag on if the bit is zero and clears it if that bit is a one. There are two forms:

BTST D*n*,*destination*
BTST #<*constant*>,*destination*

The destination may be any addressing mode except an address register or immediate data.

The bit number is specified in the source operand. Zero represents the rightmost (lowest order) bit, 1 represents the bit to the left of that (next to lowest bit), etc.

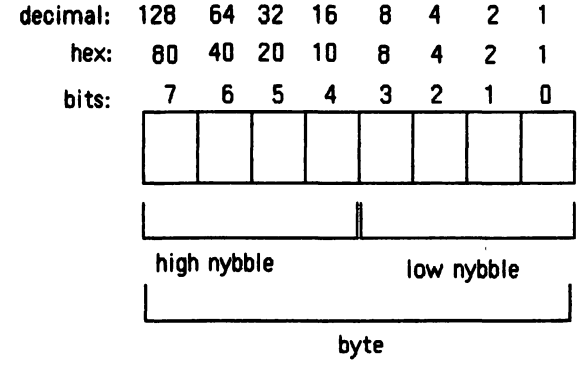


Figure 3-8 Bits in a Byte

If the destination is a data register any of the bits may be specified using a number from 0 (rightmost bit) through 31 (leftmost bit). If a number greater than 31 is used, the instruction uses the remainder after division by 32, called modulo 32, as the bit number. For example, 42 mod 32 = 10 since the remainder of 42 divided by 32 is 10 ("mod" is short for "modulo"). Don't use numbers to represent bits outside the range 0 through 31.

If the destination is not a data register, but is a place in memory, then only one byte is used. The bit number then must be between 0 and 7 where 0 is the rightmost bit, etc. If the number is larger than 7, modulo 8 arithmetic is used.

This example shows what happens when 32 bits of data in a byte are tested (we've numbered the bits from 0 on the right to 31 on the left):

```
                3322 2222 2222 1111 1111 1100 0000 0000
                1098 7654 3210 9876 5432 1098 7654 3210
D1 contains %0000 0011 0001 0000 1111 0001 1001 0010 or $0310F192
```

The 5th and 13th bits are underlined.

```
BTST #5,D1 ;looking above we see bit 5 is a zero, so the zero flag is set

BEQ BITCLR ;since the bit is zero, this branch is taken
```

This shows how you can test a single bit and branch based on its value.

```
BTST D0,D1 ;D0 has a 13 in it.
           ;since bit 13 has a one in it, the zero flag is cleared
           ;note: BTST D0,D1 would have done exactly the same thing
BEQ BITCLR ;this branch is not taken
```

Here we index to the bit to test using a register.

```
MOVE.L D1, (A0)  ;move what is in D1 to the location at address A0
BTST #12, (A0)   ;the remainder after division by 8 is 4, so this will test
                 ;the fourth bit of the byte immediately at A0. This byte is
                 ;the leftmost byte of D1 (00000011) and the fourth bit is a zero
BTST #4, (A0)    ;this will have the same effect as BTST #12 (A0)
BNE BITSET       ;this branch is not taken
```

This shows how bit testing a value in memory operates.

BSET

Bit Set is just like Bit Test only BSET sets the bit after testing it. Outside of that it is exactly the same in the way it numbers the bits, in having two forms, and in looking at 32 bits in a destination data register and only 8 bits in a memory location. There is one other difference, PC relative is not allowed.

Forms:

BSET D*n*,*destination*
BSET #<*constant*>,*destination*

For example if (A0) contains %00010000 (bit 3 has a zero), the instruction BSET #3, (A0) sets the zero flag and (A0) becomes %00011000 (bit 3 is now one).

BCLR

Bit Clear is just like Bit Set only BCLR clears the bit after testing it rather than setting it. Outside of that it is exactly the same.

Forms:

BCLR D*n*,*destination*
BCLR #<*constant*>,*destination*

For example if (A0) contains %10011010 (bit 1 has a one), the instruction BSET #1, (A0) clears the zero flag and (A0) becomes %10011000 (bit 1 is now zero).

BCHG

Bit CHanGe, BCHG, changes the bit, after testing it, to its opposite (0 becomes 1, 1 becomes 0). Outside of that it is exactly the same as Bit CLeaR in the way it handles operands.

Forms:

BCHG D*n*,*destination*
BCHG #<*constant*>,*destination*

For example, if D3 contains %0110 1101 1010 1010 11100001 0101 1011 (bit 12 has a zero), and D0 has a $0000000C in it, the instruction BCHG D0,D3 sets the zero flag and now D3 becomes %0110 1101 1010 1010 1111000101011011 (bit 12 is now one).

Shift and Rotate Operations

There are many times when you want to move the bits within a register or within memory. Briefly, *shift* moves the bits either to the left or the right, with those bits that go off the end going first into the carry/extend flags and then into oblivion. *Rotate* operations cycle those bits that are "lost," off either the right or the left end, back around so they come in through the opposite end of the register or data area.

The shift instructions are ASL for Arithmetic Shift Left, ASR for Arithmetic Shift Right, LSL for Logical Shift Left, LSR for Logical Shift Right.

The rotate instructions are ROL for ROtate Left, ROR for ROtate RIght, ROXL for ROtate with eXtend Left, and ROXR for ROtate with eXtend Right.

ASL, ASR

ASL stands for Arithmetic Shift Left and ASR stands for Arithmetic Shift Right. Why the word "arithmetic"? Because these operations are used to divide or multiply by a power of 2. If you shift a number right by 1 bit it is equivalent to dividing by 2, if you shift right by 2 bits it is equivalent to dividing by 4, while if you shift right by, say, 5 bits it is equivalent to dividing by 32. If you shift left by 5 bits, it is equivalent to mulitiplying by 32. So you might think of ASL and ASR as the "multiply/divide by power of 2 operations."

Shifting left using ASL involves no particular complications. The bit that is pushed off the left end with each shift goes into the carry and extend bits. Therefore only the last bit shifted will appear in the carry bit—you can't use the carry bit to test for overflow in shifts of more than 1 bit for this reason. Each time a shift to the left is made the empty space created to the right is filled with a zero. The ASL instruction has 3 forms.

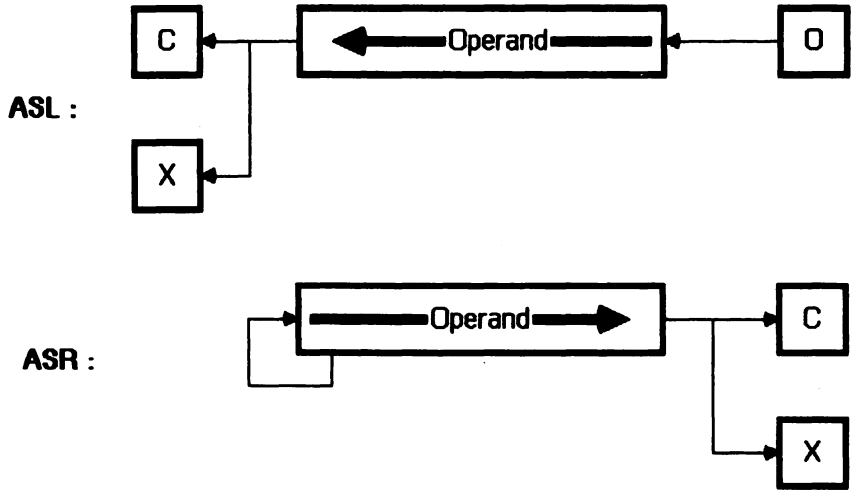


Figure 3-9 ASL/ASR

Forms:

ASL Dn,Dm
ASL #<*constant*>,Dn
ASL *operand* ;where *operand* is any addressing mode except Dn,An,PC
;relative or immediate

If the first form is used, the shift can be anywhere from 1 to 63 bits; larger numbers are treated as modulo 64. If the second form is used only shifts of 1 to 8 bits are allowed. If the third form is used, the shift is automatically one bit and only byte and word length can be used.

Example:

If D0 contains $3F4C0A0B or %0011 1111 0100 1100 0000 1010 0000 1011 then ASL #3,D0 results in the carry and extend flags being set and %1111 1010 0110 0000 0101 0000 0101 1000 or $FA605058 being in D0

By aligning the source and destination the above operation is easier to see:

```
%001111110100110000001010000001011     <=shift left 3 bits, underlined drop off
%   111110100110000001010000001011000  3 bits of zero inserted on right
```

Since we have shifted left by three bits, this is equivalent to multiplying by 2^3 or 8. This results in an overflow into the carry and extend bits. If you multiplied $3F4C0A0B times 8 you would get $1FA605058.

ASR, arithmetic shift right, is similar to ASL. The last bit shifted out of the righthand side goes into the carry and extend flags, and it uses the same three forms. The main difference is that the bits are shifted right rather than left. A further difference is that rather than filling with zeros, the fill bit is whatever was in the leftmost bit before the shift was commenced. In this way the sign of the original number is preserved. Therefore if you want a quick divide by 16 (or any power of 2) in which the sign is preserved, this is the way to do it.

Example:

If D1 contains $0000000C or 12 in decimal
and D0 contains $9F4E3A52 or %1001 1111 0100 1110 0011 1010 0101 0010
then ASR D1,D0 results in the carry and extend flags being set and
%1111 1111 1111 1001 1111 0100 1110 0011 or $FFF9F4E3 being in D0
since the highest bit in D0 before we started was 1.

Let's align the source and destination for the above operation:

```
     %10011111010011100011101001010010  source
%11111111111110011111010011100011       destination shifted right 12 bits
```

In the above shift the bold 1 bit on the left side of the source means that all the bits inserted into the destination on the left will be ones. The underlined bits of the source are lost off the right end. (You may hear the colorful programming phrase "into the bit bucket"; for reference, it means the same as "lost" does here.)

The overflow flag, V, is set if the most significant bit, the sign bit, is changed at any time during the shift operation. Therefore it can be used to

tell if the result is arithmetically valid (obviously the sign should never change during a shift). The negative flag is set if the result is negative and cleared otherwise.

LSL, LSR

LSL, Logical Shift Left, and LSR, Logical Shift Right, are nearly identical to ASL and ASR. The only difference between LSL and ASL is that in LSL the overflow flag is always cleared. The only difference between LSR and ASR is that the fill bit is always zero in LSR. Therefore LSR should not be used where you want to divide a signed number by a power of 2. It should be used where you want to divide an unsigned number by a power of 2.

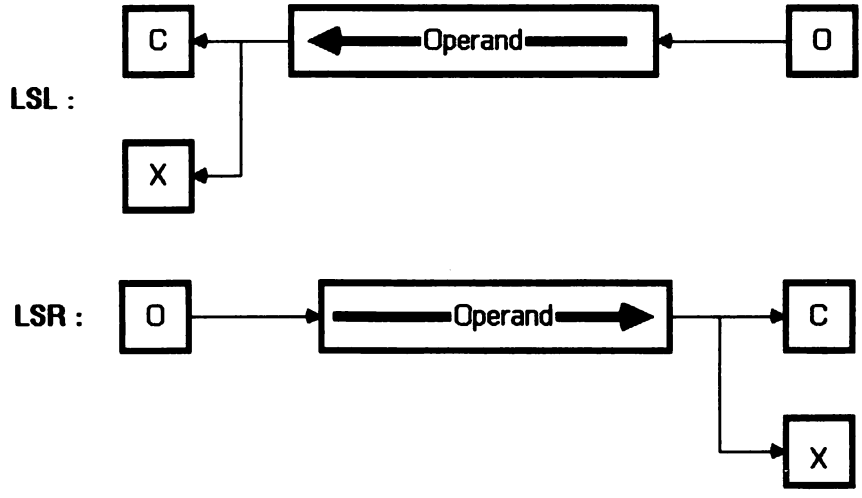


Figure 3-10 LSL/LSR

ROL, ROR

ROL stands for ROtate Left and ROR stands for ROtate Right. In similar fashion to the prior statements, LSL and LSR, the bits are shifted but the bits that fall off the end are recycled back in the other way. If you have a 5-bit rotate left, the leftmost 5 bits will become the rightmost 5 bits and the remaining bits will be shifted 5 left. If you do a 7 bit rotate right, the 7 rightmost bits will become the 7 leftmost bits and all the other bits will be shifted right by seven bits. The carry bit will be the same as the last bit rotated.

Watch a 5 bit rotate right in action operating on a single byte:

Suppose D2 contains %11011010 in the rightmost byte; then ROL.B #5,D2 operates as follows:

start point: %11011010
after 1 rotate: %10110101
after 2 rotate: %01101011
after 3 rotate: %11010110
after 4 rotate: %10101101
final result: %01011011 carry contains a 1 (set)

Note that the five leftmost bits were 11011; these are now the five rightmost bits. The three rightmost bits 010 are now the three leftmost bits.

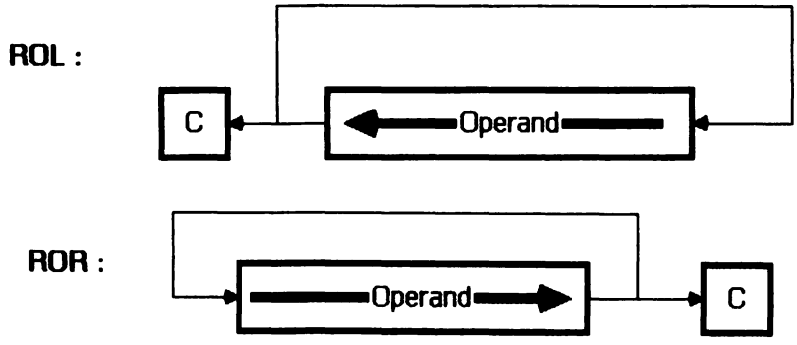


Figure 3-11 ROL/ROR

ROXL, ROXR

ROXL stands for ROtate with eXtend Left and ROXR stands for ROtate with eXtend Right. When ROL or ROR are used, the bits that fall off the end are immediately sent back into the data. In contrast, when you ROXL or ROXR, the bit that is shifted out is sent to the extend (and carry) bit. The extend bit, in turn, is sent back into the data. So the rotate takes place *through* the extend bit. There is always one bit out in limbo in the extend bit.

Let's examine how a 5-bit rotate right with extend works:

Suppose D2 contains %11011010 in the rightmost byte then ROXL.B #5,D2 operates as follows (assume the extend bit is set to 0 at the start):

start point: %11011010 (0 in extend bit at start, 1 in extend bit at end)
after 1 rotate: %10110100 (1 in extend bit at start, 1 in extend bit at end)
after 2 rotate: %01101001 (1 in extend bit at start, 0 in extend bit at end)
after 3 rotate: %11010110 (0 in extend bit at start, 1 in extend bit at end)
after 4 rotate: %10101101 (1 in extend bit at start, 1 in extend bit at end)
final result: %01011011 (1 in extend bit left after fourth rotate)

Outside of this rotating through the extend bit, ROXL and ROXR are exactly the same in form as ROL and ROR. These instructions are the same three forms with the same limits on the number of bits moved.

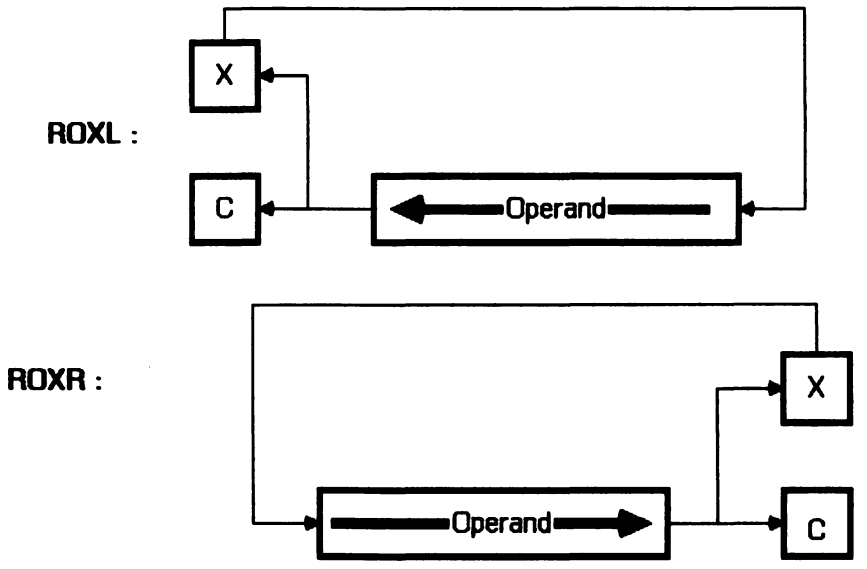


Figure 3-12 ROXL/ROXR

Forms:

ROXL Dn,Dm
ROXL #<*constant*>,Dn
ROXL *operand* ;any operand except Dn, An, PC relative, and immediate

and, of course, their complementary forms for rotate right:

ROXR Dn,Dm
ROXR #<*constant*>,Dn
ROXR *operand* ;any operand except Dn, An, PC relative, and immediate

System Control Operations

The 68000 chip has built-in means of dealing with exceptional situations such as: an overflow in a math operation (TRAPV), a subscript out of bounds (CHK), or a certain nybble ($A is most usual on the Macintosh) in the first position of an instruction. Most of these exceptions are handled automatically by the Macintosh system, or should never be used.

A table of addresses for routines is contained in memory to deal with each special case. Whenever the situation in question occurs, the processor finds the appropriate address from the table and performs the routine at that address. When any of these exceptional situations occurs, there is a standard procedure the 68000 performs. The program counter and status register are pushed onto the stack, the appropriate routine address is found, and execution continues at that address. You will see the word

Vector Number(s)	Address in Hex	Interrupt Name
0	0	Reset: Initial SSP
1	4	Reset: Initial PC
2	8	Bus Error (ID=02)
3	C	Address Error (ID=03)
4	10	Illegal Instruction (ID=04)
5	14	Zero Divide (ID=05)
6	18	CHK Instruction (ID=06)
7	1C	TRAPV Instruction (ID=07)
8	20	Privilege Violation (ID=08)
9	24	Trace (ID=09)
10	28	Line 1010 Emulator ($A Mac ROM traps)
11	2C	Line 1011 Emulator
12	30	(Unassigned, Reserved)
13	34	(Unassigned, Reserved)
14	38	(Unassigned, Reserved)
15	3C	(Uninitialized Interrupt Vector)
16–23	40–5F	(Unassigned, Reserved)
24	60	Spurious Interrupt (bus error during interrupt)
25	64	Level 1 Interrupt Autovector
26	68	Level 2 Interrupt Autovector
27	6C	Level 3 Interrupt Autovector
28	70	Level 4 Interrupt Autovector
29	74	Level 5 Interrupt Autovector
30	78	Level 6 Interrupt Autovector
31	7C	Level 7 Interrupt Autovector
32–37	80–BF	TRAP Instruction Vectors (32+trap #) (see Note)
48–63	C0–FF	(Unassigned, Reserved)
64–255	100–3FF	User Interrupt Vectors

"(Unassigned, Reserved)" means that if you use these interrupts some day Motorola will use them in a new chip and you will have to recode your application while angry users breathe down your neck.

Note: Trap $E (breakpoints) is TRAP #14 or interrupt 46.

When you see a "bomb" alert box and an ID = *nn* the number, *nn*, will usually come from the above list of interrupts. ID = 25 means out of memory or, sometimes, resource ID not found.

Figure 3-13 Interrupt Vectors Table

"vector" used to describe these addresses; it is said that the 68000 "vectors" to the appropriate address.

The $A (also called the 1010 emulator mode) exception is the only one used with real frequency on the Mac. Each exception has a number—this number is most frequently seen in the "bomb" alert box where an "ID=" number is given.

You will very rarely find yourself having to know about system control instructions unless you plan to do systems programming work on a 68000 machine. You will rarely need to know about these on the Macintosh. Perhaps the only one of these instructions that is used frequently is the CHK instruction, which is used to check for array subscript being out of bounds in various compilers.

These operations fall into three classes: trap generating, status register, and privileged.

Trap Generating Instructions

Only three trap generating instructions exist. CHK, which checks a register against bounds, TRAP, which generates a trap, and TRAPV, which traps on overflow.

CHK If you look at the output of many compilers you will find this instruction just before a subscripted variable is accessed when the range checking option is set. Its form is:

CHK *source*,D*n*

It only works on a word (subscripts must be in the range 0 through +32767). Typically the compiler sets the value as an immediate value if the arrays are static, meaning the dimensions can't change, or as a place in memory if the arrays are dynamic, meaning the dimensions can change.

CHK generates an exception processing sequence when triggered. The program counter and status register are pushed on the stack and the vector (address) set aside for the CHK instruction is the next place the program goes. Put in the more usual way, "the CHK vector is loaded into the program counter." We include the more usual way of saying this so that you won't be surprised when you encounter it in the literature.

Example: An array can only have a subscript that runs between 1 and 500.

```
SUBQ.W #1,D0    ;the subscript is in D0, subtract 1 so lower bound is zero
CHK    #499,D0  ;see if the subscript index is between 0 and 499
                ;note: 0—499 is equivalent to 1—500 after you subtract 1
                ;now use as an index into the array...
```

TRAP This instruction is never used on the Macintosh. To indicate a TRAP, the Macintosh uses an instruction that starts with an $A instead.

This is the 1010 emulator mode of the 68000 chip. For completeness we will describe the TRAP instruction. It uses the following form with a value from 0 to 15:

```
TRAP #<0-15> ;TRAP followed by an immediate value between 0 and 15
```

TRAP performs the usual exception processing; first the program counter and then the status register is pushed on the stack, then one of 16 vectors (addresses) is used to determine where to execute the next instruction. In some 68000 machines this mechanism is used to implement 16 different subroutines. See Figure 3-13, the Diagram of Vector Table for the 68000, and the discussion of traps for entry into the ROM for more information.

Example:

```
TRAP #floatadd ;do a floating point add on some 68000 machine...
```

TRAPV This instruction is never used on the Macintosh, either. If a TRAPV instruction is encountered and the overflow flag is set, you would vector to an address that is associated with the TRAPV instruction after pushing the program counter and status register. Obviously, the makers of the 68000 chip meant this to be used whenever there was a place where you had performed an arithmetic operation that would result in an incorrect result due to an overflow. Usually you print a message talking about an overflow in the last arithmetic process you performed (such as a picture of a bomb) and abort the whole process. The routine you would "vector" to would perform that function.

Form: TRAPV
Example:

```
ADD.B D0,D1 ;this will generate an overflow if the sign were wrong
            ;assuming both original numbers and the result are seen as
            ;signed.
TRAPV       ;an overflow is horrible! Abort everything and take
            ;appropriate action. If no overflow, go to next instruction
```

TAS The Test And Set (TAS) instruction is rarely used on the Macintosh. Later on, the hardware chapter will mention that this instruction can be used in conjunction with Direct Memory Access (DMA). Direct memory access means that you can get data out of memory while bypassing the 68000 completely; this procedure is usually used for interfaces with external devices such as hard disk drives that need high speed. Another use of this instruction is for synchronizing multiple processors; there is only one processor on the Macintosh.

TAS allows you to test a byte in memory (a flag usually) and set the zero and negative flags in the condition codes accordingly. The high bit of

the byte in memory is turned into a "1" to signal that this byte has already been tested. Since this instruction cannot be divided and since the Macintosh XL depends on dividing instructions for its virtual memory scheme this instruction should not be used in any program that will ever be used on the Macintosh XL.

Status Register Instructions

The Status Register contains the Condition Codes in its rightmost five bits—the status register is two bytes long. The status register is used to record the state of the system in general. The Condition Codes are used to communicate between instructions in your assembler programs. In these instructions CCR stands for Condition Code Register and SR stands for Status Register.

There are five status register instructions, none of which you will use that frequently. You might skim this section, however, so you can sound profound at the next hacker's party. The instructions are ANDI to CCR, ORI to CCR, EORI to CCR, MOVE to CCR, and MOVE from SR. Before describing these instructions, let's look at the form of the status register in more detail.

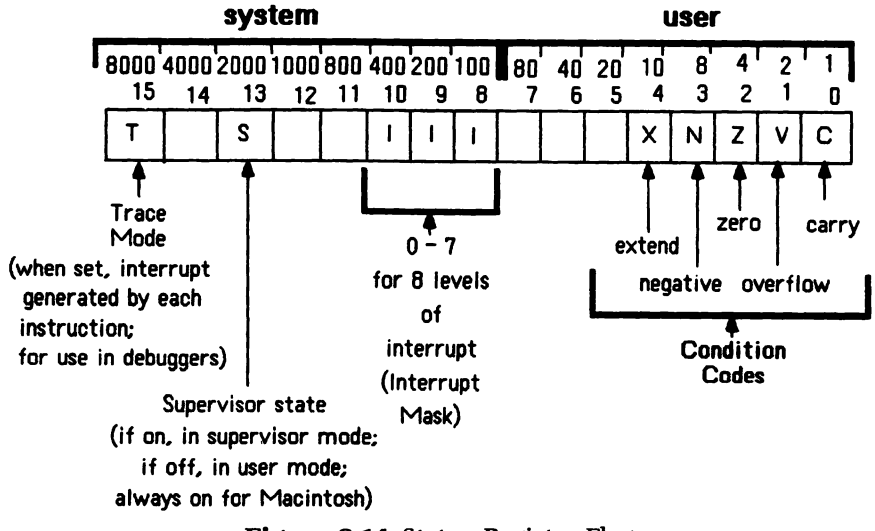


Figure 3-14 Status Register Flags

With the high bit as bit 15 and the low bit as bit 0 the status register contains:

Bit 15: Trace Mode flag. Trace mode allows you to single step through a program. Each instruction is followed by an exception processing for a trace. This mode allows you to keep tracing even when an interrupt occurs.

Bit 13: Supervisor State. The Macintosh is always in supervisor mode so all instructions are permitted. In some machines there are certain instructions reserved for the "supervisor" program that lowly "user" programs (the normal state) can't do. This state is made for multitasking or multiuser machines.

Bit 10, 9, 8: Interrupt mask. There are eight levels of interrupt. This allows some interrupts to have higher priority than other interrupts.

Bit 4: X or eXtend flag

Bit 3: N or Negative flag

Bit 2: Z or Zero flag

Bit 1: V or oVerflow flag

Bit 0: C or Carry flag

The remaining bits are unused.

The difference between moving the Status Register contents and moving the Condition Codes' contents is subtle. Two bytes are set aside for the Condition Codes just like the Status Register. The difference is that when you move the Condition Codes only the right five bits are used and the rest of the two bytes are ignored. When you move the Status Register all two bytes are moved and updated.

Since the User can only modify the Condition Codes, nothing that modifies the other parts of the status register can be done except in Supervisor Mode. On the Macintosh, which is always in Supervisor Mode, this is academic. The instructions that must be done in Supervisor Mode are called, appropriately, Privileged instructions.

ANDI to CCR In the ANDI to CCR instruction, the immediate data supplied in the source operand is ANDed with the Condition Codes (the low five bits of the Status Register). It is used to turn off Condition Code flags. If you are unsure about ANDI, see the section on the AND instruction.

Form: ANDI #<*source constant*>,CCR

As you would expect with an AND, X (eXtend) is cleared if bit 4 of the immediate operand is zero but is left unchanged if bit 4 is a one. The same rules apply for bit 3 (N or Negative), bit 2 (Z or Zero), bit 1 (V or oVerflow), or bit 0 (C or Carry).

Example:

```
ANDI #%11110,CCR ;clear the carry flag, leave other flags same
```

This example has a zero in bit zero, the carry flag, which means the carry flag will be cleared while the ones in the other bits means the other flags will remain as they were.

ORI to CCR ORI to CCR lets the immediate data supplied in the source operand be ORed with the Condition Codes (the low five bits of the Status Register). It is used to turn on Condition Code flags. If you are unsure about ORI, see the section on the OR instruction.

Form: ORI #<*source constant*>,CCR

As you would expect with an OR, X (eXtend) is set if bit 4 of the immediate operand is one but is left unchanged if bit 4 is a zero. The same rules apply for bit 3 (N or Negative), bit 2 (Z or Zero), bit 1 (V or oVerflow), or bit 0 (C or Carry).

Example:

```
ORI #%00100,CCR ;set the Zero flag, leave the other flags the same
```

Better coding would be:

```
ZERO.ON EQU %00100
ORI #ZERO.ON,CCR   ;it is much clearer what is happening here
```

EORI to CCR With EORI to CCR, the immediate data supplied in the source operand is EORed with the Condition Codes (the low five bits of the Status Register). It is used to flip Condition Code flags to their opposite. If you are unsure about EORI, see the section on the EOR instruction.

Form: EORI #<*source constant*>,CCR

As you would expect with an EOR, X (eXtend) is changed to its opposite if bit 4 of the immediate operand is one but is left unchanged if bit 4 is a zero. The same rules apply for bit 3 (N or Negative), bit 2 (Z or Zero), bit 1 (V or oVerflow), or bit 0 (C or Carry).

Example:

```
EORI #%00100,CCR ;change the.zero flag to its opposite (0 -> 1, 1 ->0),
                 ;leave the other flags the same
```

Better coding would be:

```
ZERO.ON EQU %00100
EORI #ZERO.ON,CCR   ;flip the zero flag to its opposite
```

MOVE to CCR Use MOVE to CCR to move the data in the source operand into the Condition Codes (the low five bits of the Status Register). The data size is always word, of which only the rightmost five bits are used. Only *An*, an address register, is not allowed as the source.

Form: MOVE <*source*>,CCR

Suppose the data at the address pointed to by A2 contains $0003 (rightmost 2 bits are on).

Example:

```
MOVE (A2),CCR ;and Carry bit (bit 0) being set, every other bit cleared
MOVE (A2),CCR ;this would result in the oVerflow (bit 1)
              ;and Carry bit (bit 0) being set, every other bit cleared
```

MOVE from SR MOVE from SR moves the data in the Status Register (two bytes or a word, as previously described) into the destination location. Address register, immediate, or PC relative addressing modes are not allowed as the destination (for obvious reasons).

Form: MOVE SR, <*destination*>

Suppose that the Status Register contains $2005 (Supervisor state, Zero and Carry flags on, everything else off).

Example:

```
MOVE SR,D3 ;now the low word of D3 contains $2005
```

Privileged Instructions

Privileged instructions can only be done in Supervisory mode. Although the Macintosh is always in Supervisory Mode (other 68000 computers, or even future Macintoshes, may not be) you will find yourself using these instructions only in rare circumstances.

ANDI, ORI, EORI and MOVE to SR The instructions ANDI to SR, ORI to SR, EORI to SR, and MOVE to SR are similar to the instructions which were described above for Condition Codes. The only difference is that the entire two bytes of the Status Register is updated rather than just the rightmost 5 bits which contain the condition codes. These instructions follow the form of those given in the previous sections so we will only repeat the forms and give an example of each.

Form: ANDI #<*source constant*>,SR
Example:

```
ANDI #$FFFE,SR ;turn off the Carry flag (bit 0)
```

Form: ORI #<*source constant*>,SR
Example:

```
ORI #$2000,SR ;turn on Supervisory Mode (bit 13)
```

Form: EORI #<*source constant*>,SR
Example:

```
EORI #$2000,SR ;change User state to Supervisory state, and vice versa turn
```

Form: MOVE <*source*>,SR
Example:

```
MOVE #$201F,SR
                ;turn supervisory state and all condition codes on, turn
                ;the interrupt priority level to one and trace mode off.
```

The remainder of the privileged instructions are MOVE USP for MOVE User Stack Pointer, RESET for RESET external devices, RTE for ReTurn from Exception, and STOP for STOP program execution.

MOVE USP Only a systems program would ever use the user's stack pointer. On the Macintosh you *never* want to do this because the system uses the System stack pointer (remember there are two A7 registers, a user register and a system register). Hence the user stack pointer is never used on the Macintosh. We suppose you could use this location as an extra register if you were sure that the operating system never has or would ever use it. Since few can predict what Macintosh systems programmer's will do, it is inadvisable to use these instructions. We include this instruction for the sake of completeness.

Form: MOVE USP, *An*
MOVE *An*, USP

RESET The reset line is asserted causing all external devices to be reset as at system startup. The effect of using RESET is the same as if you pressed the reset button on the Mac's programmer's switch.

RTE RTE stands for Return from Exception. The status register and program counter are pulled from the System stack. These values replace the current program counter and status register. This instruction is used like an RTS only at the end of an exception routine rather than a subroutine.

Form: RTE

STOP Stop means don't process any machine language instructions while waiting for an interrupt, trace, or reset exception to happen. The constant following STOP replaces the Status Register so that the state of the machine can be assured. The program counter points to the instruction following the STOP instruction so you can resume operation from that location in the code.

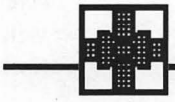
Form: STOP #<*constant*>
Example:

```
STOP #$2000 ;wait after making sure you are in Supervisor Mode by setting
            ;the Supervisor Mode flag (bit 13)
```

Unimplemented Instructions

When using the Macintosh you will find that you often encounter a pair of bytes that start with an $A in the high order nybble in a stream of instructions. The disassembler will list these bytes as a trap. Actually it is a signal for an unimplemented instruction.

The 68000 processor uses unimplemented instructions to create new commands. Any instruction that starts with an $A is sent to the address of a procedure that figures where the indicated system routine is located by way of an interrupt trap address. These instructions are called 1010 emulator mode. All the calls to the Macintosh ROM take place through the 1010 emulator mode.



Summary

The next chapter will use the instructions you have learned in this chapter to create small assembler programs. Be sure you understand all the more commonly used instructions before going on. If you want a quick summary of which op codes can be paired with which operand(s), see the table of op code, operands, and timings in Appendix B.

Memorizing op codes is only the start of doing assembler programming. The real learning begins when you read through programs written by other assembler programmers so you can feel how the op codes tie together. The learning continues when you write your first programs, of whatever size, using the knowledge of the 68000 op codes gained in this chapter.

CHAPTER 4

Sample Programs

In this chapter you will be introduced to actual program segments that are typical representations of 68000 programming. Once you have studied these program segments you will have a better understanding of how the various 68000 instructions learned in the prior chapter fit together. If the 68000 instruction formats are like English sentences (and the 68000 op codes like verbs) then this chapter will be dealing with whole paragraphs. To continue the analogy, we could say the program SimpleCalc, which you will see later and from which some of these examples are drawn, would be like writing an essay.

Let's start off with an in-depth look at a segment you have already seen; read through the following code and confirm for yourself how the routine works.

```
;-------------------------------------------------------------------
; string comparison subroutine
;-------------------------------------------------------------------
;
; Purpose: to compare 2 strings returning match/no match condition
;
; Register Usage:
;   Input  - A2        address of first of two strings to compare
;            A4        address of second of two strings to compare
;   Output -Zero flag  if zero flag true then strings are equal
;                      (If BEQ following routine taken then strings equal)
;                      if zero flag false then strings unequal
;   Registers Used - CC other than Zero flag is unknown
;-------------------------------------------------------------------
```

```
STRGCMPR  MOVEM.L  A2/A4/D0,-(SP)  ;save A2, A4, D0 on stack
          CLR.W    D0              ;set the low order word of D1 to zero
          MOVE.B   (A2)+,D0        ;move the length byte of the first
                                   ;string to D0 & move A2 to point to
                                   ;the first string byte
          CMP.B    (A4)+,D0        ;compare the lengths of the two
                                   ;strings and move A4 to point to the
                                   ;first string byte of the second string
          BNE      EXIT            ;if lengths different, exit with
                                   ;zero flag
                                   ;cleared
          BRA      COMPSTR2        ;let DBNE subtract one at start to turn
                                   ;length to index. If lengths are both
                                   ;zero, then strings are equal
COMPSTRG  CMPM.B   (A2)+,(A4)+     ;compare the two strings byte for
                                   ;byte
COMPSTR2  DBNE D0, COMPSTRG        ;keep branching until a point of
                                   ;difference found or strings are
                                   ;exhausted.
                                   ;if strings are same, then the zero flag
                                   ;is set upon coming here,
                                   ;if strings different, DBNE stopped
                                   ;looping when Not Equal (zero flag
                                   ;clear)
EXIT      MOVEM.L  (SP)+, A2/A4/D0 ;restore A2, A4, D0 from stack
          RTS                      ;exit with the zero flag indicating same
                                   ;or different strings
```

Although these are basically the same 68000 instructions that you saw used in Chapter 3 to do a string compare, there are differences. Better documentation appears at the start of this subroutine. You only need to look at the documentation at the front of this subroutine to see the routine's purpose, what registers are used, and what input you must have. Although the usual rule for commenting assembler is each line has a comment, this subroutine has more than its share. Once it's programmed, debugged, and documented, you no longer have to read through the instructions to see what the routine does. The routine is now a "black box" that magically does what you want; you don't have to think about it. Also, there are now MOVEMs at the start and end of the routine so that the only register affected is the Condition Codes (CC) register since the MOVEMs restore all registers used to their condition when you entered the subroutine. MOVEMs help to make this routine a black box since you don't have to worry about registers being changed by the subroutine.

Take three or four pairs of strings (empty strings, equal strings, unequal strings of same and different lengths) and step through the above procedures. By trying out this code, called "playing computer," you can get familiar with 68000 assembler in the speediest fashion. The easiest way to see how the code works is to take a sheet of paper and mark off an area for registers used (D0, A2, etc.) and to pen in their values as you analyze the function of the codes. You can then cross out and replace data in them each time they change. For those who have a blackboard and eraser, this may prove to be even better for "playing computer."

How could we improve this code? If we notice that the length byte is just another byte that has to be the same between the two strings we could code the main portion as follows:

```
STRGCMPR  MOVEM.L  A2/A4/D0-(SP)  ;save A2, A4,D0 on stack
          CLR.W    D0             ;set the low order word of D0 to zero
                                  ;since DBNE operates on a word
          MOVE.B   (A2),D0        ;move the length byte of the first
                                  ;string to D0
                                  ;note that the length byte includes itself
                                  ;so it is not decremented
                                  ;as is usual upon executing a DBNE loop
COMPSTRG  CMPM.B   (A2)+,(A4)+    ;compare the two strings byte for byte
          DBNE D0,COMPSTRG        ;keep branching until a point of
                                  ;difference found or strings are
                                  ;exhausted
                                  ;if strings are same, then the zero flag
                                  ;is set upon coming here,
                                  ;if strings different, DBNE stopped
                                  ;looping when Not Equal (zero flag clear)
EXIT      MOVEM.L  (SP)+,A2/A4/D0 ;restore A2, A4, D0 from stack
          RTS                     ;exit with the zero flag indicating same
                                  ;or different strings
```

Compare the new routine with the main section of the old one. It is by such observations that assembler code is reduced. In the new routine the core of a string compare is done in four instructions (with three instructions for good housekeeping)! You can, of course, go too far when reducing code and stray into a devious "trickiness," the disease of the assembly language programmer. Out of courtesy to yourself and others you should document your tricks especially well. If you don't go far enough in compacting code, however, your code will be cumbersome and run slow. As in all things in life, balance is important.

Another typical problem that you encounter in assembler is turning a 2- or 4-byte hexadecimal number in memory into a decimal number. When

you try to solve this problem you discover there are two ways of turning hexadecimal to decimal: divide the binary number repeatedly by ten resulting in decimal digits or multiply a decimal number repeatedly by two while looking at bits in the binary number.

Using the dividing by 10 method, you can use a table of the hex equivalents of 1, 10, 100, 1000, etc., then divide (via repeated subtraction) by each power of 10 starting with the highest possible power of ten; if you have a 2-byte hexadecimal number you can use the 68000 divide instruction. The 68000 divide instruction method is used in the SimpleCalc program described later in this book. Unfortunately, the 68000 divide instruction will not work with 4-byte hexadecimal numbers since the quotient which results when dividing by 10 will usually overflow the 2 bytes alotted for it.

Using the other method, you do repeated multiplications by two of a decimal number in an accumulator. You create consecutive powers of two in the accumulator by adding the number to itself in decimal; the accumulator is initialized to one. Then shift the hex number in the register right by one bit at a time so you can look at the bit indicating whether a one, two, four, eight, and so forth are in the number. If the bit that rolls off the right end of the register is 1 you add the current value in the power of 2 accumulator to a separate subtotal accumulator; if the bit is zero you loop back. You repeat for as many bits as there are in the hexadecimal number. At the end of the process, the subtotal accumulator holds the decimal equivalent of the hex number. This works best on a processor with fast decimal addition. Let's look closer at this method of conversion.

The following routine will be coded for speed, not compactness of code since it will likely be used many times during the running of the surrounding program and the average Macintosh has sufficient memory. Therefore we will code the five add decimal instructions, ABCDs, as straight line code rather than a loop. This routine will average about 6 milliseconds to convert a 32-bit hexadecimal number and 3 milliseconds for a 16-bit number.

```
;
; Convert hexadecimal to BCD
;
;
; Purpose: to convert a 4 byte hexadecimal number to Binary Coded Decimal
;
; Inputs: D0 contains the hexadecimal number to be converted
;
; Outputs: D1, D2 contain the BCD number that is the equivalent
;
; Registers Used: CC is undefined at exit
;
```

```
;  ____________________________________variables____________________________________
;
TEMPACC     EQU       -6                ;offset of temporary accumulator on
                                        ;stack
FINALACC    EQU       -12               ;offset of final accumulator on stack
;  ____________________________________code_____________________________________
HEX2BCD     MOVEM.L   D0/A1-A2,-(SP)    ;save D0,A1,A2 on the stack
            LINK      A0,#-12           ;get 12 bytes from the stack
                                        ;these 12 bytes on the stack will hold
                                        ;the final BCD result and the
                                        ;temp accumulator
            MOVEQ     #0,D1             ;clear both accumulators
            MOVE.L    D1,FINALACC(A0)   ;high 4 bytes of FINALACC
            MOVE.L    D1,FINALACC+4(A0) ;low word TEMPACC, high word
                                        ;FINALAC
            MOVEQ     #1,D2
            MOVE.L    D2,TEMPACC+2(A0)  ;low four bytes of TEMPACC
                                        ;now 6 bytes of FINALACC =0, 6
                                        ;bytes of TEMPACC = 1
            MOVE.L    #15,D2            ;loop counter - 1 (16x thru loop)
            CMPI.L    #$FFFF,D0         ;see if anything in high word
            BLS       HEXBLOOP          ;if not, branch
            MOVE.L    #31,D2            ;loop counter - 1 (32x thru loop)

HEXBLOOP    ASR.L     #1,D0             ;rightmost bit => C, X flags
            BCC       HEXBX2            ;just multiply by 2
                                        ;now you add the temp accumulator
                                        ;into the final accumulator
                                        ;this code will be executed, on
                                        ;average, 50% of the time
            LEA       TEMPACC+6(A0),A1  ;add temp accum to final accum
            LEA       FINALACC+6(A0),A2 ;point past end of BCD numbers
            ANDI      #$EF,CCR          ;clear extend bit
            ABCD      -(A1),-(A2)       ;faster than a loop (straight line)
            ABCD      -(A1),-(A2)       ;add in decimal with extend
            ABCD      -(A1),-(A2)
            ABCD      -(A1),-(A2)
            ABCD      -(A1),-(A2)
HEXBX2      LEA       TEMPACC+6(A0),A2  ;add temp accum to itself
            MOVE.L    A2,A1             ;doubling it
            ABCD      -(A1),-(A2)       ;faster than a loop (straight line)
            ABCD      -(A1),-(A2)
            ABCD      -(A1),-(A2)
```

```
ABCD      -(A1),-(A2)
ABCD      -(A1),-(A2)
DBRA      D2,HEXBLOOP        ;loop until D2 = -1

MOVEQ     #0,D1              ;clear out high word for aesthetics
MOVE.W    FINALACC(A0),D1    ;move BCD result to output
MOVE.L    FINALACC+2(A0),D2
UNLK      A0                 ;unlink 12 bytes on stack
MOVEM.L   (SP)+,D0/A1-A2     ;restore D0,A1,A2 from the stack
RTS
```

As with the string compare program, you are invited to work through this subroutine by putting a few hex values into it and seeing how they are transformed into BCD. Try the hex number $5, for example; this is binary %101. When the first rotate is performed the one bit comes off, the temporary accumulator is one; this gets added to the result accumulator which was initially zero. The temporary accumulator is doubled to two. The next time through the loop a zero bit is rotated off; the temporary accumulator is doubled to four. The third time through the loop another one comes off and the four in the temporary accumulator is added in to the result accumulator making the value there five. All the rest of the times through the loop the bit rotated off the end is zero. So the result accumulator exits with a BCD five. You should go through this example and actually see what is in each byte on the stack and in each register after every instruction. If you are more ambitious try a two- or three-digit hex number next.

This code seems satisfactory. All of the time is taken up in the ABCDs. There is a way of making an ABCD faster, however; you can use the data register, rather than the memory form, of ABCD. This would triple the speed of the ABCDs by reducing the required cycles from 18 cycles down to 6. We would need 10 data registers to do this (five for the temporary accumulator, five for the subtotal accumulator), there are only 8 in the system. That seems to kill the idea. It would take as long to move data in and out of temporary memory storage as we would save by doing the ABCDs in data registers.

However, SWAP instructions can flip two words in a data register and take only four cycles. This makes it possible to speed up the routine considerably. We can put the temporary accumulator in the low five words of five data registers and the final accumulator in the high five words of those same five data registers, but offset so that byte one of the temporary accumulator is matched with byte 2 of the final accumulator. We must do this so that we don't SWAP out the byte of the temporary accumulator

when we SWAP in the byte of the final accumulator! Each doubling now runs three times faster and even adding the temporary accumulator to the final accumulator takes place in 1¼₈th of the time (8 cycles for double SWAP plus 6 cycles for the ABCD versus 18 cycles with the old ABCD). The new code looks like this:

```
;-------------------------------------------------------------------------
; Convert hexadecimal to BCD
;-------------------------------------------------------------------------
;
; Purpose: to convert a 4 byte hexadecimal number to Binary Coded Decimal
;
; Inputs: D0 contains the hexadecimal number to be converted
;
; Outputs: D1, D2 contain the BCD number that is the equivalent
;
; Registers Used: CC is undefined at exit
;

;-------------------------------------- code --------------------------------
HEX2BCD   MOVEM.L   D0/D3-D6,-(SP)    ;save registers on the stack
                                      ;these 12 bytes on the stack will hold
                                      ;the final BCD result and the
                                      ;temp accumulator
          MOVEQ     #0,D1             ;initialize temp and final accum.
          MOVEQ     #0,D2
          MOVEQ     #0,D3
          MOVEQ     #1,D4
          MOVEQ     #0,D5
                                      ;now 6 bytes of FINALACC =0, 6
                                      ;bytes of TEMPACC = 1
          MOVE.L    #15,D6            ;loop counter - 1 (16x thru loop)
          CMPI.L    #$FFFF,D0         ;see if anything in high word
          BLS       HEXBLOOP          ;if not, branch
          MOVE.L    #31,D6            ;loop counter - 1 (32x thru loop)

HEXBLOOP  ASR.L     #1,D0             ;rightmost bit => C, X flags
          BCC       HEXBX2            ;just multiply by 2
                                      ;now you add the temp accumulator
                                      ;into the final accumulator
                                      ;this code will be executed, on
                                      ;average, 50% of the time
          ANDI      #$EF,CCR          ;clear extend bit
```

```
        SWAP    D5          ;final accum is in the high words,
        ABCD    D4,D5       ;one accumulator offset from temp
                            ;accum.
        SWAP    D5
        SWAP    D4
        ABCD    D3,D4
        SWAP    D4
        SWAP    D3
        ABCD    D2,D3
        SWAP    D3
        SWAP    D2
        ABCD    D1,D2
        SWAP    D2
        SWAP    D1
        ABCD    D5,D1
        SWAP    D1
HEXBX2  ABCD    D4,D4       ;double the temporary accumulator
        ABCD    D3,D3
        ABCD    D2,D2
        ABCD    D1,D1
        ABCD    D5,D5
        DBRA    D6, HEXBLOOP ;loop until D2 = -1
                            ;move final accumulator into D1 and
                            ;D2 in proper locations
        SWAP    D1          ;move final result to low bytes of Dn
        SWAP    D2          ;but D3 is already in proper place
        CLR.W   D3          ;so don't swap D3, but zero low byte
        CLR.W   D4          ;clear temp accum in D4
        SWAP    D4
        SWAP    D5

        MOVEQ   #$FF,D6
        AND.L   D6,D1       ;blank out high 3 bytes of D1
        AND.L   D6,D2       ;same for D2
        ROR.L   #8,D2       ;move low byte of D2 to high byte
        OR.L    D3,D2       ;and put into D2
        ASL.L   #8,D4       ;move final byte to second byte
                            ;position
        OR.W    D4,D2       ;put D4 byte into D2
        MOVE.B  D5,D2       ;put D5 byte into D2

        MOVEM.L (SP)+,D0/D3-D6 ;restore D0 thru D6 from the stack
        RTS
```

In this code the ABCD $-(A1),-(A2)$ of the prior example have been replaced with ABCD D*n*,D*n* which results in much faster code. The temp accumulator is in the low bytes of each data register, the subtotal accumulator is in the third byte of each data register. To transfer the subtotal accumulator to the low bytes of the data register we use the SWAP command.

The new code will take about two milliseconds to do a 32-bit convert—this is faster by a factor of three. Slightly more code is used to achieve this effect but it's worth it in terms of speed. This example and the string compare example show how it is often very worthwhile to consider the design of a piece of code carefully before actually executing it.

But it turns out there is a way that is faster and more compact yet. That way is to shift the bits out to the left rather than the right. The bit that is shifted out is added into an accumulator as a carry just before the accumulator is doubled. This method only uses one, rather than two accumulators! You may have seen this method when you calculate the value of a decimal number by taking a digit from the left of the number and adding it to an accumulator then multiplying this accumulator by ten.

This hexadecimal to binary procedure is the same procedure, only in binary rather than decimal. You calculate the value of a binary number by taking a digit from the left of the number and adding it to an accumulator then multiplying this accumulator by two. In the following code the main loop is one ASL, which takes the digit from the left, followed by a series of ABCDs, which multiplies the accumulator by two.

```
; ---------------------------------------------------------------
; Convert hexadecimal to BCD
; ---------------------------------------------------------------
;
; Purpose: to convert a 4 byte hexadecimal number to Binary Coded Decimal
;
; Inputs: D0 contains the hexadecimal number to be converted
;
; Outputs: D1, D2 contain the BCD number that is the equivalent
;
; Registers Used: CC is undefined at exit
;

; ---------------------------------- code ----------------------------------
HEX2BCD MOVEM.L  D0/D3-D6,-(SP)     ;save registers on the stack
        MOVEQ    #0,D1              ;initialize temp and final accum.
        MOVEQ    #0,D2
        MOVEQ    #0,D3
        MOVEQ    #0,D4
```

```
            MOVEQ     #0,D5
                                        ;now 6 bytes of FINALACC =0, 6 bytes
                                        ;of TEMPACC = 1
            MOVE.L    #31,D6            ;loop counter - 1 (32x thru loop)
            CMPI.L    #$FFFF,D0         ;see if anything in high word
            BHI       HEXBLOOP          ;if is, branch
            MOVE.L    #15,D6            ;loop counter - 1 (16x thru loop)
            SWAP      D0                ;move low to high word since bits out
                                        ;left side

HEXBLOOP    ASL.L     #1,D0             ;leftmost bit => C, X flags
            ABCD      D5,D5             ;double the accumulator
            ABCD      D4,D4             ;adding in 1 if the leftmost bit was 1
            ABCD      D3,D3             ;prior to the ASL.L
            ABCD      D2,D2
            ABCD      D1,D1
            DBRA      D6,HEXBLOOP       ;loop until D6 = -1
                                        ;move final accumulator into D1 and D2
                                        ;in proper locations
            SWAP      D3                ;put byte in low byte of high word
            ROR.L     #8,D2             ;move low byte of D2 to fourth byte of D2
            OR.L      D3,D2             ;and put D3 byte into third byte of D2
            ASL.L     #8,D4             ;move D4 byte to second byte position
            OR.W      D4,D2             ;put D4 byte into D2
            OR.B      D5,D2             ;put D5 byte into first byte of D2

            MOVEM.L   (SP)+,D0/D3-D6    ;restore D0, D3 thru D6 from the stack
            RTS
```

Try working through this method with a couple of small binary numbers. This third method results in faster and more compact code than any of the other methods; this code takes about one millisecond for a 32-bit conversion. The reason we give three different methods of achieving the same result is so that you can see how important thinking through the design of a program can be. The third method is simple, elegant, fast, and compact since it has the best design. By comparison the first design looks very clumsy. These three programs also give you a sense of how to think when programming in assembler.

The next series of code segments are adapted from the program SimpleCalc, a spreadsheet program that is restricted to integer values. SimpleCalc will be described in complete detail later, in Chapter 9.

The routine DoEvent and its subroutine MouseDown represent two different ways of creating a multiway branch in assembler. A multiway branch is like a case statement in Pascal or C; in BASIC an ON...GOTO statement is equivalent.

```
DoEvent                                 ;Process event
      MOVE    What,D0                   ;Type of event
      CMPI    #mButDwnEvt,D0            ;mouse button down is event 1
      BEQ     MouseDown
      CMPI    #keyDwnEvt,D0             ;key down is event 3
      BEQ     KeyDown
      CMPI    #autoKeyEvt,D0            ;auto-repeated key is event 5
      BEQ     KeyDown
      CMPI    #updatEvt,D0              ;update display is event 6
      BEQ     UpDate
      CMPI    #activateEvt,D0           ;activate/deactive is event 8
      BEQ     Activate
      CMPI    #abortEvt,D0              ;abort is event 9
      BEQ     Quit
NullEvent                               ;no event. check quit flag & exit
      MOVE    D5, CCR                   ;set carry if $01 set by
      RTS                               ;quit command
;_____________________________ EVENT TYPES _____________________________
MouseDown                               ;Find where mouse clicked
      CLR     -(SP)                     ;Clear space for integer result
      MOVE.L  Where,-(SP)               ;Mouse at time of GetNext Event
      PEA     EvtWind                   ;Window the event was in
      _FindWindow                       ;click point, ^window -> window part code
      MOVE    (SP)+,D0                  ;Result = section of window
      ASL.W         #1,D0               ;Window Part * 2 Bytes/Entry
      MOVE    WindowTable(D0),D0        ;Get offset from table
      JMP     WindowTable(D0)           ;Call subroutine

WindowTable

      DC.W    NullEvent-WindowTable     ;In Desk
      DC.W    InMenu-WindowTable        ;In Menu Bar
      DC.W    SystemEvent-WindowTable   ;System Window
      DC.W    Content-WindowTable       ;In Content
      DC.W    Drag-WindowTable          ;In Drag
      DC.W    NullEvent-WindowTable     ;In Grow
      DC.W    NullEvent-Window Table    ;In Go Away
```

In DoEvent, "What" holds an integer which describes an "event." The computer operator has just pressed a key or the mouse and the program must respond to it. "What" will be used to decide which way to branch. Since a CMPI instruction can't use a PC-relative address (which is what "What" is) we move "What" to data register zero.

After moving "What" to D0 there are a series of compare immediate instructions followed by branches on equal. These pairs of instructions say (using the first pairs as an example), "Is it an mButDwnEvt? If so go to the MouseDown routine. If not, is it a keyDwnEvt? If so go to KeyDown to handle it." If the subroutine DoEvent doesn't find any event worth a response, it "falls through" to the NullEvent label and returns to the loop that called it.

The Mousedown subroutine has a different way of branching to various routines which will handle the various types of mouse events. You can use this mode of branching when there is a function which returns consecutive integers starting at zero or one. This routine contains the first system macro you have seen in this book, __Find Window. For now, a system macro generates an $A-style trap (see unimplemented instructions at the end of Chapter 3, the 68000 Instruction Set Chapter) which operates just like any other subroutine.

__Find Window is passed information on the stack telling it where the mouse pointer was (on the screen) when the event occurred and in which window the event occurred. The two lines of code, MOVE.L Where,-(SP) and PEA EvtWind, push these values on the stack. __Find Window then magically returns with a word on the stack which encodes in which window and in which part of that window the event occurred. We say magically because __Find Window is a subroutine in the ToolBox ROM and we don't need to know how it performs its function; it's another "black box."

The MOVE (SP)+,D0 instruction that follows this __Find Window routine pulls the information encoding which window the mouse pointer is located off the stack and places it in register D0. The ASL instruction that follows shifts the data left by one bit effectively doubling the number in D0. This is necessary since each entry in the table that follows has two bytes per entry. These two instructions need study since they take values out of the table and jump to the appropriate place based on which part of the window was clicked into by the mouse:

```
MOVE  WindowTable(D0),D0 ; Get offset from table
JMP   WindowTable(D0)    ; Call subroutine
```

The two lines of code above taken from MouseDown subroutine involve PC relative addressing with displacement. You can tell since the register in parentheses is a data register and not an address register. The calculation of what gets moved to D0 in the first line of code can be figured out as follows:

This previous MOVE line from the example above is equivalent to

```
MOVE Windowtable-*+2(PC,D0),D0
```

where "*" means the current program counter at the start of this instruction relative to the start of this module (the assembler figures this out). So "Windowtable−*" means the assembler calculates how far from the start of this instruction to the start of Windowtable. To this value we add the program counter (PC) at this point in the code which means that the instruction is now pointing to the start of Windowtable in memory (the +2 is added since the PC is pointing *two bytes after* the start of the instruction). To the start of Windowtable in memory, we add D0 which contains the offset within the table of the data. So now the effective address of the left hand side of the MOVE is pointing at the data in the table we want. "Now move the data in the table to D0," says the MOVE instruction.

You can also think of Windowtable(D0) as "add D0 to the location of Windowtable when the program is run." So the above MOVE statement becomes "add the offset of the data we want to the address of Windowtable to get the location of the data in which we are interested; MOVE that data to D0."

The JMP Windowtable(D0) instruction takes the location of the start of Windowtable in memory (Windowtable − * + 2 + PC) and to this adds the data we pulled from the table which is the relative offset of the routine from the start of Windowtable (DC.W <routine>—Windowtable). The net result of this is the jump instruction is pointing to the routine in memory. Jump there! Now wasn't that simple! (If it wasn't simple you might read the code until you feel familiar with how the code performs.)

The next routine is also from SimpleCalc. This part of the program handles a key stroke entered by the user. There are also four routines to add, subtract, multiply, and divide which are involved with KeyStroke and which are interesting by themselves.

```
KeyStroke                ;User pressed a key
      MOVE.L  Message,D2;Get character record
                         ;Vector to operation or put digit in cell value
      CMPI.B  # '0',D2
      BCS     NotDigit   ;Not a digit
      CMPI.B  # '9',D2
      BLS     DigiKey
NotDigit
                         ;Check table for operation
      BSR     OperVect   ;Check operation table for key
      BEQ     BadKey     ;Ignore keystroke if not in table
```

```
                        ;Save address of operation. Store operation
                        ;in program. Then perform it.
        JMP     (A0)    ;go to address of operation
BadKey
                        ;Not in table so ignore key
        RTS
```

The character typed at the keyboard is inside "Message" as the low byte; this byte is next moved to D2. This character is tested by CMPIs (CoMPare Immediates) to see if it falls between 0 and 9. If the character is not numeric it falls through or branches to NotDigit. If it is numeric the program branches to DigiKey, short for "digit keyed."

In the NotDigit routine the first line does a Branch SubRoutine (BSR) to OperVect which runs through a list of operations and returns the address of the routine which will perform that operation. The zero flag tells whether an operation was found.

The following code takes the character typed at the keyboard and returns the location of the routine which will handle that character.

```
OperVect
;Return vector to operation from table
;       INPUT           D2 = Character to match
;       OUTPUT          D2 = Character matched
;                  Z flag -> character not found
;
        LEA   Opertable,A0
OpVecLoop
        CMP.B (A0),D2               ;Compare key stroke to table
        BNE   NextEntry
                                    ;Found It
        MOVE.W 2(A0),D0             ;Vector to operation
        LEA   Opertable(D0),A0      ;Actually LER
                                    ;Optable-*(PC,D0)
        RTS                         ;Return NZ
NextEntry
                                    ;Check for end of table &
                                    ;advance pointer
        TST.L           (A0)+
        BNE             OpVecLoop
        RTS                         ;Not found. Return Z flag set
```

```
OperTable
;  4 bytes per entry
;        byte 1 = ascii value of key
;        byte 2 not used
;        bytes 3&4 offset

        DC   '+'
        DC   AddOper-OperTable
        DC   '-'
        DC   SubOper-OperTable
        DC   '*'
        DC   MulOper-OprTable
        DC   '/'
        DC   DivOper-OperTable
        DC   '='
        DC   EqOper-OperTable
        DC   $0300          ; [Enter] key
        DC   Enter-OperTable
        DC   $0800          ; [BackSpace] key
        DC   Clear-OperTable
        DC   $1B00          ; [Clear] on 10-key pad
        DC   Clear-OperTable
        DC.L 0              ; End of table
```

OperVect looks through the OperTable to see if the operation is there. The OperTable is a series of 4-byte entries: the first byte has the ASCII value of the operation which is compared to the key stroke entered. The second byte is unused. The third and fourth bytes have the offset to the routine that performs the operation in the same format as we have seen before in WindowTable.

OpVecLoop starts with LEA OperTable,A0 which moves the address of OperTable at the time the program is run into A0. CMP.B (A0),D2 compares the key stroke in D2 to the byte pointed to by A0, which points to the first byte of an entry in OperTable.

If D2 and A0 are unequal a branch to NextEntry is performed. In NextEntry the entire OperTable entry is TeSTed for zero which indicates the end of the table and A0 is simultaneously bumped to the next entry in the table. If the address in A0 is not the last entry, the program loops back via BNE OpVecLoop to the start of the loop. If this is the last entry then the zero flag is set and RTS brings the program back to the routine that called OperVect; the zero flag tells that routine no match was found.

If the CMP.B (A0),D2 compares equal then we have found a match for the key stroke in the table. We move the word that is offset by 2 from the start of the entry to D0 (MOVE.W 2(A0),D0); this is the offset to the subroutine which will perform the operation. We then do an LEA Opertable(D0),A0 which we recognize as a PC relative with displacement since the register in parenthesis is a data register. This statement means take the address of OperTable at the time the program is run—add it to D0 which contains the offset from OperTable to the routine in question. This results in the address of the routine called at the time the program is run and moves it to A0. Since the last instruction that changed the zero flag was a MOVE.W of a non-zero value the RTS which exits the subroutine has the zero flag cleared signalling that an entry was found, exiting the OperVect subroutine.

Upon return from the subroutine the zero flag is checked by BEQ BadKey which branches to BadKey if the zero flag is set. BadKey simply returns from the KeyStroke subroutine with an RTS and so does nothing with the keystroke. If the keystroke represents an operation recognized by OperVect then the BEQ BadKey is not taken; instead JMP (A0) jumps the program to the selected subroutine using the address in A0.

```
DigitKey
; Digit typed in D2.        Add it onto end of number in (A3)
      MOVEQ   #$0F,D0
      AND.L   D0,D2         ; Clear upper nibble & junk
      MOVEQ   #10,D1        ; number base
      MULS    (A3),D1       ; Current value times 10
      BPL     DigiAdd       ; Is the cell number negative?
      NEG.L   D2            ; then increment is negative too
DigiAdd
      ADD.L   D2,D1         ; plus key stroke
      MOVE    D1,(A3)       ; Save value
                            ; now check for overflow
                            ; bits 15 through 30 must be the same
      LSR.L   D0,D1         ; shift down 15 bits
      ADDQ.W  #1,D1
      BEQ     DigiOK        ;OK positive number
      SUBQ.W  #D1
      BEQ     DigiOK        ; OK negative number
                            ; overflow. clear to zero and start over
      CLR.W   (A3)
DigiOK
      RTS
```

Digikey uses the algorithm we mentioned earlier when talking about mulitiplying an accumulator by ten before adding the next leftmost digit. Since digits will be entered from left to right this is the algorithm that must be used. First, we use a mask and move it to D0 to clear the high nybble of a byte. When we AND.L this mask with D2, which contains the digit entered, we have the binary value of the digit. Then we multiply the current value, addressed by A3, by ten and place the result in D1. If the number is negative, then we must add a negative digit so we NEG.L D2. Finally we add the digit into the accumulator in D1. Then we move the accumulator, D1, back to the current value, addressed by A3.

Next the program checks for overflow, which occurs when the number can't be contained in two bytes. We shift all the bits in D1, the accumulator, right 15 bits and replicate the high bit as we shift (D0 contains $0F) so the highest bit of the number is now in the lowest bit and any overflow bits are now in the next to lowest bit and up. If the number was negative, all the bits should be one bits if there was no overflow. When we add one the result should be zero. If the number was a positive number all the bits should be zero if there was no overflow. When we subtract one to counteract the one we just added, the result should be all zeros. If it is, we branch to DigiOK, and return from the subprogram; the number didn't overflow the register. If there is overflow, we clear the number, in (A3), to zero and return from the subprogram.

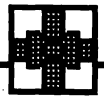
Four sample operations: add (AddOper), subtract (SubOper), multiply (MulOper), and divide (DivOper) are included in this chapter so that you can see how these operations actually look in some sample code.

```
AddOper
; Add the selected cell into the accumulator
     MOVE.W  (A3),D0
     ADD.W   D0,(A6)
     RTS

SubOper
; Subtract the selected cell from the accumulator
     MOVE.W  (A3),D0
     SUB.W  D0,(A6)
     RTS

MulOper
; Multiply the accumulator by the selected cell
     MOVE.W  (A3),D0
     MULS    (A6),D0
     MOVE.W  D0,(A6)
     RTS
```

```
DivOper
; Divide the accumulator by the selected cell
     MOVE.W  (A6),D1
     EXT.L    D1
     MOVE.W  (A3),D0
     BEQ      DivErr
     DIVS     D0,D1
     MOVE.W  D1,(A6)
     RTS

DivErr                    ; Divide by zero
     SWAP  D1             ; Return largest magnitude possible
     EORI.W  #$7FFF,D1
     MOVE.W  D1,(A6)
     RTS
```

The four operations take place using the selected cell, pointed to by A3, and an accumulator, pointed to by A6. The code is straightforward enough for add, subtract, and multiply that you should be able to examine them yourself to see what they do. Let's take a closer look at the division routine, however.

DivOper moves the accumulator value to D1. It then extends the word in D1 into a long word. MOVE.W (A3),D0 moves the selected cell value to D0. If the selected cell is zero, that's an error; a BEQ DivErr catches that error. If the selected cell is not zero we do the division. DIVS D0,D1 divides D0, the selected cell value, into D1, the accumulator value, and places the result in D1. This result is then moved back to the accumulator itself with MOVE.W D1,(A6).



Summary

This completes the examples chapter. You've now had a chance to examine the workings of typical routines as you might program them in 68000 assembly language. Assembly language routines are most often made up of many small intricate steps, and we hope the examples have given you a chance to think through several routines before you begin constructing your own. The next chapter on the 68000 hardware describes what you, as a programmer, need to understand about the Central Processing Unit of the Macintosh.

CHAPTER

5

A Programmer's Overview of the 68000 Hardware

A computer may appear quite different to a programmer than a hardware engineer. Figure 5-1, the Programmer's Block Diagram, shows the 68000 chip from a programmer's point of view. From a hardware perspective, we would be concerned with the physical size of data pathways, the electrical polarity of signals and the actual sequence of events. The programmer sees the data structures, logical meaning of signals and the logical sequence of events. We are going to examine the 68000 from the programmer's point of view.

As you remember from Chapter 1, there are 16 general purpose registers in the 68000. Eight of these are Data registers, called D0 through D7. The other eight are address registers, named A0 through A7. The A7 register is also the Stack Pointer. It behaves differently than the other Address registers. Both Address and Data registers are 32 bits wide. Although they can hold the same information, they cannot be used interchangeably. The dotted lines in Figure 5-1 divide the registers into addressable units. The Data registers can be addressed as 8, 16 or 32 bits. The address registers can only be addressed as 16 or 32 bits.

All data going into or out of the CPU passes through one data path. This path varies in width depending on the instruction. Different forms of each instruction require a data path that is 8, 16 or 32 bits wide. During a two-address instruction, such as a MOVE, this one data path is used twice.

The address bus tells where data is coming from or going to. The 68000 provides an address every time data passes over the data path. The address will select either a location in memory or a physical device. It doesn't make any difference to the 68000 whether hardware or memory is at the address. The connections to external devices, such as the printer,

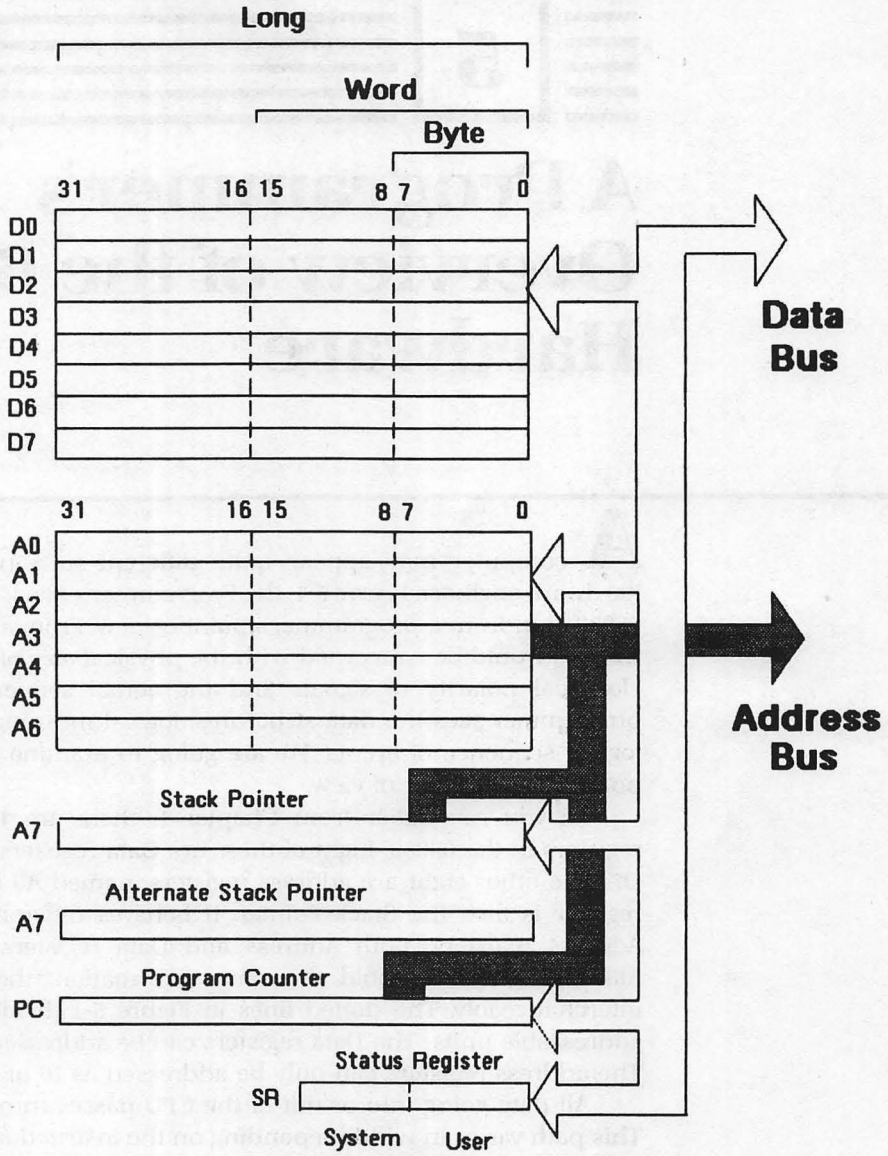


Figure 5-1 Programmer's Block Diagram

have memory addresses in the Macintosh. Reading or writing to that address actually causes the data to go out or come in over the connecting cable. This method of communicating with peripheral devices is called memory-mapped I/O.

Besides the Data and Address registers, there are two other important registers. The Program Counter points to each memory word of the program during execution. The Status register holds the state of interrupts, tracing and the arithmetic Condition Codes, such as Zero, Minus and Overflow. The Operating System can use the Alternate Stack pointer to keep its own stack in a separate location from the application stack. The Alternate Stack pointer (also called the User Stack Pointer or USP) won't be important to us.

Detailed Look at Architecture

Now let's take a closer look at the hardware which implements the programmer's world. Remember that we are still taking a software viewpoint, so don't try to design hardware based on this discussion. Figure 5-2 shows the electrical connections of the 68000.

Data Bus

The sixteen line *data bus* forms the data path. All data going into or out of the 68000 passes over the Data Bus. As mentioned before, the data path may be up to 32 bits wide. The Data Bus has to be used twice to transfer a long word (4 byte). The entire bus is used once to transfer a 16-bit (2-byte) word. Only one half of the bus is used for a byte-sized operation.

The data bus is divided into upper and lower halves. Each byte of memory is connected to only one half. Data going to a byte with an even numbered address travels over the upper half of the data bus. Data going to an odd location travels over the lower half. The address of an even, upper-half byte, is one less than the address of the corresponding location on the lower half. When an entire word is addressed, both halves are used simultaneously.

During a sixteen-bit transfer, the lower half of the data bus transfers the least significant byte (LSB) of a word. This means bits valued from one to 127. The lines of the lower half are marked D0 to D7 in Figure 5-2. These numbers correspond to the power of two that each line represents. The upper half transfers bits valued from 2^8 through 2^{15} during a 16-bit data transfer.

These lines are marked, quite logically, D8 to D15 in the figure. This convention makes it easy for hardware engineers to remember which bit each line represents.

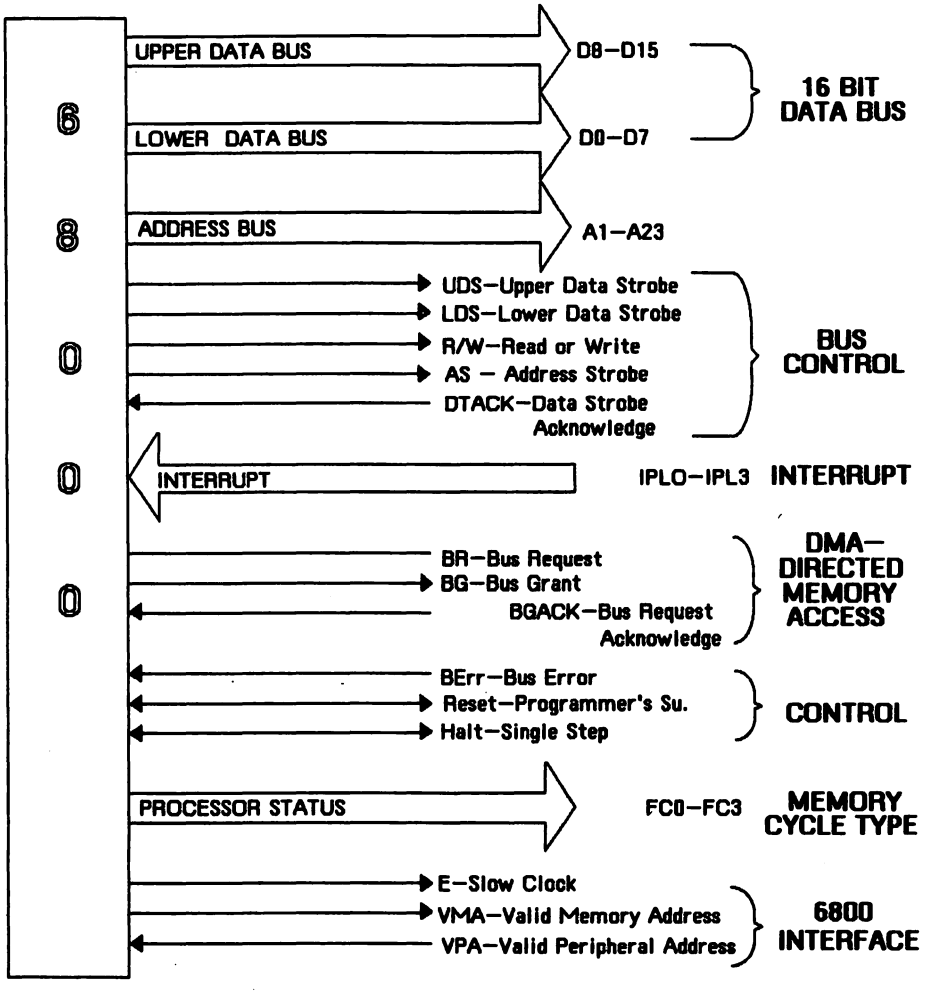
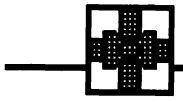


Figure 5-2 Programmer's View of 68000 Electrical Connections

The upper half of the data bus transfers the high-order bits during a word-length operation. This is why the signals are named D8 to D15. This half of the data bus is the only path to the even-numbered addresses. So during an eight-bit transfer to an even memory location, the upper half carries the only byte of data. Assume for an example, that the word $1234 is being stored at location $100. The byte $12 goes over the upper half to address $000100, while $34 goes over the lower half to $000101. But if we store just the byte $12 to address $100 then the lower half is not used at all, and again the upper half carries $12 to the even-numbered location.



Summary

Let's review what we know about the data bus so far. The lower half connects to odd numbered addresses and carries the least significant byte (LSB). The most significant byte (MSB) travels on the upper half, which connects to even numbered addresses. The even numbered address is less than the corresponding odd number. But are these important facts to the programmer? Yes, they are! They are very important because they mean a word must always start on an even address, and they let us know that the highest numbered address of a word will be odd, and contain the least significant byte. If we ever try to address a word by an odd address, the Macintosh will halt with an Address Error!

Address Bus

The signal lines labeled A1 through A23 in the figure are the *address bus*. The address bus selects the source or destination location of data moving over the data bus. Since the 68000 can address 16 megabytes(2^{24}) why are there only 23 address lines? The reason is the address bus addresses 16-bit words of memory, rather than bytes. When single bytes are addressed, the proper word is selected by the address bus, and the upper or lower byte of the data bus is used. Notice that the address lines are numbered from A1 instead of A0. Like the data bus lines, each address bus line is numbered to correspond to the bit it carries. The "1" bit, meaning even or odd, would be A0.

Bus Control Lines

The *bus control lines* coordinate the 68000 signals with the external memory and hardware. These signals indicate what data is on the buses, when the data is available, and what it should be used for. Two of the lines, called UDS (Upper Data Strobe) and LDS (Lower Data Strobe), are important to understand the way the 68000 addresses memory.

The Upper Data Strobe and Lower Data Strobe lines enable the upper and lower halves of the data bus respectively. These signals are marked UDS and LDS in the Pinout Figure 5-2 and in the Memory Selection circuit diagram in Figure 5-3. When LDS is on, it means that data is going to travel over the lower half of the bus. As mentioned above, and as visible in Figure 5-3, this means an odd memory location will be selected. Conversely, when UDS is true, data will travel over the upper half to an even memory location. The 68000 can address two bytes at the same time, by using both

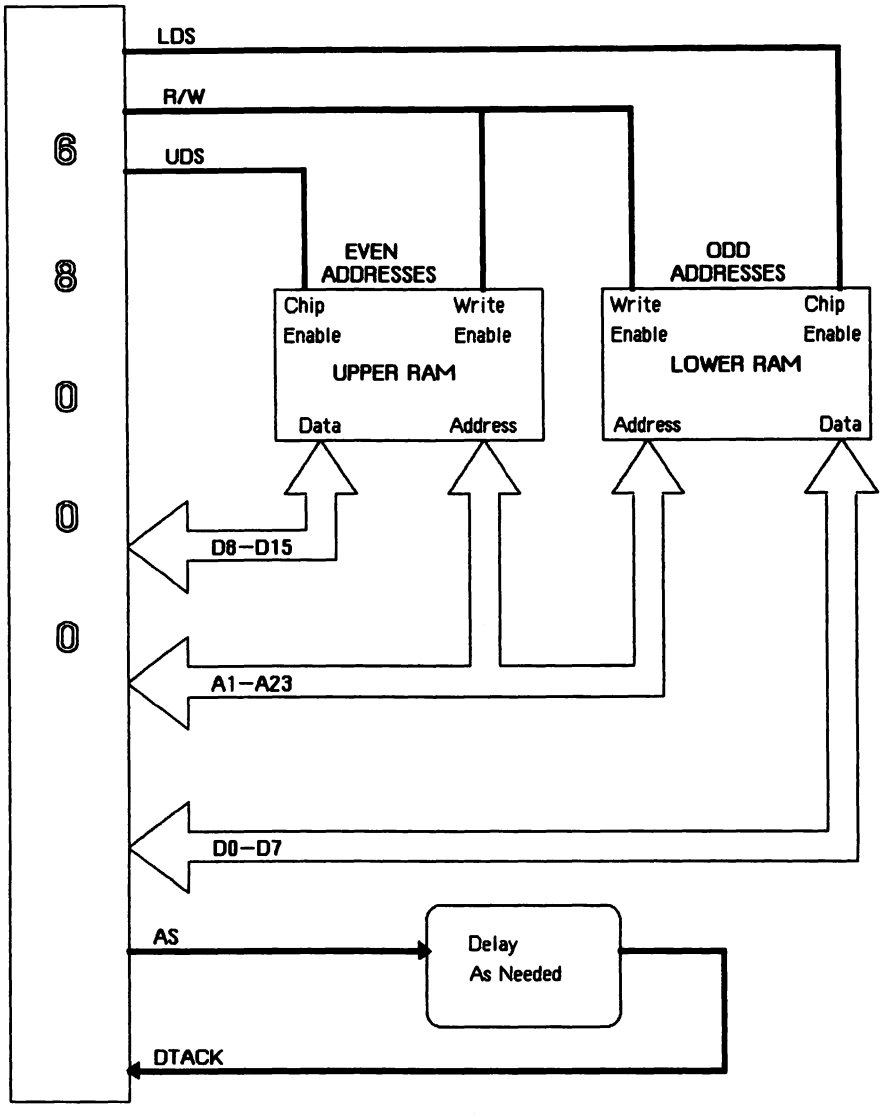


Figure 5-3 Memory Selection

LDS and UDS, provided that both bytes are in the same word. To access a word, we always use the even address, the address of the MSB.

The Read/Write line (R/W) indicates whether the address is for input or output. In the Memory Selection circuit, we can see this line is used to tell the RAMs whether they should read or write.

The Address Strobe line (AS) becomes true when the address bus is valid. The AS line approximates the OR function of LDS and UDS. Though

the timing is a little different to help hardware designers account for circuit delays, we can think of AS as turning on whenever the LDS or the UDS is active.

The Data Acknowledge input (DTACK) tells the 68000 that the memory or hardware is through with the data. By delaying DTACK for a few clock cycles, slow peripherals can slow down the 68000 to their own pace, while fast devices can go full speed ahead.

Interrupt Requests

The 3 *interrupt lines* (IPL0, IPL1, and IPL2) can make the 68000 vector to an interrupt routine. There can be seven different priorities of interrupt. Each of the seven encodes its value on IPL0 through IPL2. Priority zero means there is no interrupt occurring. The 68000 will execute the interrupt if its priority is greater than a certain number, called the Interrupt Mask, in the Status Register. The highest level, priority seven, is always executed. The Programmer's Switch on the Macintosh and the 10-Key Minus on the Lisa generate a Level 7 interrupt. If you press the Interrupt on the Programmer's Switch you will probably see a System Error ID = 13. If you have the "Inside Macintosh" manual from Apple, you can look up the meaning of the error. The meaning is "Spurious Interrupt."

DMA lines

The Direct Memory Access lines (DMA) let the hardware read and write data directly from memory. The Disk controller uses them to access memory faster than it could by going through the 68000. Since the goal of DMA is to bypass the CPU, it is usually transparent to the application programmer. We won't talk about DMA in this book. But if you are curious about it you should see the TAS instruction in Chapter 3, The 68000 Instruction Set.

Control

The *control lines* handle special situations, such as starting or restarting the processor. You often see the effect of these signals when you are testing your program, although the end user of your code should not. The control lines indicate programming errors and can restart the program after it crashes.

The Bus Error, BErr line, says something is wrong with the location selected by the address bus. The hardware uses this line to indicate trouble, instead of the DTACK line used when the memory cycle is complete. A bus error means you have gone into locations you should not be using.

The Halt line stops the processor. It can be used to single-step the processor if it is connected to the right hardware. The halt line cannot be used to single-step the 68000 in Macintosh, however, because it is connected directly to the Reset line, which is discussed next.

The Reset line restarts the system. It is connected to the peripherals as well as the 68000. The Reset connection is a type of two-way signal, called "Open Drain." The RESET instruction from the 68000 activates the Reset line to reset the peripherals. But a signal on the Reset line which the 68000 did not generate causes the 68000 to restart, as well as resetting any peripherals connected to the line. Once reset, the 68000 begins executing the boot code all over again. It starts from the boot address at memory location zero, which it used when the system was first turned on. The Programmer's Switch on the Macintosh can activate the Reset line. You can do the same thing by turning the power off and on.

State Lines

The three *state lines* indicate the type of memory cycle going on. They are not important to application programming on the Macintosh. But an explanation of these signals provides some extra insight to the 68000.

The FC0 and FC1 state lines indicate whether the address on the address bus applies to program code or data. When the 68000 is fetching the words of an instruction to execute, the FC1 line is on. Whenever the 68000 is reading or writing data, the FC0 line is on. Both lines are on while an interrupt is being acknowledged.

The hardware could use these signals to keep the program code separate from the data space. A common organization of memory requires them to be separate. The Macintosh does not do this, however. For good or bad the application can have code and data intermixed in memory!

When the operating system code is executing, the 68000 can be placed in Supervisory mode. Then the code can use the privileged instructions, to control interrupts and peripheral devices. The privileged instructions are described in Chapter 3 (The 68000 Instruction Set). Reserving these instructions for the operating system keeps the application code from affecting things it should not be allowed to change. But what if the system designer needs to keep the application from accessing certain memory locations? To do this, the designer needs a hardware signal that the 68000 is in Supervisory mode. The FC2 line provides this signal. When the 68000 is in Supervisory mode, FC2 is true whenever memory is accessed.

Those are the functions of the State lines. But remember we said they were not important to us? The FC0 and FC1 signals are not important because the Macintosh allows intermixing data and code. The FC2 signal is not used because the Macintosh is always in Supervisory mode! That is why the Alternate Stack Pointer is unimportant. And why, as we shall see, an incorrect assembly language program can wipe out the entire Operating System.

6800 Interface lines

The 6800 Interface lines are of little concern to the application programmer of the Macintosh, but they are used by the ToolBox ROM routines.

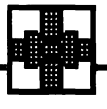
These lines let the 68000 be used with older, 8-bit, peripheral chips, that were designed originally for use with a 6800 processor. Most of the 8-bit devices are also slower than the 68000 so a special clock line, designated E, is part of the 6800 Interface. The clock signal on the E line is 1/10 the speed of the 68000 clock!

Many 16-bit systems will use some 8-bit devices. The Macintosh is no exception. The 6522 Versatile Interface Adapter and the 8530 Serial Communications Controller in the Macintosh are both 8-bit devices. The examples in this book always go through the ToolBox routines in the ROM to access these devices. But System Programmers who have to write routines to interface 8-bit chips have a special instruction at their disposal! The MOVEP instruction, described in Chapter 3 (The 68000 Instruction Set) helps them assemble the 8-bit data into 16-bit memory words.

How Does It Do All of That?

Now we have a general overview of the 68000 hardware. A brief look at the number and complexity of the instructions makes one wonder. How can so many parts be interconnected to perform so many different tasks? The answer does not show up on our Programmer's Block Diagram because it is not within the programmer's control. The control circuitry of the 68000 is implemented with micro-code.

The micro-code implements each 68000 instruction. The control circuitry which executes the micro-code is like a tiny, primitive CPU within the 68000. Executing a 68000 instruction causes a short program to run in the control circuitry. Running this program has the actual effect of the 68000 instruction. Micro-code is written in an assembly language of its own. The language is unique to the 68000, as in every other type of micro-coded processor. Micro-code instructions control the CPU at the very lowest level. They open and close gates or switches within the CPU to connect the different parts in different arrangements. It is here that hardware and software are one. Since you cannot program the micro-code built into the 68000 in the Macintosh, we won't say anything more about it in this book.



Summary

Now that you have completed this chapter you have a better understanding of 68000 assembler instructions and the workings of the Macintosh. The next chapter introduces you to the Macintosh Tools: the assemblers, linkers, and resource compilers.

CHAPTER

6

Macintosh Tools

This chapter explains the various tools available to the assembly language programmer on the Macintosh. Here we describe the assembler in fine detail: how to set aside data areas, segment your programs, and use the powerful macros and conditional assembly which makes assembly language unique. The resource compiler will also be described in detail, each one of the 12 types of resources allowed by the 68000 Macintosh development system will be shown and examples given of each. Finally, EXEC files will be examined so the series of steps necessary in a cycle of assemble/debug/assemble/debug is made quicker and easier.

Keep in mind that this is *not* meant to be a comprehensive description of the 68000 development system but rather a supplement that ties in concepts developed elsewhere in this book with a typical development system. The documentation that comes with the development system tells how the system works—here we try to tell why the system has certain features, how these features connect with other features, and why those features were included in the system. In other words, we try to make the system comprehensible to every assembly language programmer.

The Mac and Lisa Assemblers

This section describes how to handle data and memory with two different Assemblers. It includes dividing your large programs into smaller sections called segments. Some parts of the discussion are written twice—once for each assembler. If you are using only the Lisa Workshop, or only the Macintosh 68000 Development System you may want to skip the instructions for the other system. If so, just look ahead to the appropriate section heading.

You can create data space in two places in memory when you are coding in assembly language. You can define constant data right in the code of your program, or you can allocate space from the Global Data Area. The Global Data Area is a place in memory that is accessed off the A5 register; data that is shared "globally" between segments and programs is accessed here. When data appears in your code, do not modify it. It will not be very convenient to access. The data may even revert unexpectedly to its initial value if your program uses segmentation!

Macintosh Assembler Data Directives

You can create data within your program by using the "DC" (Define Constant) instructions.

DC instructions:

```
[label]  DC.B value [,value]...
[label]  DC.W value [,value]...
[label]  DC value [,value]...
[label]  DC.L value [,value]...
```

The DC creates constants at the current location in the code. Multiple values are separated by commas. The DC.W instruction creates one or several words. Each word is a two-byte integer with the value defined in the instruction. The DC.B and DC.L instructions are similar, but they create one- and four-byte values respectively. If you use "DC" without a length suffix, it is taken to mean "DC.W." You can use strings in the DC commands by putting characters between single quotes:

```
DC 'String in DC.W instruction'
```

Here is another example of DC:

```
Mylabel DC.W  $10E,20
```

This would create four bytes of data inside the program as follows: $010E0014

DCB (Define Constant Byte) instructions:

```
[label]  DCB.B length,value
[label]  DCB.W length,value
[label]  DCB length,value
[label]  DCB.L length,value
```

The "DCB" instruction creates areas in your code, filled with constant values. The "length" is the number of bytes, words or long words to create and fill. The "value" is an expression for the number of proper length, that is used to fill the area. If just "DCB" is used, the instruction defaults to

"DCB.W." Since we do not recommend that you modify data mixed in with the code, you may not use the "DCB" instruction very often.

DS (Define Storage) directives:

```
[label]   DS.B  length
[label]   DS.W  length
[label]   DS    length
[label]   DS.L  length
```

You can create data space within the Global Data Area by using the "DS" directives. The "length" refers to the number of bytes, words or long words of space allocated. If no suffix is used, DS.W is assumed. You access space created by a DS instruction relative to address register A5. The *label* will evaluate as the distance from A5 to the space assigned to that variable. When your program starts up, A5 will be set to point to the Global Data Area. The label on the DS instruction becomes the offset for a Register Indirect with Displacement addressing mode. Although this offset may be a negative number, successive DS instructions allocate successive addresses in the code. In the example below, the word SHORT and the first two-bytes of the long word, LONG, are set to 0:

```
SHORT    DS.W 1       ; two-bytes in Global Area
LONG     DS.L 1       ; four-bytes in Global Area
CLR.L    SHORT(A5)    ; clear SHORT and part of LONG
```

The Global Pointer, A5, is used by calls to the ToolBox. It is also used for the Jump Table that communicates between segments. Even if you do not use the DS instructions, leave A5 intact!

When you use DS, DC and DCB commands, the data may be automatically aligned. If the command is word or long-word sized, the first byte of the data will be on an even address, a word boundary. An extra zero byte will be added before the data if the address was an odd location. The label of the command will always point to the first data word. If you use multiple strings in a DC.W, and some of them have an odd total length, the assembler will insert zeroes between your strings to make each one start on a word boundary! Byte-width commands do no alignment. They only create the actual length you declare.

String literals can be used in LEA and PEA instructions. They are designed to duplicate the Pascal format. This is a byte of length, followed by the data. The length byte is always on a word boundary. The string data will be placed at the end of the code. Normally, a string literal will create a structure suitable for passing to the Pascal-like interfaces of the ToolBox. This instruction,

```
PEA     '123'
```

is equivalent to these three instructions:

```
        PEA     String
                ...
String  DC.B            3   ;3 bytes of data
        DC.B    $31,$32,$33 ;ASCII for "123"
```

Be sure you don't confuse the Pascal literal strings with immediate mode integers expressed as characters in quotes. Thus,

```
MOVE.L # 'ABCD',D0
```

is exactly the same as:

```
MOVE.L #$41424344,D0
```

If you write,

```
LEA       'ABCD',A0
MOVE.L  (A0),D0
```

you will not get the same effect at all. The latter instructions will leave D0 containing $03414243 instead of the desired string, and will leave a copy of the string at the end of the code.

A string in a DC instruction does not normally generate a Pascal string structure. The default form of strings in DC instructions is simply the ASCII values of the characters in the string. You can, however, change the string format for both types of string. You use the STRING__FORMAT directive to do this.

STRING__FORMAT directive:

STRING__FORMAT *value*

This sets the format for literal strings and strings in DC commands. The "value" determines the effect. The literal string formats can be set to end the string with a zero byte and not to have any length byte. The DC instructions can be set to generate the Pascal type of string. The three formats for a string look like this:

```
Pascal  DC.B  $03,$41,$42,$43  ;String or literal "ABC" with length byte
ASCII   DC.B  $41,$42,$43      ;String "ABC" in simple ASCII
CString DC.B  $41,$42,$43,$00  ;Literal "ABC" ends with a zero
```

The "value" determines the formats for both types of string according to the table below:

VALUE	LITERAL STRINGS	DC STRINGS
0	Ends with zero byte	ASCII only
1	Starts with length byte	ASCII only
2	Ends with zero byte	Starts with length byte
3	Starts wtih length byte	Starts with length byte

EQU instruction:

label EQU *value*

The EQU instruction creates a label equal to a certain constant or to a certain address in the code. The "value" is an expression for this address or number.

SET instruction:

label SET *value*

The SET instruction sets the value of a label to a constant or an address. The "value" is an expression for this address or number. Unlike the EQU instruction, SET can be used on the same label more than once. All code assembled between two SET instructions for the same label, uses the value the preceding instruction assigned.

Lisa Assembler Data Directives

.BYTE instruction:

[*label*] .BYTE *value*[,*value*]...

The .BYTE instruction creates bytes of data in your code. Each *value* must be an integer between −128 and 255, inclusive. The negative values are in two's-complement format. Therefore, minus one creates $FF. Negative 128 is the same as positive 128. You can use any number of values on a line but separate them by commas.

.WORD instruction:

[*label*] .WORD *value*[,*value*]...

The .WORD instruction creates two-byte words of data in your code. A *value* must be an integer between −32768 and 65535. The negative values are in two's-complement format. You can use any number of values on a line but separate them by commas. The .WORD instruction must begin on a word boundary (an even-numbered address). Every value will occupy two bytes of memory, so the instruction always ends on a word boundary. The odd-numbered address will contain the LSB. The *label* is the even-numbered address of the MSB.

.LONG instruction:

[*label*] .LONG *value*[,*value*]...

The .LONG instruction creates four-byte long words of data in your code. A *value* can be any integer. The assembler does arithmetic with long-word values. You can use any number of values on a line. Separate them by commas. The .LONG instruction may begin on any word boundary (and

even-numbered address). Every value will occupy four bytes of memory, so the instruction always ends on a word boundary. The "label" is the address of the most significant byte. This is the lowest numbered address used by the instruction.

.ASCII instruction:

[*label*] .ASCII *string*

The .ASCII instruction creates a string of data characters in your code. The *string* must be delimited by single or double quotes. A double quote may appear within a string delimited by single quotes and vice versa. The characters are expressed as one byte ASCII values. The following .ASCII instruction,

```
String  .ASCII "'ABC's'"
```

is the same as this .BYTE instruction:

```
String  .BYTE $41,$42,$43,$27,$73
; characters above = A B C ' s
```

.BLOCK instruction:

[*label*] .BLOCK *length,value*

The .BLOCK instruction creates areas in your code, filled with constant values. The *length* is the number of bytes to create and fill. The *value* is an expression for a number between −128 and 255 inclusive that is used to fill the area. Since we do not recommend that you modify data that is mixed in with the code, you may not use the ".BLOCK" instruction very often.

.EQU instruction:

label .EQU *value*

The .EQU instruction creates a label equal to a certain constant or to a certain address in the code. The *value* is an expression for this address or number.

Alignment

In 68000 Assembly language, instructions always have to start on word boundaries. Data handled as 2-byte words or 4-byte long words must also align with a even-address boundary. Both the assembler for the Macintosh and the assembler for the Lisa provide the same directive to help you realign after defining byte-wide data.

.ALIGN directive:

.ALIGN *boundary*

You can adjust the address counter to a word boundary with the .ALIGN directive. The next instruction will start on an address evenly divisible by the boundary. For example,

```
.ALIGN  2
```

means the next instruction starts on an even-address word boundary. If you use,

```
.ALIGN  4
```

the next address will be divisible by four. A typical use of this instruction is after data is placed within the program if the data could terminate on an odd boundary.

Segmentation

Long programs can be divided into segments. You can assemble one segment at a time, then connect them all with the Linker. If you are linking Pascal with Assembly language, you have at least two segments: one for Pascal code and the other for Assembly language. To connect code between the segments, you need to use special directives. These directives define entry points which can be called from a segment other than the segment the called routine is in.

Macintosh Assembler Segment Directives

XDEF directive:

XDEF *label*[,*label*]...

Use the XDEF directive to define an entry point for another segment, created by another assembly. A *label* must be an address which is defined in the current segment. The other segment will then be able to call that address, if it contains a matching XREF directive. The XREF directive is described below. You can put as many labels as you want in one XDEF directive separating them with commas. Any number of XDEF directives are allowed in the segment. You can put in a few extra XDEF directives to define variables for debugging. The Debugger can display addresses symbolically if they are external entry points.

XREF directive:

XREF *label*[,*label*]...

Use the XREF directive to access an entry point in another segment. A *label* must be an address which is defined in a matching XDEF directive in one other segment. You can put as many labels as you want in each XREF directive, separating them with commas. The XREF directives will usually appear before any executable code in the procedure.

The XREF directive can access an entry point in another segment, but you have to be very careful how you use the label. As you will see below in the Special Syntax section, entries to other segments point into a jump table instead of going directly to the targeted routine. In general, you should only use the label created by an XREF directive in JSR, JMP, LEA, or PEA instructions and not do any arithmetic on the label or on the address the instruction produces.

End directive:

```
END
```

The END directive comes at the very end of your assembly. It ends the program or the segment if you have multiple assemblies. Anything that appears after the END directive is ignored.

Lisa Assembler Segment Directives

The Lisa Assembler was designed to create subroutines to be called from Pascal. This means the only possible structure for a program is one or more Pascal functions or procedures. The only differences between a procedure and a function occur in the Pascal code that calls your assembly language program. In Appendix E, the Lisa Workshop, there is an example of a dummy Pascal program calling an assembly language routine. You will have to modify the programs in this book to match this structure before you can assemble them. The example uses one procedure to interface to Pascal. Data definitions are contained in each procedure or function. A label defined in one procedure will be undefined if you try to use it in another. Groups of procedures can be further divided into segments. However, every procedure and function must be entirely in one segment. To call addresses in another procedure, in the same or a different segment, you have to use the REF and DEF directives described below.

.PROC directive:

```
.PROC  label
```

A new procedure starts with the .PROC directive. The *label* is used by Pascal and other assembly language routines to access the code which follows. It addresses the first word generated after the .PROC directive. Assembly language routines in other procedures must include an .REF directive in order to use the label.

Pascal automatically has access to the label when the two programs are linked. The procedure physically ends when another .PROC, a .FUNC or an .END directive is encountered. Your code should logically end with an RTS instruction, then control will return to the Pascal program.

.FUNC directive:

.FUNC *label*

A new function starts with the .FUNC directive. This directive is the same as the .PROC directive except for the way Pascal handles it. To use a .FUNC directive you must declare your external code (your Assembly language routine), as a FUNCTION in Pascal. The Pascal code will create space on the stack underneath your return address for the value of the function. If you are using a dummy Pascal program, you don't need to pass any actual value back. Just end your code with an RTS to return to Pascal, and the value of the function will be undefined.

.SEG directive:

.SEG *'string'*

The .SEG directive divides your program into segments (a segment is a "chunk" of program that you can clear out of memory to free up space—then bring it in again from the disk when you need it). The *string* is a name for the segment. It should be in single quotes. The .SEG directive does not take effect until the next .PROC or .FUNC is reached. This guarantees that a procedure does not get split into more than one segment. The name must not be more than eight bytes long. The initial segment is called the "blank segment" because its name corresponds to eight blanks. The names are not too important (although it is nice to provide names descriptive of the segment). The Linker will show them to you, and they can be used by certain utilities to juggle the segments.

.DEF directive:

.DEF *label*[,*label*]...

Use the .DEF directive to define an entry point into a procedure. A *label* must be an address which is defined in the procedure. Another procedure, containing a matching .REF directive, will then be able to access that address. The .REF directive is described below. You can put as many labels as you want in one .DEF directive, separating them by commas. Any number of .DEF directives are allowed in the procedure. Usually they come before any executable code. They do not have to wait until the address is defined, but it must be defined within the procedure. You do not need a .DEF for the label defined by a .PROC or .FUNC directive.

.REF directive:

.REF *label*[.*label*]...

Use the .REF directive to access an entry point in another procedure. A *label* must be an address which is defined in a matching .DEF directive in one other procedure. You can have as many labels as you want in each .REF directive, separating them with commas. The .REF directives will usually appear before any executable code in the procedure. A .REF for a label must appear before the label is used.

The .REF directive may access an entry point in the same segment or in a different segment. If the code is in a different segment, you have to be very careful how you use the label. As you will see below in the Special Syntax section, entries to other segments point into a jump table, instead of going directly to the targeted routine. In general, you should only use the label for JSR, JMP, LEA, or PEA instructions and not do any arithmetic on the label nor on the address that the instruction produces.

.END directive:

.END

The .END directive comes at the very end of your program. Consequently, it ends the last segment as well as your program and anything that appears after the .END directive is ignored.

Special Syntax

There is some special syntax you use when you code for the Macintosh that is not common to all 68000 machines. This syntax was incorporated into both the Macintosh and Lisa Assemblers to help you take advantage of some of the Macintosh Operating System's features. Some kinds of instructions will be translated into others when assembled. Some addressing modes are translated to different modes. There is even one addressing mode you can use that does not seem to exist in the 68000 at all.

All program addresses on the Macintosh have to be PC-relative. The Segment Loader can load application code into any available space. It can only do that if the code is completely relocatable. You may not be aware of th PC-relative addressing when you are coding because the Assembler makes the exact addressing mode transparent to you. This is a real convenience. But it can be a source of confusion if you do not realize what is going on. For example, the 68000 does not allow PC-relative address mode for destination addresses. But Absolute mode is always allowed. So you might be surprised if you try to assemble the following instructions:

```
        MOVE D0,destination
             ...
Destination DC  0
```

You will get an addressing mode error, even though the instructions, as written, are legal. The Assembler has attempted to translate the reference to *destination* into a PC-Relative mode. The syntax it rejected is actually this.

```
MOVE  D0,Offset(PC)
```

where the PC-relative destination is in fact not a valid addressing mode. Although we don't recommend that you modify data in code space, we will show you how to do it. Load the effective address into an address register and then store the data using address indirect mode:

```
LEA   Offset,A0
MOVE.W  D0.(A0)
```

Sometimes you want to put a table of addresses into your code. This is good when you need the equivalent of ON *n* GOTO in BASIC or a CASE statement in Pascal. You have already seen how to access a data table using the program counter relative with index and displacement addressing mode (in Section 12 of Addressing Modes of the 68000, Chapter 2). But how can your table contain program addresses as data, when the code is going to be relocated? The only way to store the addresses is by relative offset. You have to save the distance from a known point in the program to the subroutine you want to call. Then, to reach your subroutine, you load the offset into a register and use a PC-relative jump. The Macintosh Assemblers are smart enough to calculate the offsets for you. Your table may look something like this:

```
CaseTable  DC   Rout1-CaseTable
           DC   Rout2-CaseTable
           DC   Rout3-CaseTable
           DC   Rout4-CaseTable
```

This table is made of words whose values are the distance in the code, between the start of the table and the four subroutines. Although the Assembler maintains all of these addresses as PC-relative offsets, it knows that the difference between two addresses will be a constant value. It can algebraically reduce an expression of the difference between PC-relative addresses.

```
Routn – CaseTable = (OffsetN+PC) – (TableOffset+PC)
```

to an expression of the difference between their offsets, or:

```
Routn –CaseTable = OffsetN – TableOffset
```

To access our table, calculate the index and then jump, using the program counter relative with index and displacement addressing mode. The instructions will look something like this:

```
MOVEQ   #3,D0               ;Go to Subroutine 3
ADD     D0,D0               ;Double index. Each entry is two bytes wide
MOVE    CaseTable(D0),D0    ;Load "Rout3-CaseTable" into D0
JMP     CaseTable(D0)
```

The very simple-looking expression in the code above,

```
...     CaseTable(D0)
```

actually assembles to:

```
...     CaseTable-*-2(PC,D0)
```

The latter address expression is in the PC-relative With Index and Displacement addressing mode, as described earlier. To use that mode directly, you have to calculate an offset between the Program Counter and the destination. Before performing the address calculation, the hardware will advance the Program Counter to the address of the extension word, the second word of the instruction. The expression takes that extra word into account, by subtracting two from the difference between the current location and CaseTable. The first expression,

```
...     CaseTable(D0)
```

makes it much simpler to use this addressing mode. The effective address is just D0 bytes beyond CaseTable.

All of the calls between segments go through the Jump Table. This table is created automatically by the XDEF and XREF (or .DEF and .REF) commands you use to communicate between segments. Calls to addresses in other segments are also automatically converted to calls through the Jump Table.

The Jump Table is located in the Global Data Area. This area is always accessed by the address register, A5. When you call an entry point in another segment, you actually JSR to the jump table. The Jump Table will usually hold a JMP instruction to the desired entry point. We say *usually* because the table is set up to load the segment containing the entry point if that segment is not in memory. In Chapter 7, The Macintosh Environment, you will learn more about how this mechanism works.

A JSR into the jump table must take the form of:

```
JSR  Offset(A5)
```

The assemblers let you code the call as though it were a destination in the same segment. So when you write,

```
            XREF OtherSeg              ; Use ".REF" with Lisa
                                       ; Assembler
            JSR  OtherSeg              ; Call routine in another
                                       ; segment
```

the code generated is something like this:

```
JumpTable JMP  ...                     ; Start of jump table
...
TabEntry  JMP  OtherSeg                ; Our entry in table
...
          LEA  JumpTable,A5            ; Before our program starts
          ...
          JSR  TabEntry-JumpTable(A5)  ; = JSR OtherSeg
```

When you use these translations you cannot use B*cc* instructions to reach addresses in other segments. These instructions have only one addressing mode, which is PC-relative. In addition, the Macintosh Assembler translates all BRA.L and BSR.L instructions into JMP and JSR instructions. It does this whether it needs to or not.

Assembly Control

Certain kinds of directives do not generate any code themselves, but affect the way the rest of the code is interpreted. They allow you to put parts of your code in different files, not assemble some lines of your program during some assemblies, and even define your own instructions. Both assemblers have these features, but they differ slightly in syntax.

Macintosh Assembler Control Directives

INCLUDE directive:

INCLUDE *filename*

If you want to use code in another source file in your program, use an include directive. When the assembler encounters the include directive, it starts taking source code from the other file. When it reaches the end of that second file, called the include file, it returns to the first or main file. The effect is the same as if you had copied the entire include file into your program in the same place where you put the include directive.

The *filename* is the name of the include file. If the include file is on a different volume (each disk is called a "volume") than the main source, you can add a volume name to the filename. The include file can be one of three types of file. You can use an ordinary text file you create with the Editor for an include file. This type of file can contain both code and

symbol definitions. If you just need the symbols from a prior assembly, you can use a symbol file created by a .DUMP directive. This directive is explained below. A special utility, PackSyms, creates a condensed form of a symbol file to use as an include file. This packed type of include file usually has the suffix ".D" and is used for the ToolBox definitions. Your INCLUDE directives can take any of these forms:

```
INCLUDE Kludge         ;More code is in the Kludge.Asm file
INCLUDE Widget:Kludge  ;Include Kludge.Asm on the Widget disk
INCLUDE QuickEqu.D     ;Include the QuickDraw definitions
```

An include file can itself contain INCLUDE statements. The newly included file can include other files too. This can go on until five levels of include file are nested. An include file cannot even indirectly include itself.

.DUMP directive:

[*label*] .DUMP *filename*

The .DUMP directive creates a symbol table file to use as an include file. You should place .DUMP at the end of a program, after all of the symbols have been defined. When you assemble the program and the Assembler reaches the .DUMP directive, it will write out all the labels in the symbol table to the file. The new symbol table file can be used as a source file for the next assembly. It is a standard text file, so you can also modify it with the Editor. The *filename* can be the same as the source file without the ".Asm" suffix. When the assembler creates the symbol table file it will add ".Sym" as a suffix to *filename*.

After creating a symbol table file with the .DUMP directive, you can compress it with the PackSyms utility. This program creates a more compact form of the symbol table file. You can use the condensed symbol table file for greater speed in an assembly, but you can't modify it with the Editor. A condensed symbol-table-file name should end with ".D" for a suffix. The definitions for using the ToolBox are supplied as condensed symbol files. To make a listing of such a file, include the ".D" file in a dummy program with the INCLUDE directive. Then use the .DUMP directive to create a new, uncondensed, symbol table file:

```
                    ;This program makes a listing of the QuickDraw
                    ;equates
.INCLUDE QuickEqu.D ;Assemble the QuickDraw symbols
.DUMP    QuickEqu   ;Write the symbols out to QuickEqu.Asm
END
```

Conditional Assembly

```
[label] IF condition
...

        [ELSE]
        ...
        ENDIF
```

If you want to assemble the same source file with slight differences to create applications, you can use conditional assembly. The *condition* is evaluated by the Assembler. If it is true, the following section of code is assembled. If the condition is false, the section is ignored. An ELSE clause is optional in the conditional assembly structure. If ELSE is used, the code following it is assembled only if the condition is false. The ENDIF marks the end of the section or sections of code affected by the conditional assembly. Everything after the ENDIF is assembled as normal.

Conditions can be either single expressions or comparisons between expressions. If a single expression evaluates to a non-zero value, the condition is true. If the comparison is correct, the condition is true. You can use any of the relational operators you use in Pascal for the comparison. These are:

```
=    equal
>    greater than
<    less than
>=   greater than or equal
<=   less than or equal
<>   not equal
```

The expressions must be constant values, rather than relocatable, PC-relative labels. You can use strings in the comparison, but they can only be tested for equality or inequality. Some examples of conditional assembly demonstrate various types of condition:

```
Five  EQU 5
Three EQU  3
Start EQU  *

      IF  Five              ;True since Five is not equal to zero
      PEA  "Always"         ;assembled
      ENDIF

      IF  Five > Three      ;True since Five is greater than Three
      PEA   "Always"
      ENDIF
```

```
IF  Five = Three        ;False since Five is not equal to Three
PEA  "Never"            ;not assembled
ELSE
PEA  "Always"           ;assembled
ENDIF

Stop IF Start = Stop
PEA "Illegal"           ;Illegal since labels are PC-relative
ENDIF
```

Several levels of conditional assembly can be nested. The inner levels are not evaluated if they are in a section of code which is not being assembled because the condition of an outer level was false. You will probably find conditional assembly very useful when you are debugging a program. You can add code to debug your program under an IF clause that tests a predefined variable, say "DEBUG." When you equate DEBUG to zero you can assemble the final version without the test code. Set DEBUG to anything else and assemble a test version with all of the debug code in place.

Lisa Assembler Control Directives

.INCLUDE directive:

.INCLUDE *filename*

If you want to use code in another source file in your program, use an INCLUDE directive. When the assembler encounters the INCLUDE directive, it starts taking source code from the other file. When it reaches the end of that second file, called the include file, it returns to the first or main file. The effect is the same as if you had copied the entire file into your program in the same place where you put the INCLUDE directive.

The *filename* is the name of the include file. The include file is an ordinary text file you create with the Editor. If you don't use ".TEXT" in the filename, the Assembler will add that suffix for you. An "INCLUDE" directive looks like this:

```
.INCLUDE Kludge         ; More code is in the Kludge.Text file
```

An include file cannot contain INCLUDE statements; include files cannot be nested.

Conditional Assembly

```
[label]  .IF  condition
...

         [.ELSE]
         ...
         .ENDC
```

If you want to assemble the same source file with slight differences to create different applications, you can use conditional assembly. The *condition* is evaluated by the Assembler. If it is true, the following section of code is assembled. If the condition is false, the section is ignored. An .ELSE clause is optional in the conditional assembly structure. If .ELSE is used, the code following it is assembled only if the condition is false. The .ENDC marks the end of the section or sections of code affected by the conditional assembly. Everything after the .ENDC is assembled as normal.

Conditions can be single expressions or tests for equality or inequality. If a single expression evaluates to a non-zero value the condition is true. If the comparison is correct, the condition is true. The expressions must be constant values, rather than relocatable, PC-relative labels. You can use strings or numbers in the comparison. Some examples of conditional assembly demonstrate various types of condition:

```
Five  .EQU  5
Three .EQU  3
Start .Equ  *

      .IF  Five                ;True since Five is not equal to zero
      PEA  "Always"            ;assembled
      .ENDC

      .IF  Five <> Three       ;True since Five is not equal to Three
      PEA  "Always"
      .ENDC

      .IF  Five = Three        ;False since Five is not equal to Three
      PEA  "Never"             ;not assembled
      .ELSE
      PEA  "Always"            ;assembled
      .ENDC

      Stop  .IF  Start = Stop  ;Illegal since labels are PC-relative
      PEA  "Illegal"
      .ENDC
```

Several levels of conditional assembly can be nested. The inner levels are not evaluated if they are in a section of code which is not being assembled because the condition of an outer level was false. You will probably find conditional assembly very useful when you are debugging a program. You can add code to debug your program under an .IF clause that tests a predefined variable, say "DEBUG." When you equate DEBUG to zero you can assemble the final version without the test code. Set DEBUG to anything else and assemble a test version with all of the debug code in place.

Macros

.MACRO Directive

```
.MACRO identifier
code
...
.ENDM
```

Macros make it easy to put groups of instructions or data structures you use frequently into your code. You define a macro to have a certain equivalent. Then, whenever you use that macro in your program, it is the same as if you had typed in the whole sequence of instructions defined as the macro.

Macros can have parameters. When you define the macro, you use special symbols for the parameters. Then when the macro is used in the code, you provide actual parameters, separated by commas. The actual parameters are substituted for the special symbols as the macro is expanded. The assemblers do this by substituting the text string of the parameter for the symbol. The symbols for the parameters are %1,%2,%3 and so forth. They correspond to the first, second, third parameter and so on.

The *identifier* for a macro can be any valid label. Use this label for the operator when you use the macro. The *code* can be any sequence of instructions or data definitions. Parameters may be in the code by using the special symbols, such as %1. Here is an example of a macro. It is defined, used, and then the equivalent code it produces is shown below:

```
.MACRO LoadBoth   ;This is the macro definition
CLR.L   D1
MOVE.W  %1,D1
CLR.L   D2
MOVE.W  %2,D2
.ENDM
```

```
...
LoadBoth #5,Var(A6);Here the macro is used
...
                        ;This is the code it generated
CLR.L   D1
MOVE.W  #5,D1
CLR.L   D2
MOVE.W  Var(A6),D2
```

The Linker

As a novice assembly language programmer you probably first came upon a linker when you discovered that you can't just assemble a program and then run it. No, instead you must assemble your program creating a ".rel" type file and this is then input to something called a linker which then creates something which you can finally, actually run. Why do they make life complicated for assembly language programmers?

Perhaps you are in that phase right now. Actually, a linker is a very useful tool. Linkers were invented for one main reason—a linker allows an assembly language programmer to write a series of small programs and link them all together into one big program.

If you only write small programs you will not link your programs together although you will still have to link up with systems routines. But when you start to write large programs that run over 30 or 40 pages of code or you work together with another person who is also writing part of the program you will suddenly appreciate the need for a linker to link your programs together.

Suppose you had no linker but had broken your program up into five smaller programs. You are in one program and you suddenly need data from another program. Whoops! You see the problem—you have no reference to where that data is in your program. So you must either have all the data references for all the programs in every program, which makes the process of assembling take forever, or you must create a special subset of the data references which will be used by more than one program and include that subset in all the programs. Suppose you take the latter course.

How do you use a routine from one program from inside another program? On the Macintosh, this is a real problem. For one thing, programs can move around in memory, even be sent out to the disk, if they are in different segments. Ok, let's assume you put everything in one segment—a real limitation. Still, where is the routine you are interested in? Let us assume that right now it is 234 bytes from the start of the program it is located in, for example. What if you place the address of the start of the routine in an address register and give an offset of 234 and go there? Suppose the code changes and the routine is moved further up in the

program? You'd have to go to every program that uses this routine and change its offset.

A better way is to have a jump table (we haven't come to the best way yet, to have a linker, but be patient!). A jump table looks like this:

```
MODULE1
ADDROUTINE   JMP   ADDRTN     ;JUMP is short for JuMP, like GO TO in
                              ;higher level languages
SUBROUTINE   JMP   SUBRTN
MULROUTINE   JMP   MULRTN
             etc.
```

In a jump table there is one entry for each routine that is called from outside the program. In the start of all the other programs there is a piece of code that looks like this (usually in an file that is included using the INCLUDE statement of the assembler):

```
ADDROUTINE   EQU   MODULE1 + 0
SUBROUTINE   EQU   MODULE1 + 4    ;there are 4 bytes to a jump instruction
MULROUTINE   EQU   MODULE1 + 8
             etc.
```

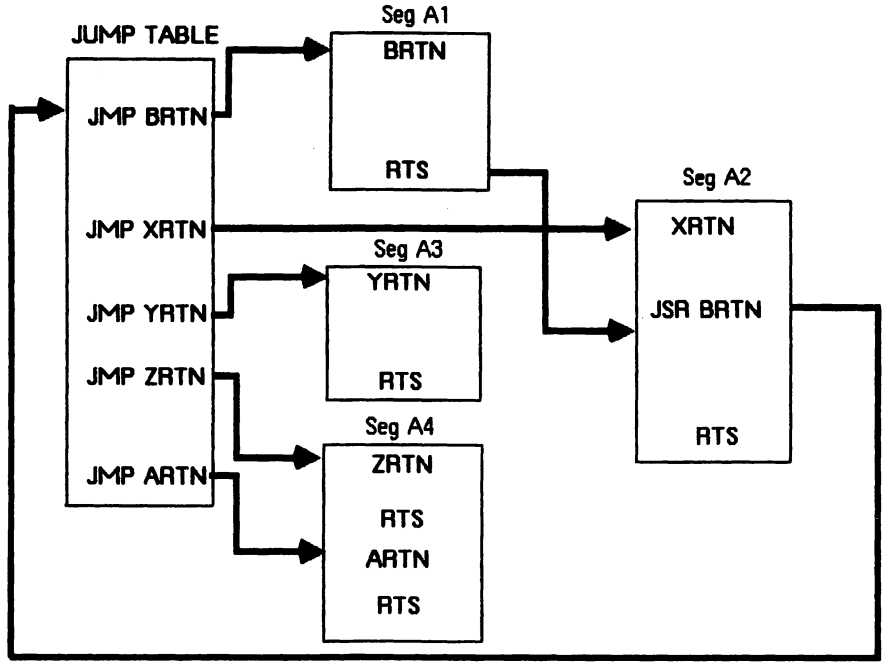
Now you simply JSR MULROUTINE (JSR means Jump SubRoutine, like GOSUB in BASIC) which lands you 8 bytes into the program at a JMP to the real location of the routine. The program itself changes that JMP each time it is reassembled so everything works. The only penalty is an extra JMP instruction is executed for each intermodule call (call from one separately assembled routine to another).

You could use the jump tables and get by without a linker, but it would be a lot of work. How would you do the same thing with a linker? With a linker you simply write commands to the assembly which tell the linker that there are certain names which will be sent to other modules or picked up from other modules (so they won't be defined in this program's text). Those names which will be sent to other modules (data and labels into the code, usually) are collected into a table by the assembler. The command for allowing other modules to use names from inside this program is XDEF, external definition. In other words, this definition is going to be used external to this program.

Example:

```
XDEF ADDROUTINE
XDEF SUBROUTINE
```

When a name is going to be pulled in from another module, the assembler has to fill the location in the code with something and create a reference to that place in the code so when the linker is run it can patch in the correct value. In other words, now when the assembler is run it creates



All subroutine calls to another segment are through the jump table. Note that for the RTS to work, the original calling subroutine must be locked in memory. ZRTN, ARTN, BRTN, etc. are subroutines.

Figure 6-1 JuMP Table and Segments in Memory

code with little "holes" in it because it has no idea what value it should use. The command that tells the assembler to create such a hole is XREF, external reference.

Example:

```
XREF ADDROUTINE
XREF SUBROUTINE
```

in the code:

```
JSR ADDROUTINE      ;we will create a hole here
JMP SUBROUTINE      ;and a hole here since we don't know the location
```

This is why the output of the assembler is called a ".rel" file, short for relocatable. Once you have a bunch of these .rel files you send them all into the linker and it goes through each program finding all the XDEF tables. Now it knows where everything really is. Then it goes through each

module and patches in all the holes created by the XREFs inside the code. Once the linker is done all of the code has real addresses in it.

Actually, some linkers cheat. They create a jump table in each module and then patch in the jump table locations into the code—this saves them a few steps. In fact, the Macintosh linker creates just such a jump table. If you look at the way the Macintosh Memory Manager works you will see that a jump table is the only way that the linker could work.

Now that you understand how a linker works you can see what an Undeclared External Reference means (the most common error when linking a program). There has to be an XDEF in some module corresponding to every XREF in another module. Otherwise, how would the linker know where to go? In general, you should eliminate all undeclared external references before running a program.

The linker on the Macintosh can be used to link together modules which were produced by the assembler. Since you can use the Macintosh assembler to create not only code but resource files, you must use the linker to tell the system which type of information it is dealing with. In general, though, it is much easier to use RMaker to create resource files rather than creating them using the assembler and linker. In fact, the easiest way to hook up an assembler program with resource data is to create your resource data in a completely separate file and then use __OpenResFile inside your assembler program in initialization so that your program is aware of that separate resource file.

A linker file in the 68000 Macintosh Development System consists of a series of commands interspersed with the names of the modules that need to be linked together. Files are assumed to end in ".Rel"—if you list a filename without ".Rel" as a suffix, that suffix is appended. So, for example both

```
myfile.Rel
```

and

```
myfile
```

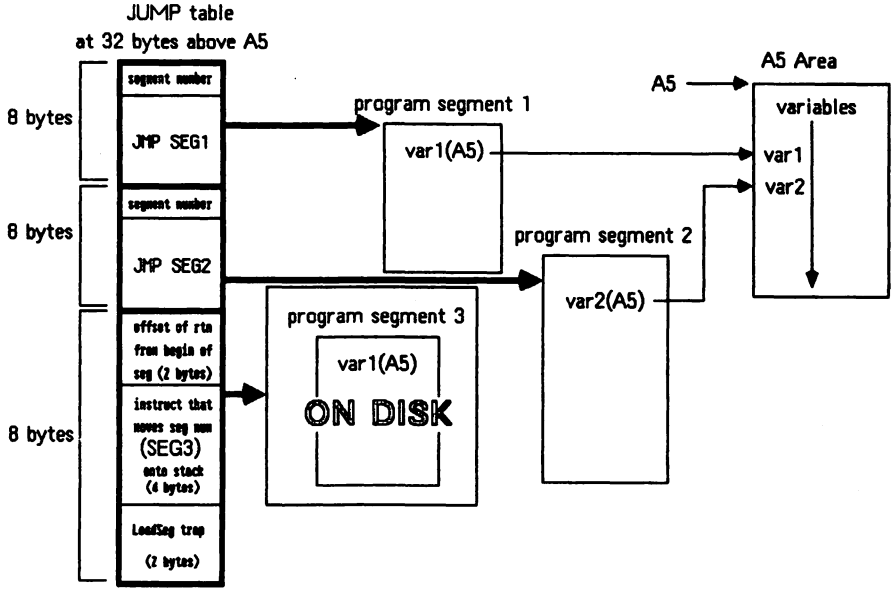
in a linker file will make the linker look for "myfile.Rel" on the disk to be linked.

Actually, there are many things you can do with the linker besides joining modules. You can place various modules into different segments. The "<" symbol in a linker file means "start a new segment." Say you had a linker file that looked like this:

```
Myfirstprog
<
Mysecondprog
```

Myfirstprog.Rel would be in one segment and Mysecondprog.Rel would be in another segment. Only one of these segments would have to

be in memory at any one time (the one in which the code was actually being executed). When you went from one segment to another it would always be through the jump table. The jump table on the Macintosh is actually a very smart jump table—if the segment isn't in memory it is brought into memory before the jump to the segment is executed.



Each segment is pointed to by an 8-byte entry in the jump table which is maintained by the segment loader. The entry is simply a jump plus segment number if the entry is in memory; if not in memory, the entry consists of a call to a routine to bring the segment in from disk. Sort of ingenious really.

Figure 6-2 JuMP Table and Segments (Advanced)

Any Macintosh program consists of code, resources (other than code), and data. Code is the concatenation of the various .Rel files created by the assembler. Resources are all the various entities that a Macintosh application calls upon—menus, dialog boxes, windows, fonts, text strings used in titles, etc. Finally, data are information in disk files that are not part of the specially formatted data of resource files.

Actually, all files on a Macintosh have two "forks" or parts. One fork contains the code, menu descriptions, dialog description, etc.—this is the

resource fork. The other fork is the data fork—here is all the random data stored just the way you are used to it on other computers as one big block of hexadecimal. Often one of these forks is empty since a file that has a resource fork is either a program or data used by a program (usually constant) while a data fork is modifiable data used by a program; programs and modifiable data files are usually in separate files.

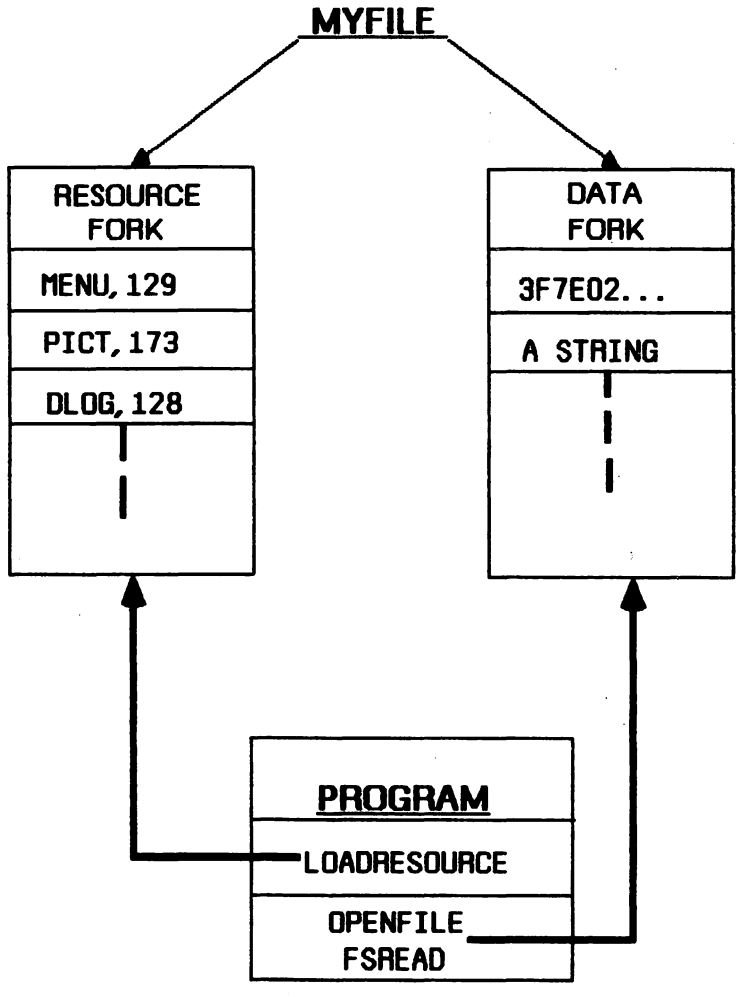


Figure 6-3 Forks

The linker puts all the code first into the resource fork, followed by all the resources, and then puts all the data into the data fork. A linker file uses the commands "/resource" and "/data" to signify the start of the respective resource (not code) and data fork information.

A typical linker file would be

```
mycode
/resources
resfile
/data
datafile
```

where mycode and resfile are put into the resource fork and datafile is put into the data fork.

There are other things the linker does, such as allow you to jump to a point other than the start of the first module to kick off that application (use an "!" followed by a symbol defined in one of the modules with an XDEF), or shunt the output of the linker to a certain file (with the /output command). Read about these in the 68000 MDS assembler manual.

The linker is also used for other things besides generating linked code. Sometimes you would put XDEFs into an assembler file to generate a linker cross reference even though you never expect them to have an XREF that will correspond. This is because the debugger uses the linker symbol table—if you have no XDEF there is no symbol placed into the linker symbol table for the debugger.

Summing up, primarily a linker glues together a whole bunch of smaller ".ref" modules created by the assembler into one big ".obj" or object module that can actually be used by the computer. In most systems, that is all there is to it. However, on a Macintosh you can't actually run your program until you learn more about Resources and the Resource Compiler.

The Resource Compiler

The Resource Compiler is really just what it sounds like. It is a compiler (like a BASIC compiler or a Pascal Compiler) and it works on resources. The rest of this section will be used in describing just what the Macintosh means by the word "resource."

Everything that is part of an application running on the Macintosh is a resource. Typical resources are menus, windows, dialog boxes, an object module (the program you just laboriously coded, assembled, and linked), and text (for messages that your program prints). Some of these resources are "templates" that describe the resource rather than the resource itself. For example, the resource file contains a description or "template" for a window, not the window itself. You can create the window by calling a

routine which uses the template made by the resource compiler as a description of what the window is to look like.

In general, all the text and every object you see on the screen such as windows, menus, and dialog boxes should be completely described in a resource file rather than inside the assembler program. Then, when you want to change a program, all you need to do is change the resource file and recompile it using the resource compiler. Or you can use the resource editor to change resources. The important point for programmers is that you do not need to change your program or even give out the source code to allow for changes in the text. The original reason for doing things this way was to make it very easy to create foreign language versions of the program. Obviously, the translator only has to translate every piece of text within the resource file into the native tongue and instantly you have a foreign version of the program.

Here is a typical resource which involves two menus, one window, and a program:

```
* Resource File comments are preceded by an asterisk (*) at the start of the line
* or by ";;" in the middle of a line.
* The first line in a resource file is the name of the output of the compiler
Drawing
APPLDRAW
INCLUDE Drawing. obj ;;include the program linked together as Drawing.obj
Type MENU
  ,300
File  ;; a file menu
  Open/O ;; Open with Propeller 0
  Close
  Quit/Q

  ,302
Edit
  Cut/X
  Copy/C
  Paste/V

Type WIND

  ,1              ;; resource ID
Drawing Window    ;; label at top of window
30 10 310 460     ;; box window will come up in
Visible NoGoAway  ;; whether window is visible, whether box in upper left corner
0                 ;; ProcID (the resource ID of the PROC that defines this window)
0                 ;; RefCon (a reference constant that is varied by the program)
Type STR#
  ,1              ;; resource ID
```

```
3                ;; number of strings following
This drawing program allows you to
use many different styles of pen,
ink, and pad.

,2
1
No Disk in the Drive!!
```

Although both the Macintosh Development System and the Lisa (Macintosh XL) have resource compilers which are very similar, there are differences between the two systems.

As you can see by looking at the above file the first three lines contain some comments preceded by asterisks. On the next line, "Drawing" is the name of the program created as an application. APPLDRAW is the type (APPL) and creator bytes (DRAW).

Then come a series of various "type"s of menu, window, and string definitions. First comes a "Type MENU" line. This means that all the entities until the next type statement are menu definitions. Each definition is preceded by a comma followed by a number (for example: ,300).

There are two menus, a File menu and an Edit menu. The first line of a menu definition describes the name of the menu that will appear on the menu bar. So the first menu will be called "File," the second menu will be called "Edit." The first menu will have three lines: Open (with command key "O" on the same line), Close, and Quit (with command key "Q" on the same line). The second menu will also have three lines: Cut (with command key "X"), Copy (with command key "C"), and Paste (with command key "V").

In your code you need to initialize the menus with a standard call to the menu manager (technically it is a trap, but the trap works just like a subroutine). For your information, it would look like this in your assembler code:

```
_InitMenus
```

Then you would call routines that would get a handle on each menu through routines that use the number in the resource file; 300 would refer to the first menu and 302 to the second menu in the above example. You would see something like the following inside your assembler program:

```
FileMenu equ 300
EditMenu equ 302
```

then something like this later on:

```
MOVE.W #FileMenu,D4  ;the MakeMenu routine expects D4 to hold the Menu ID
JSR MakeMenu         ;this routine calls two ROM routines _GetRMenu
                     ;and _InsertMenu to create the menu
```

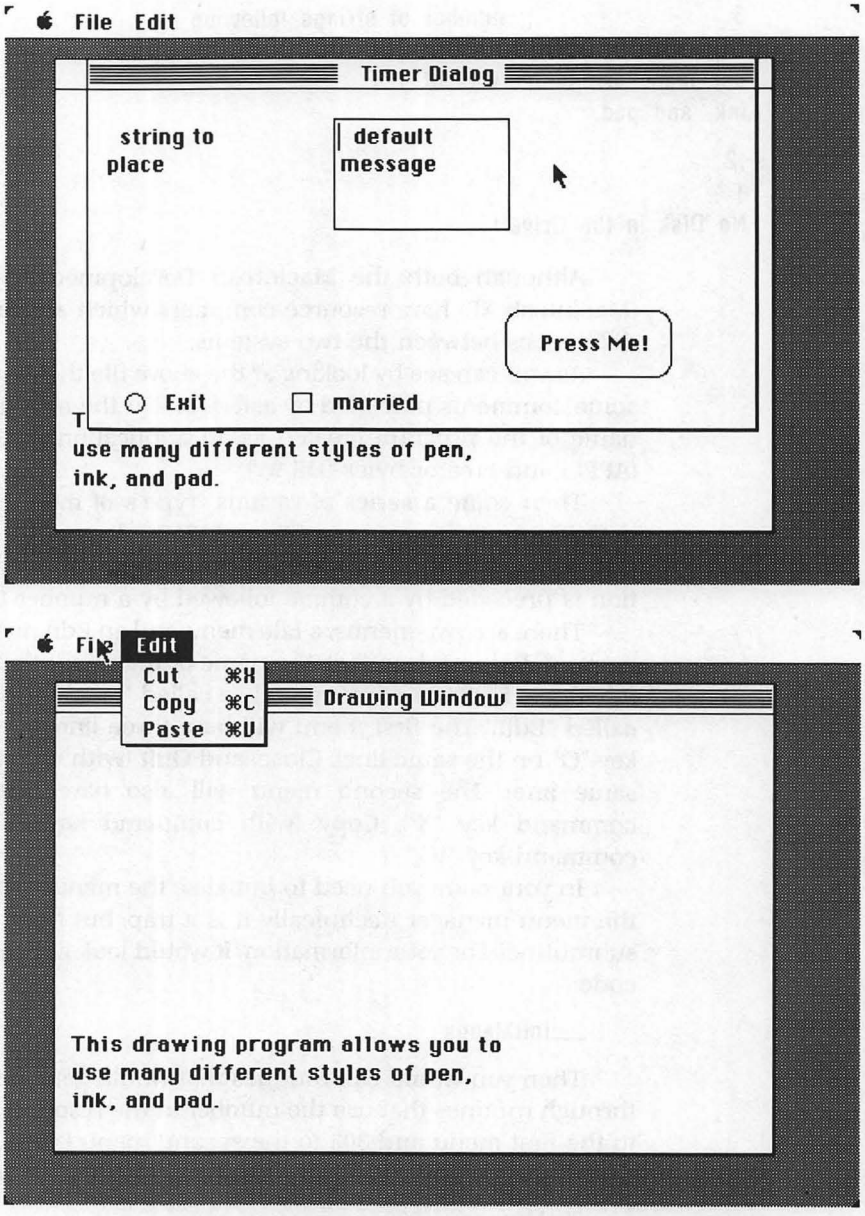


Figure 6-4 Screen Dumps of File and Edit Menus

This is how your program "connects" with resource file entries, through the resource ID numbers (the ones with the comma, such as ,300) and sometimes through the type (as in type STR#) besides. There are various ROM calls, they start with underscores as in __GetRMenu, that

require the resource ID number to find the resource. For an actual example of a resource file and the code for MakeMenu, see the example program FingerCalc later in the book.

The WIND type defines a window. Windows involve more definition than a menu. You have to tell what the title of the window will be, list four integers to define the top, left, bottom, and right coordinates of the window on the screen (the upper left corner of the screen is defined as 0,0).

The next line states whether the window is visible or invisible. If invisible, the window is there but it won't show on the screen—"visible" is the usual status since you usually want a window to be visible. NoGoAway means that there is no little "go away" box in the upper left corner of the window—GoAway means that the box will be in the upper left corner.

The next line is the ProcID (the resource ID of the type PROC procedure that will define this window) and the line that follows that is the RefCon (the reference value). In most cases the ProcID will be zero which means let the system routines that already exist handle this window. Usually the RefCon is changed under program control so this value in the definition is only an initial value.

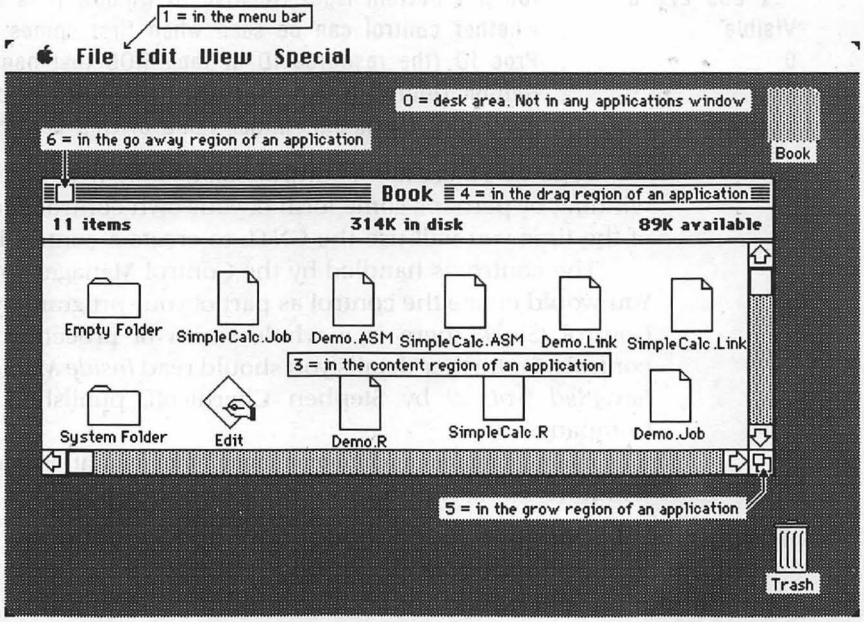


Figure 6-5 Parts of a Window

Finally, the STR# defines a series of strings. Each definition contains a count of the number of strings followed by the indicated number of strings of varying length. Then you simply use the GetIndString trap inside your program to access these strings after pushing a pointer to the string space

where the string will be returned, the resource ID, and the index of the string within that resource ID onto the stack.

There are actually twelve different types of resources that the Macintosh assembler's resource compiler recognizes. Alphabetically these are: ALRT, BNDL, CNTL, DITL, DLOG, FREF, GNRL, MENU, PROC, STR, STR#, and WIND.

We have already seen how MENU, STR#, and WIND are used. The other commonly used resources are CNTL, to create controls (such as control bars on the edges of windows), DLOG, to create dialogue boxes, DITL, for the list of parts of that dialogue box, GNRL, for creating Pascal-style records consisting of mixes of integers and strings, and STR, which creates individual strings. The remaining resource types are used less often.

CNTL creates a control template. It is similar in format to WIND. The format is:

```
Type CNTL
  ,310              ;; resource ID
vertical scroll     ;; title
-1 299 279 315      ;; top left bottom right (relative to window it is in)
Visible             ;; whether control can be seen when first comes up
0                   ;; Proc ID (the resource ID of the PROC that handles this control)
0                   ;; RefCon (four bytes of your own info which is part of the control)
0 1 0               ;; minimum, maximum, initial value of this control
```

Type CNTL creates a control bar, usually when it is associated with a window, or perhaps some form of your own control such as a clock. Most of the time you will use the CNTL to create a control bar.

The control is handled by the Control Manager routines in the ROM. You would create the control as part of your program by using __GetNewControl. Since there is a whole series of procedures associated with controls, for further details you should read *Inside Macintosh* or *Macintosh Revealed (Vol. 2)* by Stephen Chernicoff, published by Hayden Book Company.

DLOG creates a dialogue box. The format is similar to that of a window:

```
Type DLOG
  ,130              ;; resource ID
Timer Dialog        ;; message
50 50 250 350       ;; top, left, bottom, right of DLOG window from upper left of screen
Visible NoGoAway    ;; same as window
0                   ;; PROC ID (same as window)
0                   ;; RefCon (user defined value, same as window)
131                 ;; resource ID of item list (type DITL)
```

The only real difference between a DLOG and a WIND type is the last value points to a DITL or dialog item list. In this example we would expect to see a type DITL in the resource file before the DLOG with a resource ID of ,131.

A dialog item list, type DITL, defines the position and type of the various controls, check boxes, and buttons inside the dialog box.

```
Type DITL
  ,131              ;; resource ID (referred to by some type DLOG)
5                   ;; number of items in list

staticText disabled ;; text on screen (user can't modify it, create event)
20 10 150 100       ;; top left bottom right rectangle which text is placed within
string to place     ;; string to be placed inside rectangle

editText            ;; user can enter text using editing in ROMs
20 150 150 250      ;; top left bottom right rectangle which text is placed within
default message     ;; initial text that the window comes up with

radioButton         ;; radio button dialog item (control item, see control manager)
170 20 200 60       ;; top left bottom right of rectangle enclosing button
Exit                ;; message in button

checkBox            ;; a box that lights up when user clicks mouse inside it
170 80 200 120      ;; top left bottom right of checkbox
married             ;; item text to check off

button              ;; the text is surrounded by a square which is a button
180 250 210 350     ;; top left bottom right of button square
Press me!           ;; text within button
```

The above five items are defined for a dialog item list. Obviously you may have as many buttons, checkboxes, and such as you wish. The placement of the enclosing rectangle is relative to the top, left corner of the dialog box.

An item is assumed enabled unless you say otherwise. When an item is disabled nothing you do within the enclosing rectangle will have any effect. On the other hand, if you click within the rectangle of a dialog item that is enabled, or enter text into an editText item, there is an effect. If anything other than an editText item is clicked, the number of the item that was clicked is returned to the program.

Type ALRT is very similar to DLOG—ALRT is the template for an alert box. ALRT contains the following:

```
TYPE ALRT
  ,145          ;; resource ID
20 30 250 400 ;; top left bottom right of alert box relative to screen
```

```
135             ;; resource ID of a type DITL item list
7FFF            ;; stages word in hexadecimal
```

An alert has "stages" which correspond to the number of times that the alert has been called. It is possible to make an alert box do one thing the first time it is called and a different thing the second time it is called. The first three stages correspond to the first three times the dialog occurs—the fourth corresponds to every occurrence of the dialog after the third. Each stage has a default button, an indicator of whether the alert box is drawn (it may just beep and not do anything, for example), and which of four sounds, numbered zero through three, to produce. The default for the meaning of the sound numbers is that they correspond to the number of consecutive beeps in a row that the Macintosh sends out. Obviously, if there are zero beeps no sound issues forth. You can modify the sounds with __ErrorSound.

Here is how the stages word encodes the above information:

Each nybble is one of the four stages in an alert. The low-order four bits are stage one, the high-order four bits are stage four. Within each stage the high-order bit is the item number minus one of the default button—normally a zero bit means OK and a one bit means Cancel. The next bit is 1 if the alert box is to be drawn and 0 if it is not to be drawn. The last two bits range from 0 through 3, which are the sound numbers.

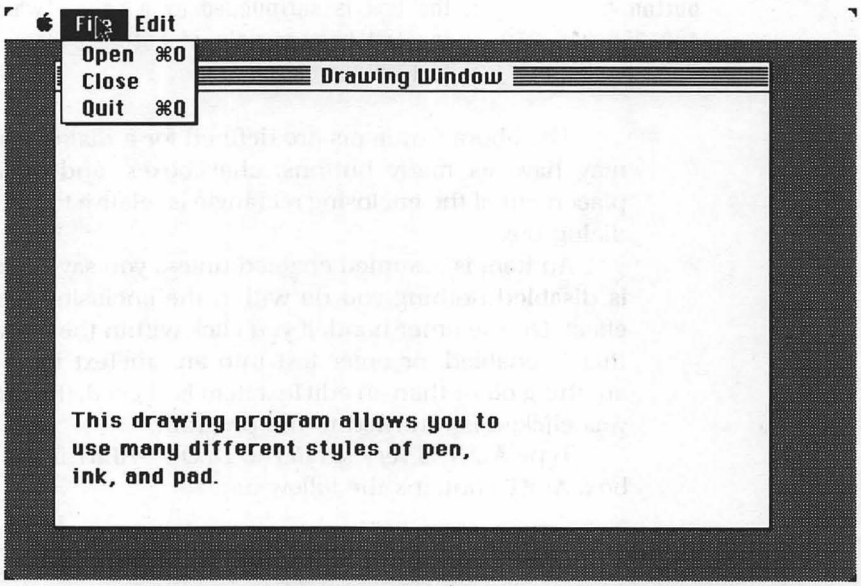


Figure 6-6 Screen Dump of Timer Dialog Box

So, the 7FFF in the example above means that for the first three times, draw the dialog box while issuing three beeps and let the default button be Cancel. After the third time, still issue three beeps and draw the box but let the default button be OK.

An ALRT is really just a special form of dialog. You must have __InitDialogs in the initialization of your program and call the function __Alert to invoke the alert. __StopAlert, __NoteAlert, and __CautionAlert are like __Alert only they place the stop, note, or caution icons in the rectangle (10,20,42,52) inside the alert. __CouldAlert and __FreeAlert might be of interest to you if you want to make the alert template unpurgeable now and later make it purgeable. ("Purgeable" means that it can be paged out of memory, "unpurgeable" means that it can't be thrown away—it must stay in memory.)

To see assembler routines that would create a dialog or an alert box see Chapter 10, with supplemental sample routines, or look in the SimpleCalc example program.

Type STR is similar to STR#. The only difference is that with type STR only one string is allowed per resource ID. Of course, you can extend a string over a series of lines by using the "+ +" convention of the RMaker program.

```
TYPE STR        ;; 'STR', space required
  ,10           ;; resource ID
A short string  ;; the string
  ,20(36        ;; resource ID (attributes = 32 (purgeable) + 04 (preload))
A long string ++;; a long string
that is continued ++
over three lines using the double plus convention.
```

The way to get one of these strings into your program is with __GetString which asks for the resource ID number and a pointer to the string space where the string is to be loaded.

The PROC type creates a resource that contains code.

```
TYPE PROC
  ,138          ;; resource ID
MyProcedure     ;; filename of a file that has a first segment of type CODE
                ;; with resource ID of 1
```

The PROC form is simple, but using it successfully is a bit more complicated. You must have a .OBJ type file created with a single segment by the linker. Then the PROC type looks for the first code segment; this means a resource of type CODE with resource ID of 1—if you only have one code segment this means the entire file. The PROC strips the first four bytes off, since these are the header bytes used by the segment loader, and saves it off inside your application.

Example:

```
TYPE DRVR = PROC ;; by using the '=' convention, you can create new types
                 ;; which behave just like the old type, in this case PROC
,150             ;; resource ID
HexCalculator    ;; a hex calculator program
```

Now, this program could be used from the Apple window which contains the desk accessories.

Type GNRL is used to create a resource file consisting of a mixture of strings, integers, long integers, hexadecimal, and other resources. This type is a grabbag of data. You define each portion of the resource using one character type designators preceded by a period.

DESIGNATORS

.P	Pascal string (one byte length followed by string of characters)
.S	string without length byte
.I	decimal integer
.L	decimal long integer
.H	hexadecimal
.R	read resource from a file. This is followed by: filename type ID

An example of GNRL:

```
TYPE GNRL
  ,258
.I
12  ;; each entry is stored as two bytes of hex in the resource file
234
17
.P
This is a string of length 29
.L
123456 ;; this will be stored as four hex bytes
```

The GNRL resource number 258 would look like this in hexadecimal (spaces added for clarity):

```
000C 00EA 0011 1D 5468697320697320612073747269E67206F66206C656E677468203239
0001E240
```

With .R you can include previously created resources in your application.

Example of .R:

```
TYPE FONZ = GNRL
  ,293
.R
```

```
System FONT 268 ;; the system font with ID 268 is brought in as type FONZ
                ;; into your application
```

You can use the resource editor and the font mover to find which fonts have which resource ID numbers. In general, you would use the resource editor to find other resources that you would like to include inside your application.

BNDL and FREF are used to make icons for your application. BNDL stands for BuNDLe and FREF stands for File REFerence. Together with ICN#, which is an icon list, you can associate an icon with your program and another icon with its data files; the icons you create will show up under the finder on the desktop. When the data file is double clicked, it will start the program.

The form of BNDL is:

```
TYPE BNDL
  ,1            ;; resource ID
  CALC 0        ;; bundle owner (CALC is the signature of SimpleCaLC)
  ICN#          ;; resource type
  0 128 1 129   ;; local ID 0 => ICN# no. 128 (icon for program),
                ;; local ID 1 => ICN# no. 129 (icon for data)
  FREF          ;; resource type
  0 128 1 129   ;; local ID 0 => FREF no. 128, local ID 1 => FREF no. 129
```

The form of FREF is:

```
TYPE FREF
  ,128          ;; resource ID
APPL 0          ;; file type, local ID of icon (APPL stands for APPLication, a program)

  ,129          ;; resource ID
DATA 1          ;; file type, local ID of icon (DATA means it is a simple calc created
                ;; file)
  ,130          ;; resource ID (no BNDL for this in above example)
TSTR 2 testfile ;; file type, local ID of icon, filename (filename optional)
```

If you used the above bundle and file reference and also included two icons (ICN#s) your assembler application would have its own icon (icon number 128) and the files it created would have their own icon (icon number 129). When the program creates data files, it must set the type to DATA, and the creator to CALC. There must be two icons in each icon list: one is the data for the icon and the other is the mask for the icon. You must have:

```
TYPE CALC = GNRL
,0
```

in the resource file for the finder (the "=" convention is described in the next paragraph). Any information may be in this item, but traditionally a string describing your program is placed there. Now you can have files that autostart your application in true Macintosh fashion.

NOTE
CALC is the signature of SimpleCalc
DATA is FREF 129

If you want to create new types with your own four character names you would simply use the '=' convention as shown by this example:

```
TYPE MYRE = GNRL ;;create a new type called MYRE
,1
.H
23AF0C
```

To get a handle to these three bytes of hexadecimal you would include the following in your program:

```
  CLR.L -(SP)          ;clear four bytes on the stack for the handle to be returned
MOVE.L 'MYRE',-(SP)    ;push the type 'MYRE' onto the stack
MOVE.W #1,-(SP)        ;push to the data 0001 onto the stack
__GetResource          ;call the procedure that loads a handle to the resource into memory
```

After completion, a handle to the procedure would be returned on the stack. In other words, this new resource has its own resource IDs and the resource name can be used in your program just like any other resource name.

In general, the resource compiler for the Lisa looks similar. There are minor differences—the names of dialog items are different, for instance, and there is a different way of including code. Further, there are more resource types allowed by the Lisa compiler.

The Debugger

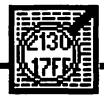
A debugger allows you to pull bugs out of your programs. Since you have done some programming you know by now that a "bug" is a teeny mistake that somehow has crept into the program causing it to do unexpected things. Computers will do exactly what they are told—unfortunately for the programmer it will do *exactly* what it is told, not always what you *think* it should do.

For some reason, many beginning programmers don't see any need to use "tools," such as debuggers. After you have spent hours trying to debug a program that you could have debugged in minutes with a good programming tool, you begin to see the value in the tools.

We have seen four people in a small company, with adequate tools, do more work than 20 programmers with no adequate tools in a large corporation. If you want to get a complex assembly language program up and running in a reasonable time you will have to learn how to use a debugger. In fact, even for small programs you will need tools such as debuggers to create a working program. Once you learn how to use one you will never want to work without it.

Fortunately, the assembly language development system from Apple has a debugger. Actually, there are a confusing series of debuggers; one for each different type of development system. The smallest debugger is for a 128K Mac (MacsBug). There is a better debugger for the 512K Mac (MaxBug) and there is a similar debugger for the MacWorks environment on the Macintosh XL (a.k.a. Lisa) called LisaBug.

Each of these different brands of debugger has different abilities. Obviously, the larger the memory available the more powerful the debugger that can be used. Some of the debuggers require two Macintoshes or a Macintosh and a Lisa in order to operate at all. In this way you use one computer to "monitor" the other computer—very little memory in the computer being monitored is used. This can be important when you have a program that fills all of the memory. Most of the debugger is in the computer that is doing the monitoring. Only enough of the debugger to interrupt instructions and send information about the state of the machine being monitored is left there. The two machines are connected by a cable—the information about the machine being monitored is sent along the cable one way and commands to set interrupts or feed back certain information is sent the other way. There are two debuggers called TermBugA and TermBugB that work this way.



Discussions and Comparisons

Specifically, the five versions of the MacsBug debuggers are:
MacsBug for a 128K Mac, displays eight lines at the bottom of the screen (uses 18K).
Maxbug for a 512K Mac, uses a 40-line display and shows trap names rather than numbers (uses 40K).
TermBugA/TermBugB TermBugA is for use between two Macintoshes or a Macintosh and a Lisa over the modem port while TermBugB is for use over the printer port (uses 12K).
LisaBug for use on MacWorks on a Lisa, is just like MaxBug. This is the only debugger that will work in the MacWorks environment.

There is also a *MacDB* debugger. Like the TermBugA and TermBugB debuggers this debugger requires two machines to operate. One of the machines can be a Lisa using MacWorks—the other way to operate this debugger is with two Macintoshes. The debuggers that go from one machine to another should not be used with machines hooked into Applenet.

The *MacDB* debugger is much more powerful than the MacsBug series of debuggers. You can use multiple windows, menus, symbolic debugging and other tools. A description of symbolic debugging comes later.

To get one of the debuggers up on your Macintosh, rename it "MacsBug" and restart the system. The welcome screen will now have the message "MacsBug installed" on it. Press the interrupt button on the programmer switch (the rear button) or press the "-" key on the numeric keypad if you have a Macintosh XL and are using MacWorks. If you put a line into your assembler program with _Debugger on it a break into the debugger will occur when that instruction in the program is performed.

The most important reason for having a debugger is for the dynamic situation in which you are running a program and in thousandths of a second something goes drastically wrong and the entire screen dissolves into a mass of swirling patterns.

What you have to do is run the program at normal speed right up to the point prior to the problem, stop it immediately, then go through the program one instruction at a time. At each instruction you verify that the program is doing what you think it should.

This implies that you both know what the program should be doing and what it is doing. You can look at all the registers since they are shown with each step. You can further check what is happening in memory since you have the DM command (explained later). So you know what is happening. Therefore the important point is to be aware of what should be happening.

To debug a program speedily takes both intuition and detective work. Your main clue consists in watching the program run and looking to see the last thing that worked correctly just before things went wrong. If the problem happens intermittently was there something that nearly always happens just before things go crazy? If there was, then single step through that part of your code.

To get started debugging you should know how to do the following:

1. list out programs in memory,
2. list data in memory as hexadecimal, strings of characters, etc.,
3. set breakpoints,
4. single step, and
5. view and change registers, the stack, etc.

With these five functions you are ready to debug programs. Now we will describe how you can use these functions to debug a program.

The ability of a debugger to turn the raw hexadecimal in memory into a program listing helps you immeasureably in finding the answers to questions such as "Where is the code in memory that I have just assembled?" and "Where am I now in executing the code?". This ability is called "disassembling."

If assembling is the ability to turn code as text into hexadecimal op codes/data then disassembling reverses the process and turns hexadecimal op codes and data back into text. Some disassemblers within debuggers can look into the linker files and find the original names of various areas in memory and thus can turn the disassembly into a listing using the original names you used in the linkage file. Such debuggers are called "symbolic debuggers" since they are aware of the original symbolic names of things. In the *MacsBug* debugger, the PX command toggles symbols on or off. (*Toggling* means that if something is on it is turned off and if off it is turned on when you use the command—it works just like a toggle switch.) Only the *MacDB* series of debuggers gives symbolic debugging for assembly language; the *MacsBug* series gives symbolic debugging only for Pascal and even then only when the ($D+) option is used.

The command to disassemble in the various Apple debuggers is "IL (*address*) (*number*)" where (*address*) is the location in memory and (*number*) is the number of lines. The parentheses are a convention; they mean "optional." So the way to disassemble a particular location in memory (say $20F5) would be "IL 20F5." Since PC stands for the program counter (where the computer is currently executing code) the most usual command is "IL PC" which shows the present place code is executing as well as the next few instructions. If you want to know what commands were just executed (or probably were, you may have JuMPed here!) you would type "IL PC-10" which would start listing out code starting $10 bytes back from your current program location. If the number of lines is omitted, a screenful of lines will be shown.

Be careful with disassemblers. If you start in the middle of an instruction it will take them a while to "synch up" —the first few instructions may be meaningless. Start disassembly 20 or 30 bytes in front of the area you think the code may be to get a true picture of the code at that location. If you have a symbolic debugger and have a label in the assembly (in this case "mylabel") then you simply type:

```
IL mylabel
```

The debugger will automatically disassemble when it is single stepping through a program. Each "step" it takes is one instruction, that instruction is disassembled.

In reality, code is only one of many forms of data that is in memory. Other data are integers, stored as two bytes, long integers, in four bytes, string data, stored as ASCII codes (for example,$20 is a space) often with a leading length byte, floating point numbers, etc. A good debugger can tell you what is in memory without any need to look up ASCII code.

A good debugger knows about each different format that data can take and gives back an appropriate display of that data. It takes much programmer time to look up each ASCII code to figure out what a string contains, for example. The *MacDB* debugger allows you to format the display of data in the window while the *MacsBug* and *MaxBug* debuggers do not.

The "DM (*Address*) (*Number*)" command allows you to Display Memory as both hexadecimal and the ASCII equivalents; each line shows 16 bytes. You will very often use an address consisting of a register.

You can actually have addresses that use symbolic numbers such as RA0 which means register A0 or RD3 which means register D3. When you write:

```
DM RA1
```

it means "show the memory pointed to by register A1." If you write:

```
DM 1004 32
```

the program would display 32 bytes (two lines of display) starting at memory location hex 1004.

Besides register A0 through register A7 (RA0-RA7) and register D0 through register D7 (RD0-RD7) there is the program counter (PC) which we have already seen. There is also "." which is the last address referenced by one of the comands.

You can also add or subtract any of these amounts together. For example:

```
DM RA0+7
```

which would show the data at seven bytes beyond the address in register A0.

You may place a breakpoint (a stopping point where you fall into the debugger) in your program with the __Debugger macro or, if you are fleet of finger, hit the rear programmer's switch just before things are about to go bad. The first step in debugging, therefore, is to get a consistent sequence which causes the blowup.

There is another way to set a breakpoint, the BR command. Its form is:

BR *address* (*count*)

This sets a breakpoint at the address you have given it. The count is used to bypass the breakpoint a series of times without stopping. If you set

the count to 3 then the first two times the breakpoint is encountered the program will not stop, but the program will stop on the third encounter. The count is optional; it is used when you have a loop which is fine until the 354th (or some other large number) time. You do not want to step through this loop needlessly hundreds of times when you know the problem always occurs after a specific large number of loops.

There are only eight breakpoints allowed at a given time.

You may clear up a breakpoint with the CL command:

CL (*address*)

If the optional *address* is omitted, all breakpoints are cleared.

Once you have stopped at a breakpoint there are a number of things you can do. You can start single stepping. Simply type in the trace command:

T

and one single instruction in your program will be performed. If you hit return another instruction will be executed. To keep stepping through the program just keep hitting the Return key. Traps are treated as single instructions (which means you don't see what is happening inside the trap handling routine). They are identified as "TOOLBOX" $AXXX in the code. All the registers are displayed with each step in the state they are before the disassembled command is executed.

If you want to go a number of steps without having to type return many times there is the "S" command:

S (*number*)

If you want to go for 100 steps you would type:

S 100

If you want to go fast up to a certain point, use the "GT" or Go Till Command. This goes full speed up to the location you have specified (a temporary breakpoint is set there, it is automatically cleared when it is encountered).

GT *address*

Sometimes you want to stop single stepping and revert to full speed operation. Then simply type "G" for Go. If you want to start executing at a specific address, specify that as follows:

G *address*

Most of the time when you use the G command you will simply want to go full speed to get out of the program or go to the next breakpoint.

The ST or Step Till command allows you to step through code without setting breakpoints. Since a breakpoint can't be set in ROM this is the

equivalent of the Go Till (GT) command to use when you want to step through a ROM. Its format is:

ST *address*

An example is ST 1F70 which would step until 1F70 and then stop.

Sometimes when you are executing the program you will want certain registers to be particular values in order to test some condition. To change a register simply type:

D*n* *expression*

to change a data register.

Example:

```
D2 100AF
```

would set register D2 to hex 000100AF. The same procedure and format apply to address registers:

A*n* *expression*

or to the program counter:

PC *expression*

or status register:

SR *expression*

Examples of these commands are:

```
A0 FE70
PC RA0+4
SR 0
```

The PC command above would add four to the contents of address register A0 (which probably points to a location in the program in this instance) and sets the program counter to that value. After the above commands, address register A0 would be set to $FE70 and the status register would be zeroed out.

To change a value in memory the Set Memory (SM) command is provided. Its form is:

SM *address expression* (*expression*) (*expression*)...

The series of expressions are translated into hexadecimal and placed sequentially in memory starting at Address. String expressions are placed into memory for their length, although any string over four characters is truncated to four characters. The width of each hex or decimal number is

the minimum number of bytes that can express the value (four bytes is the maximum).

Examples:

```
SM 1002 &258 102F 'AB' 10 'CDEFG'
```

would result in:

```
1002: 01 02 10 2F 41 42 10 43 44 45 46
```

being in memory at location hex 1002 (1002: means "starting at memory location 1002 the data is..."). Since decimal 258 is equivalent to hex 0102 this number takes up two bytes, hex 102F is just placed directly in memory, the ASCII for "A" is 41 and for "B" is 42, the value of hex 10 takes up one byte, and then the first four characters of the string "CDEFG" are placed in memory and the "G" is ignored.

There are other abilities incorporated in the *MacsBug* version of the debugger. There are commands that allow you to search through memory (F), convert data to hex, signed hex, signed decimal, and text (CV), and to monitor the Toolbox (ROM) calls and the heap (the various Ax and Hx commands). These heap commands are useful for finding what went wrong when various resources or files have incorrect pointers. Since most of the really difficult bugs involve the heap and handles that point nowhere, this is most important.

The most powerful debugger for the Macintosh is the *MacDB* series of debuggers. We haven't gone into these debuggers in great detail since the majority of our readers will have only one Macintosh. However, if you have the luxury of owning two Macintoshes or a Macintosh and a Lisa (Macintosh XL) then you will find the *MacDB* debugger much more useful and powerful.

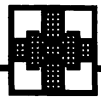
The EXEC File

The EXEC files on the Macintosh allow you to program a series of "jobs" for the machine to do. These jobs run through an assembly and link. Within an EXEC file you don't have to type in the full names of all the files to be assembled and linked. You simply run the execute file and everything is automatic.

The ability to use EXEC files is not only a time saver but avoids many typing mistakes made when you repetitively type in the same series of commands. An EXEC file looks like this:

```
Asm     Mysasm.Files    Exec    Edit
Link    Mysasm.Link     Exec    Edit
```

This would assemble the files in Mysam.Files and link them up using the commands and modules specified in Mysasm.Link. In case of difficulty, this file exits to the editor.



Summary

This concludes the Macintosh Tools chapter. By reading this chapter you have learned the intracacies of the Macintosh and Lisa assemblers, technical details of how to link programs together, and how to use resource files. The next chapter describes the Macintosh environment from the point of view of an assembly language programmer.

CHAPTER

7

The Macintosh Environment

The unique Macintosh environment was designed to be easy on the operator. Consequently, the Macintosh presents a unique environment to the programmer as well. You may think that operator simplicity will translate into difficulties for the programmer, but this is not true about the Macintosh. The special environment is very well supported in the Macintosh ToolBox. Programming it is just a matter of calling the ROM routines. The ToolBox is divided into the various *managers*, such as the Window Manager which handles windows and the Menu Manager which controls menus. These managers call each other. To understand how the ToolBox calls work, we have to pay close attention to the environment created for the operator.

The Macintosh is easy for the operator to use. Graphics mixed with text gives a pleasing appearance. Almost infinite combinations of text fonts, faces and type sizes, make the screen resemble a magazine page more than a traditional computer screen. Icons and windows make code and data seem like tangible objects, safe and reliable. Finally the fast response to menus and controls makes the operator feel in control.

A good Macintosh program stays in one "mode." The operator always knows what to expect, and always has the same freedoms. Windows can always be moved, menu items selected and so forth. These functions should be available both while the operator is actually entering data and during computation. By contrast, a traditional program switches frequently between input and processing modes. When the cursor isn't there, there is little the operator can do. Even when input is allowed, there may be no way to start a new process or quit the program directly. The traditional program may not return to a menu until it completes the current process. This can cause real frustration for the operator, but the

Macintosh presents the menus at all times, giving the operator a real sense of freedom.

An event loop is usually the center of a Macintosh program. "Events" are actions by the operator or the system which require the response of the program. The event loop receives a notice of each event and acts accordingly. Operator actions are handled first. Then, when there are no events, the loop calls routines to handle computations and update the display. Computation must be performed in small sections, between checks for recent events. This lets the program respond quickly and uniformly to operator actions.

The graphics package in the ToolBox ROM is called QuickDraw. Simple calls to QuickDraw handle both geometric drawing and text display. All QuickDraw drawing occurs in what are called "ports." A port contains all the information for drawing in a certain environment. Usually that environment is a window displayed on the screen, but sometimes a printer or other peripheral will use a port which is not displayed. The actual image the port displays is a memory area called the "bit map" of the port.

All graphics and text are drawn into the bit map of a QuickDraw port. There can be many ports, set up at different places on the screen. You can also set up a port in off-screen RAM to draw data for printing or copying to the screen later on. Each port has its own data. The data contains such information as text font, pen location and pattern for the port. It also includes a coordinate system, which defines the top left corner, or "origin" of coordinates in the port. Everything that influences drawing is saved in the port data, so drawing actions in different ports can never interfere with each other.

The desk top is the background below all of the windows on the screen. Like a window, the desk top is a port. The desk coordinate system is called the *Global* coordinate system. The ToolBox uses Global coordinates to define points on the desk top such as where the mouse was clicked, or where a new window should be placed. Whenever you make calls to the ROM for actions outside of a window, you use the Global coordinates.

The "content area" of a window is a QuickDraw port. There an application can draw the information it needs to display in the window. The coordinate system of the content area is called the *Local* coordinates. By drawing the same image but changing the origin of the Local coordinates, you can make a document scroll as though it were moving behind the window frame.

The Window Record is the same as the Port data with more information added to the end. Pascal programmers must struggle with the differences between the two data types, but they are interchangeable in Assembly language. The records are usually handled by pointers. For example, when you need a pointer to a Grafport to pass to QuickDraw, you can just pass the pointer to the Window Record the Grafport is in.

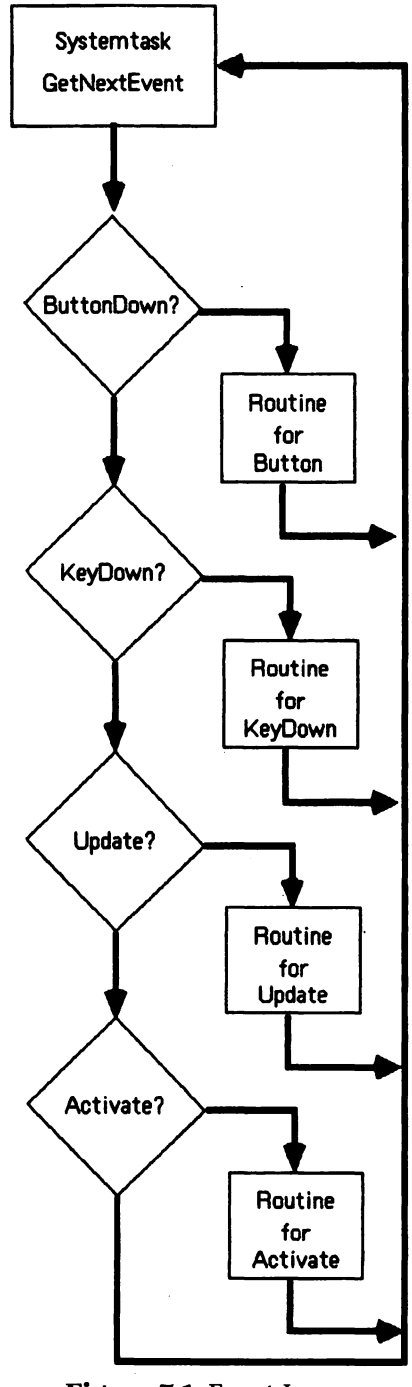


Figure 7-1 Event Loop

The window may be active or inactive. The active window is where the operator can enter data. This is usually the top window. The programmer should highlight the active window in some fashion, so the operator is sure when the window activates. Highlighting usually includes inverting or darkening any selected items, and, if scroll bars are used, displaying them only on the active window.

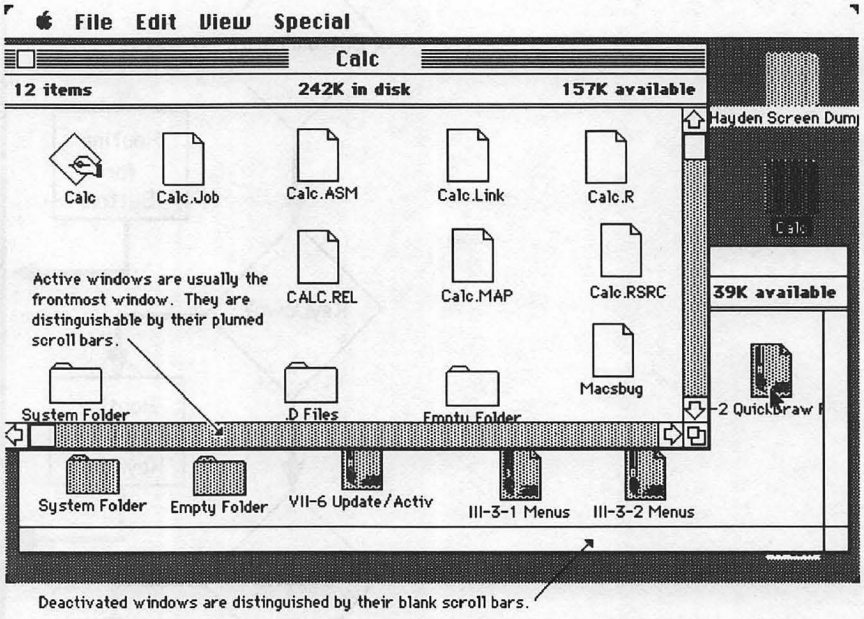


Figure 7-2 Active/Deactive Windows

By now you may have realized that a Macintosh application often draws into odd-shaped regions. When you de-highlight an inactive window, you have to confine your drawing to the area of the window that actually shows on the screen. Since that region could be under any number of other windows it could be a very strange shape indeed.

Of course Macintosh applications don't have to redraw the top windows just to make some change to windows underneath like some window-package products have to do. QuickDraw makes it easy to keep from drawing outside the window. A port has a Visible Region, the "VisRgn" data, set up by the Window Manager. The application draws all of its data, but only the bits within the Visible Region actually go onto the screen.

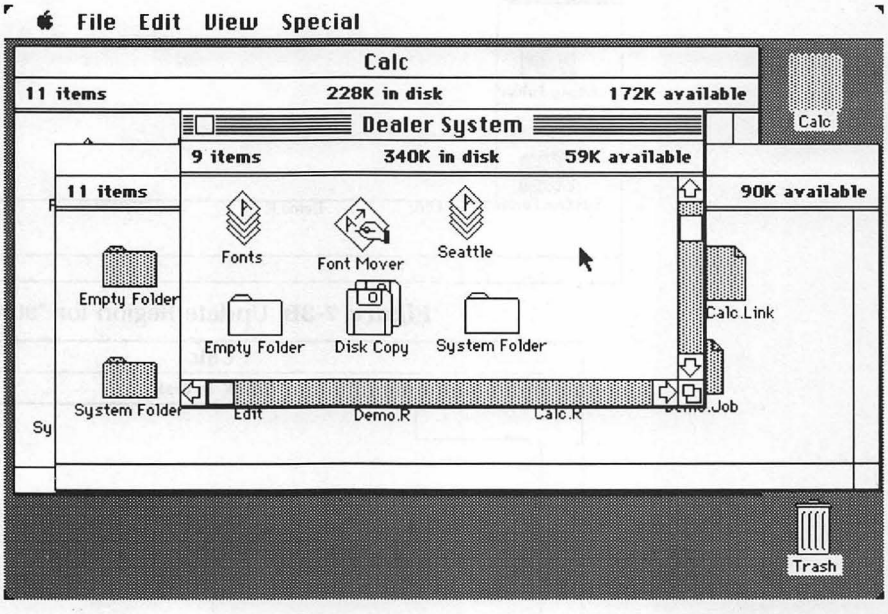


Figure 7-3 Overlapping Windows and Regions

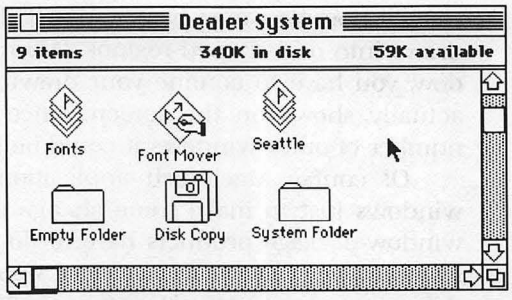


Figure 7-3A Update Region for "Dealer System"

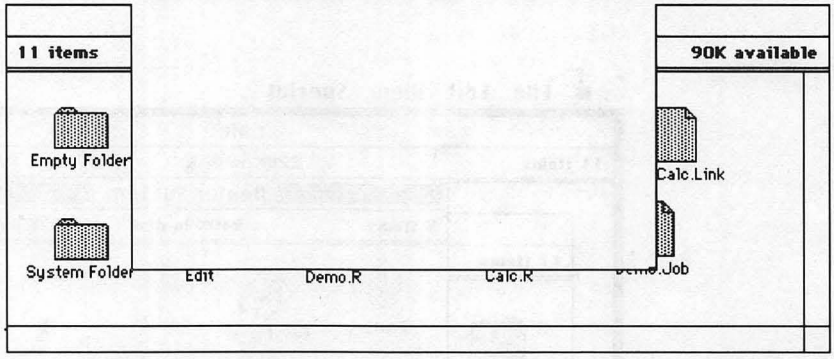


Figure 7-3B Update Region for "90K"

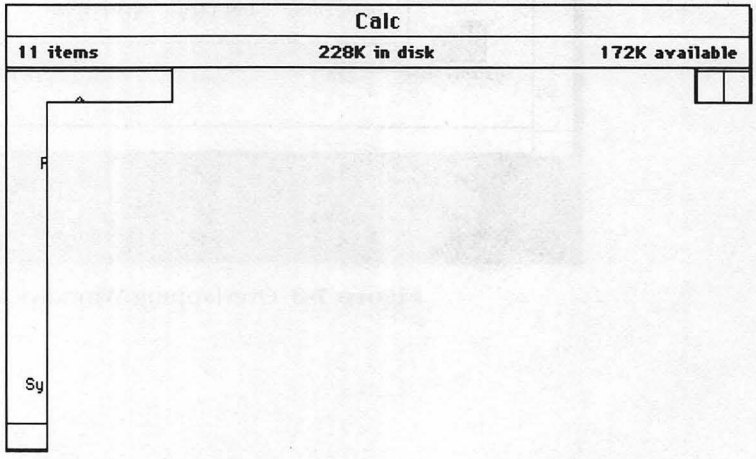


Figure 7-3C Update Region for "Calc"

Of course the graphic display of a Macintosh application may require a lot of fine-tuning. When computer screens had only text in precise rows and columns, you could design your screens on a sheet of graph paper. But you will probably find it easier to assemble and run the Macintosh

program you are working on to get a look at the screen, than to try to create it with paper and pencil. To make it easy to adjust the graphics and text displays, the Macintosh lets you keep the parameters in a special "resource" file. The source of the resource file is written in a simple language of its own. A special compiler, RMaker, quickly compiles the source into a form your program can use. When you want to vary the sizes of windows, or change text strings, keep them in the resource file. Then you only have to run the resource compiler to make adjustments, instead of assembling and linking the whole program. If you want your program to be used in foreign countries, put all of your messages in the resource file. Then, as we mentioned before when talking about the resource compiler, someone else can easily translate the messages into a foreign language. And that person won't have to have any access to your program source code!

An important goal of the system is a fast and consistent operator environment. Event-driven programs can achieve this most easily. When the operator clicks the mouse or types a character it generates an interrupt. The Event Manager services the interrupt, but instead of taking specific action, it just records the event and puts the record into the Event Queue.

A queue is a line-up of items to be worked on. It differs from a stack because the items are serviced in the order they are received. This means that new items are added to one end of the queue. Items to be processed are taken off the other end of the queue. This type of arrangement is called "First In First Out," or FIFO.

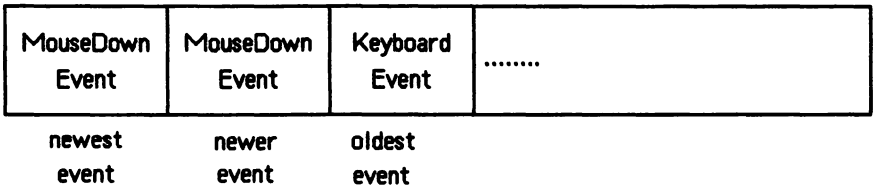


Figure 7-4 Event Queue

Both the application and the ToolBox can post events, as well as the Event Manager itself. This allows the application to send notes to itself, and lets the Window Manager signal the application when a window needs to be updated.

Making your program event-driven is really very easy. First you write the event loop which detects each kind of event. It calls routines to handle each one. Many types of events can be ignored in most programs. When

the event loop finds no events, it calls the process routines to do any computations. The only restrictions on the process routines are that they return relatively quickly to allow new events to be seen, and that they save their program state with data rather than by return addresses.

The user will cause several types of events, which you should be able to handle. These are a menu-selection event, a content-selection event, a window-control event and a keystroke. Some of these can be caused in more than one way. A menu item, for example, can be selected with either the mouse or a command key. The Window Manager can post two other types of event, an activate/deactivate event or an update event. You respond to these events by redrawing all or part of a window.

Menu-selection events are started when the mouse clicks in the menu bar, or the user presses a key with the command key held down. The Menu Manager will handle pulling down the menu and returning the result. If you have a mouse-down event in the menu bar you just call the Menu Manager routine, __MenuSelect. Then the Menu Manager will display the menus and highlight the items as the user moves the mouse up and down. Finally, when the user releases the button, the Menu Manager restores the screen and returns the ID of the menu and the item number that was chosen. When a Command key combination is detected, just pass the key stroke to __MenuKey. The Menu Manager will find the corresponding item and return the menu ID and item number.

How you respond to the menu selection depends on the application, of course. But certain menus are standard and should appear in a standard order. The Apple menu displays the "About" box and activates the desk accessories; it should be the first menu. The special Menu Manager call, __AddResMenu, will put all the desk accessories into the Apple menu for you. The About box describes your program and shows the copyright message. The File menu is second. It controls opening and closing files or ending the application. The Edit menu is the third standard menu. It contains the Copy and Paste commands. These should be given in the standard order with the standard Command key equivalents so the Desk Accessories can use them properly.

A content-selection event will start when the user pushes the mouse button inside the content area of a window. If that window is not the active window, you will first have to select it, by calling the Window Manager routine __SelectWindow. In fact, that may be all you do in this case; highlight the window with the first click which you then throw away. With the window active, you have to highlight the chosen item. You may have to do some interpretation of the event, if your application supports double-clicks or drags. A drag is when the mouse goes down in one spot and up in another. The event record has both the time and place of a mouse-down or a mouse-up event to make it easy to decipher these actions.

Window-control events start when the mouse is clicked in a control area of a window. If the mouse goes down in the top bar of the window, just call the Window Manager to drag the window around the desk top until the mouse is released. If your window has a Grow Icon, and the user selects it, the Window Manager can drag the lower right corner around the desk top for you. The Control Manager handles scroll bars and buttons for you in a similar way. Your only responsibility is to redraw the document. In fact the ToolBox will even identify which type of action a mouse click requires for you. All you have to do is call the proper routine based on the result!

A simple keystroke may be the hardest type of event to handle. However, if you are using the Text Editor, it can be the easiest. Either way, the character has to be drawn on the screen, and saved in an input buffer. You will probably need to handle Backspace and Clear as well. The ASCII value of the key is passed in Message field of the Event Record. This is a four-byte field that has different uses for different event types. The lowest byte has the ASCII value of a keystroke. The third byte has a scan code, which corresponds to the physical location of that key. This is important since keyboards for different languages may have the same letters in different positions. The two-byte Modifier field of the Event Record bit-maps which shift keys were down when the key was pressed. The shift keys include the Command, Option, Shift and Caps Lock keys. Another bit tells whether the Mouse was up or down. Generally if the Command key is down, treat the event as a menu selection rather than a key stroke.

The Window Manager may post two different kinds of events. When windows are moved around, it will post Update events. These signal you to redraw a particular window. The Window Manager creates Update events when windows are below a window which has been changed in size or moved. When the active window is changing, the Window Manager will post Activate and De-Activate events. These two event types respectively signal your application that a window is becoming the active window or is no longer the active window. Under normal circumstances, the Window Manager will move the active window to the top. if it is not already there. When you detect an Activate event, you should expect Update events to follow.

When you receive an Activate or Deactivate event, you have to figure out which kind it is, and which window it is for. This information is passed in the Message and Modifier fields of the Event Record. The Message field will be the Window Pointer of the affected window. The lowest bit of the Modifier field will be set for an Activate event. Activate and Deactivate events always come in pairs. First a window is deactivated, then the new active window will be set. When you have multiple windows you will need to keep track of the data for each one. A special, four-byte field in the

Window Record called "RefCon," can be used for a pointer to the window data. When you create a window and its matching data, store a pointer to the data in the RefCon field of the Window Record. Now when you have an Activate event, get the pointer from RefCon and use it for the current data to work on. Don't forget to show the user that the window is active by highlighting it.

An Update event is a signal to redraw a window. You have to be able to redraw any window at any time. This means you cannot draw items on the screen without a complete record of what goes where. As in the Activate event type, the Message field of Update events contains the Window Pointer. If you have more than one window, you can use this pointer to find the RefCon of the Window Record that points to the data to regenerate the window display. But even if you have only one window, you should check to make sure the Update event is for that window. It could possibly be for a desk accessory which does not need to draw itself.

When you redraw the window in response to an Update event, there is a good chance that all of the window does not need to be regenerated. A window may have been covering only a small corner of your window, and has now moved away. Or, if you have a Grow Box, a small area may have been added to the right or bottom of the visible portion of the document.

The ToolBox provides a special trick to make updating easy. The Window Manager keeps track of the region that needs to be redrawn. This region is known as the update region. When it is time to update the display, the Window Manager will substitute the update region for the normal VisRgn, the visible region of the window. Now, when the Update event appears, you just draw the whole document. Only that part which needs to be updated will actually go to the screen, since QuickDraw only draws within the visible region. After the update, the Window Manager will clear the update region and restore the original VisRgn.

The Update and Activate events that the ToolBox posts are caused indirectly by your application's own activity. When you call the Window Manager to select a window (which means to make it to the top, active window), you can expect that some Activate and Update events will probably follow. Passing a mouse click to a desk accessory might also cause a Deactivate event if the desk accessory becomes selected.

The Event Manager helps you implement the one-mode environment we describe at the beginning of this chapter, but sometimes you do need a more specialized environment. You may need answers to some specific questions before you can execute a function that the operator has selected, for example. For these situations you can use dialog boxes.

A dialog box is a special kind of window. It is designed to pose a question and return some sort of reply. But you can also use a dialog box to simply display some information and ignore the response. The About box, which shows the copyright message for a typical program, is usually a

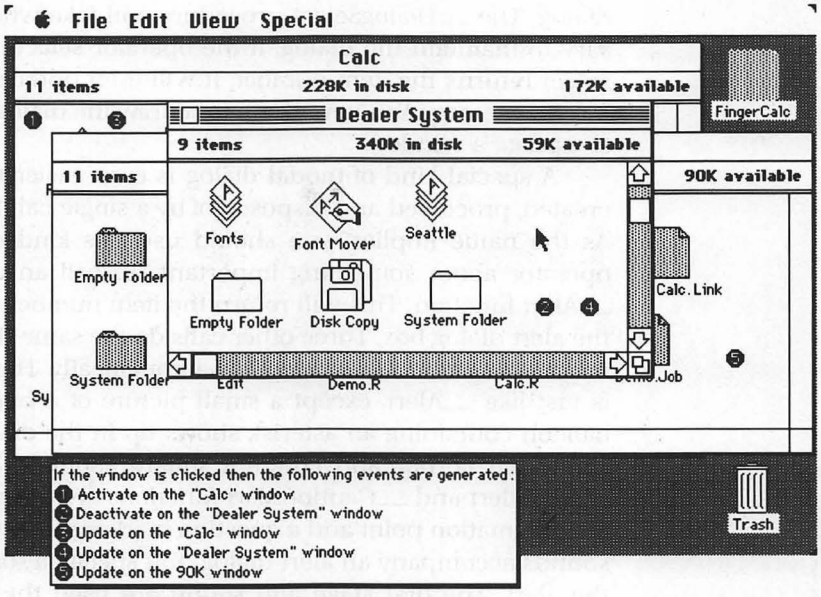


Figure 7-5 Update/Activate Events for a Series of Windows

dialog box used this way. Dialog boxes are easy to use because the Dialog Manager handles all of the details for you. You don't even have to draw the box!

You can declare the items to be displayed in a dialog box in the resource file. To display the dialog, you just call the Dialog Manager with the ID of the dialog resource. The Dialog Manager will draw the box and display the items inside, following the list in the resource file. The item list can include text, buttons, controls, icons and even fields that the operator can edit. If you want to wait for a reply from the operator, you call __ModalDialog. Now the operator can only choose items from the dialog box. Clicking the mouse outside the box just causes a beep sound. When a valid choice is made, __ModalDialog returns with an integer, the item number of the selection. The item number is set by the order the items are declared in the resource file. A value of one is passed if the operator presses the Return key. This is the default item. You should always make the safest choice be the first item, but you can place it anywhere in the box that seems logical.

If you don't call __ModalDialog, the dialog box is like a window but with some important differences. The Dialog Manager can tell you which item in which dialog box the operator is working on. For each event you call __IsDialogEvent to see if it belongs to a dialog. You call __DialogSelect if it does, or handle the event for your window if it does not belong to a

dialog. The __DialogSelect procedure will take whatever action is necessary to maintain the dialog. If the operator selects something, __DialogSelect returns the item number. It will even update or activate the dialog box if necessary. You never have to redraw the dialog box the way you have to update a window.

A special kind of modal dialog is even easier to use. An alert box is created, processed and disposed of by a single call to the Dialog Manager. As the name implies, you should use this kind of dialog to warn the operator about something important. To call an alert you can use the __Alert function. This will return the item number of the item selected in the alert dialog box. Three other calls do the same thing, but each adds an impressive icon to the dialog box automatically. The __NoteAlert function is just like __Alert, except a small picture of a face with a cartoon style balloon containing an asterisk shows up in the dialog. The icon is in the upper left corner, and covers a square about 50 pixels on a side. The __StopAlert and __CautionAlert functions are the same, except they have an exclamation point and a question mark respectively. You may even have sounds accompany an alert dialog. You specify a sound for each "stage" of the alert. The first stage and sound are used the first time the alert is called. If it is called twice in a row it goes to stage two. This continues up to the highest stage, stage number four. Calling any other alert sets the stage counter back to zero. The next time you use this alert, you'll hear the first stage sound.

The Dialog Manager provides another service, too. It will report a system error. You will probably get used to seeing the "Bomb" message indicating a system error while you are debugging your program. The error numbers, shown in the "ID=" field, are listed in Appendix D. After the error dialog, you can arrange to have control pass back to a special "Restart Procedure" that you have coded. This gives some chance to correct the error or at least salvage the data in an otherwise fatal situation. To do this, when you initialize the Dialog Manager, pass the address of your restart procedure. But if you don't want to bother with all this, pass zero instead.

Aside from calling dialogs and alerts, you should keep the screen as nearly consistent as possible. Still your program will probably have to have some "modality." Some menu items will not be appropriate at all times. Certain actions with the mouse may not always be possible. You should indicate these changes to the operator in a standard fashion. Usually, you will put all possible items into the menus. When certain items are not appropriate, you can disable them with the __DisableItem call. When they are allowed again, use __EnableItem to turn them back on. The editing states can be handled by changing the mouse cursor. Calls to __InitCursor turn the cursor into an arrow. You can change the cursor to predefined shapes by calling __SetCursor. An I-Beam for editing text, a plus for editing grahics, a watch for delays, and more cursors are available. If you want to

design your own cursor you can do it in the resource file. An example in Appendix F (Traps) shows how to do this.

We have already described the resource file. Many of the ToolBox calls can take their parameters from the resource file. They do this by calling the Resource Manager. But your program can also call the Resource Manager directly. The most important thing you may need to do, is to open a resource file.

There are usually several resource files open, when your applicaton is running. The System resource file is opened first. Your application code itself is a resource file. The __OpenResfile call opens additional files for your application. When it looks for a resource, the Resource Manager always starts from the most recently opened file. This means you can supercede definitions in the System resource file if you want to.

We mentioned that the application code is in a resource file. In fact, every Macintosh file has both a data and a resource "fork." This amounts to two files for every catalog entry, one for the data and one for the resource. The code of an application is in the resource fork of a file. The resources used by that code can be in the same file, or in a different file. The Resource Compiler on the Lisa accepts the type "CODE." You can declare a linked object with this type, and the compiler will build one file containing both application and resource data. In Appendix E, Using The Lisa WorkShop, we show how to do this.

Each resource in the resource file belongs to a four-character type. For example, dialogs are type "DLOG" and windows are type "WIND." You can even make up your own types for storing specialized data for your application. Each specific resource has an ID integer. This number only needs to be unique among other resources of the same type. The exact syntax of the Resource Compiler language was described in Chapter 6, Macintosh Tools. An example of a working resource source appears in the next chapter.

You should follow a few standards in creating your resource file to be compatible with the System code of the Macintosh. To avoid conflicting with the System Resource File unintentionally, you should make all of your ID numbers greater than 256, with one exception. The standard Apple menu should always be ID number one. The edit menu can be any ID greater than 256, but the contents should be compatible with the desk accessories so they can use it also. The desk accessories may support up to five actions, Cut, Copy, Paste, Clear and Undo. These should be arranged in your edit menu in the standard order. Some of the commands have standard Command-key equivalents. Figure 7-6 shows the Standard Edit Menu. The way it looks on the screen is shown to the left. The resource file entries are shown in the center column. The suggested command keys appear in the resource entries, separated by "/" (a slash). The right column of numbers shows the values passed to the Desk Accessories with the

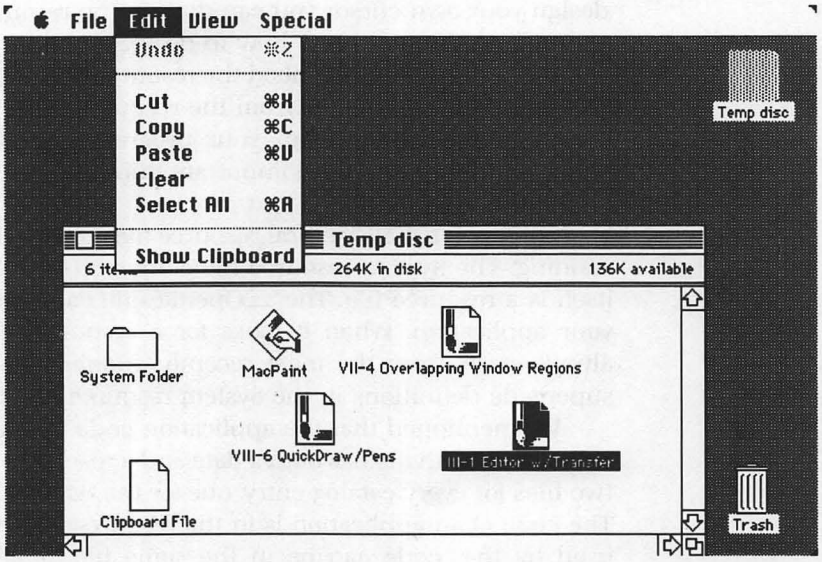


Figure 7-6 Edit Menu

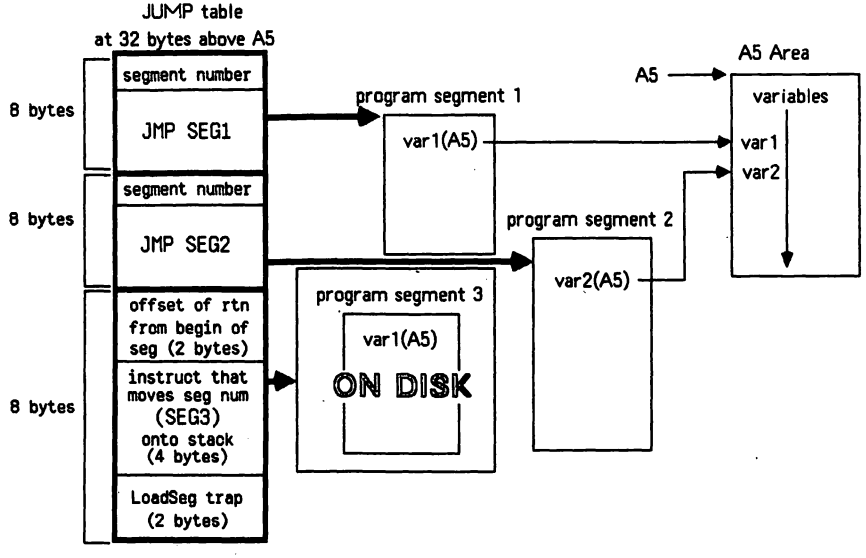
__SystemEdit call. If you use the Standard Edit Menu, you must subtract one from the item number before calling __SystemEdit. If you create a different menu, try to keep it similar, and calculate the values for __SystemEdit as necessary.

Memory management is particularly flexible on the Macintosh System calls to the Memory Manager will give you large blocks of RAM, which can be either relocatable or in fixed locations. The same system keeps track of space allocated by calls to the Dialog or Window Managers. If you want to allocate your own memory space, there are several ways to do it. You can allocate space in your program, allocate space from the Global Variable area, or simply take space from the stack area. None of these mechanisms will interfere with the memory manager itself.

The Memory Manager can recover space used by inactive code to create new blocks or to expand old ones. But you don't have to worry about part of your program disappearing without warning. The only application code that will be purged from memory are segments that you have marked purgeable with the __UnLoadSegment call. These must be separate segments you have created with the Linker on the Macintosh, or with the .SEG statement in the Lisa Assembler. You cannot have any return addresses or pointers into unloaded segments, but you can enter the segment at any time.

All of the calls between segments go through the Jump Table. This was explained in Chapter 6 (Macintosh Tools). If the segment is not in memory,

the Jump Table does not contain a JMP to the entry point. Instead, it sends the call first to the Segment Loader, and then to its destination in the segment. The Segment Loader will call the Resource Manager to read the code into memory. Then the Segment Loader will change all the entry points for that segment to the proper JMP instructions, before branching to the destination routine. If you have marked a segment as purgeable with the _UnLoadSegment call, the memory manager may reclaim the space. Then it will change the Jump Table entry back to the trap instruction which calls the Segment Loader the next time the entry is accessed.



Each segment is pointed to by an 8-byte entry in the jump table which is maintained by the segment loader. The entry is simply a jump plus segment number if the entry is in memory; if not in memory, the entry consists of a call to a routine to bring the segment in from disk. Sort of ingenious really.

Figure 7-7 Jump Table and Segments in Memory

You can allocate memory within your program by using the "DC" directives as described in Chapter 6 (Macintosh Tools). This is especially good for defining constants. You can also use it for variable areas which

need to be initialized to a constant value, but we do not recommend this. The 68000 was designed with Pascal in mind. Pascal implementations usually separate the code and data areas of memory quite distinctly; the instruction set of the 68000 was not geared to modifying the program space. Although you can keep variables in the code space on the Macintosh, they will be more difficult to access. What is worse, if you use segmentation in your program, the variables may mysteriously revert to their initial values if the segment is paged out and reloaded.

If you don't allocate variables from code space, how do you do it? The most versatile way is to take space from the stack. You can set an address register to point to your block and address your variables as constant offsets from that register. The LINK and UNLK of the 68000 were especially designed for allocating memory this way. But notice that when you use the LINK instruction, the constant should be a negative number, and your offsets will have negative values. You might want to review the LINK and UNLK instructions in Chapter 3 (68000 Instruction Set) before going on. Pascal uses LINK and UNLK to allocate space for local variables. Pascal always uses address register A6 for these instructions. Let's use a typical Pascal procedure, such as this do-nothing procedure we'll call NADA:

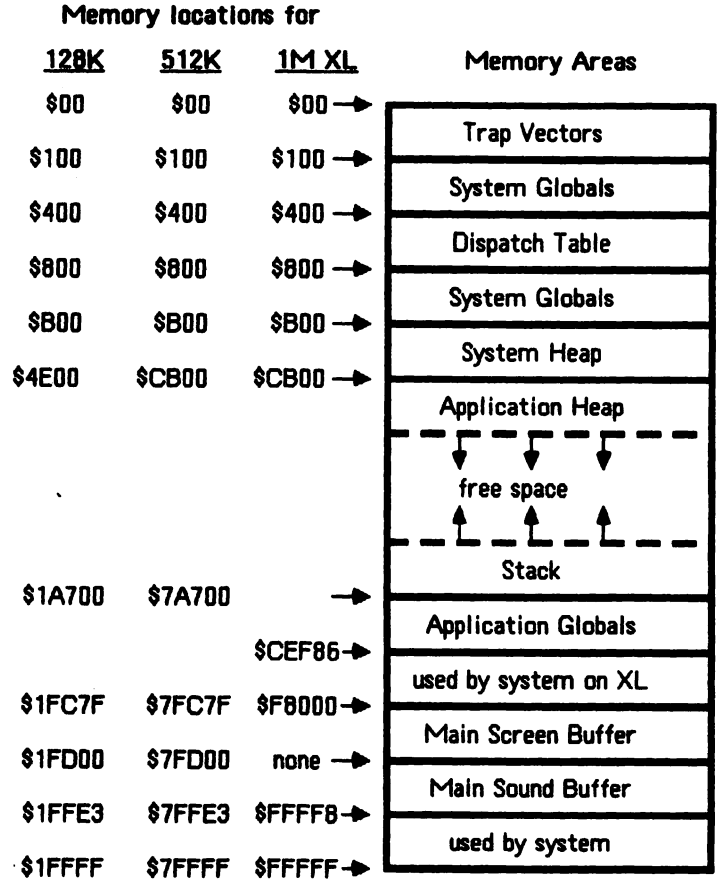
```
PROCEDURE NADA;
VAR X,Y : INTEGER;
BEGIN
  X := Y;
END
```

The NADA procedure will compile to a structure such as this:

```
NADA  LINK  #-4,A6          ; Allocate four bytes from the stack
      MOVE  -2(A6),-4(A6)   ; Assign Y to X, both local variables
      UNLK  A6              ; Return stack space.
      RTS
```

You can use this mechanism repeatedly, only taking up the memory space for which code is actually in use. This sits well with the Memory Manager also. The memory space allocated by the Memory Manager is taken from the Application Heap. If you refer to Figure 7-8, you will notice that the Application Heap builds up toward the stack pointer, while the stack builds downward. This allows the stack and heap to grow into the same free space, by nibbling away at it from both ends. The ToolKit even checks to be sure your stack does not collide with the heap. It does this during every vertical retrace interrupt, which happens sixty times per second.

You can also allocate memory during assembly. If you use the "DS" directives in the Macintosh Assembler, described in Chapter 6 (Macintosh Tools) global memory space will be allocated relative to register A5 when



Note 1: There is no sound buffer in a Macintosh XL.
Note 2: There is no gap between Application Globals and the Mac Screen Buffer on a Macintosh, but there is gap on the Macintosh XL (used by system).

Figure 7-8 Memory Map

your program starts. You can achieve the same effect in the Lisa Workshop Assembler by defining global variables in your Pascal dummy program.

Whether you use global variables or not, you should never alter A5. This register is used by the ToolKit for system variables as well as for the Jump Table. Since you cannot use A5 anyway, the "DS" directives give you RAM space for your variables without dedicating an extra address register.

There is one more way of allocating memory. When you use a literal string in an instruction in the Macintosh Assembler such as,

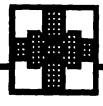
```
PEA "Where did it go?"
```

you may wonder where the ASCII went. The Assembler converted the string to a Pascal form and placed it at the end of the code. The PEA instruction above is equivalent to:

```
        PEA LengthByte
        ...
LengthByte DC.B 18
```

Since this structure is equivalent to just defining strings yourself with the "DC" or ".ASCII" directives, it should be used with the same restrictions put on data defined in code space.

To use the ToolBox you need lots of definitions. The appropriate include files have to be in every program. These contain macro definitions for the traps as well as data definitions. If you are using the Lisa Workshop, you have a lot of RAM space, and must have a hard disk. A Macintosh Assembler environment may be a little more limited, so a special type of file, a dump file, helps the Assembler shuffle through all of this information much faster.



Summary

Now that we have taken a brief look at the Macintosh environment, we are ready to start delving into the ToolBox. In Chapter 8 you will see how to call the ToolBox and Operating System from your program.

CHAPTER

8

The Macintosh ROM Calls

The key to programming on the Macintosh is learning how to call the subroutines provided for you in the ROM and Operating System. At first, you may have thought the Macintosh environment would be difficult to create. But in the last chapter you saw that is simply a matter of calling the routines already written for you. Now you will examine the structure of those calls. You will study how parameters are passed and how the routines are accessed. Along the way you will learn about QuickDraw and learn some of the more common ROM calls. After seeing these examples, you will be able to figure out how to use most other Macintosh system calls in the Appendices or Apple documentation.

Calling the Toolbox

The Toolbox calls were designed to interface to Pascal. Since all parameters are passed on the stack in Pascal, the Toolbox type of interface is called the stack-based interface. Some of the calls are like Pascal procedures which return no result. Other calls are like Pascal functions which can return only one value. We are going to discuss the parameters for some of the calls. Appendix F, the Trap Appendix, shows a few of the more common calls in Assembly language. The rest of the calls are shown only in their Pascal form. After you have a little experience, you will be able to use the call in Assembly language by translating the Pascal form.

Pascal procedures and functions use the stack for passing parameters and returning results. After the required data is pushed onto the stack, a JSR calls the routine, placing the return address on top of the stack. The ROM calls use the same data structure, but they don't use a JSR to reach the subroutine. Instead, a special instruction called a "Trap" transfers

control to the subroutine. A special macro has been defined for each trap. This is usually the name of the Pascal call preceded by an underscore. The routine called "InitCursor" in Pascal, for example, has a macro called "__InitCursor" in assembly language. The trap macro works just like a JSR from the point of view of the application. Just write "__InitCursor" (for example) to call the ROM routine with the return address on the stack.

You don't have to know exactly how the traps work to use them. The Toolbox makes them function just like JSR subroutine calls. Although you don't have to understand the traps to use the macros, we think a brief outline will help you get a better grasp on Toolbox calls. If you feel you don't need to worry about this, you can skip the next three paragraphs.

The trap calls do not use the "TRAP" instructions. Instead, they use a special op code which is unimplemented in the 68000 instruction set. When this op code is encountered, the 68000 vectors to a special handler, the same way it does for a "TRAP" or "CHK" instruction. The unimplemented op code is a hex ten in the first four bits of the instruction word. Thus, a trap can be any instruction of the form:

$A*xxx*.

where "x" means any hex value. The last three nibbles are used to specify the trap being called. This allows 16 to the third power, or 4096 possible traps. The "TRAP" type instructions can only accommodate 16 different calls, one for each instruction and matching vector.

The unimplemented instruction vector leads to a routine called the Trap Dispatcher. The job of this routine is to decode the trap call and jump to it, just as though the application code had used a JSR to the subroutine. As you remember from Chapter 3, 68000 Instructions, the vector operation leaves the address of the instruction and the status register contents on the stack. The Trap Dispatcher uses the instruction address from the stack to get the original instruction word. The upper four bits are always $A of course, but the lower bits contain the trap number, telling which routine needs to be called. The Trap Dispatcher uses the trap number to look up the address of the routine in a special table, the Dispatch Table. Finally, the Trap Dispatcher sets up the stack to look as if a JSR has just been executed. A return address on top of the stack points to the address just after the trap instruction in the user code, then it jumps to the address it found in the Dispatch table. This completes the interface just as though it had been done by a JSR.

The Dispatch Table contains the addresses of the trap routines. These routines are normally in ROM, and their addresses don't change, but Apple had the forethought to send all of the routines through a table anyway. Now, when any changes have to be made to the ROMs, the chips don't need to be replaced. Instead, a new software release can load the patches into RAM and change the table to point to the improved code, instead of the old ROM routine.

A Toolbox procedure is called with the parameters on the stack. You push all the parameters in the order in which they appear in the procedure declaration. Then you execute the trap to call the procedure. When the call returns, all of the parameters will be gone. The stack is left as it was before the call. If the procedure does not have any parameters, use just the trap macro. This procedure, InitCursor, does not have any parameters. Here is the Pascal form:

```
procedure InitCursor;
```

To call InitCursor from assembly language, use:

```
__InitCursor
```

Routines that have parameters described as "INTEGER" in Pascal, need a two-byte, two's-complement integer on the stack. The Pascal type "LONGINT" is a long integer. It requires a four-byte integer. Other types of parameters need different data on the stack. The data and size for each type of parameter is shown in Appendix F, Trap Appendix. We will explain the most common ones later on in this chapter. For now, we can call a Pascal procedure such as,

```
procedure HiLiteMenu (menuD: INTEGER);
```

In Assembly language:

```
MenuID DC.W 302
...
MOVE.W MenuID,-(SP) ;ID of menu, an integer
__HiLiteMenu
```

A Toolbox function is also called with the parameters on the stack, but first you have to make room for the result. A Pascal function definition specifies the type of the result the same way as the type of a parameter. All you have to do to make space for the result, is clear the same space on the stack that a parameter of the same type would occupy. When you have loaded the result space and parameters, just use the trap macro to call the function. When the call returns, the parameters and return address will have been removed. The result will remain on the stack, just where you created space for it. You can either pop the result into its destination, or leave it on the stack if it will be the parameter of a future call. A routine to read the Macintosh clock, in ticks or intervals of 1/60 second is a function in Pascal:

```
function TickCount: Longint
```

Use this code to load the number of ticks since power-on into D0:

```
CLR.L  -(SP)      ;Make space for long integer result
__TickCount       ;Call the TickCount function
MOVE.L (SP)+,D0 ;Result to D0
```

Pascal only allows a function to have one two-byte or four-byte result. You will always be clearing two or four bytes on the stack for a function result. Many routines return a longer structure but cannot be defined as functions. Instead they are defined as procedures with the results returned in variables. A few routines return two values packed into a four-byte result. Pascal has difficulty separating the values, but in Assembly language you can actually pop one integer at a time from the stack.

The Operating System calls are usually different from the Toolbox calls. The parameters are passed in registers or in memory, instead of being passed on the stack. This type of interface is called the register based interface. If a single data value is passed, it is placed in D0 before the trap. If an address value is needed, it is passed in A0. Longer data structures are first set up in memory. Then the trap is called with a pointer in A0. The pointer should be the address of the first word of the data structure. For example, the Operating System call to drain the Event Queue is defined in Pascal as:

```
procedure FlushEvents (eventMask,stopMask: INTEGER);
```

Since this is an Operating System call, the data is passed in D0, like this:

```
MOVE.W  #StopMask,D0  ;Events to stop scan
SWAP    D0
MOVE.W  #EventMask,D0 ;Events to remove
__FlushEvents
```

Both kinds of call preserve the same registers. The scratch registers are registers D0, D1, D2, A0 and A1. The data registers numbered D3 and above, and the address registers numbered A2 and above will not be altered by the call. The lower registers may be changed, or used to pass parameters. You can keep your data in the higher numbered registers, and know it will be the same while you call the Toolbox or Operating System.

The register A5 has special uses throughout the system. This register points to the Global Data Space. You should never change A5 because QuickDraw, the Segment Loader, and many other routines require data from the Global Data Space. The Macintosh Assembler will allocate variables there if you use the DS instruction. If you are using the Lisa Assembler, you can create global variables in Pascal, and access them from A5 in assembly language.

The length and structure of parameters to pass on the stack is sometimes difficult to figure out. A table in Appendix F, Trap Appendix, shows each type of parameter. Some general rules make it easy to remember what to pass in most cases. If the Pascal parameter is defined with a VAR, use a four-byte pointer to the data. It could be modified by the

call. If the data is longer than four bytes, use a pointer. It won't be modified by the call if the Pascal definition does not describe the parameter as a VAR. Pass a two-byte value for a Pascal "INTEGER." Pass a four-byte value for a Pascal long integer, "LONGINT." Always pass four bytes for an address or a pointer. These rules describe most cases, but there are some special cases.

Some calls may accept a pointer but do not require it. In Pascal, you may pass NIL if you are not using a pointer. The equivalent of the empty pointer, NIL in Pascal, is a four-byte zero. When you initialize the Dialog Manager, you pass a pointer to a restart procedure to take care of system errors. If you don't want to bother with a restart procedure, you pass NIL instead. The Pascal definition is,

```
procedure InitDialogs (restart:ProcPtr);
```

where a ProcPtr is a pointer to a procedure, in other words, a code address. If you have a restart procedure named "ReStart" call the InitDialogs procedure like this:

```
        PEA         ReStart   ;Pointer to ReStart subroutine
        _InitDialogs          ;Initialize Dialog Manager
        ...
ReStart
                              ;This is the restart procedure. It
                              ;has no parameters.
        ...
        RTS
```

If you do not have a restart procedure, and usually you don't need to have one, you can pass NIL in the call like this:

```
        CLR.L  -(SP)          ;NIL pointer for restart procedure
        _InitDialogs          ;Initial Dialog Manager
```

Pascal programmers sometimes have to go through complex machinations to defeat the language's type enforcement. There are several different types of pointer defined for the Toolbox in Pascal. Usually, the symbol for any type of pointer will end in "...PTR" to give you a clue that it is a pointer. All pointers are the same in Assembly language, a four-byte address. But in Pascal, a particular pointer type can only point to one kind of record. A four-byte integer, a "LONGINT" in Pascal, is also not interchangeable with a pointer. The Pascal functions, "ORD4" and "POINTER" convert between the two types. These functions have no effect on the data so you need no equivalent in Assembly language.

Two other Pascal functions which have no effect on the data, are the "CHR" and the "ORD" functions. These two convert between characters

and integers. A character is represented by its ASCII value as a two-byte integer. This means the data is in the lower byte instead of the upper byte. The DrawChar procedure draws one character on the screen. It is defined in Pascal as:

```
procedure DrawChar(ch:CHAR);
```

If we want to put a zero on the screen, which is the ASCII character $30, we can use this:

```
MOVE.W #$30,-(SP)   ;Put $0030 which is a '0' on the
                    ;stack
__DrawChar          ;Draw the character
```

True or false values are called Boolean variables in Pascal. Although only one bit is required to represent them, the interface passes an entire word. The true or false value is in bit zero of the high-order byte. The two possible values are:

```
$0100 means TRUE
$0000 means FALSE
```

We recommend that you always pass Boolean parameters with these values. But since only one bit is significant, you should check that bit only, without counting on the other bits in the word being set to zero. To test a Boolean parameter returned on the stack:

```
BTST #0,(SP)+   ;Check result
BNE TRUE        ;Function returned True
BEQ FALSE       ;Function returned False
```

One of the most useful Toolbox variable types is the *handle*. A handle is simply a pointer to a pointer. This may seem like a needless complication at first, but it helps you off-load much of the work of space allocation to the System. Handles are always four bytes long. Those four bytes contain the address of another four bytes. This second four bytes contain the address of the data. The reason why handles are so useful though, is that you hardly ever have to unravel them this way. Instead, the Toolbox will allocate memory space for a structure such as a menu, and pass a handle back to your application. When you call the Menu Manager again, you just pass the handle and the Manager can reference the data.

How do handles help the Memory Manager allocate memory? The key to this mystery is that allocating memory is not as big a problem as recovering space for reuse. An application starts with one large block of memory it can use, which is called the heap. When the program is

running, smaller blocks of memory are constantly being created from the heap and returned. They are not necessarily returned in the order they are created. Scattered blocks throughout the heap will remain till the end of the program. This kind of activity tends to break usable space into smaller and smaller pieces. Eventually, a new block of modest size cannot be created, because the heap is divided into small sections. The blocks that divide the heap occupy little space in themselves, but they divide the heap so that a larger block cannot be created in one piece. This situation is called a "fragmented" heap. An illustration of a fragmented heap is in Figure 8-1.

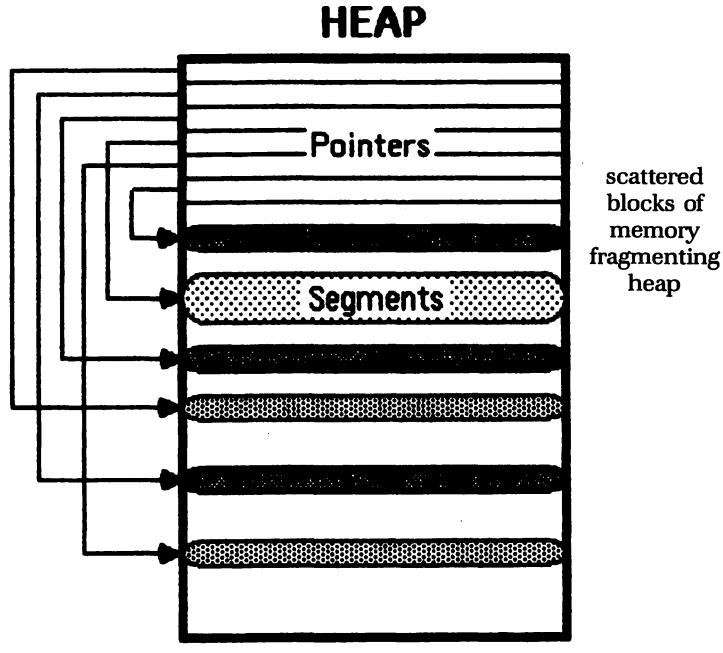


Figure 8-1 Heaps and Pointers (Fragmented)

If the data blocks in the fragmented heap can be moved, the heap can be repaired. All we have to do to make a fragmented heap usable, is shift all of the blocks as far as possible toward one end of the heap. This process is called compacting the heap. Figure 8-2 shows the heap after compacting.

A fragmented heap is easy to repair by compacting it. But the Memory Manager cannot compact the heap if it has been passing out pointers to

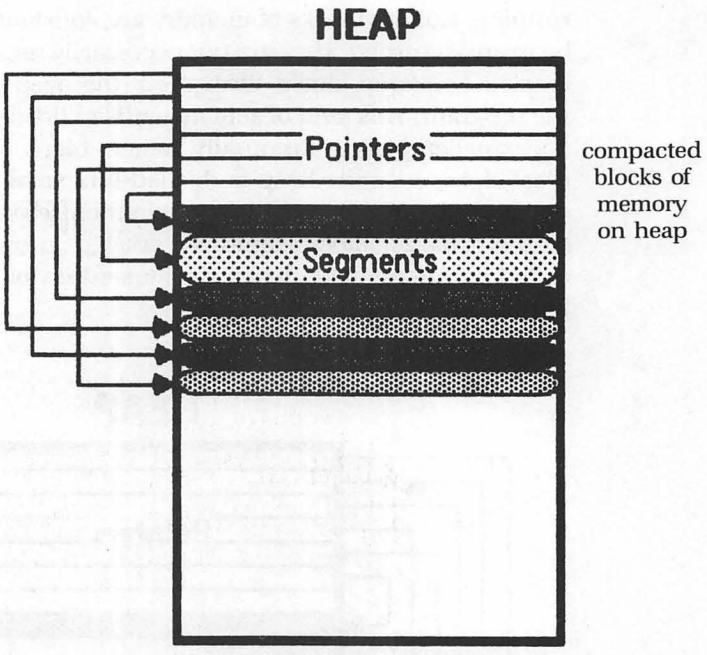


Figure 8-2 Heaps and Pointers Compressed

the blocks. The application program has saved the pointers which the Memory Manager provided. If the blocks move, all is lost. Instead, the Memory Manager puts all the pointers in a small reserved area. It tells the application where a particular pointer is by passing a pointer to that pointer, a handle. Now when the Memory Manager has to move the blocks, it just changes the pointers to point to the new data locations. The blocks with handles are shown in Figure 8-3.

These blocks are called relocatable, because the Memory Manager can move them to make space. All you have to do is always access them through the handles. If you ever need to dereference a handle, the code to do it is simple. This example resolves the handle to a byte, and puts the byte into D0.

```
MOVE.L   Handle(A5),A0   ;Get handle from storage
MOVE.L   (A0),A0         ;Get pointer
MOVE.B   (A0),D0         ;Get byte
```

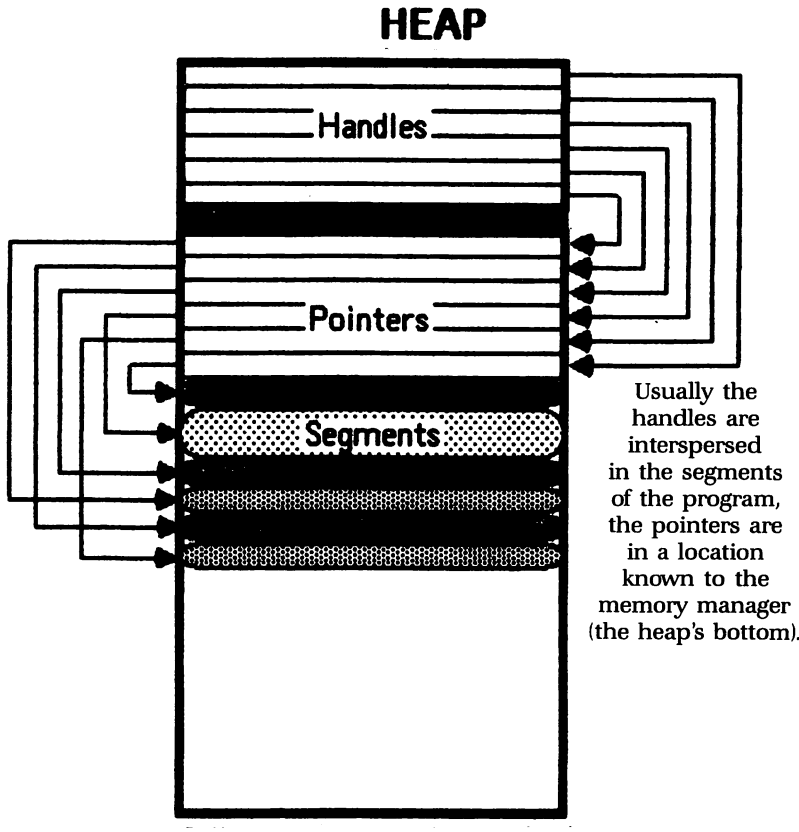


Figure 8-3 Heaps and Handles (Handles Pointing to Pointers...)*

Some calls use a complex structure defined as a "RECORD" in Pascal. To use a record as a parameter, you have to create a similar structure in memory and place your values in the proper fields. If the record parameter is identified as a VAR, values may be returned to you in some fields of the record as well. You pass a pointer to the record as the actual parameter for the call, unless the record is no more than four bytes long and not a VAR.

The fields of one variable are stored in memory sequentially, in the order in which they are declared in the record. The space taken by each field depends on its type. The size of a field is generally the same as the size of the data you would pass for a parameter of the same type. A Pascal record type containing two integers may be defined like this in Pascal:

```
TwoInts = RECORD
  ValOne : INTEGER;
  ValTwo : INTEGER
END;
```

To create a variable of this type, you just define space for two integers:

```
VRValOne    DS.W   1          ;Space for ValOne integer
VRValTwo    DS.W   1          ;Space for ValTwo integer
ValRecord   EQU    VRValOne   ;Address record by first field
```

Once you know the length of each field, you can create and use record parameters for any type of Pascal call.

As we said above, the size of a field is usually the same as the size of a parameter of the same type. There is one important exception to this rule; two sequential Boolean fields can occupy one word. Remember that we pass a Boolean value in the upper byte of a stack word. A Boolean is really only one byte, containing a one or a zero. The lower byte of the stack word is, in fact, just fill. The stack stays aligned on a word boundary, whether we move a byte or a word onto it. Likewise, most of the fields in memory have to be aligned to word boundaries, so an extra byte of fill will usually be added after a Boolean. It is only when two one-byte fields appear in a row that the extra byte of fill can be omitted. Here we have a record that contains a Boolean between two integers:

```
Fill= RECORD
      ValOne   : INTEGER;
      LongBool : BOOLEAN;
      ValTwo   : INTEGER
  END;
```

The equivalent Assembly language definition uses two-bytes for the Boolean:

```
FValOne     DS.W   FValOne    ;Space for ValOne integer
FLongBool   DS.B   1          ;Boolean value
            DS.B   1          ;Fill to maintain alignment!
FValTwo     DS.W   1          ;Space for ValTwo integer
FillRec     EQU    FValOne    ;Address record by first field
```

But if there are two Booleans together, like this,

```
NoFill = RECORD
  ValOne : INTEGER;
  BoolOne : BOOLEAN;
  BoolTwo : BOOLEAN;
  ValTwo : INTEGER
END;
```

the equivalent definition squeezes both Booleans together:

```
NFValOne    DS.W   1          ;Space for ValOne integer
NFBoolOne   DS.B   1          ;First Boolean value
NFBoolTwo   DS.B   1          ;Second Boolean value
```

```
NFValTwo    DS.W   1          ;Space for ValTwo integer
NFillRec    EQU    NFValOne   ;Address record by first field
```

The key to remember is word boundary alignment. One-byte Booleans don't need to be aligned, but most other fields do.

The fields of most records are defined for you in the include files. The fields are defined as offsets from the start of the record. You can use these labels in expressions to address the fields. Just add the offsets to the address of the start of the record. When you have an address register pointing to the record, you can use the offset in an indexed addressing mode. If the TwoInts record as defined above, were a part of the Toolbox there would probably be offset definitions for both fields such as these:

```
                                ;TwoInts Structure
ValOne      EQU    0
                                ;ValOne : INTEGER
ValTwo      EQU    2            ;ValTwo : INTEGER
```

You can make it easier to keep track of the fields with these labels. Put the name of the variable together with the offset label to make your own name for the field, as we have been doing above:

```
VRValOne    DS.W   1          ;Space for ValOne integer
VRValTwo    DS.W   1          ;Space for ValTwo integer
ValRecord   EQU    VRValOne   ;Address record by first field
```

Now you can address the fields symbolically in two ways. For example, you can clear and test the second integer like this:

```
CLR.W    VRValTwo(A5)      ;Set ValTwo of the record to zero
...
LEA      ValRecord(A5),A0  ;Point to a record
TST.W    ValTwo(A0)        ;Check ValTwo of that record
```

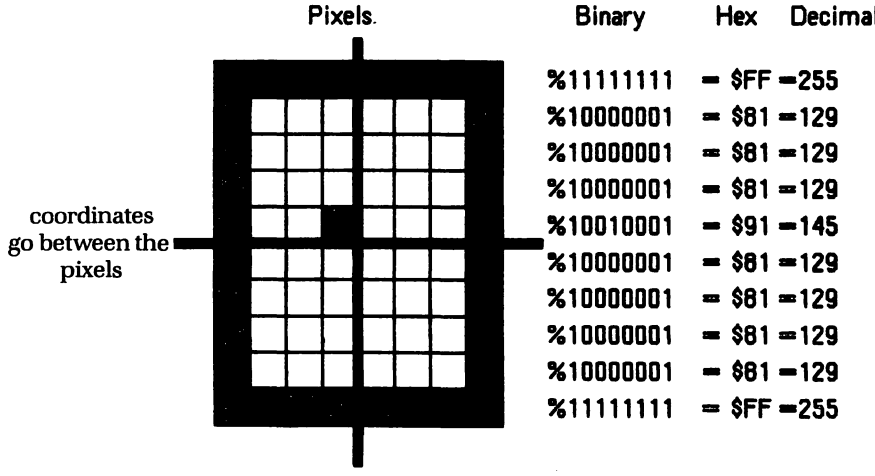
This is as simple as a higher level language, only more efficient!

Calling the QuickDraw Graphics Package

The QuickDraw drawing environment is an unusually powerful graphics package. We cannot describe all of the capabilities of QuickDraw in this book. We are going to show you some of the most useful features however. When you are ready to do even more, you can refer to *Macintosh Revealed* (Vol. 1), by Stephen Chernicoff, published by Hayden Book Company.

A pixel is a dot described by one bit. If the bit is one, the dot is black. If the bit is zero, the dot is white. When you erase an area of the screen, you make it white. As you draw text and graphics, the pixels change from zero to one, as you add black lines and letters.

A bit image is a section of memory used to store pixels. QuickDraw drawing commands, change the bits of the words in the bit image. The Macintosh screen is the most common bit image, but QuickDraw can operate on any section of RAM. The pixels use all the bits of a memory word. They are arranged to make the horizontal scanning circuitry simple. Figure 8-4 shows how pixels are arranged in a bit image. You may never need this information, because QuickDraw takes care of all of the bit manipulation for you.

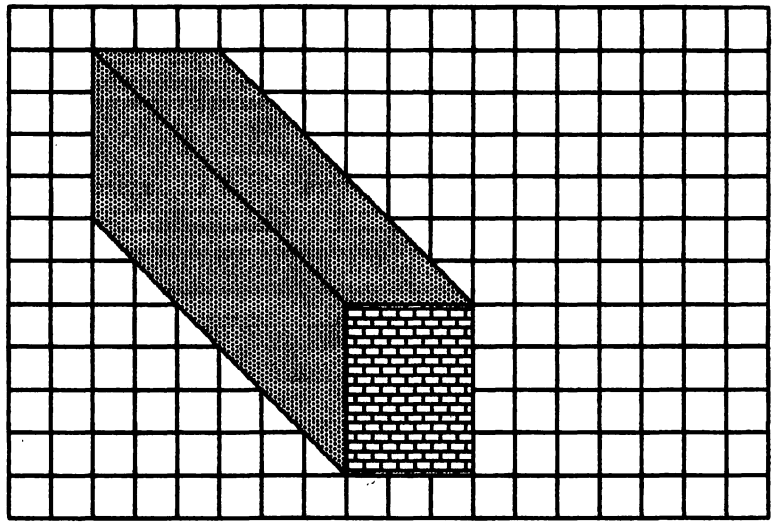


As you can see, there is a one-to-one correspondence between pixels on the screen and bits inside a word of memory. It only becomes obscure in hexadecimal and even more obscure in decimal.

Figure 8-4 QuickDraw Bit Image

Coordinates are vertical and horizontal mathematical lines. They are described by two-byte integers, running from −32768 to 32767. Although there are horizontal and vertical coordinates for each pixel, the coordinates go between the pixels, rather than through them. This makes QuickDraw unusual. In most graphics packages, the coordinates go through the pixels, like beads on a string. In QuickDraw, the coordinates separate the pixels, like the lines between squares on a checkerboard. The QuickDraw system is much easier, once you are familiar with it, because you don't have to make special efforts to include the end points of a line you are drawing. Figure 8-5 shows how the coordinates divide the pixels.

The QuickDraw vertical coordinate system is opposite to the standard Cartesian coordinates. The vertical coordinates get larger going down on



This drawing shows a 3×5 pen drawing along a diagonal downward and to the right. The area in represents the pen and the area in represents the area affected by the diagonal line.

Figure 8-5 QuickDraw and Pen

the Macintosh. The upper left corner of the image will have the lowest coordinate values. The lower right corner will always have the highest coordinate values. This makes sense when drawing text, because we like to think of the top of the screen as the first line rather than the last line.

A point is the intersection of a horizontal and a vertical coordinate. This is not the same as a pixel. Instead, a point is the place where four pixels touch. This can be seen in Figure 8-5. A point is described by two integers, the vertical coordinate followed by the horizontal coordinate. Here we define a point close to the X-Axis:

```
HighPoint       DC.W 5      ;vertical coordinate equals 5
                DC.W 20     ;horizontal coordinate coordinate
                            ;equals 20
```

A rectangle is formed by two horizontal and two vertical coordinates. The coordinates surround the pixels contained in the rectangle. The rectangle is described by these four coordinates in a specific order: top, left, bottom, right. You can also think of this as two, diagonal, corner points to describe the rectangle, the top, left point and the bottom, right point. Here we define a tall, narrow rectangle close to the X-Axis:

```
ThinRect    DC.W 5      ;top coordinate equals 5
            DC.W 40     ;left coordinate equals 40
            DC.W 50     ;bottom coordinate equals 50
            DC.W 45     ;right coordinate 45
```

An imaginary drawing implement, the QuickDraw Pen, controls how drawing takes place. The pen has many characteristics you can control. It is rectangular, and you can change its height and width. It has a location, a point in the bit image. When you draw, the bits of the pen are combined with the bits underneath according to a rule or pen mode. You can set the pen mode to be "AND," "OR," etc. depending on how you want the new drawing to combine with the image already there. Finally it has a pattern, which is the shading such as black, gray or white, with which the new image is drawn. Setting these characteristics gives you great variety in the drawings you can make. We will discuss each of them later on in this chapter.

There are three kinds of drawing in QuickDraw. You can draw using text, lines or rectangles. Text drawing puts solid letters on the screen at the current pen location. Line drawing usually starts from the pen location and moves diagonally to a new location. Rectangular drawing creates different kinds of figures within a rectangle. All of the QuickDraw commands relate to drawing in one of these three fashions.

Text drawing always starts at the current pen location. Each character is created to the right of the pen. Then the pen is moved to the left by the width of the character. Text drawing does not use any pattern. The text is always solid black or white. The pen mode is not used, but a special text mode can be set which has the same effect. (Pen mode is discussed later on in this chapter.) You can draw single characters, Pascal strings, or blocks of text from a larger structure.

We have already seen how to use DrawChar to put a single character on the screen. To draw a Pascal type of string, you use the DrawString procedure. In Pascal this procedure is defined as:

```
procedure DrawString(s:Str255);
```

Don't worry about the type definition, "Str255." This is just a string with a maximum size of 255 bytes. We only have to pass a pointer to any Pascal-type string to use the call:

```
PEA "Literal String"   ;Push pointer on stack
_DrawString            ;Draw the string
```

If the data is not formatted as a Pascal string, there is a more generalized procedure, DrawText. It is defined this way in Pascal:

```
procedure DrawText(textBuf: QDPtr;FirstByte, ByteCount : INTEGER);
```

The type definition, "QDPtr," is as you might expect, a pointer. The

"FirstByte" is the number of characters to skip before drawing the number in "ByteCount" characters. This call prints "YES" on the screen:

```
        PEA     Text                ;Pointer to start of data
        MOVE    #7,-(SP)            ;Start at the eighth byte
        MOVE    #3,-(SP)            ;Draw three characters
        __DrawText                  ;Draw the characters
Text    DC.B    "NoNoNo YES No"
```

If you want to know the width of text without actually drawing it, there are three similar routines to measure text. It is important to always get the width of the text this way, because the width of characters depends on the font and size, as you shall see below. The text width calls are three functions which correspond to the text drawing procedures:

```
function CharWidth (ch: CHAR): INTEGER;
function StringWidth (S: Str255): INTEGER;
function TextWidth (textBuf: Ptr; firstByte,byteCount: INTEGER):INTEGER;
```

To put the width of a letter "A" into D0, you can call CharWidth like this:

```
CLR.W  -(SP)                ;Space for integer result
MOVEQ  #' A",D0             ;Character "A"
MOVE.W D0, -(SP)            ;Parameter to stack
__CharWidth                 ;Call CharWidth function
MOVE.W (SP)+,D0             ;Character width to D)
```

Line drawing depends greatly on the pen characteristics. The line is created using the pen mode and pattern. You can imagine that the drawing area already contains the pen pattern, but it is covered with another surface. As the pen moves, it scrapes off the top layer, leaving the pattern exposed. The width of the line is set by the dimension of the pen. You can think of the pen as a block that hangs below and to the right of the pen location during line drawing. As the pen moves along a coordinate line, the pixels below or to the right for the height or width of the pen are affected. If the pen moves along a diagonal, the affected area is more difficult to describe, but the analogy still holds. The upper left corner of a rectangle the height and width of the pen, is dragged along the path. Figure 8-5 illustrates the QuickDraw Pen making a diagonal line.

There are four commands most useful in line drawing. In Pascal they are:

```
procedure MoveTo(h,v : INTEGER);
procedure LineTo(h,v : INTEGER);
procedure Move (dh,dv : INTEGER);
procedure Line (dh,dv : INTEGER);
```

The Line routines actually draw a line. The Move routines just move the pen to the new location. The pixels along the way are not affected. The MoveTo and LineTo routines use absolute coordinates. The other two give a distance to move on each axis from the current pen location. Here is how you might move the pen to the point (5,3):

```
MOVE.W   #5,-(SP)        ;X axis coordinate
MOVE.W   #3,-(SP)        ;Y axis coordinate
__MOVETO                 ;Move pen without drawing
```

Since the horizontal coordinate is put on the stack first, it is at a higher address than the vertical coordinate once the coordinates are on the stack. You can achieve the same structure by simply moving a point onto the stack. Recall that a point is two integers, vertical followed by horizontal. So to draw a line to a given point you could use:

```
MOVE.L   Spot(A5),-(SP)  ;Load X & Y of spot
__LineTo                 ;Draw line
Spot     DS  4           ;Storage for point
```

Rectangular drawing is the most powerful. You can draw rectangles, of course. They may be solid or open-line frames. You also use rectangular drawing to make circles, ellipses, and rounded corner boxes. There are commands (described below) for rectangular drawing that use the pen characteristics, and other commands that let you use different modes and patterns.

Rectangular drawing only affects pixels within the target rectangle. For example, the __FrameRect call draws a square-cornered box entirely within the given rectangle. The pen during rectangular drawing does not just hang by the top left corner. Instead, it moves to create the largest possible figure of the proper type that will fit completely inside the rectangle. The result of a FrameRect call is shown in Figure 8-6.

Notice how the border of the input rectangle has been traced by a pen that always stayed inside of the rectangle. Here is the Pascal definition for FrameRect, and how we might use the call in Assembly language:

```
;procedure FrameRect (r: Rect); (Outline the rectangle "r")
          PEA     Box        ;Always pass a pointer to a rectangle
          __FrameRect
          ...
Box                          ;Constant rectangle
          DC      5          ;top coordinate
          DC      100        ;left coordinate
          DC      50         ;bottom coordinate
          DC      200        ;right coordinate
```

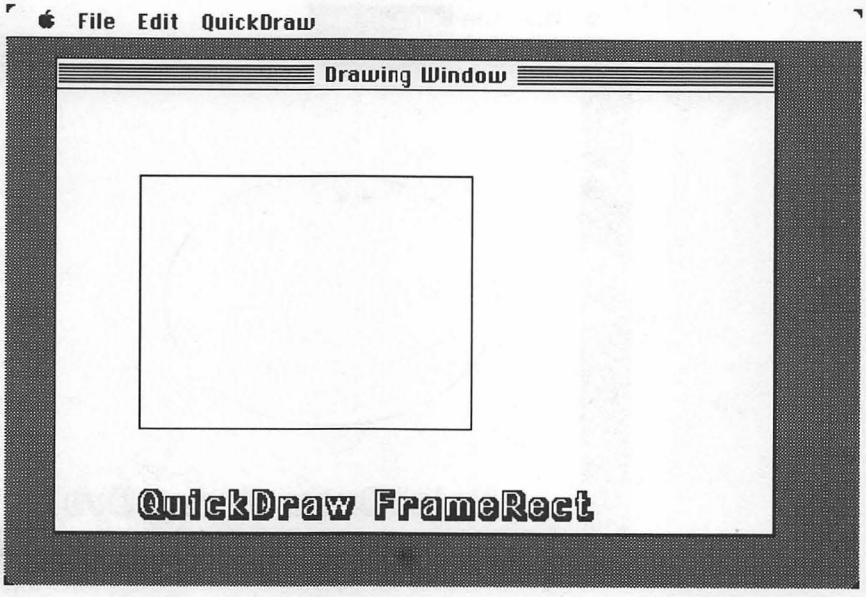


Figure 8-6 FrameRect Call

When you use rectangular drawing to draw circles, you specify a square rectangle. QuickDraw will then fit a circle within that square. The width of the circle will depend on the width and height of the pen. The circle will include pixels next to the coordinates at its tangent points, but nothing will be drawn outside of the rectangle. To draw an ellipse, you use the same calls as for drawing circles, but the rectangle does not have to be square. The __FrameOval call outlines an ellipse within a rectangle. The result of a call to __FrameOval is shown in Figure 8-7. The syntax of this call is identical to FrameRect, which draws rectangles.

Partial circles or ellipses are called *arcs*. To draw an arc you specify the rectangle that would make the complete circle. You also pass an *angle* where the arc should start and one for how far it should go. The angles are given in degrees. Vertical is zero degrees. Moving clockwise increases the angle. An angle can also be specified as a negative value. This is measured counterclockwise from the vertical. When you specify a negative angle for the distance, the arc is drawn counterclockwise from the start angle. If you use a positive angle for the distance, the arc is drawn clockwise from the start angle.

If the rectangle is square, the angles correspond to actual measurements like those you would make with a protractor. But if the rectangle is not square, the angles are scaled by its dimensions. The forty-five degree angle is always in the upper right corner. Ninety degrees is always

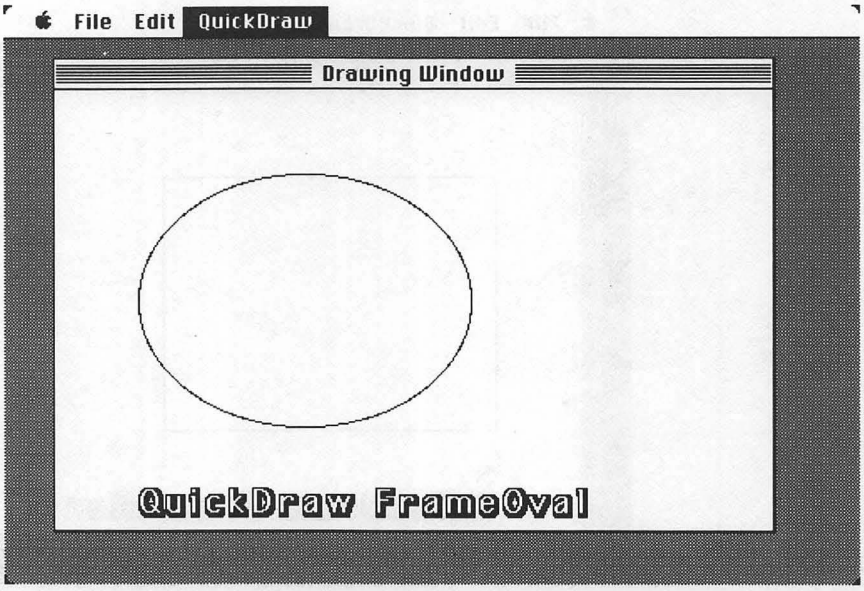


Figure 8-7 FrameOval

perpendicular to the right edge of the rectangle. The effect is that of drawing the angles on a circle within a square, and then tilting the plane until the square and circle look like the desired ellipse and rectangle.

One more class of shape can be created with rectangular drawing. The rounded-corner boxes are made by the RoundRect commands. These rounded rectangles are ordinary boxes, with ellipses in the corners. QuickDraw makes them by drawing 90 degrees of the oval in each corner. Then it connects them with edges reduced in length by the height or width of the ovals. Rounded-rectangle calls need two additional parameters to describe the corner ovals, the width and height of the four ellipses. Figure 8-8 shows the result of a call to __FrameRoundRect.

The syntax of this call in Pascal and Assembly language is given below:

```
;  procedure FrameRoundRect(r: Rect; ovWd,ovHt :INTEGER);
        PEA     Box             ;Always pass a pointer to a rectangle
        MOVE    wide,-(SP)      ;Width of corner rectangle
        MOVE    high,-(SP)      ;Height of corner rectangle
        __FrameRoundRect
        ...
Box                             ;Constant rectangle
        DC      40              ;top coordinate
        DC      100             ;left coordinate
        DC      120             ;bottom coordinate
```

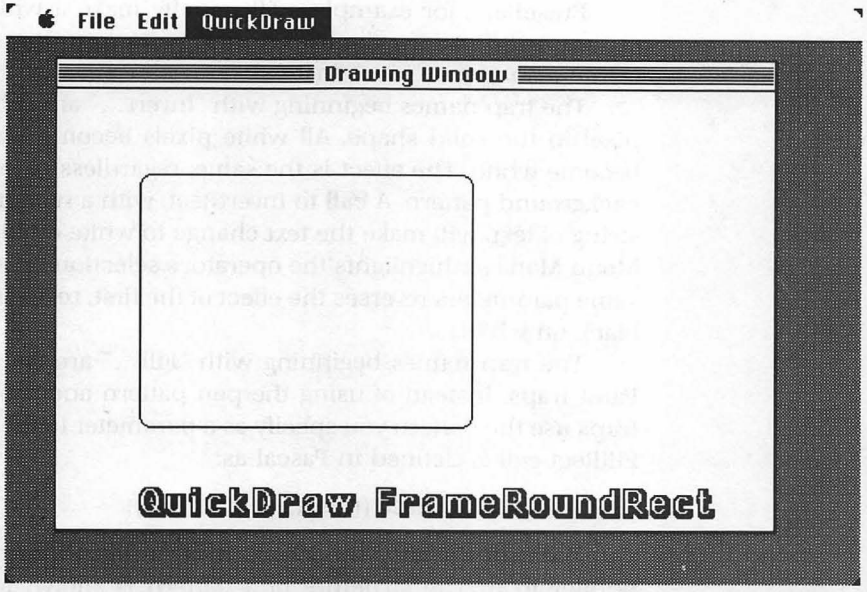


Figure 8-8 FrameRoundRect

```
        DC      200         ;right coordinate
High    DC      20          ;corner height
Wide    DC      15          ;corner width
```

You can make solid shapes as well as outlines with rectangular drawing. The various calls for creating solid shapes are shown in Appendix F (Trap Appendix). You should have no trouble calling them from Assembly language, because the parameters are the same as calls we have already discussed. There are several calls for each shape. Happily, Apple made the first word of the name consistently describe what each call does.

All trap names that begin with "Paint..." are calls that fill in a solid shape, using the pen pattern and mode. PaintRect will draw a black rectangle, if the pen is in the normal state. PaintOval can likewise be used to draw a solid circle or ellipse. The size of the pen does not matter for these, or any solid drawing calls, as long as neither the pen height nor width is zero.

All trap names that begin with "Erase..." are calls that normally clear a solid shape to white. This means all pixels within the shape are set to zero. We say "normally" because the commands actually fill the shape with a special pattern defined as the background pattern for the port. This is normally the white pattern of all zeroes, but it can be changed with a procedure:

```
procedure BackPat (pat; Pattern);
```

EraseRect, for example, will initially make a white rectangle on the screen. But if you set the background pattern to Gray, then EraseRect will create a solid gray block instead.

The trap names beginning with "Invert..." are calls that change every pixel in the solid shape. All white pixels become black. All black pixels become white. The effect is the same, regardless of pen pattern, mode or background pattern. A call to InvertRect, with a rectangle drawn around a string of text, will make the text change to white on black. This is how the Menu Manager highlights the operator's selections. A second call with the same parameters reverses the effect of the first, restoring the menu item to black on white.

The trap names beginning with "Fill..." are calls that work like the Paint traps. Instead of using the pen pattern and mode however, the Fill traps use the pattern you specify as a parameter to fill the solid shape. The FillRect call is defined in Pascal as:

```
procedure FillRect (r:Rect; pat; Pattern);
```

To use this call, you could create a pattern in memory and pass a pointer to it. The structure of a pattern is shown in Appendix F (Trap Appendix). An easier way is to use one of the standard patterns initialized by QuickDraw. We will show you how to do this later on this chapter.

The pen characteristics vary the way these drawing traps work. You can set the pen characteristics with a few simple calls. A similar group of calls set the characteristics for drawing text. You can set the size, mode and pattern of the pen. For text drawing, you can set the font, face, as well as the size, and mode. We are going to explain what each of these terms means and give some example calls.

The pen size is the height and width of the pen. There is one call with two parameters to set the pen size:

```
; procedure PenSize(width,height):INTEGER);
            MOVE.W      #width,-(SP)
            MOVE.W      #height,-(SP)
            _PENSIZE
```

You can also think of the pen shape as a point. That point defines the lower right corner of the pen when the upper left corner is at (0,0).

A drawing mode is the logical operation that puts the new drawing into the current bit image. The mode controls how the new drawing covers up or blends in with the old. The new drawing, or source data, is combined with the contents of the destination, the bit image. The result is placed back into the destination. If the logical operation is "OR" for example, the black parts of the new figure will be drawn over the old. The old drawing will show through any blank spots in the new figure. The "OR" mode is just like a pen and ink. With different modes you can figuratively

draw with an eraser, using the "BiC" or bit clear mode, or paste text over the drawings with the "Copy" mode.

You set the pen mode by passing an integer with a __PenMode call. This mode is used for line and rectangular drawing. A separate mode for drawing text is set by passing an integer with a __TextMode call. The integer values for the same modes are different for the two types of call. The Pattern transfer modes for the __PenModes call are the Source transfer modes for __TextMode plus eight. The values for the modes are shown in Appendix F (Trap Appendix). Since the "OR" mode for text, SrcOr, has a value of one, we can set the text mode to "OR" like this:

```
MOVE.W  #1,-(SP)
__TextMode
```

A Quickdraw pattern is a small design, eight bits wide by eight bits tall. As you draw, this square is repeated across the screen. You can use the five patterns QuickDraw defines, or make your own. The pattern data is just eight bytes starting on a word boundary. Each byte represents one horizontal line of the pattern. A pattern of wide and narrow diagonal strips, as shown in Figure 8-9, can be defined like this:

```
            .ALIGN  2
Stripes     DC.B    $27     ;Binary                 00100111
            DC.B    $4E     ;                       01001110
            DC.B    $9C     ;                       10011100
            DC.B    $39     ;                       00111001
            DC.B.   $72     ;                       01110010
            DC.B.   $E4     ;                       11100100
            DC.B    $C9     ;                       11001001
            DC.B    $93     ;                       10010011
```

To set the pen to this pattern, use the __PenPat procedure like this:

```
; procedure PenPat  (pat: Pattern);
            PEA Stripes
            __PenPat
```

When the pen draws, it copies the pattern into the bit map according to the current drawing mode. It always aligns the pattern according to the coordinates rather than where you start drawing. This way solid shapes drawn separately blend together well where they touch.

If you don't want to define your own patterns, you can use the patterns QuickDraw defines for you. These are created when you call __InitGraf to initialize QuickDraw. A pointer passed to __InitGraf tells QuickDraw where to keep its variables. The Finder has an area reserved for this, if you just pass a pointer to −4(A5).To access the patterns, you get a pointer to the QuickDraw Globals from a global location that stays put.

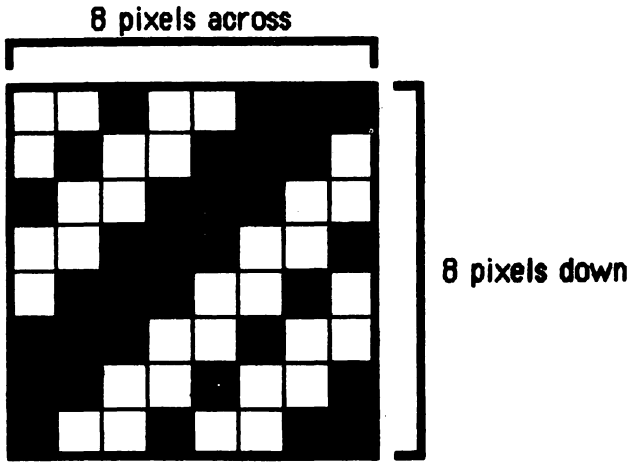


Figure 8-9 Pattern Picture

Then you use a predefined offset to access the pattern. This sounds complicated in words, but the code is very simple. To initialize QuickDraw, use:

```
PEA  -4(A5)                    ;System QD port
__InitGraf                     ;Initialize Quickdraw
```

Then, to set the pen pattern to Gray, use:

```
MOVE.L      GrafGlobals(A5),A0  ;Pointer to QD globals
PEA         Gray(A0)            ;Standard pattern
__PENPAT
```

There are five standard patterns, white, black and three shades of gray. The offsets for these patterns are defined in the include file QDEqu. They are:

```
Black     EQU     -16
White     EQU     -8
Gray      EQU     -24
LtGray    EQU     -32
DkGray    EQU     -40
```

The offset values are also listed in Appendix F (Trap Appendix). The shade each pattern represents is self-explanatory. You can use these offsets in the same way we used "Gray" above.

Text drawing has its own characteristics. There is a text mode, font, size and typeface. We have already talked about setting the text mode in the discussion of the PenMode call. The text font is the form of the characters. A set of drawings, one for each character, make up a font. The

text face is the way the basic font can be modified. Any font can be turned into italics by slanting the characters, or made bold or darker by widening them. The text size is also a modification to a standard font. By changing the text font, size and face, you can create displays on the Macintosh which are as attractive as a printed page.

Although it is possible to define your own fonts, it is much easier to use a standard font provided from the System Disk by the Font Manager. Each font is identified by an integer. To make them easier to remember, the fonts also have the name of a city. The font the System uses for things like the menus can be selected as font #0. The default application font is font #1. These two are not actual fonts, but rather tell the Font Manager to return one of the two default font numbers set up by the Finder. You pass the font number as an integer to TextFont to change the font. For example, the call to set the font called *Venice* (font #5):

```
MOVE.W  #5,-(SP)
_TextFont
```

Some fonts are proportionally spaced, while others have fixed spacing. Proportional spacing means the width of a character depends on its shape. A capital letter "W" can take more space than a lower case "i" in a proportional font. Fixed spaced fonts make all letters the same width, like a typewriter.

Any text size can be achieved by scaling a standard font. Some sizes look much better than others, however. The Font Manager has access to several different sizes of many fonts, already scaled by a human artist. If the Font Manager does not have the current font in the size you request, another size of the same font is scaled up or down. The pre-defined sizes look much better than the intermediate sizes the computer calculates on its own. The size of a font is the number of printer's points. One point is 1/72 inch. An integer passed to _TextSize sets the text size. The integer must either be zero or a number between 6 and 127. Passing zero lets the Font Manager use the default size. To set the font size to 12 points, use:

```
MOVE.W  #12,-(SP)
_TextSize
```

The text face can be plain or different combinations of styles, such as italic and boldface. The different styles are given by powers of two, so that you can combine them into one style word. The style numbers given below are also repeated in Appendix F (Trap Appendix):

```
Bold        EQU     $01
Italic      EQU     $02
UnderLine   EQU     $04
OutLine     EQU     $08
Shadow      EQU     $10
```

```
Condense    EQU    $20
Extend      EQU    $40
```

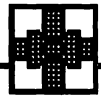
The procedure for setting the text face is,

```
procedure TextFace (face: Style);
```

where "Style" is an integer. To set italic text with underline use:

```
MOVE.W #6,-(SP) ;Underline +italic = 4+2 = 6
_TextFace
```

Of course if you want to go back to plain text, just pass _TextFace a zero.



Summary

In this chapter we discussed calling the Toolbox and Operating System. We had a summary of some of the basic commands in QuickDraw. Now we are ready to go on to Chapter 9, SimpleCalc, where we will see how an actual Macintosh application is put together.

CHAPTER

9

SimpleCalc— A Sample Application

How to Use SimpleCalc

SimpleCalc is a simplified spread sheet. Although it is fun to play with, it can't be used to keep the books of a multi-national corporation. But since we wrote it especially as an example, you can use it as a model when you start on your own best-selling software.

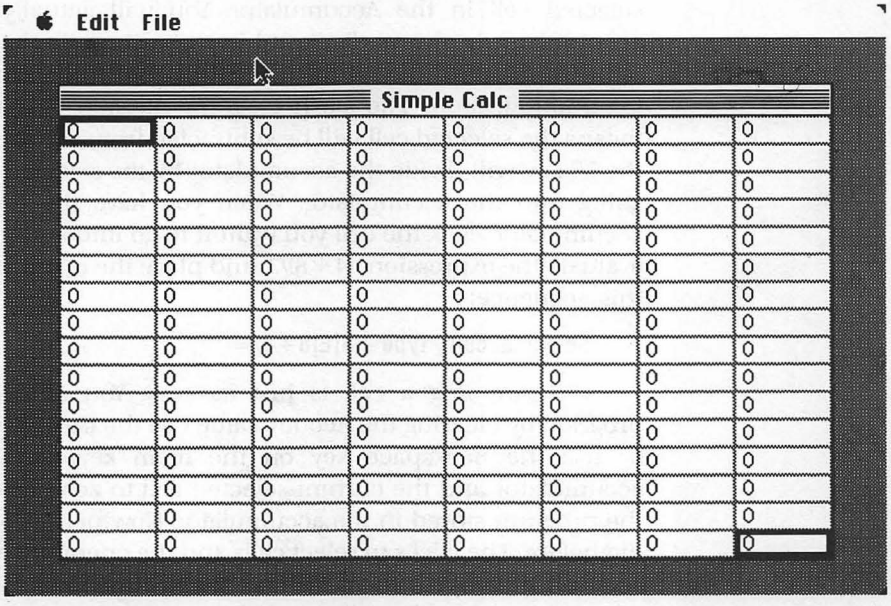


Figure 9-1 SimpleCalc Screen

SimpleCalc only uses integers; there are no legends, no decimal points. In fact the only values you can key into the 127 cells are digits. SimpleCalc does arithmetic. You can even program cells to update themselves by stored formulae. All these computations use an algebraic system known as *Reverse Polish* notation.

Reverse Polish notation is used in some calculators. In Polish notation operations are performed in the exact order in which they appear in the expression. When you see a plus sign you add. When you see a multiplication sign you multiply. When several signs are combined in an expression, you perform the first operation first, regardless of the type of operation. Reverse Polish notation is "reversed" because the operands appear before the operators in an expression. The expression,

```
4[E]8+3/
```

(where [E] = Enter)

means to take four and eight and add them. Then divide the result by the number three. This is the same as the standard algebraic expression, (4+8)/3. Reverse Polish notation is easy for a program to evaluate. In fact compilers and interpreters usually turn expressions into a Polish form using a stack algorithm when they evaluate them.

Using SimpeCalc is easy. Double-click the icon to start the program. Then just click the mouse in any cell to select that cell. Now you can type a value into the cell or use it in an operation. Use the "Enter" key like the [E] in the Reverse Polish expression above. That will put the value of the selected cell in the Accumulator. You will actually see it go in. The accumulator is always displayed in the cell on the lower right.

Use the plus and minus signs to add and subtract. The asterisk (*) is for multiplication. The slash (/) is for division. Pressing the minus key means the selected cell will be subtracted from the accumulator. Similarly, the "/" key will divide the accumulator by the selected cell, with the result going into the accumulator. When you have the desired result in the accumulator, click the cell you want it to go into and press the "=" key. To evaluate the expression, (4+8)/3, and place the result in cell *X*, you can use this sequence:

```
Select a cell, type "4[E]8+3/="
```

Programming a cell is just as easy. You should always start the program by clearing the Accumulator. Use the Clear key on the 10-key pad, or use the Backspace key on the main keyboard. This will set the accumulator and the current selected cell to zero, as well as clearing out the program stored in the accumulator. Now perform calculations as you did before. The clicks to select cells and the operator keys you press will be saved in a 28-byte record behind the accumulator. When you finish the expression, select the cell you wish to program with that expression. Then

choose Program from the Edit Menu, or just press "Command-P." The program will be copied from the accumulator area to the program storage area behind the selected cell. Any "=" operators won't be stored in the program, so the calculation won't have any side effects.

To indicate the Backspace or Clear key, we use the notation, "[C]." For "Command-P," we use "[P]." As you remember we use "[E]" for the "Enter" key. This sequence programs cell *X* to be the sum of cells *Y* and *Z*:

```
Select a cell (X), type "[C]",
select a second cell (Y), type "[E],"
select a third cell (Z), type "+",
go back to the first cell (X) and type "[P]'.
```

When you set a cell to a constant value, it clears out its program. You do this whenever you use the Equals key, the Clear key, or type a digit into a cell. You can also set a cell to a constant using the Edit Menu. The Cut, Copy and Paste commands work with an invisible Clipboard. The Invert command changes the sign of the cell and makes the cell a constant. It is the only way to enter negative numbers directly, because the minus key will just subtract the cell from the accumulator.

The programmed cells are continuously updated, but only while there are no events waiting in the event queue. Each time the Event Queue is polled and found to be empty, one cell is recalculated and redrawn if its value is changed. The next time the Event Queue is polled the next cell may be redrawn, until the whole spreadsheet has been updated. This process goes on even when the window is inactive and a desk accessory is being used. By making cells serve as input to their own calculations, you can make counters, oscillators, and compute converging values.

One type of converging calculation is the square root function. We can calculate the square root of X by this algorithm.:

```
Y = 1
REPEAT Y = ((X/Y)+Y)/2
```

In Reverse Polish notation the second line would be

```
2[E]Z=X[E]Y/Y+Z/Y=
```

This works because if you divide a number by what you think is its square root, the result should also be the square root. If the first guess was too small, the result of the division will be larger than the root, and vice versa. By averaging these two values and repeating, Y gets closer and closer to the actual square root.

When we implement this algorithm in SimpleCalc, we can omit the first step. Setting Y to one is only necessary to avoid division by zero. Our program returns a large but finite value on division by zero, so Y does not have to be initialized. Assuming that the value of X has already been entered, the second line becomes:

select a cell, type "2", select a second cell(Y), type "[C]",
select a third cell(X),type "[E]",
go back to the second cell (Y), type "/+", then go back
to the first cell selected, type "/",
finally go to the second cell (Y) and type "[P]".

Here we have used the first cell selected to hold a constant value, two. Once it is started, the calculation will continue updating the second cell until a stable value is reached. Then you can change the third cell and watch a new value be calculated. What do you think will happen if the third cell is set to a negative number? Try it. Did you find the result in the second cell never stops changing?

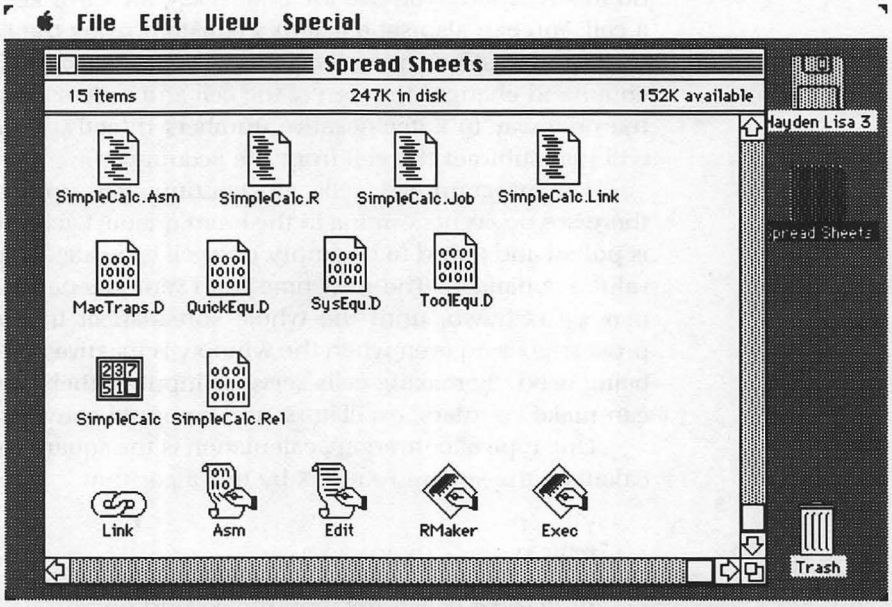


Figure 9-2 SimpleCalc Icons

Description of the Code for SimpleCalc

There are three major sections to SimpleCalc, the Main Program, the User Interface and the Spreadsheet Functions. The Main Program and the User Interface are similar to most Macintosh programs. Most of our discussion centers on these two sections. The functions of these sections are common to all Macintosh programs. Whether you are writing a word processor or a program to simulate an accordian, many aspects of the user interface will

remain the same. The Spread Sheet Functions have many entry points, but they are grouped together so you can modify them easily, or even replace the section entirely with your own code. This kind of organization is easy to create in Assembly language, but almost impossible to achieve in Pascal.

Our Main Program itself has three parts: the initialization, a main loop and the termination. The functions of the first and last are obvious. The main loop just calls the three main tasks, checking for events, displaying the total, and calculating programmed cells.

The User Interface contains most of the code you could transport to another application. It detects events, interprets them, manages the window and the Apple menu functions.

The Spreadsheet Functions section is unique to SimpleCalc. This section includes editing the cell values, drawing the spreadsheet, and calculating the programmed cells.

Look at the code in the accompaning figures as we go through the listing and describe each routine. As we do this you should notice not only the code, but the documentation in the program. Documentation is even more important in Assembly language than it is in BASIC or Pascal. Fortunately, comments are easy to type with the Assembler: everything on a line after a semicolon is considered to be a comment. The semicolon ";", is much easier to insert than the pinky-pulling, curly braces "{ }" or the "(* *)" syntax demanded by Pascal. Inserting many more comments is so much easier. Some Assembly language programmers put a comment on each line. Some put a paragraph at the head of each routine. Some people do both. The more you can say about your program, the easier it will be to debug. It will also be easier for other programmers to understand, or for you to understand it yourself, many months later.

Conventions

The documentation in SimpleCalc uses some conventions which we should explain. The Pascal symbol for a pointer, an upward arrow, (^), is used to symbolize an address or pointer. The result of a function is shown with a left arrow made of a minus and a greater-than sign, "->". The trap macro of each Toolbox call shows the parameters passed on the stack using those two symbols where necessary. Some programmers prefer to reproduce the Pascal procedure definition before each call. This is also a good method of describing the data on the stack. Register usage is only documented when it is different from the initial description.

The Main Program

SimpleCalc begins with a few lines of documentation which list the other files needed to create the application. The resource source file, SimpleCalc.R, must be compiled with RMaker. The linking file, SimpleCalc.L, tells the Macintosh Linker to put the only segment, SimpleCalc.Rel, into

the final object file. The exec file, SimpleCalc.Job, runs first the Assembler to create SimpleCalc.Rel, and then the Linker to create the final application. If you are using the Lisa Workshop, the exec file has to be a little bit different. You will also need to modify a few statements that come later. You can look in Appendix E, Using the Lisa Workshop, for more information.

```
;
;------------------------------------------------------------------------
; SimpleCalc  - A Simplified SpreadSheet Example by Harland Harrison
; and Ed Rosenzweig
;
; Other files needed are
;     SimpleCalc.R          Resource source
;     SimpleCalc.Job        Exec file
;     SimpleCalc.Link       Linkage list file
```

The include file comes next. The files included in SimpleCalc contain traps and data definitions for calling the Toolbox. They are in the compressed format as explained in Chapter 2, Addressing Modes of the 68000. If you are working with a Lisa the "include" statements will be different.

```
;------------------------------ INCLUDE ------------------------------
Include     MacTraps.D          ; Include equates and traps files
Include     ToolEqu.D
Include     QuickEqu.D
Include     SysEqu.D
```

Data storage is defined under the next three headings and under the last heading at the end of the program. Local data will be accessed from A6. Global data will be accessed from A5. Program data will be PC-relative.

The register usage is only documentation, but it is just as important to clarify as the offsets we will use to access memory. You will probably want to refer back to this section often while we go through the code.

```
;------------------------------ LOCAL DATA ------------------------------
DataSize   EQU    4110            ; Space needed for variables
Clip       EQU    4102            ; Clipboard for editing cells
ItemHit    EQU    4104            ; Dialog item chosen
AppleHand  EQU    4106            ; Handle for Apple menu
AccCell    EQU    127             ; Cell number of accumulator
; Stored spread sheet program format
;   30  bytes for each cell
;   $80 bit means select a cell. Otherwise arithmetic operation
;   $00 means end of program
Prg        EQU    2                    ; Offset to start of program in cell
ProgLast   EQU    29                   ; Maximum entries. Last byte must be 0
; Flag bits stored in Flag register, D5
FrontFlag  EQU    2                    ; Bit set if our window selected
Redraw     EQU    1                    ; Redraw cell bit. = overflow in SR
QuitFlag   EQU    0                    ; Exit program bit. = carry in SR
;------------------------------ GLOBAL DATA ------------------------------
DeskName            DS    16          ; Desk accessory's name
WindowStorage       DS.B  WindowSize  ; Storage for window
;---------------------------- REGISTER USAGE ----------------------------
;
;           MainProgram               Initialization
;     D4    Selected Cell             Menu ID
;     D5    Flag register             Menu handle
;     D6    Program byte counter
;     D7    Cell to draw or calculate
;
;
;     A3    Selected Cell
;     A4    Cell being calculated
;     A5    Global Variables
;     A6    Variable Base/ Accumulator
;
```

The memory usage in SimpleCalc is intentionally not optimal. We designed this program as an example. We tried to define data in all possible ways to give you examples to create your own programs. We are sure you can think of better ways to assign the variables in SimpleCalc. In particular, you may not want to use any PC-relative data defined in the last section for variables. As you will see, it is hard to modify these locations. And as pointed out in Chapter 6, it cannot be used with segmented code.

```
; ---------------------------- Data Storage -------------------------

CurrentEvent                    ; Event record
What            DC    0         ;  Type of event
Message         DC.L  0         ;  Info about event
When            DC.L  0         ;  Tick when it happened
Where                           ;  Mouse location when it happened
WhereV          DC    0         ;   Vertical coordinate
WhereH          DC    0         ;   Horizontal coordinate
Modify          DC    0         ;  Control keys down when it happened

EvtWind         DC.L  0         ; Window with event

Menu            DC    0         ; Menu that item is in
MenuItem        DC    0         ; Menu item selected

WindowPointer   DC.L  0         ; Pointer to spread sheet window

DragLimit                       ; Boundary rectangle for dragging window
                DC    30        ;  top
                DC    5         ;  left
                DC    350       ;  bottom
                DC    500       ;  right

CellRect        DCB.W 4,0       ; Rectangle enclosing selected cell
TxtPnt          DC.L  0         ; Point in cell where text starts

END
```

The Main Program is only 15 lines long. It initializes, runs the application and then terminates to return to the Finder. The LINK instruction sets up the local data space, just as a Pascal application would. Unlike a Pascal application, we follow the LINK with an LEA instruction. This LEA makes A6 point to the lowest address of the data space instead of the highest address, so we can use positive offsets to reach our variables. Pascal programs use negative offsets, and you can too, but we thought that keeping the address calculations in positive numbers makes the program easier to understand. Once set-up, the data space is initialized by a call to InitMain. This subroutine will be explained soon, when we get to that code. Then the program falls into the main loop where it stays until it is ended.

```
;------------------------------ MAIN PROGRAM ------------------------------

      LINK   A6,#-DataSize          ; Make space for spread sheet
      LEA    -DataSize(A6),A6       ; Address memory from low end
      BSR    InitMain               ; Initialize
MainLoop
      BSR    Calculate              ; Calculate values
      BSR    GetEvent
      BCS    MainExit               ; Carry Set is signal to quit
      BVC    MainLoop               ; Overflow is signal to redraw
      BSR    DrawSelect
      BRA    MainLoop
MainExit
      LEA    DataSize(A6),A6        ; Point to top of memory
      UNLK   A6                     ; Return memory to stack space
      RTS
```

Main Loop

The MainLoop calls Calculate to compute the values of programmed cells. Then it calls GetEvent to look for any operator activity. These routines are designed to keep the program operator-responsive. Calculate only computes one cell for each call. But GetEvent handles all events in the queue. This minimizes the time between operator actions and their results. GetEvent returns results to the main loop in the Condition Codes. This is the fastest way to communicate between subroutines, but it is also the hardest to keep track of. Here the Carry Flag (Carry Set) means to exit the program. The Overflow Flag means the selected cell or the accumulator have been changed and must be drawn again by the DrawSelect subroutine before continuing with the loop. The main loop keeps cycling in this way, until the operator wants to stop using SimpleCalc. Then the Carry Flag will be set, and the loop will fall through to the termination.

Termination

When the operator chooses "Quit" from the menu, MainExit ends the program. This example returns the data space by instructions that mirror its creation. The LEA instruction moves A6 back to the end of the data space. The UNLK instruction restores both A6 and the stack pointer. Finally an RTS returns control to the finder. If your program is more complicated than SimpleCalc, you may have more things to do in the termination. An editor, for example, will ask the operator if he wants to save any open documents before quitting.

Initialization

The initialization, InitMain, is a subroutine called only once. It could have been included as part of the Main Program. Separating the two makes the Main Program more clear. First InitLoop fills the data space with zeros. This routine is entered at the DBRA labeled InitChk with the count in D0, because DBRA will cycle until D0 is minus one. Branching to the decrement-branch instruction this way is the best practice when the count might possibly be zero. Next, InitMain initializes the Toolbox Managers.

```
;------------------------------ INITIALIZE ------------------------------

InitMain                            ; Main initialization
; Clear data area
      MOVE.L   A6,A0                ; Point to start of area
      MOVE.W   #DataSize,D0         ; Byte count to clear
      BRA      InitChk              ; DBRA counts to -1
InitLoop
      CLR.B    (A0)+                ; Clear one byte of memory
InitChk
      DBRA     D0,InitLoop          ; Repeat DataSize times
;Initialize Managers
      PEA      -4(A5)               ; System QD port
      _InitGraf                     ; Init Quickdraw, ^ Grafport
      _InitFonts                    ; Init Font Manager
      _TEInit                       ; Init Text Editor
      _InitWindows                  ; Init Window Manager
      CLR.L    -(SP)                ; No restart procedure
      _InitDialogs                  ; Init Dialog Manager
      _InitMenus                    ; Init Menu Manager
;Open the Resource File
      CLR      -(SP)                ; Clear space for reference number result
      PEA    'SimpleCalc.Rsrc'      ; [Volume: ]Resource Name
      _OpenResFile                  ; File-name string -> Reference Number
      MOVE     (SP)+,D0             ; Get rid of the Ref Num
;Set Up Apple Menu
      MOVEQ    #1,D4                ; Resource ID 1
      JSR      MakeMenu
      MOVE.L   D5,AppleHand(A6)     ; Save Apple menu handle
;Set Up File Menu
      MOVE.W   #303,D4              ; Resource ID 303
      JSR      MakeMenu
```

```
;Set Up Edit Menu
      MOVE.W   #302,D4              ; Resource ID 302
      JSR      MakeMenu
;Add Desk Accessories To Apple Menu
      MOVE.L   AppleHand(A6),-(SP); Apple menu handle
      MOVE.L   #'DRVR',-(SP)        ; Accessory resource type
      _AddResMenu                   ; MenuHandle,Type
;Draw the completed Menu Bar
      _DrawMenuBar
;Initialize the Window
      CLR.L    -(SP)                ; Make space for the window-pointer result
      MOVE     #301,-(SP)           ; Resource ID #301
      PEA      WindowStorage(A5)    ; Push address for window-data storage
      MOVE.L   #-1,-(SP)            ; Put window on top of any other windows
      _GetNewWindow                 ; ID,^ Storage, ^ window above -> ^ window
      LEA      WindowPointer,A0
      MOVE.L   (SP)+,(A0)           ; Save the window pointer in memory
      BSR      SelWindow            ; Make it the top window
; Empty event queue of old keystrokes and mouse clicks
      CLR.L    D0                   ; No type of event stops flush
      MOVE.W   #$FFFF,D0            ; Flush any type of event
      _FlushEvents                  ; Stop, Event in D0
      _InitCursor                   ; Initialize the cursor into an arrow
; Clear registers
      CLR.L    D6                   ; No Accumulator program
      CLR.L    D7                   ; Start update from cell 0
      CLR.L    D5                   ; Clear flags
      CLR.L    D4                   ; Select cell 0
; Draw spread sheet in window and exit InitMain
      BRA      DrawInside           ; Draw sheet of zeros
```

Every program may not use all of the Managers, but you can initialize them anyway. It is important to initialize them in a certain order because some of the managers depend on others. For example, the Dialog Manager needs the Text Edit Manager which needs the Font Manager. The QuickDraw Manager needs space to create the GrafGlobals data. InitMain passes a standard area reserved by the Finder at −4(A5). The Dialog Manager could accept a restart procedure, but SimpleCalc does not need one; we

pass NIL instead. Finally initialize the Menu Manager, but don't draw the menus yet.

Next open the resource file. We are using a separate resource file in this example to show how to open one. If you make the code and resource data into one resource file, then you won't have to open the resource file separately. The name of the file, "SimpleCalc.Rsrc," is the parameter for the "__OpenResFile" call. We could also include a volume name in the parameter. If the disk were named "Spreadsheets" the parameter would be "Spreadsheets:SimpleCalc.Rsrc."

Once the resource file is open we can set up the menus. The menus are defined in the resource file. Although you can create menus by defining the data in your code, the resource file is the easiest and most flexible way to do it. The menus are set up in the standard order: Apple menu, File menu and Edit menu. Let's look down a few lines to the MakeMenu subroutine.

The MakeMenu subroutine gets a handle for the menu data in the Resource File, by calling __GetRMenu with the ID number of the menu. Then it adds the menu to the Menu List with __InsertMenu. The second parameter of __InsertMenu tells where in the Menu List you want the new menu inserted. We just pass a zero to add the new menu to the end of the list. You can pass the ID number of a menu you have already created, if you want to add the new menu to the left of the existing one. MakeMenu returns with the menu handle in D5. If you ever need to modify a menu after it is created, you have to have the handle. Let's look back a few lines and see what happens to the Apple Menu.

```
MakeMenu
;Install Menu
;    Input  D4 = Menu ID
;   Output D5 = Menu Handle
;          D4 = Menu ID
      CLR.L     -(SP)              ; Clear space for menu handle
      MOVE      D4,-(SP)           ; Resource ID input in D4
      _GetRMenu                    ; MenuID -> MenuHandle
      MOVE.L    (SP),D5            ; Return in D5 & leave on stack
      CLR.W     -(SP)              ; Put menu after all others
      _InsertMenu                  ; MenuHandle,BeforeID
      RTS
```

The Apple Menu selects the program description dialog, the About Box, and activates the desk accessories. The "About..." item is in the Resource File, but the names of the desk accessories have to be added to the Apple Menu. After calling MakeMenu for the Apple Menu, we save the handle. A few lines later we add the desk accessories with __AddResMenu. This Toolbox call adds the text of items from all open resource files to the end of a menu. You pass a menu handle and a resource type to __AddResMenu. For the desk accessories, the type is "DRVR." The type is passed as four bytes, not as a Pascal string which would start with a length byte. The __AddResMenu call puts all of the desk accessories available into the Apple Menu. This means you can call any accessory from the program which is available on your system, even desk accessories which were not available at the time the program was written!

Finally, after the menus are all set up, we call __DrawMenuBar. This call displays the menu titles at the top of the screen. Now we can go on to set up the window.

Like the menus, the window is defined in the Resource File. We initialize the window with a call to the __GetNewWindow function. First we make a four-byte space on the stack for the window-pointer result of the function. Next we pass the resource ID, 301. The next parameter is a pointer to a storage area for the window data. We have defined an area for it, but you can also pass NIL if you want the system to allocate the space. The last parameter is negative one. This means our new window will be created on top of all the others. A zero puts the new window at the bottom of the pile. To put it in between two windows, you pass the ID of the last window on top of the new one. After calling __GetNewWindow, we save the window pointer. We have to use an LEA to address the WindowPointer field because we defined it in program space. Finally we make our window the active one with the SelWindow subroutine, which is explained later. Now our screen is starting to shape up. Both the menu bar and the empty window are displayed.

Before proceeding, we empty the event queue. This is not absolutely necessary, but it is a good idea. The operator may have tried to make something happen—by blindly clicking the mouse, for instance—while waiting for the program to load. The __FlushEvents routines is a very general call. It is a good example of an Operating System call, where the parameters are passed in registers. Since the parameter is not an address it is passed in D0 to __FlushEvents. The lower part of the long word is a bit-mask of the types of events to remove from the queue. The upper part of the word is a mask for an event type that will stop the flushing. Since the upper part is zero and the lower part is $FFFF our call will remove all types of events from the entire queue.

The next few lines complete the initialization. First we call __InitCursor to set the cursor to an arrow. Then we clear the data registers we are

going to use. Finally we jump to the DrawInside routine in the Spreadsheet Functions section to present the initial spreadsheet. That completes the initialization. Now lets go on to the User Interface.

The User Interface

The heart of the User Interface is the event loop which begins with GetEvent. This routines processes all of the events. It keeps going round and round until there are no more events left. Handling all available events at once this way minimizes keyboard response time. GetEvent begins with a call to __SystemTask. This call allows desk accessories to function during the idle loop. Next we check for available events with __GetNextEvent. A mask like the one we used for Flush events chooses the events to accept. Minus one on the stack here means to take any kind of event. The function returns a Boolean value, true or false. Notice that the result is stored in bit 8 of the word, not bit 0. We check the bit with a BTST instruction because the lower byte may contain garbage.

```
;-------------------------- USER INTERFACE -------------------------

GetEvent                            ; Get next event
      _SystemTask                   ; Update desk accessories
      CLR     -(SP)                 ; Clear space for result
      MOVE    #-1,-(SP)             ; Mask to accept all events
      PEA     CurrentEvent          ; Pointer to event record
      _GetNextEvent                 ; Mask, ^ event record -> TRUE if any events
      BTST    #0,(SP)+              ; Event returned?
      BEQ     NullEvent             ; No event.  Return to main program
      BSR     DoEvent               ; Respond to event
      BRA     GetEvent              ; Check for more events in queue

DoEvent                             ; Process event
      MOVE    What,D0               ; Type of event
      CMPI    #mButDwnEvt,D0        ; mouse button down is event 1
      BEQ     MouseDown
      CMPI    #keyDwnEvt,D0         ; key down is event 3
      BEQ     KeyDown
      CMPI    #autoKeyEvt,D0        ; auto-repeated key is event 5
      BEQ     KeyDown
      CMPI    #updatEvt,D0          ; update display is event 6
      BEQ     UpDate
      CMPI    #activateEvt,D0       ; activate/deactive is event 8
      BEQ     Activate
```

```
        CMPI    #9,D0                   ; abort is event 9
        BEQ     Quit
NullEvent                               ; no event. check quit flag & exit
        MOVE    D5,CCR                  ; Set carry if $01 set by
        RTS                             ;  quit command
```

The event loop will exit through NullEvent to the main program, if there is no event. But as long as events keep coming, it calls the DoEvent routine to process them. The events are described by their hardware types. The Event Manager reports that the mouse button was clicked or a key was pressed. It is up to the application to determine if the mouse click should move the window or the keystroke should actually signal a menu item selection. DoEvent will decode and execute each event type.

There are fifteen possible event types. We only handle the most important types in SimpleCalc. The meaning of some event types is obvious from the name. An Update event means we have to redraw a window. An Activate event means a new window is becoming the top, or active window. An AutoKey event means a key was held down to repeat the keystroke continuously. This is treated the same way as the initial keystroke in most programs. But certain kinds of programs, such as an accordian simulation, would ignore an auto-key event. Instead the accordian program would look for a key-up event to terminate the sound, an event which other programs ignore.

The NullEvent entry point is the exit from the event loop. We fall through to NullEvent if there is some kind of event we can't identify. GetEvent branches here when there are no events left at all. NullEvent returns to the main loop with Condition Codes set to request certain actions. It uses the move-to-CCR instruction to set the Condition Codes from the flags in D5. The move-to-CCR instruction is explained in Chapter 3, 68000 Instruction Set.

The various event types are decoded when they occur in the section called "Event Types." An Activate event, indicating a new top window, comes first. As a precaution we check to see if the event applies to our window by calling the WindChk subroutine. The subroutine, a few lines below, shows how to get the window pointer from the message field of the event record. This would be very important in an application with more than one window. WindChk also selects the port to make sure we draw in the right window. Returning to Activate, we see that it handles two different cases. If bit zero of the event modifier field is set, the event is the signal to activate the new window. But if bit zero is clear the event is a de-activate, a signal that the old active window should be de-selected.

SimpleCalc highlights its window when it is active by drawing a black box around the accumulator and the selected cell. The FrontWindow flag is set in D5 so we can draw this box if necessary whenever we update the cells.

```
;------------------------- EVENT TYPES -------------------------

Activate
      BSR    WindChk             ; Is it our window?
      BNE    NullEvent           ; No. Ignore event
      BTST   #0,Modify+1         ; Activate or Deactivate event flag
      BEQ    Deactive            ; Activate if $01 set, else DeActivate
      BSET   #FrontFlag,D5       ; Remember window is active
      BRA    DrawSelect          ; Highlight selected cell
Deactive
      BCLR   #FrontFlag,D5       ; Remember window is inactive
      BRA    DrawSelect          ; Un Highlight selected cell
```

An Update event means a window needs to be redrawn. Again we first check to make sure it is our window. A program with multiple windows would call a routine to select the proper one. Two Toolbox calls, __BeginUpdate and __EndUpdate, bracket the subroutine call to draw the spreadsheet. The call to __BeginUpdate exchanges the VisRgn, the part of the port that shows on the screen, with a special update region. This update region shows just the part of the window that has been recently uncovered. Then we draw the whole document, but only the QuickDraw calls that fall within the new VisRgn are actually executed. The call to __EndUpdate puts everything back to normal, with the newly exposed area freshly drawn.

```
Update                             ; Our VISible ReGioN changed
      BSR    WindChk               ; Is it our window?
      BNE    NullEvent             ; No. Ignore event
      MOVE.L WindowPointer,-(SP)      ; Pointer to window being updated
      _BeginUpDate                 ; Start drawing in Update Region
      BSR    DrawInside            ; Only part needing update actually drawn
      MOVE.L WindowPointer,-(SP) ; Pointer to window whose update is finished
      _EndUpdate                   ; Return to normal mode drawing
      RTS
```

A KeyDown event can have one of two different meanings. If the Command key was held down during the keystroke it means the event is really a menu item. If the command key was up it means data is being entered. The Modifier word in the event record maps the state of the control keys, the Command, Shift, Option and Caps Lock keys. If the Command key is down we go to CommandKey subroutine to process the keystroke. If not, we call the Spreadsheet Function part to handle the data. CommandKey turns a keystroke into a Menu Event. The Menu Manager keeps track of the keyboard equivalents for the menu items. The value of the key is in the Message field of the event record. The call to the _MenuKey function returns a menu ID and item number of the selected event. Pascal programs see these two numbers as one four-byte integer on the stack. Our MenuCommand subroutine will accept the two values on the stack as input, and perform the proper function.

```
KeyDown                              ; User pressed a key
      MOVE   Modify,D0               ; Modifier word maps shift keys
      BTST   #8,D0                   ; If the Command key is down
      BNE    CommandKey              ;  it is a menu event
      BRA    KeyStroke               ; Spread-Sheet data

CommandKey                           ; Command key event. Short-cut menu choice
      CLR.L  -(SP)                   ; Get space for menu choice
      MOVE   Message+2,-(SP)         ; Put message byte on stack
      _MenuKey                       ;Identify Key. ASCII -> MenuID,MenuItem
      BRA    MenuCommand             ; Menu ID & Item now on stack
```

Mouse down events are the most complicated. Where the mouse went down determines the meaning of the event. Fortunately the Window Manager identifies the area a point is in. A call to _FindWindow returns both a code for the type of area on the desk top the point is in, and, if it is in a window, the window pointer of the top window that contains the point. The Window Manager has a list of window pointers in the order of their appearance on desk top. It checks from top to bottom for a window containing the point. The Window Manager uses this same list to generate Activate and Update events too. The parameters for _FindWindow may seem a little clumsy. The area-type code is passed on the stack, but the window pointer is returned in a memory location. You pass a pointer to that memory location as an input parameter, and the window pointer, if returned, will be placed there. This interface comes about because Pascal only allows a function to return one two- or four-byte result.

```
MouseDown                          ; Find where mouse clicked
        CLR     -(SP)              ; Clear space for integer result
        MOVE.L  Where,-(SP)        ; Mouse at time of GetNextEvent
        PEA     EvtWind            ; Window the event was in
        _FindWindow                ; click point, ^ window -> window part code
        MOVE    (SP)+,D0           ; Result = section of window
        ASL.W       #1,D0          ; Window Part * 2 Bytes/Entry
        MOVE    WindowTable(D0),D0 ; Get offset from table
        JMP     WindowTable(D0)    ; Call subroutine

WindowTable

        DC.W    NullEvent-WindowTable   ; In Desk
        DC.W    InMenu-WindowTable      ; In Menu Bar
        DC.W    SystemEvent-WindowTable ; System Window
        DC.W    Content-WindowTable     ; In Content
        DC.W    Drag-WindowTable        ; In Drag
        DC.W    NullEvent-WindowTable   ; In Grow
        DC.W    NullEvent-WindowTable   ; In Go Away
```

We use a table to arrive at a specific routine for each type of mouse down event. The table stores each address in a relative form. The offset between WindowTable itself and the desired routine is stored in each entry of WindowTable. This syntax is legal with either assembler, because the difference between two addresses is a constant. The assembler cannot possibly put an absolute, four-byte address into the relocatable code, because the code can wind up anywhere in memory. But, the result of subtracting one address from another can be stored. To find an address from the table, we first double the index, using ASL.W, because each entry occupies two bytes. We use the double index as an offset to load the table entry. Finally we use the table entry as an offset to jump to the address. The instructions which use the offset,

```
MOVE  WindowTable(D0),D0
JMP   WindowTable(D0)
```

were explained in Chapter 4, Sample Programs. In Chapter 2, The Addressing Modes of the 68000, we learned that this syntax is actually a form of Program Counter relative with index and displacement. These two instructions could also be written as:

```
MOVE  Windowtable-*+2(PC,D0),D0
JMP   Windowtable-*+2(PC,D0)
```

The subroutines called through the window table appear next. Each represents the consequence of a mouse down event in a particular area of the screen. Some mouse clicks are ignored. For example, the program ignores clicks on any part of the desk top window, which shows at the edges of the active SimpleCalc window. Clicking in a grow-box is not possible, since our window does not have one. Clicking in a go-away box normally means to close or switch documents in a multiple window environment. Since we have only one window, the go-away box could have been used to exit the program. We did not implement that because it would be too easy for the operator to make a mistake and end up back in the Finder.

A mouse click in the menu bar means the start of a pull-down type of menu selection. The InMenu routine calls __MenuSelect with the starting point of the mouse operation. The Menu Manager now takes over. It will display the menu, track the mouse and highlight items, and retain control until the operator selects an item or releases the mouse outside of a menu. The __MenuSelect call will return with the same values that __MenuKey leaves on the stack. If the operator does not complete a selection, zero is returned for the menu ID.

```
InMenu
      CLR.L       -(SP)                ; Get space for menu choice
      MOVE.L      Where,-(SP)          ; Mouse at time of event
      _MenuSelect                      ; Click Point -> MenuID,MenuItem
      BRA     MenuCommand              ; Menu choice now on stack
```

A click in a system window is intended for a desk accessory. The SystemEvent routine passes the event and the window it is in to __SystemClick. Then the desk accessory takes care of its own mouse click. If the click activates the accessory's window, activate and update events may be loaded into the event queue.

```
SystemEvent                            ; Action for desk accessory
      PEA     CurrentEvent             ; Pointer to event record
      MOVE.L      EvtWind,-(SP)        ; Load window pointer onto stack
      _SystemClick                     ; System takes care of own click
      RTS
```

A click in the content region is meant for our application. The Content routine calls the Spreadsheet function, SelCell, to select a cell. But if the window is not active, we use the first click to activate the window. This means you can click anywhere on an obscured window to bring it to the front without having to worry about the normal consequences of clicking in that particular spot.

```
Content                                 ; Click inside a window
      MOVE.L EvtWind,A0                 ; Event Window
      CMP.L  WindowPointer,A0           ; Is it our Window?
      BNE    NullEvent                  ; No. Ignore this event
      BTST   #FrontFlag,D5              ; Is our window top dog?
      BEQ    SelWindow                  ; No. Just select window for first click
      BRA    SelCell                    ; Yes. Change cell in top window
```

The Drag routine is called when the operator is moving the window. Here again the ToolKit does most of the work. The start point and window pointers are parameters to _DragWindow. The rectangle, DragLimit, is the range of motion permitted. We don't want to let the window slide completely away.

```
Drag                                    ; Let user move the window
      MOVE.L EvtWind,-(SP)              ; Window pointer
      MOVE.L Where,-(SP)                ; Current mouse location
      PEA    DragLimit                  ; Boundary rectangle
      _DragWindow                       ; ^ Window,Start point,Bounds rect
      RTS
```

Three subroutines for windows and ports appear next. The Windchk subroutine makes sure an event is for our window, as already explained. The SelWindow and SelPort subroutines select the window and the port respectively. These two actions are very different things. The selected window becomes the active, top window. Calling _SelectWindow will cause Activate events. The selected port is the port we are drawing into. It can be any window or the desk top, regardless of its position. In an

application with more than one window, these three subroutines would be more elaborate. Instead of working on one window, they would pick out the proper window and port to use.

```
WindChk
      MOVE.L Message,A0              ; Find event window
      CMP.L  WindowPointer,A0        ; Is it our window?
      BNE    @10                     ; If not ignore it
      BSR    SelPort                 ; Select our port
      CMP    D0,D0                   ; Return Z set
@10   RTS

SelWindow
      MOVE.L WindowPointer,-(SP)     ; Window Pointer
      _SelectWindow                  ; Put the window in front

SelPort
      MOVE.L WindowPointer,-(SP)
      _SetPort                       ; Set the drawing port to the window
      RTS
```

The next section handles all of the menu selections. Each menu has its own routine. This section could be table driven instead, but as an example, it shows more variety the way the section is coded. The entry point is MenuCommand. Its input is a menu resource ID and an item number on the stack. The ChooseMenu subroutine selects the function to execute. The Toolbox call, __HiLiteMenu, turns off the menu bar highlighting when the function has been performed.

```
;--------------------------- MENU EVENTS ---------------------------

Menucommand
; Choose selections from Menu Bar
;    Input  (SP) = Menu ID
;           2(SP) = Menu Item
;
      LEA    Menu,A0             ; Point to menu variable
      MOVE.L    (SP)+,(A0)       ; Save menu and item
      BSR    ChooseMenu
      CLR    -(SP)               ; Menu 0 means unhighlight
      _HiLiteMenu                ;  menu highlighted now
      RTS

ChooseMenu                       ; Find menu & item
      MOVE.W     Menu,D0         ; Menu resource ID
      CMPI   #1,D0               ; Is it menu 1?
      BEQ    AppleCmd            ;  Apple menu command

      CMPI   #302,D0             ; Is it menu 302?
      BEQ    FileCmd             ;  File menu command

      CMPI   #303,D0             ; Is it menu 303?
      BEQ    EditCmd             ; Edit menu command
      RTS                        ; No item selected
```

ChooseMenu does a simple compare and branch to the handler for each menu. The compare is by Menu ID; not the order of the menu in the bar. Selections from the three menus are handled by FileCmd, EditCmd and AppleCmd.

```
FileCmd                         ; Only item is quit
Quit                            ; Exit to finder
      BSET   #QuitFlag,D5       ; Set quit bit in flag register
      RTS

EditCmd                         ; Could be for SimpleCalc or desk accessory
      CLR.L       -(SP)         ; Space for TRUE/FALSE result & item
      MOVE   MenuItem,D0        ; Action for system to try
      ADDQ   #1,D0              ; Adjust to desk accessory standard order
      CMPI   #6,D0              ; Make NEGATE = 0 for UNDO
      BCC    @10
      MOVE   D0,(SP)            ; Edit commands 2..5
@10   _SysEdit                  ; Menu Item -> TRUE if accessory used event
      BTST   #0,(SP)+           ; Check result. If system used the menu
      BNE    NullEvent          ;  item event we are all done
      MOVE   MenuItem,D0
      CMPI   #2,D0
      BEQ    Copy               ; Item 2?
      BCS    Cut                ; Item 1?
      CMPI   #4,D0
      BEQ    ClearCmd           ; Item 4?
      BCS    Paste              ; Item 3?
      CMPI   #7,D0
      BEQ    Program            ; Item 7?
      BRA    Invert             ; Must be item 5

AppleCmd

      MOVE   MenuItem,D0        ; Check item number
      CMP    #1,D0              ; Item 1?
      BEQ    About              ;  Yes. Do About...

; Desk accesory. The name in the menu is the same as the program file name

      MOVE.L AppleHand(A6),-(SP) ; Apple-Menu handle
      MOVE.W D0,-(SP)           ; Number of chosen item
      PEA    DeskName(A5)       ; Pointer to place for name
      _GetItem                  ; Get string for item
      CLR    -(SP)              ; Space for reference number
      PEA    DeskName(A5)       ; Open desk acc
      _OpenDeskAcc              ; Name -> Reference number
      MOVE   (SP)+,D0           ; Discard result
      BRA    SelPort            ; Restore our graph port
```

```
About                                   ; Display "about" box
      CLR.L  -(SP)                      ; Clear space for dialog pointer
      MOVE   #301,-(SP)                 ; Dialog resource ID 301
      CLR.L  -(SP)                      ; Let Dialog Mgr provide storage area
      MOVE.L #-1,-(SP)                  ; Place Dialog box above all other windows
      _GetNewDialog          ; DialogID, ^ Storage, ^ Window above->Dialog pointer
      CLR.L  -(SP)                      ; No filter procedure
      PEA    ItemHit(A6)                ; ^ Area for Item Hit
      _ModalDialog                      ; filter procedure, ^ item chosen
      _DisposDialog                     ; Dialog pointer still on stack
      CMPI.W #2,ItemHit(A6)             ; End program if button 2 chosen
      BEQ    Quit
      RTS
```

The File menu has only one command, so the FileCmd label is the same as Quit, which ends the program. Quit just sets a bit in D5. Remember D5 will be moved into the CCR after we run out of events, then the top routine will see the flag and end the program.

The Edit menu is shared with the desk accessories. Before EditCmd uses the selection it checks to see if a desk accessory can accept it. The __SysEdit function will handle the item if it can. The item number must be adjusted to the standard item number values before calling __SysEdit. If the SimpleCalc window is on top, the Edit menu item won't be taken by a desk accessory. Then EditCmd will compare the item number to find the proper subroutine.

The AppleCmd routine processes selections from the Apple Menu. The first item is the program description dialog, the "About..." box. If the item number is one, we branch to About, which will be discussed later. All other items are desk accessories. The __GetItem call returns the text of a menu item in DeskName. This string is the name of a desk accessory, since only desk accessory names were put into the Apple Menu by the call to __AddResMenu above. The desk accessory is started by a call to __OpenDeskAcc, which has the same parameters as a call to open a data or resource file. The reference number returned by __OpenDeskAcc could be used to close, that is terminate, the desk accessory, but we just discard it here. When the accessory is activated, it draws its window. The call to SelPort makes sure the SimpleCalc window is selected again, before continuing.

The About routine puts up a dialog box. We start by calling __GetNewDialog to create the dialog box. This call has the same parameters as __GetNewWindow, already explained. A dialog box is in fact, just a special

kind of window. Calling _ModalDialog restricts the operator to dealing only with the dialog box next. It will return an item selected by the operator. As an example, we give a choice to continue or end the program. Normally an About box just shows the information and does not offer any options. The filter procedure parameter allows screening the events before they are handled by _ModalDialog. We don't use it here, but in another program, such as accordian simulation, you could use the filter procedure to force a beginner to play only harmonious notes. The ItemHit area contains the item number of the chosen item after _ModalDialog returns. The Return key is the same thing as item one. Item number two causes a branch to end the program, so the operator can just press Return to leave the box. Then the call to _DisposDialog erases the dialog box and returns the memory the Dialog Manager allocated.

Now we are done with the User Interface. The code we have just discussed has features required of any good Macintosh application. You should understand this part of the program well, because you can apply it to any Macintosh program you write. The Spreadsheet Function section is not as universal. All subroutines there are called by the User Interface. They are specific to a spreadsheet application. They consist of editing, drawing and updating the cell display.

Spreadsheet Functions

SelCell is called when the operator clicks the mouse to select a cell. First we convert the global-coordinate point to local coordinates which are relative to the upper corner of the document. The _GlobalToLocal procedure uses the same point, passed by a pointer, for input and output. We call DrawCell to draw the currently selected cell without a frame. Then CalcNum computes the new selected cell, returning a number from zero to 127 in D2. We store a command to select the cell in the stored program with ProgSel. Finally we call DrawSelect to draw the newly selected cell.

```
;------------------------ SPREADSHEET FUNCTIONS ---------------------------

SelCell                         ; User clicked to select a cell
      PEA     Where             ; Adjust Mouse location to window coordinates
      _GlobalToLocal            ; pointer to the point for input and output
      BSR     DrawCell          ; Un Highlight current selection
      BSR     CalcNum           ; Which cell to select
      MOVE.W D2,D4              ; Save the selected cell
      BSR     ProgSel           ; Record the action
      BRA     DrawSelect        ; Outline cell darkly
```

The Keystroke subroutine begins when the operator types a key. The key could be a digit for the selected cell value, or an editing operation. Since both digits and operations change the value of the cell or of the accumulator, we set the Redraw flag right away. When the operator types a digit, Keystroke passes it to the DigiKey routine to set a cell value. An operation key, such as a plus sign, is decoded by the NotDigit routine.

```
KeyStroke                          ; User pressed a key
      BSET   #Redraw,D5            ; Redraw selected cell when done
      MOVE.L Message,D2            ; Get Character record
; Vector to operation or put digit in cell value
      CMPI.B #'0',D2
      BCS    NotDigit              ; Not a digit
      CMPI.B #'9',D2
      BLS    DigiKey
```

The message field of the Event Record has the ASCII value of the keystroke in its lowest byte. This is all we use in SimpleCalc, as will usually be the case. The next higher byte of the Message field contains a key-cap code. You can use that byte for an accordian simulation, because it identifies an actual location on the keyboard, regardless of language. A French keyboard, for example, may have the "A" and "Z" where you are used to seeing the "Q" and "W." The ASCII values returned are the same, no matter where the key is found on the particular keyboard in use. The key code identifies the location of the key, regardless of the letter which corresponds to that location.

NotDigit is called when the operator types a key other than "0" through "9". The routine tries to match the keystroke to an operation. The call to the OperVect subroutine returns the address of an arithmetic routine or sets the Zero flag if none is found. The vector address is pushed onto the stack so that it will be executed at the end of the next call. That call, to ProgOper, saves the operation in the Accumulator program.

```
NotDigit
; Check table for operation
      BSR    OperVect          ; Check operation table for key
      BEQ    BadKey            ; Ignore keystroke if not in table
; Save address of operation. Store operation in program. Then perform it.
      PEA    (A0)              ; Push address on stack
      BRA    ProgOper          ; Save operation in accumulator program

BadKey
; Not in table so ignore key
      RTS
```

DigiKey is called with one of the characters "0" through "9" in register D2. This digit will be tacked onto the end of the value in the currently selected cell. We get a decimal value for the character by subtracting $30, the ASCII value of zero. After multiplying the current value by ten, we increase its magnitude by the new digit. This means we add the new digit to a positive number, but subtract it from a negative number by inverting it first. If the result can't be expressed in 16 bits, we have an overflow. SimpleCalc clears the number to zero. You could also refuse the digit and sound an alarm.

```
DigiKey
;  Digit typed in D2.  Add it onto end of number in (A3)
      MOVEQ  #$0F,D0
      AND.L  D0,D2             ; Clear upper nibble & junk
      MOVEQ  #10,D1            ; number base
      MULS   (A3),D1           ; Current value times 10
      BPL    DigiAdd           ; Is the cell number negative?
      NEG.L  D2                ;  then increment is negative too
DigiAdd
      ADD.L  D2,D1             ; plus key stroke
      MOVE   D1,(A3)           ; Save value
; now check for overflow
; bits 15 through 30 must be the same
      LSR.L  D0,D1             ; shift down 15 bits
      ADDQ.W #1,D1
      BEQ    DigiOK            ; OK negative number
      SUBQ.W #1,D1
      BEQ    DigiOK            ; OK positive number
```

```
; overflow. clear to zero and start over
      CLR.W  (A3)
DigiOK                          ; New cell value in range
; fall through to CellConst
```

DigiKey falls through to the CellConst routine. It is called whenever a cell has been assigned a new, constant value. CellConst clears out the program for the cell, so the value won't change automatically. It sets the Redraw bit in D5 because the cell must be drawn again with its new value. This routine will be called later on by the edit routines, which also assign constants rather than programmed values to a cell.

```
CellConst                       ; Cell set to constant value
      BSET   #Redraw,D5         ; Redraw selected cell after editing
      CLR.B  Prg(A3)            ; No program for cell
      RTS
```

The OperVect subroutine looks up the address of a routine to perform an operation. It returns the address in A0. The Zero flag is set if the character is not found in the table. The most important part of this subroutine is the structure of the table. Let's look ahead to OperTable, the table of operations and addresses. Each entry is four bytes long. The first byte is the operation code of the actual ASCII keystroke value we are going to match. The second byte is not used in order to maintain word alignment. The next two bytes indicate an address in a relative form, like that we saw in the window table. The OperVect routine scans this table until it finds a match. Then the vector address is calculated the same way as for the window table. The Zero flag is set before exiting if the TST.L finds the four-byte zero that ends the table. When a match is found, moving the offset to D0 sets clear the Zero flag. Remember that an LEA, the next instruction, does not affect the Condition Codes.

```
OperVect
; Return vector to operation from table
;       INPUT     D2 = Character to match
;       OUTPUT    A0 = Vector address
;                 D2 = Character matched
;              Z flag -> character not found
;
      LEA   Opertable,A0
OpVecLoop
      CMP.B (A0),D2              ; Compare key stroke to table
      BNE   NextEntry
; Found it
      MOVE.W 2(A0),D0            ; Vector to operation
      LEA    Opertable(D0),A0    ; Actually LEA Optable-*(PC,D0)
      RTS                        ; Return NZ
NextEntry
; Check for end of table & advance pointer
      TST.L  (A0)+
      BNE    OpVecLoop
      RTS                        ; Not found. Return Z flag set

OperTable
;  4 bytes per entry
;    byte 1 = ascii value of key
;    byte 2 not used
;    bytes 3&4 offset

      DC     '+ '
      DC     AddOper-OperTable
      DC     '- '
      DC     SubOper-OperTable
      DC     '* '
      DC     MulOper-OperTable
      DC     '/ '
      DC     DivOper-OperTable
      DC     '= '
      DC     EqOper-OperTable
      DC     $0300                    ; [Enter] key
      DC     Enter-OperTable
      DC     $0800                    ; [BackSpace] key
      DC     ClearCmd-OperTable
      DC     $1B00                    ; [Clear] on 10-key pad
      DC     ClearCmd-OperTable
      DC.L   0                   ; End of table
```

The arithmetic operations are addition, subtraction, multiplication and division. They are all similar, because they calculate with the selected cell and the accumulator. The result always goes into the accumulator. The next four routines, AddOper, SubOper, MulOper and DivOper, implement the arithmetic functions. They do all of the calculations in temporary registers. None of them check for overflow, but the division routine has to check for division by zero. The 68000 will vector to a trap if we try to divide by zero. The DivErr routine returns the highest number of the appropriate sign if division by zero is attempted. This works out rather well for converging algorithms, such as the square root formula.

```
AddOper
;  Add the selected cell into the accumulator
       MOVE.W (A3),D0
       ADD.W  D0,(A6)
       RTS

SubOper
;  Subtract the selected cell from the accumulator
       MOVE.W (A3),D0
       SUB.W  D0,(A6)
       RTS

MulOper
;  Multiply the accumulator by the selected cell
       MOVE.W (A3),D0
       MULS   (A6),D0
       MOVE.W D0,(A6)
       RTS

DivOper
;  Divide the accumulator by the selected cell
       MOVE.W (A6),D1
       EXT.L  D1
       MOVE.W (A3),D0
       BEQ    DivErr
       DIVS   D0,D1
       MOVE.W D1,(A6)
       RTS
```

```
DivErr                          ; Divide by zero
      SWAP   D1                 ; Return largest magnitude possible
      EORI.W #$7FFF,D1
      MOVE.W D1,(A6)
      RTS
```

The Enter operation just sets the accumulator to the value of the selected cell. The EQQper routine, called by the equal-sign key, sets the selected cell to the value of the accumulator. We also clear the cell program and take the equal-sign operator out of the accumulator program. Later on, when the stored program is assigned to a cell, we don't want the program execution to have any side effects on the cell selected now.

```
Enter
;  Set the accumulator to the value in the selected cell
      MOVE.W (A3),(A6)          ; Set value
      RTS

EQOper
;  Set the selected cell to the value of the accumulator
;  Clear the program in the cell
;  delete the "="from the accumulator program
      CLR    Prg(A3)
      MOVE.W (A6),(A3)
      SUBQ.W #1,D6              ; Delete the = operator
      RTS

;  User can select these Edit Menu items

Cut                             ; CUT / X
      MOVE.W (A3),Clip(A6)      ; ClipBoard = Cell
      CLR.W  (A3)               ; Cell = 0
      BRA    CellConst

Copy                            ; COPY / C
      MOVE.W (A3),Clip(A6)      ; ClipBoard = Cell
      RTS
```

```
Paste                                   ; PASTE / V
      MOVE.W Clip(A6),(A3)              ; Cell = ClipBoard
      BRA    CellConst

ClearCmd                                ; CLEAR  or Clear key
      CLR    (A3)                       ; Cell = 0
      CLR    (A6)                       ; accumulator  = 0
      CLR    D6                         ; Clear accumulator equation
      BSR    ProgSel                    ; Accumulator equation = select
      BRA    CellConst

Invert                                  ; NEGATE / N
      NEG    (A3)                       ; cell = -cell
      BRA    CellConst

Program                                 ; PROGRAM / P
; Copy the accumulator program into the selected cell
      LEA    Prg(A3),A1                 ; Point to cell program area
      LEA    Prg(A6),A0                 ; Point to accumulator program area
      MOVE   D6,D0                      ; Count of valid bytes
CopyPLoop
      MOVE.B (A0)+,(A1)+
      DBRA   D0,CopyPLoop               ; Move D0+1 bytes
      CLR.B  -(A1)                      ; Program ends with 0
      RTS
```

The Edit items can be selected by menu or keystroke. All the Edit items except Program exit through the CellConst routine because they assign constant values to the selected cell. Cut, Copy, and Paste work with the value stored in the ClipBoard variable. The equations in the comment fields indicate the action. A more elaborate program would display the Clip Board in a separate window. The Negate item just inverts the cell's value. It also functions as Undo for desk accessories.

One Edit item, Clear, can be called from the keyboard without using the Command key. This arrangement is normally against the Macintosh user-interface guidelines. However, if you have a ten-key pad, the Clear key will set the cell to zero, by calling the Clear routine. The Backspace key performs the same function if you don't have the ten-key pad. The Clear routine sets the accumulator and selected cell to zero. Then it restarts the accumulator program by setting it to one command which selects the current cell.

The Program routine is unique. This is the only way that a program is assigned to a cell. The routine uses a simple block move to copy the program from the accumulator into the selected cell. The last byte is set to zero to mark the end of the cell program.

Drawing Routines

The Drawing Routines come next. They draw the spreadsheet within the window. A subroutine draws one cell. It gets called 128 times to create the complete spreadsheet. The coordinates for the cell being drawn are set up by subroutines which appear later in the program.

```
;------------------------- DRAWING ROUTINES -------------------------

;
DrawCell
;  Put numbers into selected cell. Erase existing numbers
      PEA    CellRect
      _ERASERECT
      BSR    FrameCell          ; Draw gray box
;  Go to start of cell
      MOVE.L TxtPnt,-(SP)       ; X,Y of start of cell
      _MOVETO                   ; horiz, vertical
;  Set up to draw a value
      CLR.L  D0
      MOVE.W (A3),D0            ; First integer is value
      BPL    DrawValue
;  Precede negative number with a minus sign
      MOVE.W #$2D,-(SP)         ; minus sign
      _DRAWCHAR                 ; Draw the -
      CLR.L  D0
      MOVE   (A3),D0            ; Get value again
      NEG.W  D0                 ;  and make it positive
```

DrawCell is the subroutine to draw one cell. The cell value is pointed to by A3. The coordinates must be already set up by the CalcCellRect subroutine. DrawCell erases the current contents of the cell and puts up a light gray border by calling FrameCell. If the cell is marked later on, the black border will cover up the gray border already there. Next, DrawCell puts the digits into the cell. It calls __MoveTo with the pen location which CalcCellRect put into Txtpnt. There it draws a minus sign if necessary,

leaving the pen at the start of the number. It loads the value and, if the number is not positive, draws the minus sign and then makes the number positive. Notice that it reloads the number into D0 after calling __DrawChar because D0 is likely to be altered.

Next the number is converted to digits. It is done by the same algorithm you probably have used to convert numbers by hand to bases other than 10. It can be summarized by these three steps:

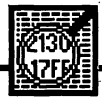
1. Divide the number by the base.
2. Write remainder to left of last digit.
3. Repeat from step 1, using the quotient for the number.

DrawValue implements this algorithm to convert to base ten from binary, because the 68000 can divide in binary but not in base ten. People usually find it easier to divide in base 10 than in any other base. DrawValue does one thing a little bit differently. Instead of writing the results from right to left, it saves each digit on the stack. Then when it is all done, it can write the digits in the proper order, by popping them from the stack as it draws characters from left to right.

```
DrawValue
      DIVU   #10,D0              ; Draw in base 10
      SWAP   D0                  ; Get remainder
      ORI.B  #'0',D0             ; Make digit a character by oring in zero
      MOVE.W D0,-(SP)            ; Save digit to draw
      SWAP   D0                  ; Restore quotient
      EXT.L  D0                  ; Leading digits?
      BEQ    DrawDigit           ; No more digits. Draw them all now
      BSR    DrawValue           ; Calculate next higher digit
DrawDigit
      _DRAWCHAR                  ; Draw a byte from stack
      RTS                        ; Return to draw next byte or exit
```

Let's go through DrawValue step by step. We start with a 32-bit integer for the divide instruction, DIVU. The unsigned divide leaves the quotient in the lower word of D0 and the remainder in the upper word. SWAP and ORI.B make the remainder into a printable character. The digit is saved on the stack for a future call to __DrawChar. Notice that character parameters on the stack are in the lower byte of the word. SWAP restores the quotient. Now here is the tricky part. EXT.L expands the quotient to a 32-bit integer, but it also sets the Condition Codes. If the quotient is zero, then there are

no more digits to the left of the current digit. BEQ goes to DrawDigit to draw the last character and return. But if the quotient is not zero we do a BSR back to the start of DrawValue. This will draw the characters to the left. Then the RTS after __Drawchar will return to the instruction after the BSR—which is DrawDigit itself. So the various digits will be stacked up by repeated passes through BSR DrawValue. Then after the last digit, repeated loops of DrawDigit will print them all out.



Discussions and Comparisons

DrawValue is a recursive algorithm. There are other, non-recursive, ways to print an integer in decimal. But some computations can only be done by recursive means. We often think of higher-level languages, such as Pascal, when talking about recursion. Here is a Pascal implementation of the algorithm:

```
DrawValue(ValueToDraw:INTEGER);
 VAR DigitToDraw :CHAR;
 BEGIN
  DigitToDraw := CHR(ORD('0')+ValueToDraw MOD 10);
  ValueToDraw := ValueToDraw DIV 10;
  IF ValueToDRAW <> 0 THEN DrawValue(ValueToDraw);
  DrawChar(DigitToDraw)
 END;
```

In the Assembly language version, we used six bytes of the stack for each digit printed. There was a four-byte return address and a two byte character. But the Pascal implementation uses twice as much space on the stack! Besides the two-byte variables, ValueToDraw and DigitToDraw, there is a return address and a four-byte link for each pass. The link is the old value of A6 that has to be saved on the stack by the LINK instruction which begins every Pascal procedure. So writing in Assembler can be advantageous for memory and speed, especially on recursive algorithms which may require high performance.

Just before leaving DrawValue, there is one more change we can make. Would you like the spreadsheet to work in octal? Changing just one constant in DrawValue will do that, since it is a recursive algorithm. Just change the #10 in the divide instruction to #8 and presto, the values are drawn in octal. Make the same change to the MULS in Digikey to input in octal. If you make these values a variable, you have a decimal to octal converter.

MarkCell and FrameCell do basically the same thing. FrameCell draws a thin gray border, whereas MarkCell draws a thick black border. MarkCell defers to FrameCell if the SimpleCalc window is not selected. Thus the window is highlighted by the black border around the selected cell and accumulator.

```
MarkCell
;  Draw Selected cell border Dark so we know it is selected
       BTST    #FrontFlag,D5       ; Is our window active?
       BEQ     FrameCell           ;  No. Don't highlight
       _PENNORMAL                  ; Normal Width,Mode,Black Color
       MOVE.L #$00030003,-(SP)     ; Width,Height=3
       _PENSIZE
       PEA     CellRect            ; pointer to separating rectangle
       _FRAMERECT                  ; Draws box INSIDE CellRect
@10    RTS

FrameCell
;  Outline cell with light border
       _PENNORMAL                  ; Normal Width,Mode,Black Color
       MOVE.L GRAFGLOBALS(A5),A0 ; Pointer to QD globals
       PEA     GRAY(A0)            ; Standard pattern
       _PENPAT
       PEA     CellRect            ; pointer to separating rectangle
       _FRAMERECT                  ; Draws box INSIDE CellRect
       RTS
```

To understand how these routines work, recall how QuickDraw coordinates define the screen. The coordinates are thin lines between the pixels. The drawing of rectangles takes place inside a box formed by the coordinates. The rectangle, CellRect, is a box around the cell. The walls of this box are common to the walls of the surrounding cells. FrameCell and MarkCell call __FrameRect to draw this box. That makes the gray lines between two cells two pixels thick.

We give examples of several other QuickDraw calls in these subroutines. The __PenNormal call sets the default pen. This is a solid black pattern, one pixel wide by one pixel high. __Pensize is used to set height and width to any value chosen, for MarkCell we choose three by three. You must have a pattern to set the pen to with the __PenPat. Some standard patterns are available from QuickDraw. Do you remember the parameter we passed to __InitGraf during initialization? Well, QuickDraw set up the default patterns in some of the space provided by that pointer. The variable, GrafGlobals, holds a pointer to the QuickDraw global area. The patterns, such as Gray, are at an offset into that area. The constants we used for GrafGlobals and Gray are defined in the QuickEqu.D include file.

DrawInside draws the entire spreadsheet. It uses repeated calls to DrawCell to draw each one. When all done, it falls through to DrawSelect to mark the selected cell and accumulator if the window is active.

```
DrawInside
; Draw the entire spreadsheet
      MOVE   D7,-(SP)          ; Save cell to recalculate
      MOVEQ  #$7F,D7           ; Highest number cell
InsideLoop
      MOVE   D7,D2
      BSR    CalcCellRect
      BSR    DrawCell          ; Draw numbers
      DBRA   D7,InsideLoop
      MOVE   (SP)+,D7          ; Restore cell to recalculate
; Draw the selected cell and Accumulator.  Restore pointers
;    Fall through to DrawSelect
```

DrawSelect draws just the selected cell and the accumulator. It clears the update flag in D5 which is the signal for the MainProgram to call DrawSelect. Then it sets up and draws first the accumulator and then the selected cell. This leaves the rectangle and data pointer of the selected cell set up, which is the normal state, when it finishes.

```
DrawSelect
; Draw the selected cell and Acc. Exit with pointers set up for selected cell
      BCLR   #Redraw,D5          ; Clear update needed flag
; Redraw accumulator
      MOVEQ  #AccCell,D2         ; Cell num of accumulator
      BSR    CalcCellRect
      BSR    DrawCell
      BSR    MarkCell            ; Mark accumulator
; Redraw Selected cell
      MOVE   D4,D2               ; Selected cell
      BSR    CalcCellRect
      BSR    DrawCell
      BRA    MarkCell            ; Highlight cell
```

Spreadsheet Updating

The next section is called Spreadsheet Updating. It has only one entry point, Calculate, which computes the value from the program for one cell. To execute the saved program, it uses the same subroutines called when the user recorded the program. First it has to save the current state of things, because the user may be in the middle of editing a different cell. The registers are saved on the stack. The accumulator is saved too, by loading into the D0 register before the MOVE multiple instruction. When complete, Calculate will restore the registers and accumulator.

```
;----------------------- SPREAD SHEET UPDATING --------------------------

Calculate
      MOVE   (A6),D0              ; Save accumulator
      MOVEM.L D0/D6/A3/A4,-(A7) ; Save registers
      ADDQ.B #1,D7               ; Next cell to update
      BPL    CalcNext
      CLR    D7                  ; Start with cell 0
CalcNext
      MOVE.W D7,D2
      BSR    CalcPntr            ; Point to cell data
```

```
        BEQ     CalcExit            ; No need to calc accumulator
        TST.B   Prg(A3)             ; Any program ?
        BEQ     CalcExit
        MOVE.L  A3,A4               ; Pointer to update cell
        CLR     D6                  ; Start of program
        CLR.W   (A6)                ; Start with clear acc
```

Calculate finds the next cell to work on by adding 1. If the byte goes negative, at 128, it starts over from cell zero. The CalcPntr subroutine addresses the data, but also returns a zero flag if it is the accumulator. There is no need to recalculate the accumulator, so we exit in that case. We also exit if there is no program, which means the first byte is the ending zero. If there is a program we use A4 to point to it. We zero the accumulator, because the programs start to record after a clear command. The next section, CalcLoop, executes each program byte.

CalcLoop performs one saved instruction with each cycle. It keeps repeating until it finds the zero byte that ends the program. There are only two types of program byte: an arithmetic command or a command to select a cell. The BCLR instruction distinguishes them. Remember that it not only clears a bit, but checks its current state at the same time! If the bit was already clear, we go on to execute the operation. If not, we call CalcPntr to change the selection. CalcLoop goes round and round until the little program is finished.

```
CalcLoop
        MOVE.B  Prg(A4,D6),D2       ; Get Code byte
        BEQ     CalcDone
        ADDQ    #1,D6               ; Point to next byte
        BCLR    #7,D2               ; Is it Select or arithmetic?
        BEQ     CalcCheck
; select cell for input to calculator
        BSR     CalcPntr
        BRA     CalcLoop
```

CalcCheck has to execute all of the arithmetic instructions. A call to the same OperVect routine which handled the original keystroke does this readily. After the command, CalcCheck normally returns to CalcLoop for the next program byte. But if the command is not found in the table, we hit a STOP instruction. This is not ordinarily possible, of course. The operations were put into the program after they had already been found in the table and executed. The instruction is here to demonstrate STOP as a debugging tool. To use a STOP instruction, the data value should have the $2000 bit set. This value goes into the Status Register setting the Supervisor State bit. If the $2000 bit is zero, the Supervisor State is reset, causing an immediate error.

```
CalcCheck
; Find operation in table
      MOVE.B D2,D0
      BSR    OperVect
      BEQ    CalcOper          ; Found it
; Not in table. Impossible !!
      STOP   #$2000            ; No privilege violation
CalcOper
      JSR    (A0)              ; Execute operation
      BRA    CalcLoop
```

When we reach CalcDone, the entire program for the cell has been executed. To minimize redrawing, we check to see if the value of the cell has changed. If it has changed and the cell is the selected one, we just set the Redraw flag to draw it later. To update another cell, we temporarily make it the selected cell and call DrawCell. The CalcExit routine just restores the registers and accumulator and returns. Remember that we saved the accumulator in D0.

```
CalcDone
;  end of prog. assign value. Redraw cell if it changed
       MOVE    (A6),D0              ; Get result
       CMP.W   (A4),D0              ; Compare to old value
       BEQ     CalcExit             ; No change
       MOVE.W D0,(A4)               ; Assign new value
;  new value. redraw cell
       CMP     D7,D4                ; Calculating selected cell?
       BNE     CalcUpdate
       BSET    #ReDraw,D5           ; Set flag to do later
       BRA     CalcExit

CalcUpdate                          ; Redraw the re calculated cell
       MOVE    D7,D2
       BSR     CalcCellRect
       BSR     DrawCell
       MOVE    D4,D2                       ; Selected cell
       BSR     CalcCellRect

CalcExit
       MOVEM.L (A7)+,D0/D6/A3/A4 ; Restore saved registers
       MOVE    D0,(A6)              ; Restore accumulator
       RTS
```

The Spreadsheet Subroutines provide services for the previous routines. These subroutines find the cell a point is in, calculate the dimensions for a cell and store data in the cell program.

The CalcNum routine tells which cell a point is in. To convert a point to a cell number we first calculate the column the horizontal coordinate is in. There are 56 pixels in each cell, so we divide the coordinate by 56. Then calculate the row by dividing the vertical coordinate by the height of each cell, 16 pixels. We use an LSR instruction to divide by 16 because it is faster than the divide instruction. Eight times the row number plus the column number gives the cell number. We use an ASL instruction instead of MULU here because it, like LSR, is faster than the corresponding arithmetic instruction.

```
;----------------------- SPREAD SHEET SUBROUTINES --------------------------

CalcNum
;  Calculate the cell number that contains a point
;  Input   Where = point in local coordinates
;  Output  D2    = cell number
     CLR.L  D0
     MOVE.W WhereH,D0
     DIVU   #56,D0              ; Horiz pixels/cell
     MOVE.W WhereV,D2
     LSR.W  #4,D2               ; 16 Vert pixels/cell
     ASL.W  #3,D2               ;  8 cells  Horiz per line
     ADD.W  D0,D2               ;  Cell number = 8 * y + x
     RTS
```

The CalcCellRect routine calculates the box that encloses a cell. It also figures the pen location to start drawing the numbers inside the box. At the end, it falls through to CalcPntr which loads A3 with the pointer to the cell's data. Calculating the rectangle is similar to CalcNum in reverse. Dividing the cell number by eight rows per column gives the row number as the quotient and the column number as the remainder. The equations for the box dimensions are simple:

Top = cell height * row number
Left = cell width * column number
Bottom = Top + cell height
Right = Left + cell width

These values are set using the offset constants defined in QuickEqu.D. When the point and rectangle are complete, CalcCellRect goes on to the next routine.

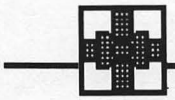
```
CalcCellRect
;  Calculate a cell Rectangle from  a cell number
;  Input   D2       = cell number
;  Output  D2       = cell number
;          A3       = pointer to cell data
;          TxtPnt   = first text pixel of cell
;          CellRect = rectangle separating cell
;
      MOVE.W D2,D1
      EXT.L  D1                 ; Divide always 32 bits
      DIVU   #8,D1              ; 8 columns per row
      MOVE.L D1,D0
      SWAP   D0                 ; Remainder in upper word
      LEA    CellRect,A0        ; Point to cell rectangle
      LEA    TxtPnt,A1          ; Point to start of text
      MULU   #56,D0             ; 56 pixels per column
      MOVE   D0,left(A0)        ; Left separator
      ADDQ   #4,D0              ; Margin before text
      MOVE.W D0,h(A1)           ; First text location
      ADD.W  #52,D0             ; Advance to next column
      MOVE.W D0,right(A0)       ; Right separator
      ASL.W  #4,D1              ; 16 pixels per row
      MOVE.W D1,top(A0)         ; Top separator
      ADD.W  #12,D1             ; Ascent of text + top margin
      MOVE.W D1,v(A1)           ; First text location
      ADDQ   #4,D1              ; Advance to next row
      MOVE.W D1,bottom(A0)      ; Bottom separator
; Fall through to calculate cell data pointer
```

The CalcPntr subroutine sets up a pointer to the cell data. The calculation is simple. An offset of 32 bytes for each cell is figured from A6. The exclusive-or instruction adjusts the accumulator location to cell 127, in the bottom corner of the screen. You could also use a subtraction in this case. But the EOR will work for any value of AccCell, the cell-number constant of the accumulator.

```
CalcPntr
;  Calculate the pointer to a cell value
;  Input   D2   = cell number
;  Output  D2   = cell number
;          A3   = pointer to cell data
;          Z flag -> cell number is accumulator
      MOVEQ #AccCell,D0          ; Make accumulator = cell number 127
      EOR.W D2,D0                ; Calc relative location
      ASL   #5,D0                ; 32 bytes / cell
      LEA   0(A6,D0.W),A3        ; Does not affect Condition codes
      RTS
```

The ProgSel subroutine stores a command to select a cell in the program. It just turns on the $80 bit in the cell number, indicating a selection command instead of an operation, and continues with ProgOper to actually store the byte. The ProgOper subroutine is also called to store an operation command. Actually it just stores the byte in D2 into the program area, then ProgOper advances the program counter, unless the storage area is full.

```
ProgSel                          ; Store selection operation in program
; Put selected cell, D4, into program with $80 flag
      MOVE   D4,D2
      BSET   #7,D2
; Fall thru to ProgOper to put byte into program

ProgOper                         ; Store arithmetic operation in program
; Put byte in D2 into accumulator program
      MOVE.B D2,Prg(A6,D6.W)     ; Store byte in program list
      CMPI.B #ProgLast,D6        ; More program space?
      BEQ    ProgOVF
      ADDQ.B #1,D6               ; Advance program pointer
      RTS
ProgOVF                          ; Program too big
      RTS
```



Summary

Now we have gone completely through the sample program. You may want to go back and review parts of the code you are not sure about. We think this example will help you to write your own programs. In Chapter 10, Advanced Routines, we will show a few more routines that may be useful. Happy coding!

CHAPTER

10

Some Advanced Subroutines Not in SimpleCalc

In the last chapter we took a look at the sample program, SimpleCalc. Although the sample program has many useful calls in it, there are still many things you may want to do which have not been covered yet. In this chapter we are going to go over several useful subroutines. Once you understand the sample program, it will be very easy for you to adapt and add these subroutines to programs you create yourself. Some more important Toolbox calls are shown in these examples. After you have mastered them, you can try using the other traps in Appendix F, based on what you already know about similar calls. But even Appendix F does not begin to exhaust the power of the Macintosh! For an even deeper summary, you may consult *Macintosh Revealed*, by Stephen Chernicoff, and the encyclopedic technical manual, *Inside Macintosh*, available from Apple Computer Inc.

Using the Memory Manager

The Memory Manager always allocates the space for code and system data. You can get the best performance from your program by calling the Memory Manager to allocate the application data space as well. Some system calls optionally let you provide your own space for system data when you can do it more efficiently. If you just pass NIL in those cases, the Memory Manager will take over. You can request both fixed and relocatable blocks of memory. The two types have different calls and uses.

When you request a non-relocatable block, the Memory Manager returns a pointer to the first word of your data space. The area is reserved

at the bottom of the heap. It will never move and will be available until you dispose of it. Non-relocatable blocks are the easiest to use, but repeated calls to create and dispose can split the available heap into small fragments. The relocatable blocks are designed to avoid this.

A relocatable block is always addressed by its handle. The handle is a pointer to a master pointer. The master pointer in turn, contains the current location of the block. When the Memory Manager has to move a relocatable block, it puts the new address in the master pointer location. This way, the handle is always valid even though the block may be moving around. When you want to be sure the block stays put, a special call to "lock" the handle prevents the Memory Manager from moving it until it is unlocked.

```
                                ;Using a non-relocatable block
                                ;Create a new, 1000-byte, non-relocatable
                                ;block
MOVE.L      #1000,D0            ;Size in bytes
__NewPtr                        ;Create a fixed block
MOVE.L      A0,BlockPtr         ;Save pointer returned in A0
TST.W       D0                  ;Enough space for new block?
BNE         Error               ;D0 contains error code or 0
                                ;Expand the non-relocatable block to
                                ;2000 bytes
MOVE.L      BlockPtr,A0         ;Get pointer to existing block
MOVE.L      #2000,D0            ;New size in bytes
__SetPtrSize                    ;Expand the block
TST.W       D0                  ;Enough room to grow?
BNE         Error               ;D0 contains error code or 0
                                ;Store some data in the block
                                ;Get a pointer to the first word of the
                                ;block
MOVE.L      BlockPtr,A0
                                ;Put four characters into the start of the
                                ;block
MOVE.L      # 'Data',(A4)
                                ;Return the non-relocatable block
MOVE.L      BlockPtr,A0         ;Get handle to old block
__DisposePtr                    ;Return block space

                                ;Using a relocatable block

                                ;Create a new, 1000-byte, relocatable
                                ;block
MOVE.L      #1000,D0            ;Size in bytes
                                ;Create block and fill it with zeros
__NewHandle ,CLEAR
```

```
            MOVE.L      A0,BlockHand ;Save handle to new block
            TST.W       D0           ;Any error creating block?
            BNE         Error        ;D0 contains error code or 0

                                     ;Expand the relocatable block to 2000
                                     ;bytes

            MOVE.L      BlockHand,A0 ;Get handle to existing block
            MOVE.L      #2000,D0     ;Size in bytes
            __SetHandleSize          ;Expand the block
            TST.W       D0           ;Was there enough room?
            BNE         Error        ;D0 contains error code or 0

                                     ;Lock and modify the relocatable block
            MOVE.L      BlockHand,A0 ;Get handle to existing block
            MOVE.L      (A0),A4      ;Get a pointer to the data
            __HLock                  ;Keep the block from moving

                                     ;Modify the block. You can make any
                                     ;kind of system call here
            MOVE.L      # 'Data',(A4) ;Put some data into block
                                     ;Unlock the block when you are done
            MOVE.L      BlockHand,A0 ;Get handle to locked block
            __HUnlock                ;Unlock the block

                                     ;Return the relocatable block
            MOVE.L      BlockHand,A0 ;Get handle to old block
            __DisposHandle           ;Return block space

                                     ;Data area
BlockPtr    DS.L        1            ;Four-byte pointer to start of block
BlockHand   DS.L        1            ;Handle to relocatable block
```

Variable Text in a Dialog Box

The dialog boxes we have used so far have all used constant text, hard-coded into the resource file. Often you will want to put different messages into the same geometric box, or provide some data in the box which you can't know until the program is running. Here we will show two ways to vary the text in a dialog box.

```
                                     ;Using variable text in a dialog box
                                     ;Create the dialog
                                     ;Clear space for the dialog pointer
            CLR.L       -(SP)

                                     ;Pass the ID number of the dialog box,
                                     ;#283
```

```
MOVE        #283,-(SP)
                                ;Pass NIL to let the Dialog Manager
                                ;provide the storage area
CLR.L       -(SP)
                                ;Place this dialog box above all other
                                ;windows
MOVE.L      #-1,-(SP)
__GetNewDialog                  ;Start the dialog
                                ;Save the dialog pointer for future calls
                                ;with this dialog box
MOVE.L      (SP)+,DiaPtr(A5)
                                ;Set the parameter strings
PEA         'You can set four variable strings to use'
PEA         'in Static Text items in a dialog box.'
PEA         'Refer to the strings as 0 through 3'
PEA         'in the Resource File'
__ParamText
BSR         @10                 ;Wait for operator to see
                                ;Change two of the parameter strings.
                                ;The other two stay the same
CLR.L       -(SP)               ;String^0 won't be changed
CLR.L       -(SP)               ;String^1 won't be changed
PEA         'You can change some of the strings'
PEA         'and leave others as they are.'
__ParamText
BSR         @10                 ;Wait for operator to see
                                ;Changing a text item directly

                                ;Get a handle to a dialog item. All we need is
                                ;the handle, but a call to
                                ;__GetDItem returns the item type and the
                                ;enclosing rectangle as well
MOVE.L      DiaPtr(A5),-(SP)    ;Load the dialog pointer
MOVE.W      #2,-(SP)            ;Second item in list
                                ;A code number for the type of this item
                                ;will be returned in ItemType
PEA         ItemType
PEA         ItemHand(A5)        ;Item handle will be returned
PEA         ItemBox             ;Item rectangle will be returned
__GetDItem                      ;Get info for the item

                                ;Set the text of item number two
MOVE.L      ItemHand(A5),-(SP)  ;Item handle
PEA         'Or you can change the text directly'
__SetIText                      ;Set dialog item text
```

```
            BSR         @10

                                        ;All done. Close up and go home
            MOVE.L      DiaPtr(A5),-(SP)
            __DisposDialog
            RTS

                                        ;Update the dialog and wait for the
                                        ;operator to press [Return]
@10         CLR.L       -(SP)           ;No filter procedure
            PEA         ItemHit         ;^Area for item hit
            MOVE.L      DiaPtr(A5),-(A7) ;Redraw the box if necessary and wait
                                        ;for operator action
            __DrawDialog
            __ModalDialog               ;filter procedure,^item chosen
                                        ;ItemHit now contains the item chosen.
                                        ;Pressing [Return] is the
                                        ;same as selecting item number one
            RTS

                                        ;Data areas
ItemHit     DC.W        1               ;Item chosen.
                                        ;Item parameters. These items are set by
                                        ;__GetDItem. They are defined
                                        ;with the values for item two already in
                                        ;them as an example.
ItemType    DC.W        8               ;Type of changed item.
ItemBox                                 ;Rectangle enclosing text item
            DC.W        36              ;top
            DC.W        30              ;left
            DC.W        80              ;bottom
            DC.W        330             ;right

                                        ;Handle and pointer
DiaPtr      DS.L        1               ;Storage for dialog pointer
ItemHand    DS.L        1               ;Storage for item handle
```

The resource for the dialog should look like this:

```
Type DLOG
,283
*Outside corners of the box, relative to the desk top
100 62 200 450
Visible NoGoAway
1
```

```
0
*Item list is number 283
283
*Item list for the dialog
Type DITI
,283
*Number of items in list
2
*First item says "Note:" followed by parameters zero and one
 StatText Disabled
 20 30 52 330
Note: ^0^1
*Second item starts as parameters two and three, but will be changed
 StatText Disabled
 52 30 84 330
^2^3
```

Setting the Cursor

One friendly invention in the Macintosh is the pictorial cursor. The cursor on the Macintosh is an image that moves with the mouse. A blinking vertical line where text may be entered is called a caret. The cursor symbol can tell the operator what is going on in the machine. Less advanced machines sometimes use the height and blink rate of the caret to designate the current mode. But with the QuickDraw cursor images, the Macintosh can provide a picture for every occasion.

A Macintosh cursor is actually two small patterns. One bit pattern, called the image, shows the detail of the picture. The second bit pattern is the cursor mask. It blocks out the image underneath the cursor. Both bit patterns together make each cursor pixel appear black, white, clear or inverting. To draw the cursor, the Toolbox first does a bit-clear operation using the mask. Then it draws the image pattern in Exclusive-Or mode. The resulting cursor can have holes where the underlying image shows through, and inverted areas where the drawing below changes color, as well as detail on the cursor itself in black or white.

The center of the cursor is called the hot spot. The cursor is always drawn with the hot spot at the mouse location. The hot spot is defined relative to the top left corner of the cursor. If the cursor is an arrow pointing North by NorthWest, the hot spot should be at the upper left corner, since the operator will see it pointing to the upperleft. But if the cursor is a finger pointing to the right, the hotspot should be at the tip of the finger.

Several handy cursors are already built in. They come in the System Resource file as standards. They have certain standard use conventions

which you can follow to make your programs easy to learn. If you need even more variety, you can define your own cursor in a resource file. This example shows how to load and use one of the standard cursors. Figure 10-1 shows a cursor along with its image and mask. The format for a cursor in memory is shown in Appendix F (Trap). In that Appendix you can also see how to put a cursor into your own Resource File.

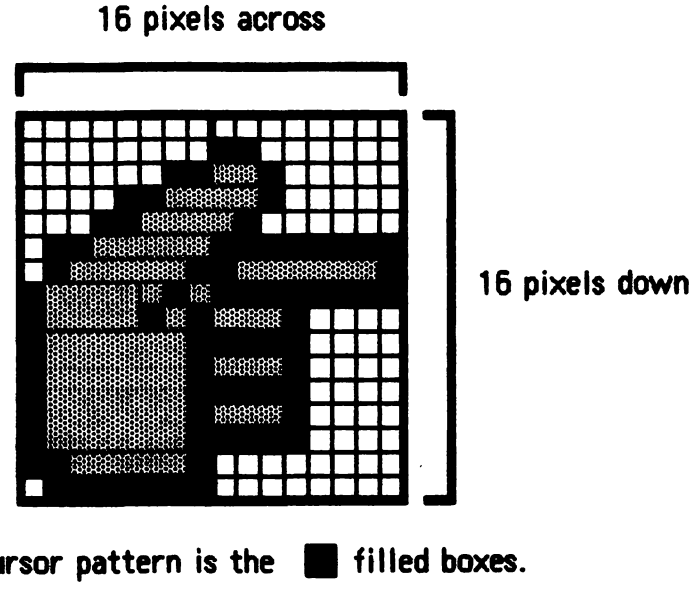


Figure 10-1 Cursor Image and Mask

```
                                ;
                                ;Set the cursor
                                ;
                                ;resource IDs for standard cursors
                                ;These are predefined in SysEqu.D
iBeamCursor     EQU     1       ;for selecting text
crossCursor     EQU     2       ;for selecting graphics
```

```
plusCursor    EQU       3              ;for drawing graphics
watchCursor   EQU       4              ;for indicating a long delay

CursData      DS.B      68             ;Storage for cursor data

FirstWatch                             ;Enter here to load the cursor data when
                                       ;you use
                                       ;the watch cursor for the first time
              CLR.L     -(SP)
                                       ;Use the ID for the watch cursor defined
                                       ;in the System Resource file
              __GetCursor              ;Get handle to cursor
              MOVE.L    (SP)+,A0
              MOVE.L    (A0),A0        ;Get pointer to cursor
              LEA       CursData(A5),A1 ;Point to storage
              MOVEQ     #16,D0         ;Copy 68 bytes
@10           MOVE.L    (A0)+,(A1)+    ;Move four bytes at a time
              DBRA      D0,@10
SecondWatch
                                       ;Set the cursor. Enter here to set the
                                       ;cursor after CursData is loaded
              PEA       CursData(A5)   ;Pointer to data we stored
              __SetCursor              ;Change the cursor
              RTS
```

Changing the Cursor Shape with Its Position on the Screen

The previous example showed the basics of setting the cursor. Many applications use a cursor within the document, which is not appropriate for selecting menu items or scrolling the window. This example shows you how to change the cursor back to an arrow for use on the desk top when the mouse is moved off the document. When the mouse comes back into the content area of the window, the special cursor is set again. To achieve this effect, we add a small routine to the main loop which handles events. When there are no events to process, we adjust the cursor to the proper shape.

```
                                       ;
                                       ;Adjust the cursor.
                                       ;Make it change to an arrow when not in
                                       ;the document
                                       ;Make it a cross-hair cursor when in the
                                       ;window
CursData      DS.B      68             ;Storage for cursor data
```

```
InitMain
                                        ;This is the main initialization called once
                                        ;at the start of the program
                                        ;Get the System Cross-hair cursor to use
                                        ;for graphic selection
        CLR.L-(SP)
        MOVE.W      #crossCursor,-(SP)
        __GetCursor                     ;Get Handle to cursor
        MOVE.L      (SP)+,A0
        MOVE.L      (A0),A0             ;Get pointer to cursor
        LEA         CursData(A5),A1     ;Point to storage
        MOVEQ       #16,D0              ;Copy 68 bytes
@10     MOVE.L      (A0)+,(A1)+         ;Move 4 bytes at a time
        DBRA        D0,@10
                                        ;Continue with other initialization
        ...

GetEvent
        ...                             ;Start event loop
        __GetNextEvent                  ;Check for any activity
        BTST        #0,(SP)+
        BEQ         NoEvent             ;No new event. Update cursor
        ...                             ;Handle new event

NoEvent
        __SystemTask                    ;Call System task when idle
        PEA         Where(A5)
        __GetMouse                      ;Find mouse location
        CLR.W       -(SP)               ;True/False result
        MOVE.L      Where(A5),-(SP)     ;Mouse point
        MOVE.L      WindowPointer       ;A0
        PEA         portRect(A0)        ;Pointer to port rect
        __PtInRect                      ;point,^rect -> True/False
        BTST        #0,(SP)+            ;Mouse in window?
        BNE         InDoc               ;Yes. Set crosshair
        BEQ         NotInDoc            ;On desk. Set arrow

InDoc                                   ;Set cross-hair cursor while inside the
                                        ;window
        PEA         CursData(A5)        ;Pointer to data we stored
        __SetCursor                     ;Change the cursor
        RTS                             ;Back to main loop

NotInDoc                                ;Set arrow cursor while on the desk top
        __InitCursor
        RTS                             ;Back to main loop
```

Marking a Selection on the Screen by Making It Blink

When an item is selected for editing, you should inform the operator by drawing it distinctively. A selected item can be made to blink, be redrawn in gray, or both. In this example we show how to make an item blink. Like adjusting the cursor, this is done by inserting a routine in the event loop.

The continuous clock, available by a call to __TickCount, provides a convenient timer with a period of about one second. If your application has a caret you can use the same routine to make the caret blink. A more elaborate routine could adjust the blink time to the system variable, CaretTime. This four-byte integer is stored in parameter RAM so it remains even when the machine is restarted.

```
                                        ;Make a drawing item blink every ½
                                        ;second
IdleBlink
            CLR.L       -(SP)
            __TickCount
            MOVE.L      (SP)+,D0
            LSR         #6,D0           ;Divide by 64
            BCS         @10
            BSET        #OnOff,D7       ;Cursor should be on
            BEQ         @20             ;Draw it to display
            RTS                         ;already on exit
@10         BCLR        #OnOff,D7       ;Cursor should be off
            BNE         @20             ;Draw again to erase it
            RTS                         ;already off exit

                                        ;Draw your structure. Here it is a Box
                                        ;around a circle
@20         MOVE        #patXor,-(SP)   ;Set XOR mode, number 10
            __PenMode
            PEA         theBox
            __FrameRect                 ;Open square
            PEA         theBox
            __PaintOval                 ;Solid circle
            __PenNormal                 ;Restore pen
            RTS
theBox      DC          50,50,250,250
GetEvent
            ...                         ;Start event loop
            __GetNextEvent              ;Check for any activity
            BTST        #0,(SP)+
```

```
            BEQ         NoEvent         ;No new event. Update cursor
            ...                         ;Handle new event
NoEvent
            BSR         IdleBlink       ;Make selection blink
            _SystemTask                 ;Let the System blink any desk
                                        ;accessories
            ...                         ;Any other processing
            RTS
```

Dragging Selections with the Mouse

Some applications let the operator move a selected item. To be most in keeping with the Macintosh style, the item should move smoothly following the mouse. The routine remembers the mouse location when it first starts. Then, as the mouse is moved, it applies the same relative motion to the location of the selected item. This lets the operator press the button anywhere on the selected item without the item jumping to align itself with the mouse. The subroutine should be called from the main loop. We saw how to do this in earlier examples. The item has to be drawn the first time by a call to the Select routine below, before you can start to move it with the DragItem routine.

```
Select
                                        ;Initialize the item that is going to be
                                        ;moved
            MOVE.L      #$00AC0100,D0   ;Start item in middle of screen
            MOVE.L      D0,ItemLoc(A5)  ;Store the item location
            MOVE.L      D0,-(SP)        ;Item location onto stack
            _MoveTo
            BSR         DrawItem        ;Draw the original image
            RTS
NullEvent
            ...
DragItem
                                        ;Enter here periodically if an item is
                                        ;selected to see if the
                                        ;mouse button is down
            CLR.W       -(SP)
            _Button
            BTST        #0,(SP)+        ;Is operator holding the button?
```

```
        BNE         @10             ;Yes. Track mouse with selection
                                    ;Button up. Return to main loop
        RTS

                                    ;Button went down. Start to drag the
                                    ;selected item. You may want
                                    ;to add code here to check that the
                                    ;mouse is close enough to it.

                                    ;Get the starting location
@10     PEA         OldMouse(A5)
        _GetMouse

                                    ;Drag item with the mouse
@20     PEA         NewMouse(A5)
        _GetMouse

                                    ;If mouse hasn't moved don't draw again
        MOVE.L      NewMouse(A5),D0
        CMP.L       OldMouse(A5),D0
        BEQ         @30

                                    ;Mouse moved. Draw in new location.
                                    ;Erase in old location.
        MOVE.L      ItemLoc(A5),-(SP) ;Old location onto stack

                                    ;Compute new location. Item = Item +
                                    ;New Mouse - Old Mouse
                                    ;Calculate vertical
        MOVE.W      NewMouse(A5),D0
        SUB.W       OldMouse(A5),D0
        ADD.W       D0,ItemLoc(A5)
                                    ;Calculate horizontal
        MOVE.W      NewMouse+2(A5),D0
        SUB.W       OldMouse+2(A5),D0
        ADD.W       D0,ItemLoc+2(A5)

                                    ;NewMouse location will be the next
                                    ;OldMouse location
        MOVE.L      NewMouse(A5),OldMouse(A5)

                                    ;Draw new version before erasing old so
                                    ;item can't disappear
        MOVE.L      ItemLoc(A5),-(SP) ;New location onto stack
        _MoveTo
        BSR         DrawItem        ;Draw new image
        _MoveTo
```

```
            BSR         DrawItem        ;Erase old image
                                        ;Repeat if button still down
@30         CLR.W       -(SP)
            __Button
            BTST        #0,(SP)+
            BNE         @20             ;Still down. Repeat
                                        ;All done. Go back to main loop
            RTS
                                        ;Draw your structure. Here it is the
                                        ;letter "A"
DrawItem    MOVE        #srcXor,-(SP)   ;Set XOR text mode, number 2
            __TextMode
            MOVEQ       # 'A',D0        ;Make a CHAR to draw
            MOVE.W      D0,-(SP)
            __DrawChar                  ;Draw or erase letter "A"
            RTS
                                        ;Data area
NewMouse    DS.W        2               ;Current mouse location
OldMouse    DS.W        2               ;Previous mouse location
ItemLoc     DS.w        2               ;Item location
```

Marking, Disabling, and Changing Menu Items

Many of the options in your program can only be available at certain times. You can keep the operator informed about the conditions and available options by varying the menu items. The Menu Manager lets you put checks or other marks in front of menu items. You can also change the text of an item to indicate what is available. For example, an item may read "Show Clock." When that item is selected, a clock may appear on the screen. Now you can either put a check mark beside "Show Clock," in which case a second click would take away the clock and check mark, or else you can change the item to read "Hide Clock" while the clock is visible. A third scheme maintains two menu items, "Show Clock" and "Hide Clock." When the clock is visible you disable "Show Clock," causing it to display in gray and be unselectable. When the clock is hidden again, you enable "Show Clock" again and disable "Hide Clock." All three methods accomplish the same goal, to show the operator what is available and prevent improper selection.

This example shows how to manipulate menu items all three ways. Individual items are disabled. An entire menu is disabled and enabled as well. As you will see, the menu is disabled independently from the

individual items. When the entire menu is turned back on, some items remain disabled, as they were before the whole menu was inhibited. The example also shows how to change the text and mark menu items. You can mark an item with any character just by passing the character's ASCII value. The system font has some special characters intended for menu item marks. They are defined in the ToolEqu include file and also listed below.

```
                                        ;These chars are pre-defined in ToolEqu.D
commandMark EQU     $11                 ;Command "cloverleaf"
checkMark   EQU     $12                 ;Check mark
diamondMark EQU     $13                 ;Diamond
appleMark   EQU     $14                 ;Desk accessory menu title
noMark      EQU     0                   ;Use for removing any mark

                                        ;Get menu handle to use in all routines
            CLR.L   -(SP)               ;Space for handle
            MOVE.W  #301,-(SP)          ;Menu ID from resource file
            _GetMHandle
            MOVE.L  (SP)+,MenuHand(A5)  ;Menu handle

                                        ;Mark the first menu item with an apple
                                        ;character
            MOVE.L  MenuHand(A5),-(SP)  ;Menu handle
            MOVE.W  #1,-(SP)            ;First menu item
            MOVE.W  #AppleMark,-(SP)
            _SetItmMark                 ;Make your mark

                                        ;Disable the second menu item
            MOVE.L  MenuHand(A5),-(SP)  ;Menu handle
            MOVE.W  #2,-(SP)            ;Second item in list
            _DisableItem

                                        ;Enable the third menu item
            MOVE.L  MenuHand(A5),-(SP)  ;Menu handle
            MOVE.W  #3,-(SP)            ;Third item of menu
            _EnableItem

                                        ;Disable the entire menu
            MOVE.L  MenuHand(A5),-(SP)  ;Menu handle
            CLR.W   -(SP)               ;Zero means entire menu
            _DisableItem

                                        ;Re-enable the menu again. Item two
                                        ;stays disabled
            MOVE.L  MenuHand(A5),-(SP)  ;Menu handle
            CLR.W   -(SP)               ;Entire menu
            _EnableItem
```

```
                                    ;Change the text of item four
MOVE.L    MenuHand(A5),-(SP)        ;Menu handle
MOVE.W    #4,-(SP)                  ;Fourth menu item
PEA       'New Item Four'           ;Text to display
_SetItem
```

Drawing Text in Gray, as the Menu Manager Does

On occasion you may want to draw text in a shade other than black. This is usually only legible with large font sizes. The Menu Manager uses gray text to indicate items that are not selectable. Here the difficulty in reading actually helps to enforce the idea that the gray items cannot be chosen. The QuickDraw package does not provide any direct way of making gray text. But if you would like to know how to do it you can study the example here.

The method for drawing gray is simple. A gray mask is first XOR'ed over the area where the characters are to go. Then the text is drawn in Bit Clear mode. This erases the mask wherever the letters are solid. Finally the gray mask is XOR'ed over the area again. The two exclusive-or operations cancel each other, leaving the text in gray.

```
                                    ;Draw text in gray
_PenNormal                          ;Normal Width, Mode, Black Color
PEA TextBox                         ;pointer to surrounding rectangle
_FRAMERECT                          ;Frame box in black for visibility
MOVE.W    #patXor,-(SP)             ;XOR pen mode, number 10
_PenMode
                                    ;Bit Clear text mode, number 3
MOVE.W    #srcBiC,-(SP)
_TextMode
MOVE.L    GRAFGLOBALS(A5),A0        ;Pointer to QD globals
PEA       GRAY(A0)                  ;Standard gray pattern
_PENPAT
MOVE.W    #48,-(SP)                 ;Gray characters look better large
_TextSize

PEA       TextBox                   ;Draw gray mask
_PaintRect
MOVE.L    TextStart,-(SP)
_MoveTo
PEA       'GRAY'                    ;Clear text bits
_DrawString
PEA       TextBox                   ;Erase excess mask
_PaintRect
_PenNormal                          ;Normal Width,Mode,Black Color
RTS
```

```
TextStart   DC      150
            DC      200
TextBox     DC      100
            DC      150
            DC      180
            DC      350
```

Coding for the Undo Command

Standard edit operations are Cut, Copy, Paste, Clear and Undo. The last operation, Undo, should reverse whatever operation was done before. It is usually adequate to make only the most recent operation reversible. In a good Macintosh application the operator can see the effect of a mistake immediately, and select Undo before doing anything else. Repeated calls to Undo can alternately reverse the operation and then perform it again, in case selecting Undo was a mistake. This example shows how to maintain data about one previous operation in order to reverse it.

```
                                    ;The Standard Edit items, Cut, Copy,
                                    ;Paste, Clear and UnDo
                                    ;Your menu should use this order to be
                                    ;compatible with the Desk Accessories
                                    ;
UndoCmd     EQU     1               ;UNDO/Z
                                    ;
CutCmd      EQU     3               ;CUT /X
CopyCmd     EQU     4               ;COPY /C
PasteCmd    EQU     5               ;PASTE/V
ClearCmd    EQU     6               ;CLEAR
                                    ;
                                    ;Data Areas. Initialize these to 0 when
                                    ;program starts
                                    ;Clipboard area
                                    ;The size depends on the type of data
                                    ;the program uses
Clip        DS.L    1
                                    ;Undo recovery area
                                    ;This should be the same size as the
                                    ;clipboard
Save        DS.L    1
SelectPtr   DS.L    1               ;Pointer to selected item
SaveCmd     DS.W    1               ;Last menu item to undo
SaveSelect  DS.L    1               ;Selected item to undo

                                    ;Enter with Menu item in edit menu
                                    ;in D0
```

```
DoCmd   MOVE.L   Select Ttr(A5),A0   ;Point to selected item
        CMPI     #UnDoCmd,D0         ;Check for Undo First
        BEQ      Undo
        MOVE.L   A0,SaveSelect(A5)   ;Remember what was selected to undo it
        MOVE.W   D0,SaveCmd(A5)      ;Remember what was done to undo it
        CMPI     #CutCmd,D0
        BEQ      Cut
        CMPI     #CopyCmd,D0
        BEQ      Copy
        CMPI     #PasteCmd,D0
        BEQ      Paste
        CMPI     #ClearCmd,D0
        BEQ      Clear
                                     ;Item not on list. Impossible!
        STOP     #$2000              ;No privilege violation

                                     ;Operator can select these Edit Menu
                                     ;items
Cut                                  ;CUT / X
        MOVE.L   Clip(A5),Save(A5)   ;Save=ClipBoard
        MOVE.L   (A0),Clip(A5)       ;ClipBoard=Selection
        CLR.L    (A0)                ;Selection=0
        RTS
Copy                                 ;COPY/C
        MOVE.L   Clip(A5),Save(A5)   ;Save= ClipBoard
        MOVE.L   (A0),Clip(A5)       ;ClipBoard=Selection
        RTS
Paste                                ;PASTE/V
        MOVE.L   (A0),Save(A5)       ;Save=Selection
        MOVE.L   Clip(A5),(A0)       ;Selection=ClipBoard
        RTS
Clear                                ;CLEAR/Z
        MOVE.L   (A0),Save(A5)       ;Save=Selection
        CLR.L    (A0)                ;Selection=0
        RTS
Undo                                 ;UNDO
        MOVE.W   SaveCmd(A5),D0      ;Remember what was done last & Undo it
        MOVE.L   SaveSelect(A5),A0   ;Remember to which item it was done
        MOVE.L   A0,Select(A5)       ;Restore old selection
        BCHG     #15,D0
        MOVE.W   D0,SaveCmd(A5)      ;Mark savedcommand as undone
        BPL      DoCmd               ;Undoing an Undo? Just do it again!
```

```
            BCLR        #15,D0
            CMPI        #CutCmd,D0
            BEQ         UnCut
            CMPI        #CopyCmd,D0
            BEQ         UnCopy
            CMPI        #PasteCmd,D0
            BEQ         UnPaste
            CMPI        #ClearCmd,D0
            BEQ         UnClear
                                        ;There could be nothing to undo right
                                        ;after initialization
            CLR.W       SaveCmd(A5)
            RTS
UnCut
            MOVE.L      Clip(A5),(A0)   ;Selection=ClipBoard
            MOVE.L      Save(A5),Clip(A5)   ;ClipBoard=Save
            RTS
UnCopy
            MOVE.L      Save(A5),Clip(A5)   ;ClipBoard=Save
            RTS
UnPaste
            MOVE.L      Save(A6),(A0)   ;Selection=Save
            RTS
UnClear
            MOVE.L      Save(A6),(A0)   ;Selection=Save
            RTS
```

Double Precision Division

A 68000 processor can add and subtract up to 32 bits at a time. Multiplication and division are limited to 16-bit input and output respectively. A special carry flag is provided for adding and subtracting large numbers by stages, using the ADDX and SUBX instructions, as described in Chapter 3, the 68000 Instruction Set. Multiplication is fairly straightforward, but division which yields an answer greater than 16 bits is a little more tricky, so it is presented here.

This example calculates the time elapsed since the Macintosh was turned on. The Macintosh maintains the time in units called "ticks" which are 1/60 second. This routine divides the current time in ticks by 60 to obtain the number of seconds. Then it divides the latter figure by 60 again to arrive at the number of minutes plus the number of seconds as a remainder. The result of the first division will be more than two bytes long after about 18 hours.

```
                                        ;Get time since power-up in minutes and
                                        ;seconds
                                        ;Return minutes in D0 and seconds in D1

                                        ;Get tick count. 60 ticks/second
        CLR.L       -(SP)
        __TickCount

                                        ;Divide by 60 to get total seconds
        MOVEQ       #60,D2              ;Divisor=60
        BSR         Divide

                                        ;Divide again get minutes and seconds
        MOVE.L      D0,-(SP)
        BSR         Divide
        RTS                             ;All done

Divide
                                        ;Divide the 32 bit integer on the stack by
                                        ;the 16 bits in D2
                                        ;Return a four-byte quotient in D0 and a
                                        ;two-byte remainder in D1
        MOVE.L      (SP)+,A0
        CLR.L       D0                  ;Division always uses 4 bytes
        MOVE.W      (SP)+,D0            ;Get upper word
        DIVU        D2,D0               ;Divide upper word
        MOVE.L      D0,D1               ;Save remainder for next division
        SWAP        D0                  ;Save quotient for upper result
        MOVE.W      (SP)+,D1            ;Get lower word
        DIVU        D2,D1               ;Divide upper remainder and lower word
        MOVE.W      D1,D0               ;Result into D0.L
        SWAP        D1                  ;Remainder into D1.W
        JMP         (A0)
```

Using the Print Package

The Macintosh provides a very advanced printer driver. Other small computers may have a few BIOS calls that perform the equivalent of a BASIC "PRINT" statement, but the Macintosh Print Manager comes complete, with graphics, spooling and application-program independence of the printer, paper size, and type.

Before you start to print, you call two dialogs provided by the system. One dialog, called the Style dialog, gets all the information about the printer and paper. In the print-record field, it saves the operator's answers along with data specific to the printer. You should call this dialog before printing the first time. The Job dialog lets the operator choose the number of copies, select the pages to be printed, and choose the print quality

where this is an option, as it is on the ImageWriter printer. If you use the Job dialog, you should call it every time a document is printed.

When the print record has been completed, you open a special graphic port for the printer. Then you simply draw the data you want to go on the paper, just as though you are drawing to the screen! If the operator selected spool printing, the data will go to a disk file. For a draft printing operation, the data goes directly to the printer. When you are all done, you close the printer port and further drawing goes back to the top window. If you have a spool file, you can call the Print Manager to print it, or just leave the data on the disk. The operator can print it later from the finder.

This example shows how to print multiple pages of text and graphics. The operator can choose to spool or draft the output. If the document is spooled, the routine detects it and prints the spool file immediately. This is only a small sample of what you can do with the Print Manager. If you would like to learn more about the Print Manager, you can find it in *Macintosh Revealed,* or *Inside Macintosh*.

```
                                ;Printing Text and Graphics
                                ;Set up the printer manager
LINK        #-iPrintSize,A6     ;Make space for the print record
LEA         -iPrintSize(A6),A6  ;Point to start of record
LINK        #0,A6               ;Make a handle to print record
MOVE.L      A4,-(SP)            ;Save register A4
CLR.L       -(SP)
PEA         (SP)
__GetPort                       ;Save the port
JSR         PrOpen
                                ;Put up a style dialog to learn the type
                                ;of the printer and paper
MOVE.L      A6,-(SP)            ;Print record handle
JSR         PrintDefault        ;Initialize record
CLR.W       -(SP)               ;Result
MOVE.L      A6,-(SP)
JSR         PrStlDialog         ;Handle->Result
BTST        #0,(SP)+            ;Did operator click Cancel?
BEQ         @86
                                ;Put up a job dialog to learn page range,
                                ;quality and number of copies
CLR.W       -(SP)               ;Result
MOVE.L      A6,-(SP)
JSR         PrJobDialog         ;Handle->Result
BTST        #0,(SP)+            ;Did operator click Cancel?
BEQ         @86
                                ;Start a document to print into
CLR.L       -(SP)               ;Printing Port Result
```

```
MOVE.L      A6,-(SP)          ;Handle
CLR.L       -(SP)             ;System allocates port space
CLR.L       -(SP)             ;System allocates print buffer space
JSR         PrOpenDoc         ;Handle,^port,^print buffer->^port
MOVE.L      (SP)+,AU          ;Pointer to printing port
                              ;Print on the first page
BSR         @10               ;Begin the first page
MOVE.L      #$00200020,-(SP)  ;Coordinates 32,32
__MoveTo                      ;Move pen
PEA         'This is page 1'  ;Text to print
__DrawString                  ;Goes into printer port bit map
                              ;Add more drawing commands here!
NOP
;
BSR         @20               ;End the first page
                              ;Print on second page
BSR         @10               ;Begin the second page
MOVE.L      #$00200020,-(SP)  ;Coordinates 32,32
__MoveTo                      ;Move pen
MOVE.L      #$00800080,-(SP)  ;Coordinates 128,128
__LineTo                      ;Goes into printer port bit map
                              ;Add more drawing commands here!
NOP
;
BSR         @20               ;End the second page
                              ;End of the document
MOVE.L      A4,-(SP)          ;^port
JSR         PrCloseDoc        ;^port
                              ;Print spool file if doing spool printing.
                              ;If draft printing just exit now
MOVE.L      (A6),A4           ;Get pointer to print record
LEA         PrJob(A4),A4      ;Point to job sub-record
LEA         bjDocLoop(A4),A4  ;Point to printing method
MOVE.B      (A4),D0           ;Get printing method code
CMPI.B      #bSpoolLoop,D0    ;Spool printing if 1
BNE         @86               ;Draft printing. Already done
                              ;Open and print spool file
LINK        #-iPrStatSize,A4  ;Make space for status record
LEA         -PrStatSize(A4),A4 ;Point to start of record
MOVE.L      A6,-(SP)          ;Handle to print record
CLR.L  -(SP)                  ;System allocates NEW printing port
CLR.L       -(SP)             ;System allocates file buffer
CLR.L  -(SP)                  ;System allocates printer buffer
MOVE.L      A4,-(SP)          ;^status record
```

```
        JSR         PrPicFile       ;handle,^port,^file buf,^print buf,^status
                                    ;Restore the stack and regsiter A4
        LEA         iPrStatSize(A4),A4
        UNLK        A4
        BRA         @86             ;All done
                                    ;Begin a page of the document
@10     MOVE.L      A4,-(SP)        ;^port
        CLR.L -(SP)                 ;No scaling rectangle. Print normal size
        JSR         PrOpenPage      ;^port,^scale rect.
        RTS
                                    ;End a page of the document
@20     MOVE.L      A4,-(SP)        ;^port
        JSR         PrClosePage     ;^port.
        RTS
                                    ;All done printing. Return to the main
                                    ;program
@86     JSR         PrClose         ;Close the print driver
        _SetPort
        MOVE.L      (SP)+,A4        ;Restore A4
                                    ;Restore the stack and register A6
        UNLK        A6
        LEA         iPrintSize(A6),A6
        UNLK        A6
        RTS
```

Using the SANE Numeric Package

Apple Computer Inc. has a standard floating point package for all their machines. Called the Standard Apple Numeric Environment, or SANE, the package provides consistent floating-point arithmetic on all Apple machines, running in the most popular languages. The SANE package is based on a standard from the IEEE, the Institute of Electrical and Electronics Engineers.

The package handles numbers in six different precisions. Three of these types, single precision, double precision, and extended precision, are floating point numbers of different lengths. They contain 32, 64 and 80 bits respectively. The fourth type, dubbed "comp," is an eight-byte integer, useful for accounting purposes. The other two types are two-byte and four-byte integers. All arithmetic is performed in extended precision. The results are only stored in the other types. The examples below show how to perform calculations and convert between the various types.

The SANE routines require SANEMacs.txt to be included in the assembly—this package holds the macro definitions for the SANE operations. These macros expand to two instructions. The first instruction pushes a constant "opcode" onto the stack. The opcode tells SANE which

operation to perform. The second instruction is another macro, JSRFP, which in turn expands to a trap which calls SANE.

Converting Numeric to String

The SANE package itself will not convert a number directly to a string representation. The numeric formats required by high-level languages vary so much that it would be nearly impossible for one routine to support all of them. Instead, SANE converts between binary numbers and an exponential, decimal form, called a "decimal record." Then the higher-level language converts between the decimal record and the actual string representation required.

A decimal record contains a sign, an exponent and a mantissa, the significant digits. The decimal record below represents the number *pi*. The sign is one byte. It is either zero for positive or one for negative. The exponent is a two-byte binary integer. It contains the power of ten. The mantissa is presented as a Pascal string. The first byte is the length. The decimal digits follow, in ASCII, with no decimal point, commas or other formatting. You can think of the mantissa as one large integer. Multiply that integer by ten to the power of the exponent to get the final number.

```
DecStr
;This decimal record contains pi
;Sign byte. 1=negative. 0=positive
          DC.B          0
;Fill
          DC.B          0                    ;not used
;Exponent
          DC.W          -6
;Number of decimal digits
          DC.B          7
;Decimal digits
          DC.B          '3141593'
```

This routine will convert the decimal record above to an extended precision number:

```
DecimalToExtended
;This routine converts a decimal record to an extended precision number
;Input:    DecStr=decimal record
;Output:   Accumulator=extended precision number
;Push the address of the input decimal record
          PEA           DecStr               ;^dec rec
;Push address of output extended precision number
          PEA           Accumulator          ;^extended
Convert the decimal record to a number
          FDEC2X
```

Here Accumulator is a 10-byte field which holds the result. When making decimal records for conversion, you have to be sure that the first digit is not a zero. When SANE finds a leading zero in the decimal record, it evaluates the decimal record as zero.

To convert a binary representation to a decimal record efficiently, SANE has to know how many digits of the result will actually be used. When you call the conversion routine you pass a format record. This record tells whether the result will be in a fixed-point or floating-point format, and how many digits of accuracy are needed. For a fixed-point format, the number of digits to the right of the decimal point is given. For a floating-point format, the total number of digits is passed. The format record is not needed to convert from a decimal record to a binary representation, because the number of digits is contained in the decimal record. Here is a format record specifying 7 digits of accuracy for a floating-point number.

```
;Description of format for conversion
FormRec
;conversion style. 0=floating point, 1=fixed point
                DC.B    0
                DC.B    0   ;not used
;number of significant digits
                DC.W    7   ;7 significant digits
```

This routine converts a four-byte, single precision number for output. Only the mantissa is converted to a string. The exponent is returned, in binary, in register D1. The exponent will be zero using the data in this example. If you use this routine to display a number in a window, you will probably want to show the exponent as a superscript. To display the number, first draw the mantissa on the screen. Then draw "X 10," move the pen up and draw the exponent. You can convert the exponent with the "Integer to String" routine, DrawValue, shown in Chapter 9. The routine below returns a pointer to the mantissa in A0, as well as the exponent in D1. The mantissa string lies on top of the input decimal record. Trailing zeros have been removed. In case the input number is zero, the routine forces the exponent to zero, because the exponent returned by SANE is undefined in that case.

```
SingleToString
;This routine converts a single precision number to ASCII string
;for display.
;Input:     Single=four-byte single precision number
;Output:    (A0) =normalized mantissa in string form
;           D1 =exponent in binary
```

```
                                        ;
                                        ;push address of format description
        PEA         FormRec             ;^formrec=19 digits
                                        ;push address of input single precision
                                        ;number
        PEA         Single              ;^single
                                        ;push the address of the output decimal
                                        ;record
        PEA         DecStr              ;^dec rec
                                        ;Convert the number to a decimal record
        FS2DEC
                                        ;Set the sign
        LEA         DecStr,A0
        MOVEQ       # '+',D2             ;assume positive sign
                                        ;Check the sign byte of the result.
                                        ;0=positive. 1=negative
        TST.B       (A0)
        BEQ         @10
        MOVEQ       # '-',D2             ;negative sign
@10     MOVE.L      (A0)+,D1            ;get exponent
                                        ;get length
        MOVE.B      (A0),D0             ;count
                                        ;adjust the exponent for the number of
                                        ;significant digits
        ADD         D0,D1
        SUBQ        #1,D1
                                        ;strip trailing zeroes
@20     CMPI.B      #$30,0(A0,D0)
        BNE         @30
        SUBQ        #1,D0               ;subtract one from length
        BNE         @20
                                        ;SANE returned zero for a result. The
                                        ;exponent may be undefined
        MOVEQ       #1,D0               ;use one digit of zero
        CLR         D1
                                        ;insert the decimal point
@30     MOVE.B      1(A0),A0
        MOVE.B      # '.',1(A0)         ;adjust start for sign and decimal point
        MOVE.B      D2,-(A0)
        ADDQ        #2,D0
                                        ;place count into string
```

```
        MOVE.B      D0,-(A0)        ;length
                                    ;all done. Return A0 and D1
        RTS
                                    ;Data areas
                                    ;Description of format for conversion
                                    ;FormRec
                                    ;conversion style. 0=floating point,
                                    ;1=fixed point
        DC.B        0
        DC.B        0               ;not used
                                    ;number of significant digits
        DC.W        7               ;7 significant digits
                                    ;Single precision number
Single  DC.L        $40490FDB       ;3.141593
                                    ;Decimal record
DecStr
        DC.B        0               ;Sign byte. 1=negative. 0=positive
        DC.B        0               ;Fill
        DC.W        -6              ;Exponent
        DC.B        7               ;Number of decimal digits
        DC.B        '3141593'       ;Decimal digits
```

Converting Between SANE Types

The SANE package provides routines to convert between each of the numeric types and the extended type. To convert between two types you must first convert to extended. To perform calculations you may need to convert some of the numbers to extended precision first.

Converting from the extended type to a less precise type may require rounding the result. SANE lets you control the method of rounding. The rounding mode is set by the SANE "environment word." Bits 14 and 13 in this word tell how to round a number when it is necessary. The other bits have meanings too, so be careful not to change them. The environment word is described fully in the SANE chapter of *Inside Macintosh*. The rounding modes are shown below, along with the results of rounding some numbers to integers. "Round-to-nearest" is the default. It gives the greatest accuracy in calculations.

Rounding mode	Environment word value	Rounded values of 1.6	1.2	−1.2	−1.6
Round to nearest	0	2	1	−1	−2
Round toward zero	$6000	1	1	−1	−1
Round upward	$2000	2	2	−1	−1
Round downward	$4000	1	1	−2	−2

The routine below converts a double precision number to extended precision, single precision and an integer. All of the SANE conversion routines use address pointers for operands. Even the two-byte integer is passed as a pointer. The environment word is set once. It applies to all of the conversions which take place until it is set again.

```
ConvertNumbers
;This routine converts a double precision number to three types,
;               an integer, an extended and a single precision number.
;Input:         Double    =double precision number
;Output:        Single    =single precision number
;               Integer   =two-byte integer
;               Extended  =extended precision number
                                        ;Get the current environment word
                LEA       Envword,A4
                PEA       (A4)
                FGetEnv
                                        ;Change the rounding mode to "round to
                                        ;nearest"
                MOVE.W    (A4),D0
                ANDI.W    #$AFFF,D0     ;Clear current mode
                MOVE.W    D0,(A4)
                                        ;Set the new environment word
                PEA       (A4)
                FSetEnv
                                        ;Convert a double precision number to an
                                        ;extended number
                PEA       Double,-(SP)
                PEA       Accumulator
                FD2X                    ;call SANE routine to convert
                                        ;Convert the extended number back
                                        ;to double precision
                PEA       Accumulator
                PEA       Double,-(SP)
                FX2D                    ;call SANE routine to convert
                                        ;Convert extended to single precision
                PEA       Accumulator
                                        ;Push the address of the single precision
                                        ;number on the stack
                PEA       Single,-(SP)
                FX2S                    ;call SANE routine to convert
                                        ;Round an extended precision number to
                                        ;an integer
                PEA       Accumulator
                                        ;Push the address of the integer on the
                                        ;stack
```

```
                PEA     Integer,-(SP)
                FX2I                    ;call SANE routine to convert
                                        ;Data areas
Single          DS.B    4
Integer         DS.B    2
Double          DS.B    8
Accumulator     DS.B    10
Envword         DS.B    4
```

Using the SANE Floating Point Package for Arithmetic

The SANE package does all arithmetic in extended precision. You can, however, call binary operations with one parameter of lesser precision. Conversion is automatic. In this example we use one location for an extended precision number as a kind of accumulator. Single, double and extended precision numbers are combined with the accumulator to arrive at a final result. You can use the routines shown above to convert to a string for display or to a lesser precision. The equation evaluated in this example is:

Accumulator = (Single1*Extended+Double)/Single2

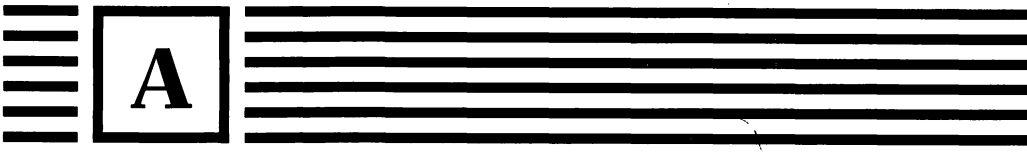
where Single, Double and Extended represent numbers of the implied precision and Accumulator is an extended precision number.

```
Equation
;This routine evaluates the equation:
;               Accumulator=(Single1*Extended+Double)/Single2

;Input:         Single1 and Single2 = single precision numbers
;               Double = double precision number
;               Extended = extended precision number
;Output:        Accumulator = extended precision number
                                        ;
                                        ;Move an 80-bit number called Extended
                                        ;into Accumulator
                PEA     Extended
                PEA     Accumulator
                FX2X                    ;Moves a 10 byte block
                                        ;Multiply Accumulator by a single
                                        ;precision number
                PEA     Single1
                PEA     Accumulator
                MULS
                                        ;Add Double to Accumulator
                PEA     Double
                PEA     Accumulator
```

```
                ADDD
                                        ;Divide Accumulator by a single precision
                                        ;number
                PEA     Single2
                PEA     Accumulator
                DIVS
                                        ;Exit with the result in the Accumulator
                RTS
                                        ;
                                        ;Data definitions
Accumulator     DS.B    10
Extended        DS.B    10
Double          DS.B    8
Single1         DS.B    4
Single2         DS.B    4
```

These routines are just a sample of what you can accomplish with the Macintosh. Using the sample program, SimpleCalc, as a skeleton, and what you already know, you can create powerful programs in the friendly style of the Macintosh. Now is the time for you to start programming on your own. We hope you have enjoyed our book. Good luck and happy programming!

APPENDIX A



The Binary and Hexadecimal Numbering Systems

Why couldn't the people who make hardware use the decimal system? Why do I have to learn binary and hexadecimal? Well, it turns out that when computers were first being created there were some abortive attempts to create ten way switches. It is much easier to deal with two way switches than ten way switches. Electricity with its positive and negative charges and magnetism with its two poles, north and south, are inherently binary. The universe down at the electromagnetic level is inherently dualistic, not "decimalistic." Therefore, any hardware based on this dualism is much easier to turn into a workable model than hardware based on any other number base.

Disk Drives store data as a series of magnetic fluxes pointed one of two ways. Computer memory stores data as a series of either charged or uncharged cells. Logic and arithmetic are formed by a series of two way switches. Data is communicated by pulses of electricity that are in one of two states. On the Macintosh screen every dot is either black or white. When a modem is used the data is transmitted down the phone line as one of two tones. Everything in today's computers is done in pairs.

The binary numbering system is a mathematical way of representing data based on pairs. In our normal way of counting we use ten digits—the numbers from 0 through 9. In the binary system we only use two digits—the numbers 0 and 1. Maybe you will find it amazing that any number, even very large numbers, can be turned into a string of zeroes and ones.

Actually, it is not that surprising. The way it is done exactly parallels the way in which you normally work with the decimal system. All you have to do is break out of your normal way of thinking in the decimal system and generalize a bit.

Let us analyze what happens when you work in decimal. In the decimal system the first place is the units place, the next place is the tens

place, the next is the hundreds place, etc. Or, to put it another way, the first is the units place, the next is the tens place, the next is the ten times ten place, the next is the ten times ten times ten place, etc.

When you count you start in the units place and count from 0 through 9 (one less than ten, our number base), then you bump the place to the left by one, and repeat counting from 0 through 9, then bump the place to the left again. Eventually you get to where the tens place itself reaches 9 so this time you bump the next place to the left, the hundreds (ten times ten) place. Perhaps you have never examined what you were doing in such excruciating detail before.

Now everywhere in the above two paragraphs where we used the number ten let us use the number two and you will be counting in binary! In the binary system the first place is the units place, the next place is the twos place, the next is the fours place, etc. Or, to put it another way, the first is the units place, the next is the twos place, the next is the two times two place, the next is the two times two times two place (or eights place), etc.

When you count in binary you start in the units place and count from 0 through 1 (one less than two, our number base), then you bump the place to the left by one, and repeat counting from 0 through 1, then bump the place to the left again. Eventually (and this happens a lot faster in binary) you get to where the twos place itself reaches 1 so this time you bump the next place to the left, the fours (2x2) place.

Getting to one in binary is like getting to nine in decimal. When you add one you have to bump the place to the left and turn this place to zero.

Think about the four paragraphs you have just read. Now look at the example of counting in binary below:

0 is zero.

1 is one (after this we must bump the next place, just like getting to 9 in decimal!).

10 is two (one in the two's place, zero in the units place).

11 is three (one in the two's place, one in the units place, 2 +1 = 3). This is like getting to 99 in decimal—get ready to bump the next place to the left and turn both these places to zero.

100 is four (one in the four's place).

101 is five (one in the four's place plus one in the one's place 4 + 1 = 5).

110 is six (one in the four's place plus one in the two's place 4 + 2 = 6).

111 is seven (one in the four, two, and units place 4 + 2 + 1 = 7). Get ready to turn all three places to zero and add one to the place to the left. This is just like 999 in decimal.

1000 is eight (one in the eight's place).

By now we hope you have the idea. If you want to test your mettle, try counting to 16 in binary.

Binary is the system that the actual computer hardware works in. Each binary digit is called a bit. So the binary number 1000 (8 in decimal) has 4 bits while the number 11 (3 in decimal) uses 2 bits. So if you wanted to have a piece of hardware that could store numbers between 0 and 7 you could use 3 on-off switches. When the switch is on that would be a 1, when off that would be a 0. Then all the switches on would represent a 7 (111) while all the switches off would represent a zero (000) —if only the middle switch were on that would be a 2 (010). Usually these "switches" are small microscopic transistors or capacitors that store charges inside a memory chip.

So if somebody asks you why do computers count in binary while we count in decimal just point to your hands and say, "we have ten fingers, computers only have two!"

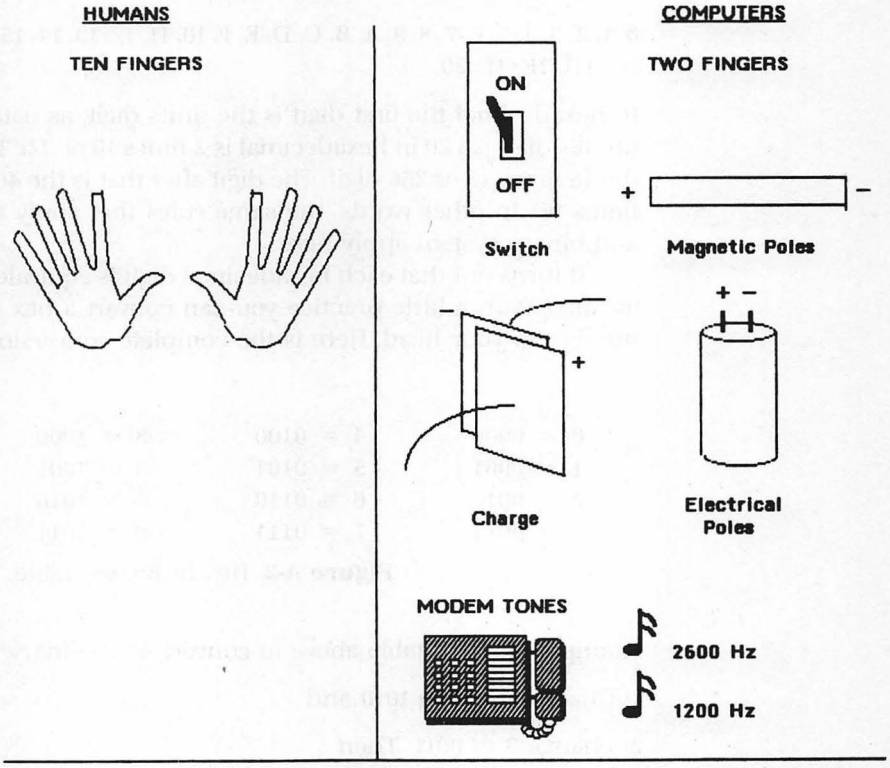


Figure A-1 Ten Fingers versus Two Fingers

The Hexadecimal Numbering System

When dealing with binary numbers of any real size, there is a problem. What is the difference between 01101010001110 and 01101010011110? It turns out that there is another numbering system called hexadecimal (or "hex" for short) with a base of 16. This is easy to remember since "hex" means six and "decimal" means 10 so hexadecimal is base 6 + 10 or 16. This numbering system is easier for people to use and at the same time it is very simple to translate into binary.

In the decimal system the digits are 0 through 9 (1 less than the base). In the hexadecimal system the digits run from 0 through a digit that represents 15 (1 less than 16). Since there are no digits for 10 through 15, people use the first letters of the alphabet. So you should know that A is a digit of 10, B is a digit of 11, C is a digit of 12,...and F is a digit of 15. Here is how we would count from 0 to 32 in hexadecimal.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20.

In hexadecimal the first digit is the units digit, as usual. The next digit is the 16s digit (so 20 in hexadecimal is 2 times 16 or 32). The digit after that is the 16 times 16 or 256 digit. The digit after that is the 4096 digit (16 times 16 times 16). In other words, the same rules that apply to both the decimal and binary system apply here.

It turns out that each hexadecimal digit is equivalent to 4 binary digits (or bits). With a little practice you can convert a hex number to a binary number in your head. Here is the complete conversion table:

0 = 0000	4 = 0100	8 = 1000	C = 1100
1 = 0001	5 = 0101	9 = 1001	D = 1101
2 = 0010	6 = 0110	A = 1010	E = 1110
3 = 0011	7 = 0111	B = 1011	F = 1111

Figure A-2 Hex to Binary Table

Example: Use the table above to convert A3 to binary.

1) Change the A to 1010 and

2) change 3 to 0011. Then

3) string the two numbers together making 10100011 (which equals A3).

The number 2F3C would be 0010, 1111, 0011, and 1100 strung together. So 2F3C would equal 0010111100111100.

Nearly everything we do in 68000 assembly language, the assembly language of the Macintosh, will be done with hexadecimal notation. Oftentimes, people place a dollar sign in front of a number to tell you that it's hexadecimal. So in the assembler $3F would mean the hexadecimal number 3F. You may come across the notation 3FH. This also means "the hexadecimal number 3F."

What the Data in Memory Looks Like

Data is arranged in memory 8 bits at a time. Each set of 8 bits is called a *byte*. Since four bits is equivalent to one hexadecimal digit, two hex digits make up one byte. Some people call half a byte a *nybble* (sometimes spelled "nibble"). Obviously, a nybble is just 4 bits or 1 hex digit.

One byte can hold a number from 0 through 255, or 256 possible values. Since there are only 26 upper case letters, 26 lower case letters, 10 digits, and a few assorted special characters such as "?" or "/" we can assign each one of these letters, numbers, etc. a value from 0 to 255 and have plenty of numbers left over.

A typical series of bytes will look like this:

10011010, 00010110, 10101110 in binary or

9A, 16, AE in hexadecimal.

So to be more specific, characters are arranged in memory as bytes. So every time you type a letter into your word procesor you should know that each letter, number, and even space that you see on the screen is represented inside the computer as a single byte.

For example, the words "Hello there!" would look like this inside the computer:

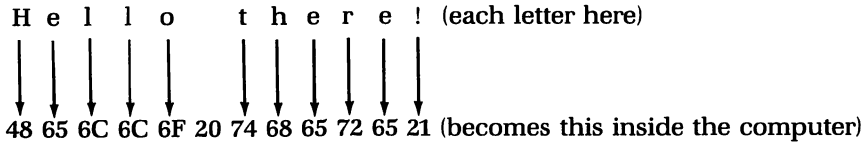


Figure A-3 Letter/ASCII Conversion

When you see advertisements saying that a computer has 128K of memory you should know that means that it has 1024 times 128 bytes of memory. Oftentimes the advertisements will tell the reader that means there are 128 thousand "characters" of memory in the machine.

Fortunately, the way in which characters are encoded in a given byte are not arbitrary—there are standards. The primary standard is that of the American National Standards Institute, abbreviated as the ANSI standard. All microcomputers use the ANSI standard, with some modifications.

Other objects besides characters are represented by bytes of memory. If you have worked with integers in a higher level language you will notice that the integers usually go from −32,768 to +32,767. This is because integers are almost universally represented as 2 bytes of data—2 bytes of data comprise 64 times 1024 possible values (for a total of 65536 possible values). Or, put another way, there are 256 possible values for each byte so two bytes represent 256 times 256 possible values, which is 65536.

How are negative integers represented? It turns out that there is a very elegant way to represent numbers so that negative numbers can be represented by the larger numbers within the range of possible numbers. This is called two's complement notation. When you add such a negative number to another negative number or to a positive number the result comes out correct. By using this little trick, computers never need to have a subtraction circuit, only an addition circuit.

Let's say you had a number that was positive which you wanted to make negative, using two's complement notation. You would "flip the bits" which means changing all bits that are 1 to 0 and all bits that are 0 to 1. Then you would add 1 to the number and ignore the carry.

Look at how you would turn ten to negative ten. Ten is 000A in hexadecimal. In binary this would be:

```
0000000000001010
1111111111110101      which becomes when you flip the bits
              +1
----------------
1111111111110110   = FFF6 which is negative ten in two's complement
                     notation.
```

Another way of finding the two's complement is to subtract from one greater than the maximum number you can represent within the number of bytes you are using.

```
10000
-  A
----
FFF6  = the two's complement of A
```

Now, suppose you wanted to make the following addition in decimal:

```
  15
-10
---
   5
```

In hexadecimal this becomes:

```
  000F
+ FFF6   you add a minus 10 (in two's complement form) rather than
  0005   subtracting 10!
```

Note that the carry of 1 is ignored and everything works out just the way we would expect.

Now for another example:

In decimal:

```
  35
 -44
  -9
```

In hexadecimal:

```
0023  (this is the hexadecimal equivalent of decimal 35)
FFD4  (add a 2's complement 44 which is equivalent to subtracting 44)
FFF7  which is the two's complement of 9, hence this number is -9.
```

Please take some time to go over these examples in detail before going on. Try out some of your own examples as well.

Although they will never tell you this in grade school, this same trick works in decimal. For example, to add 123 to −17 do the following:

```
  123
+ 983  = 1000 - 17
 1106  = 106 if you ignore the carry into the thousands digit
```

Another way of turning 17 into 983 is to subtract each digit from 9 (the so-called 9's complement) and then add 1 to the result. This works because another way of saying 1000 minus a number is 1 plus 999 minus a number or 999 minus a number plus 1. By using this method you would never need to subtract in decimal again (except when subtracting single digits from 9). For those of you who hated subtraction in grade school, you now have a way out. Again, try a few more examples using a calculator in decimal. After you are done, think about the hexadecimal version of this way of subtracting once again. It is vital that you understand hexadecimal and binary arithmetic before going on to learning assembly language.

Getting back to integers represented as two bytes in higher level language let us see what happens when two integers are added in BASIC or Pascal, say:

```
X = 3 - 1
```

Internal to the machine the 3 is turned into 0003 and the −1 becomes FFFF in two's complement. Adding 0003 to FFFF and ignoring the carry the result is 0002. Then the two bytes of 00 and 02 are moved into X. When the user wants to see what is in X the BASIC or Pascal does a fast translation from hex to decimal and informs the user that there is 2 in X. Part of the fascination of learning assembly language is that you start to understand what is really happening deep inside the machine.

APPENDIX

B

Instruction Format & Cycle Timing

This appendix contains a table of every possible format of operands which a given 68000 instruction can use. The instructions are arranged in alphabetical order.

For each instruction, this table shows the number of cycles that an instruction with a given set of operands will take to perform an operation on a byte (.B suffix, 1 byte), word (.W suffix, 2 bytes), or long word (.L suffix, 4 bytes). A byte or word instruction takes the same number of cycles; this is in the first series of columns. Long words usually take more cycles than bytes or words; long word timings are in the second series of columns.

For a byte/word or long the number of reads and writes are given. Each read or write takes 4 cycles of time.

D*m* or D*n* mean data registers D0 through D7. Example: D4.
A*m* or A*n* mean address registers A0 through A7. Example: A2.
D*n*/A*n* means either a data register or an address register may be used. Examples: A3,D7.
#Imm is immediate data. Example: #24

Here are some examples from the table. An ADD D*m*,d(A*n*,D*n*/A*n*) means that the possible formats for an ADD include:

```
ADD.B  D3,10(A2,D4)
ADD.L  D0,7(A5,A2)
ADD.W  D4,-3(A0,D0)
```

There are hundreds of possible combinations for this one format alone. This format will take 18 cycles for the byte and word form (three reads and one write) and 26 cycles for the long form (four reads and two writes).

To know how long an instruction will take multiply the number of cycles by the time per cycle. On the Macintosh, a cycle is ⅛ of a microsecond (160 nanoseconds), approximately. So 18 cycles takes about 3 microseconds; 26 cycles takes about 4⅓ microseconds.

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
ABCD	Dm,Dn	06	1	0			
ABCD	-(Am),-(An)	18	3	0			
ADD	Dm,Dn	04	1	0	08	1	0
ADD	Am,Dn	04	1	0	08	1	0
ADD	(Am),Dn	08	2	0	14	3	0
ADD	(Am)+,Dn	08	2	0	14	3	0
ADD	-(Am),Dn	10	2	0	16	3	0
ADD	d(Am),Dn	12	3	0	18	4	0
ADD	d(Am,Dn/An),Dn	14	3	0	20	4	0
ADD	Abs.W,Dn	12	3	0	18	4	0
ADD	Abs.L,Dn	16	4	0	22	5	0
ADD	d(PC),Dn	12	3	0	18	4	0
ADD	d(PC,Dn/An),Dn	14	3	0	20	4	0
ADD	#Imm,Dn	08	2	0	14	3	0
ADD	Dm,(An)	12	2	1	20	3	2
ADD	Dm,(An)+	12	2	1	20	3	2
ADD	Dm,-(An)	14	2	1	22	3	2
ADD	Dm,d(An)	16	3	1	24	4	2
ADD	Dm,(An,Dn/An)	18	3	1	26	4	2
ADD	Dm,Abs.W	16	3	1	24	4	2
ADD	Dm,Abs.L	18	4	1	28	5	2
ADDA	Dm,An	08	1	0	08	1	0

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
ADDA	Am,An	08	1	0	08	1	0
ADDA	(Am),An	12	2	0	14	3	0
ADDA	(Am)+,An	12	2	0	14	3	0
ADDA	-(Am),An	14	2	0	16	3	0
ADDA	d(Am),An	16	3	0	18	4	0
ADDA	d(Am,Dn/An),An	18	3	0	20	4	0
ADDA	Abs.W,An	14	3	0	18	4	0
ADDA	Abs.L,An	20	4	0	22	5	0
ADDA	d(PC),An	14	3	0	18	4	0
ADDA	d(PC,Dn/An),An	18	3	0	20	4	0
ADDA	#Imm,An	12	2	0	14	3	0
ADDI	#Imm,Dn	08	2	0	16	3	0
ADDI	#Imm,(An)	16	3	1	28	5	2
ADDI	#Imm,(An)+	16	3	1	28	5	2
ADDI	#Imm,-(An)	18	3	1	30	5	2
ADDI	#Imm,d(An)	20	4	1	32	6	2
ADDI	#Imm,d(An,Dn/An)	22	4	1	34	6	2
ADDI	#Imm,Abs.W	20	4	1	32	6	2
ADDQ	#Imm,Dn	04	1	0	08	1	0
ADDQ	#Imm,An	08	1	0	08	1	0
ADDQ	#Imm,(An)	12	2	1	20	3	2
ADDQ	#Imm,(An)+	12	2	1	20	3	2
ADDQ	#Imm,-(An)	14	2	1	22	3	2
ADDQ	#Imm,d(An)	16	3	1	24	4	2
ADDQ	#Imm,d(An,Dn/An)	18	3	1	26	4	2
ADDQ	#Imm,Abs.W	16	3	1	24	4	2

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
ADDQ	#Imm,Abs.L	20	4	1	28	5	2
ADDX	Dm,Dn	04	1	0	08	1	0
ADDX	-(Am),-(An)	18	3	1	30	5	2
AND	Dm,Dn	04	1	0	08	1	0
AND	(Am),Dn	08	2	0	14	3	0
AND	(Am)+,Dn	08	2	0	14	3	0
AND	-(Am),Dn	10	2	0	16	3	0
AND	d(Am),Dn	12	3	0	18	4	0
AND	d(Am,Dn/An),Dn	14	3	0	20	4	0
AND	Abs.W,Dn	12	3	0	18	4	0
AND	Abs.L,Dn	16	4	0	22	5	0
AND	d(PC),Dn	12	3	0	18	4	0
AND	d(PC,Dn/An),Dn	14	3	0	20	4	0
AND	#Imm,Dn	08	2	0	14	3	0
AND	Dm,(An)	12	2	1	20	3	2
AND	Dm,(An)+	12	2	1	20	3	2
AND	Dm,-(An)	14	2	1	22	3	2
AND	Dm,d(An)	16	3	1	24	4	2
AND	Dm,d(Am,Dn/An)	18	3	1	26	4	2
AND	Dm,Abs.W	16	3	1	24	4	2
AND	Dm,Abs.L	20	4	1	28	5	2
ANDI	#Imm,Dn	08	2	0	16	3	0
ANDI	#Imm,(An)	16	3	1	28	5	2
ANDI	#Imm,(An)+	16	3	1	28	5	2
ANDI	#Imm,-(An)	18	3	1	30	5	2
ANDI	#Imm,d(An)	20	4	1	32	6	2

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
ANDI	#Imm,d(An,Dn/An)	22	4	1	34	6	2
ANDI	#Imm,Abs.W	20	4	1	32	6	2
ANDI	#Imm,Abs.L	24	5	1	36	7	2
ANDI	#Imm,CCR	20	3	0			
ANDI	#Imm,SR	20	3	0			
ASL	Dm,Dn	6+2n	n	0	8+2n	n	0
ASL	#Imm,Dn	6+2n	n	0	8+2n	n	0
ASL	(An)	12	2	1			
ASL	(An)+	12	2	1			
ASL	-(An)	14	2	1			
ASL	d(An)	16	3	1			
ASL	d(Am,Dn/An)	18	3	1			
ASL	Abs.W	16	3	1			
ASL	Abs.L	20	4	1			
ASR	Dm,Dn	6+2n	n	0	8+2n	n	0
ASR	#Imm,Dn	6+2n	n	0	8+2n	n	0
ASR	(An)	12	2	1			
ASR	(An)+	12	2	1			
ASR	-(An)	14	2	1			
ASR	d(An)	16	3	1			
ASR	d(Am,Dn/An)	18	3	1			
ASR	Abs.W	16	3	1			
ASR	Abs.L	20	4	1			
		Bcc byte			Bcc word		
Bcc	Label(branch not taken)	08	1	0	12	2	0
Bcc	Label(branch taken)	10	2	0	10	2	0

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
BCHG	Dm,Dn	08	1	0			
BCHG	Dm,(An)	12	2	1			
BCHG	Dm,(An)+	12	2	1			
BCHG	Dm,-(An)	14	2	1			
BCHG	Dm,d(An)	16	3	1			
BCHG	Dm,d(An,Dn/An)	18	3	1			
BCHG	Dm,Abs.W	16	3	1			
BCHG	Dm,Abs.L	20	4	1			
BCHG	#Imm,Dn	12	2	0			
BCHG	#Imm,(An)	16	3	1			
BCHG	#Imm,(An)+	16	3	1			
BCHG	#Imm,-(An)	18	3	1			
BCHG	#Imm,d(An)	20	4	1			
BCHG	#Imm,d(An,Dn/An)	22	4	1			
BCHG	#Imm,Abs.W	20	4	1			
BCHG	#Imm,Abs.L	24	5	1			
BCLR	Dm,Dn	10	1	0			
BCLR	Dm,(An)	12	2	1			
BCLR	Dm,(An)+	14	2	1			
BCLR	Dm,-(An)	14	2	1			
BCLR	Dm,d(An)	16	3	1			
BCLR	Dm,d(An,Dn/An)	18	3	1			
BCLR	Dm,Abs.W	16	3	1			
BCLR	Dm,Abs.L	20	4	1			
BCLR	#Imm,Dn	14	2	0			
BCLR	#Imm,(An)	16	3	1			

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
BCLR	#Imm,(An)+	16	3	1			
BCLR	#Imm,-(An)	18	3	1			
BCLR	#Imm,d(An)	20	4	1			
BCLR	#Imm,d(An,Dn/An)	22	4	1			
BCLR	#Imm,Abs.W	20	4	1			
BCLR	#Imm,Abs.L	24	5	1			
BRA	Label	10	2	0			
BSET	Dm,Dn	08	1	0			
BSET	Dm,(An)	12	2	1			
BSET	Dm,(An)+	12	2	1			
BSET	Dm,-(An)	14	2	1			
BSET	Dm,d(An)	16	3	1			
BSET	Dm,d(Am,Dn/An)	18	3	1			
BSET	Dm,Abs.W	16	3	1			
BSET	Dm,Abs.L	20	4	1			
BSET	#Imm,Dn	12	2	0			
BSET	#Imm,(An)	16	3	1			
BSET	#Imm,(An)+	16	3	1			
BSET	#Imm,-(An)	18	3	1			
BSET	#Imm,d(An)	20	4	1			
BSET	#Imm,d(Am,Dn/An)	22	4	1			
BSET	#Imm,Abs.W	20	4	1			
BSET	#Imm,Abs.L	24	5	1			
BSR	Label	18	2	2			
BTST	Dm,Dn	06	1	0			
BTST	Dm,(An)	08	2	0			

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
BTST	Dm,(An)+	08	2	0			
BTST	Dm,-(An)	10	2	0			
BTST	Dm,d(An)	12	3	0			
BTST	Dm,d(An,Dn/An)	14	3	0			
BTST	Dm,Abs.W	12	3	0			
BTST	Dm,Abs.L	16	4	0			
BTST	Dm,d(PC)	12	3	0			
BTST	Dm,d(PC,Dn/An)	14	3	0			
BTST	#Imm,Dn	10	2	0			
BTST	#Imm,(An)	12	3	0			
BTST	#Imm,(An)+	12	3	0			
BTST	#Imm,-(An)	14	3	0			
BTST	#Imm,d(An)	16	4	0			
BTST	#Imm,d(Am,Dn/An)	18	4	0			
BTST	#Imm,Abs.W	16	4	0			
BTST	#Imm,Abs.L	20	5	0			
BTST	#Imm,d(PC)	16	4	0			
BTST	#Imm,d(PC,Dn/An)	18	4	0			
		CHK not taken			CHK taken		
CHK	Dm,Dn	08	1	0	40	5	3
CHK	(Am),Dn	12	2	0	44	6	3
CHK	(Am)+,Dn	12	2	0	44	6	3
CHK	-(Am),Dn	14	2	0	46	6	3
CHK	d(Am),Dn	16	3	0	48	7	3
CHK	d(Am,Dn/An),Dn	18	3	0	50	7	3
CHK	Abs.W,Dn	16	3	0	50	7	3

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
CHK	Abs.L,Dn	20	4	0	52	8	3
CHK	d(PC),Dn	16	3	0	50	7	3
CHK	d(PC,Dn/An),Dn	18	3	0	50	7	3
CHK	#Imm,Dn	12	2	0	44	6	3
CLR	Dn	04	1	0	06	1	0
CLR	(An)	12	2	1	20	3	2
CLR	(An)+	12	2	1	20	3	2
CLR	-(An)	14	2	1	24	3	2
CLR	d(An)	16	3	1	26	4	2
CLR	d(An,Dn/An)	18	3	1	28	4	2
CLR	Abs.W	16	3	1	26	4	2
CLR	Abs.L	20	4	1	28	5	2
CMP	Dm,Dn	04	1	0	06	1	0
CMP	Am,Dn	04	1	0	06	1	0
CMP	(Am),Dn	08	2	0	14	3	0
CMP	(Am)+,Dn	08	2	0	14	3	0
CMP	-(Am),Dn	10	2	0	16	3	0
CMP	d(Am),Dn	12	3	0	18	4	0
CMP	d(Am,Dn/An),Dn	14	3	0	20	4	0
CMP	Abs.W,Dn	12	3	0	18	4	0
CMP	Abs.L,Dn	16	4	0	22	5	0
CMP	d(PC),Dn	12	3	0	18	4	0
CMP	d(PC,Dn/An),Dn	14	3	0	20	4	0
CMP	#Imm,Dn	08	2	0	14	3	0
CMPA	Dm,An	06	1	0	06	1	0

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write	
CMPA	Am,An	06	1	0	06	1	0	
CMPA	(Am),An	10	2	0	14	3	0	
CMPA	(Am)+,An	10	2	0	14	3	0	
CMPA	-(Am),An	12	2	0	16	3	0	
CMPA	d(Am),An	14	3	0	18	4	0	
CMPA	d(Am,Dn/An),An	16	3	0	20	4	0	
CMPA	Abs.W,An	14	3	0	18	4	0	
CMPA	Abs.L,An	18	4	0	22	5	0	
CMPA	d(PC),An	14	3	0	18	4	0	
CMPA	d(PC,Dn/An),An	16	3	0	20	4	0	
CMPA	#Imm,An	10	2	0	14	3	0	
CMPI	#Imm,Dn	08	2	0	14	3	0	
CMPI	#Imm,(An)	12	3	0	20	5	0	
CMPI	#Imm,(An)+	12	3	0	20	5	0	
CMPI	#Imm,-(An)	14	3	0	22	5	0	
CMPI	#Imm,d(An)	16	4	0	24	6	0	
CMPI	#Imm,d(An,Dn/An)	18	4	0	26	6	0	
CMPI	#Imm,Abs.W	16	4	0	24	6	0	
CMPI	#Imm,Abs.L	20	5	0	28	7	0	
CMPM	(Am)+,(An)+	12	3	0	20	5	0	
DBcc	Dn,Label	12	2	0	(cc true, branch not)			
DBcc	Dn,Label	14	3	0	(cc false, branch not)			
DBcc	Dn,Label	10	2	0	(cc false, branch)			
DIVS	Dm,Dn	158	1	0	(DIVS, DIVU are max			
DIVS	(Am),Dn	162	2	0	values)			
DIVS	(Am)+,Dn	162	2	0				

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
DIVS	-(Am),Dn	164	2	0			
DIVS	d(Am),Dn	166	3	0			
DIVS	d(Am,Dn/An),Dn	168	3	0			
DIVS	Abs.W,Dn	166	3	0			
DIVS	Abs.L,Dn	170	4	0			
DIVS	d(PC),Dn	166	3	0			
DIVS	d(PC,Dn/An),Dn	168	3	0			
DIVS	#Imm,Dn	162	2	0			
DIVU	Dm,Dn	140	1	0			
DIVU	(Am),Dn	144	2	0			
DIVU	(Am)+,Dn	144	2	0			
DIVU	-(Am),Dn	146	2	0			
DIVU	d(Am),Dn	148	3	0			
DIVU	d(Am,Dn/An),Dn	150	3	0			
DIVU	Abs.W,Dn	148	3	0			
DIVU	Abs.L,Dn	152	4	0			
DIVU	d(PC),Dn	148	3	0			
DIVU	d(PC,Dn/An),Dn	150	3	0			
DIVU	#Imm,Dn	144	2	0			
EOR	Dm,Dn	04	1	0	08	1	0
EOR	Dm,(An)	12	2	1	20	3	2
EOR	Dm,(An)+	12	2	1	20	3	2
EOR	Dm,-(An)	14	2	1	22	3	2
EOR	Dm,d(An)	18	3	1	24	4	2
EOR	Dm,d(An,Dn/An)	20	3	1	26	4	2
EOR	Dm,Abs.W	18	3	1	24	4	2

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
EOR	Dm,Abs.L	20	4	1	28	5	2
EORI	#Imm,Dn	08	2	0	16	3	0
EORI	#Imm,(An)	16	3	1	28	5	2
EORI	#Imm,(An)+	16	3	1	28	5	2
EORI	#Imm,-(An)	18	3	1	30	5	2
EORI	#Imm,d(An)	20	4	1	32	6	2
EORI	#Imm,d(An,Dn/An)	22	4	1	34	6	2
EORI	#Imm,Abs.W	20	4	1	32	6	2
EORI	#Imm,Abs.L	24	5	1	36	7	2
EORI	#Imm,CCR	20	3	0			
EORI	#Imm,SR	20	3	0			
EXG	DA,Dn/An	06	1	0			
EXT	Dn	04	1	0			
ILLEGAL		34	4	3			
JMP	(An)	08	2	0			
JMP	d(An)	10	2	0			
JMP	d(An,Dn/An)	14	2	0			
JMP	Abs.W	10	2	0			
JMP	Abs.L	12	3	0			
JMP	d(PC)	10	2	0			
JMP	d(PC,Dn/An)	14	3	0			
JSR	(An)	16	2	2			
JSR	d(An)	18	2	2			
JSR	d(An,Dn/An)	22	2	2			
JSR	Abs.W	18	2	2			
JSR	Abs.L	20	3	2			

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
JSR	d(PC)	18	2	2			
JSR	d(PC,Dn/An)	22	2	2			
LEA	(An)	04	1	0			
LEA	d(An)	08	2	0			
LEA	d(An,Dn/An)	12	2	0			
LEA	Abs.W	08	2	0			
LEA	Abs.L	12	3	0			
LEA	d(PC)	08	2	0			
LEA	d(PC,Dn/An)	12	2	0			
LINK	An,#displace	18	2	2			
LSL	Dm,Dn	6+2n	n	0	8+2n	n	0
LSL	#Imm,Dn	12	2	1			
LSL	(An)	12	2	1			
LSL	(An)+	12	2	1			
LSL	-(An)	14	2	1			
LSL	d(An)	16	3	1			
LSL	d(An,Dn/An)	18	3	1			
LSL	Abs.W	16	3	1			
LSL	Abs.L	20	4	1			
LSR	Dm,Dn	6+2n	n	0	8+2n	n	0
LSR	#Imm,Dn	12	2	1			
LSR	(An)	12	2	1			
LSR	(An)+	12	2	1			
LSR	-(An)	14	2	1			
LSR	d(An)	16	3	1			
LSR	d(An,Dn/An)	18	3	1			

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
LSR	Abs.W	16	3	1			
LSR	Abs.L	20	4	1			
MOVE	Dm,Dn	04	1	0	04	1	0
MOVE	Dm,(An)	08	1	1	12	1	2
MOVE	Dm,(An)+	08	1	1	12	1	2
MOVE	Dm,-(An)	08	1	1	14	1	2
MOVE	Dm,d(An)	12	2	1	16	2	2
MOVE	Dm,d(An,Dn/An)	14	2	1	18	2	2
MOVE	Dm,Abs.W	12	2	1	16	2	2
MOVE	Dm,Abs.L	16	3	1	20	3	2
MOVE	Am,Dn	04	1	0	04	1	0
MOVE	Am,(An)	08	1	1	12	1	2
MOVE	Am,(An)+	08	1	1	12	1	2
MOVE	Am,-(An)	08	1	1	14	1	2
MOVE	Am,d(An)	12	2	1	16	2	2
MOVE	Am,d(An,Dn/An)	14	2	1	18	2	2
MOVE	Am,Abs.W	12	2	1	16	2	2
MOVE	Am,Abs.L	16	3	1	20	3	2
MOVE	(Am),Dn	08	2	0	12	3	0
MOVE	(Am),(An)	12	2	1	20	3	2
MOVE	(Am),(An)+	12	2	1	20	3	2
MOVE	(Am),-(An)	12	2	1	20	3	2
MOVE	(Am),d(An)	16	3	1	24	4	2
MOVE	(Am),d(An,Dn/An)	18	3	1	26	4	2
MOVE	(Am),Abs.W	16	3	1	24	4	2
MOVE	(Am),Abs.L	20	4	1	28	5	2

Instruction	Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
MOVE (Am)+,Dn	08	2	0	12	3	0
MOVE (Am)+,(An)	12	2	1	20	3	2
MOVE (Am)+,(An)+	12	2	1	20	3	2
MOVE (Am)+,-(An)	12	2	1	20	3	2
MOVE (Am)+,d(An)	16	3	1	24	4	2
MOVE (Am)+,d(An,Dn/An)	18	3	1	26	4	2
MOVE (Am)+,Abs.W	16	3	1	24	4	2
MOVE (Am)+,Abs.L	20	4	1	28	5	2
MOVE -(Am),Dn	10	2	0	14	3	0
MOVE -(Am),(An)	14	2	1	22	3	2
MOVE -(Am),(An)+	14	2	1	22	3	2
MOVE -(Am),-(An)	14	2	1	22	3	2
MOVE -(Am),d(An)	18	3	1	26	4	2
MOVE -(Am),d(An,Dn/An)	20	3	1	28	4	2
MOVE -(Am),Abs.W	18	3	1	26	4	2
MOVE -(Am),Abs.L	22	4	1	30	5	2
MOVE d(Am),Dn	12	3	0	16	4	0
MOVE d(Am),(An)	16	3	1	24	4	2
MOVE d(Am),(An)+	16	3	1	24	4	2
MOVE d(Am),-(An)	16	3	1	24	4	2
MOVE d(Am),d(An)	20	4	1	28	5	2
MOVE d(Am),d(An,Dn/An)	22	4	1	30	5	2
MOVE d(Am),Abs.W	20	4	1	28	5	2
MOVE d(Am),Abs.L	24	5	1	32	6	2
MOVE d(Am,Dn/An),Dn	14	3	0	18	4	0
MOVE d(Am,Dn/An),(An)	18	3	1	26	4	2

Instruction		Byte/ Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
MOVE	d(Am,Dn/An),(An)+	18	3	1	26	4	2
MOVE	d(Am,Dn/An),-(An)	18	3	1	26	4	2
MOVE	d(Am,Dn/An),d(An)	22	4	1	30	5	2
MOVE	d(Am,Dn/An),d(An,Dn/An)	24	4	1	32	5	2
MOVE	d(Am,Dn/An),Abs.W	22	4	1	30	5	2
MOVE	d(Am,Dn/An),Abs.L	26	5	1	34	6	2
MOVE	Abs.W,Dn	12	3	0	16	4	0
MOVE	Abs.W,(An)	16	3	1	24	4	2
MOVE	Abs.W,(An)+	16	3	1	24	4	2
MOVE	Abs.W,-(An)	16	3	1	24	4	2
MOVE	Abs.W,d(An)	20	4	1	28	5	2
MOVE	Abs.W,d(An,Dn/An)	22	4	1	30	5	2
MOVE	Abs.W,Abs.W	20	4	1	28	5	2
MOVE	Abs.W,Abs.L	24	5	1	32	6	2
MOVE	Abs.L,Dn	16	4	0	20	5	0
MOVE	Abs.L,(An)	20	4	1	28	5	2
MOVE	Abs.L,(An)+	20	4	1	28	5	2
MOVE	Abs.L,-(An)	20	4	1	28	5	2
MOVE	Abs.L,d(An)	24	5	1	32	6	2
MOVE	Abs.L,d(An,Dn/An)	26	5	1	34	6	2
MOVE	Abs.L,Abs.W	24	5	1	32	6	2
MOVE	Abs.L,Abs.L	28	6	1	36	7	2
MOVE	d(PC),Dn	12	3	0	16	4	0
MOVE	d(PC),(An)	16	3	1	24	4	2
MOVE	d(PC),(An)+	16	3	1	24	4	2

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
MOVE	d(PC),-(An)	16	3	1	24	4	2
MOVE	d(PC),d(An)	20	4	1	28	5	2
MOVE	d(PC),d(An,Dn/An)	22	4	1	30	5	2
MOVE	d(PC),Abs.W	20	4	1	28	5	2
MOVE	d(PC),Abs.L	24	5	1	32	6	2
MOVE	d(PC,Dn/An),Dn	14	3	0	18	4	0
MOVE	d(PC,Dn/An),(An)	18	3	1	26	4	2
MOVE	d(PC,Dn/An),(An)+	18	3	1	26	4	2
MOVE	d(PC,Dn/An),-(An)	18	3	1	26	4	2
MOVE	d(PC,Dn/An),d(An)	22	4	1	30	5	2
MOVE	d(PC,Dn/An),d(An,Dn/An)	24	4	1	32	5	2
MOVE	d(PC,Dn/An),Abs.W	22	4	1	30	5	2
MOVE	d(PC,Dn/An),Abs.L	26	5	1	34	6	2
MOVE	#Imm,Dn	08	2	0	12	3	0
MOVE	#Imm,(An)	12	2	1	20	3	2
MOVE	#Imm,(An)+	12	2	1	20	3	2
MOVE	#Imm,-(An)	12	2	1	20	3	2
MOVE	#Imm,d(An)	16	3	1	24	4	2
MOVE	#Imm,d(An,Dn/An)	18	3	1	26	4	2
MOVE	#Imm,Abs.W	16	3	1	24	4	2
MOVE	#Imm,Abs.L	20	4	1	28	5	2
MOVE	Dn,CCR	12	2	0			
MOVE	(An),CCR	16	3	0			
MOVE	(An)+,CCR	16	3	0			
MOVE	-(An),CCR	18	3	0			

Instruction	Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
MOVE d(An),CCR	20	4	0			
MOVE d(An,Dn/An),CCR	22	4	0			
MOVE Abs.W,CCR	20	4	0			
MOVE Abs.L,CCR	24	5	0			
MOVE #Imm,CCR	16	3	0			
MOVE Dn,SR	12	2	0			
MOVE (An),SR	16	3	0			
MOVE (An)+,SR	16	3	0			
MOVE -(An),SR	18	3	0			
MOVE d(An),SR	20	4	0			
MOVE d(An,Dn/An),SR	22	4	0			
MOVE Abs.W,SR	20	4	0			
MOVE Abs.L,SR	24	5	0			
MOVE d(PC),SR	20	4	0			
MOVE d(PC,Dn/An),SR	22	4	0			
MOVE #Imm,SR	16	3	0			
MOVE SR,Dn	06	1	0			
MOVE SR,(An)	12	2	1			
MOVE SR,(An)+	12	2	1			
MOVE SR,-(An)	14	2	1			
MOVE SR,d(An)	16	3	1			
MOVE SR,d(An,Dn/An)	18	3	1			
MOVE SR,Abs.W	16	3	1			
MOVE SR,Abs.L	20	4	1			
MOVE USP,An	04	1	0			
MOVE An,USP	04	1	0			

Instruction	Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
MOVEA Dn,An	04	1	0			
MOVEA Am,An	04	1	0			
MOVEA (Am),An	08	2	0			
MOVEA (Am)+,An	08	2	0			
MOVEA -(Am),An	10	2	0			
MOVEA d(Am),An	12	3	0			
MOVEA d(Am,Dn/An),An	14	3	0			
MOVEA Abs.W,An	12	3	0			
MOVEA Abs.L,An	16	4	0			
MOVEA d(PC),An	12	3	0			
MOVEA d(PC,Dn/An),An	14	3	0			
MOVEA #Imm,An	08	2	0			
MOVEM RegList,(An)	8+5n	2	n	8+10n	2	2n
MOVEM RegList,-(An)	8+5n	2	n	8+10n	2	2n
MOVEM RegList,d(An)	12+5n	2	n	12+10n	2	2n
MOVEM RegList,d(An,Dn/An)	14+5n	2	n	14+10n	2	2n
MOVEM RegList,Abs.W	12+5n	2	n	12+10n	2	2n
MOVEM RegList,Abs.L	16+5n	2	n	16+10n	2	2n
MOVEM (An),RegList	12+4n	3+n	0	12+8n	3+2n	0
MOVEM (An)+,RegList	12+4n	3+n	0	12+8n	3+2n	0
MOVEM d(An),RegList	16+4n	4+n	0	16+8n	4+2n	0
MOVEM Abs.W,RegList	16+4n	4+n	0	16+8n	4+2n	0
MOVEM Abs.L,RegList	20+4n	5+n	0	20+8n	5+2n	0
MOVEM d(PC),RegList	16+4n	4+n	0	16+8n	4+2n	0
MOVEM d(PC,Dn/An),RegList	18+4n	4+n	0	18+8n	4+2n	0
MOVEP Dm,d(An)	16	2	2	24	2	4

Instruction	Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
MOVEP d(Am),Dn	16	4	0	24	6	0
MOVEQ #Imm,Dn	04	1	0			
MULS Dm,Dn	70	1	0	(MULS, MULU are max		
MULS (An),Dn	74	2	0	values)		
MULS (An)+,Dn	74	2	0			
MULS -(An),Dn	76	2	0			
MULS d(An),Dn	78	3	0			
MULS d(An,Dn/An),Dn	80	3	0			
MULS Abs.W,Dn	78	3	0			
MULS Abs.L,Dn	80	3	0			
MULS d(PC),Dn	78	3	0			
MULS d(PC,Dn/An),Dn	80	3	0			
MULS #Imm,Dn	74	2	0			
MULU Dm,Dn	70	1	0			
MULU (An),Dn	74	2	0			
MULU (An)+,Dn	74	2	0			
MULU -(An),Dn	76	2	0			
MULU d(An),Dn	78	3	0			
MULU d(An,Dn/An),Dn	80	3	0			
MULU Abs.W,Dn	78	3	0			
MULU Abs.L,Dn	82	4	0			
MULU d(PC),Dn	78	3	0			
MULU d(PC,Dn/An),Dn	80	3	0			
MULU #Imm,Dn	74	2	0			
NBCD Dm	06	1	0			
NBCD (An)	12	2	1			

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
NBCD	(An)+	12	2	1			
NBCD	-(An)	14	2	1			
NBCD	d(An)	16	3	1			
NBCD	d(An,Dn/An)	18	3	1			
NBCD	Abs.W	16	3	1			
NBCD	Abs.L	20	4	1			
NEG	Dn	04	1	0	06	1	0
NEG	(An)	12	2	1	20	3	2
NEG	(An)+	12	2	1	20	3	2
NEG	-(An)	14	2	1	22	3	2
NEG	d(An)	16	3	1	24	4	2
NEG	d(An,Dn/An)	18	3	1	26	4	2
NEG	Abs.W	16	3	1	24	4	2
NEG	Abs.L	20	4	1	28	5	2
NEGX	Dn	04	1	0	06	1	0
NEGX	(An)	12	2	1	20	3	2
NEGX	(An)+	12	2	1	20	3	2
NEGX	-(An)	14	2	1	22	3	2
NEGX	d(An)	16	3	1	24	4	2
NEGX	d(An,Dn/An)	18	3	1	26	4	2
NEGX	Abs.W	16	3	1	24	4	2
NEGX	Abs.L	20	4	1	28	5	2
NOP		04	1	0			
NOT	Dn	04	1	0	06	1	0
NOT	(An)	12	2	1	20	3	2
NOT	(An)+	12	2	1	20	3	2

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
NOT	-(An)	14	2	1	22	3	2
NOT	d(An)	16	3	1	24	4	2
NOT	d(An,Dn/An)	18	3	1	26	4	2
NOT	Abs.W	16	3	1	24	4	2
NOT	Abs.L	20	4	1	28	5	2
OR	Dm,Dn	04	1	0	08	1	0
OR	(An),Dn	08	2	0	14	3	0
OR	(An)+,Dn	08	2	0	14	3	0
OR	-(An),Dn	10	2	0	16	3	0
OR	d(An),Dn	12	3	0	18	4	0
OR	d(An,Dn/An),Dn	14	3	0	20	4	0
OR	Abs.W,Dn	12	3	0	18	4	0
OR	Abs.L,Dn	16	4	0	22	5	0
OR	d(PC),Dn	12	3	0	18	4	0
OR	d(PC,Dn/An),Dn	14	3	0	20	4	0
OR	Dm,(An)	12	2	1	20	3	1
OR	Dm,(An)+	12	2	1	20	3	1
OR	Dm,-(An)	14	2	1	22	3	1
OR	Dm,d(An)	16	3	1	24	4	1
OR	Dm,d(Am,Dn/An)	18	3	1	26	4	1
OR	Dm,Abs.W	16	3	1	24	4	1
OR	Dm,Abs.L	20	4	1	28	5	1
ORI	#Imm,Dn	08	2	0	16	3	0
ORI	#Imm,(An)	16	3	1	28	5	2
ORI	#Imm,(An)+	16	3	1	28	5	2
ORI	#Imm,-(An)	14	2	1	22	3	1

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
ORI	#Imm,d(An)	20	4	1	32	6	2
ORI	#Imm,d(Am,Dn/An)	22	4	1	34	6	2
ORI	#Imm,Abs.W	20	4	1	32	6	2
ORI	#Imm,Abs.L	24	5	1	36	7	2
ORI	#Imm,CCR	20	3	0			
ORI	#Imm,SR	20	3	0			
PEA	(An)	12	1	2			
PEA	d(An)	16	2	2			
PEA	d(Am,Dn/An)	20	2	2			
PEA	Abs.W	16	2	2			
PEA	Abs.L	20	3	2			
PEA	d(PC)	16	2	2			
PEA	d(PC,Dn/An)	20	2	2			
RESET		132	1	0			
ROL	Dm,Dn	6+2n	1	0	8+2n	1	0
ROL	#Imm,Dn	6+2n	1	0	8+2n	1	0
ROL	(An)	12	2	1	16	3	1
ROL	(An)+	12	2	1	16	3	1
ROL	-(An)	14	2	1	18	3	1
ROL	d(An)	16	3	1	20	4	1
ROL	d(An,Dn/An)	18	3	1	22	4	1
ROL	Abs.W	16	3	1	20	4	1
ROL	Abs.L	20	4	1	24	5	1
ROR	Dm,Dn	6+2n	1	0	8+2n	1	0
ROR	#Imm,Dn	6+2n	1	0	8+2n	1	0
ROR	(An)	12	2	1	16	3	1

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
ROR	(An)+	12	2	1	16	3	1
ROR	-(An)	14	2	1	18	3	1
ROR	d(An)	16	2	1	20	4	1
ROR	d(An,Dn/An)	18	3	1	22	4	1
ROR	Abs.W	16	3	1	20	4	1
ROR	Abs.L	20	4	1	24	5	1
ROXL	Dm,Dn	6+2n	1	0	8+2n	1	0
ROXL	#Imm,Dn	6+2n	1	0	8+2n	1	0
ROXL	(An)	12	2	1	16	3	1
ROXL	(An)+	12	2	1	16	3	1
ROXL	-(An)	14	2	1	18	3	1
ROXL	d(An)	16	2	1	20	4	1
ROXL	d(An,Dn/An)	18	3	1	22	4	1
ROXL	Abs.W	16	3	1	20	4	1
ROXL	Abs.L	20	4	1	24	5	1
ROXR	Dm,Dn	6+2n	1	0	8+2n	1	0
ROXR	#Imm,Dn	6+2n	1	0	8+2n	1	0
ROXR	(An)	12	2	1	16	3	1
ROXR	(An)+	12	2	1	16	3	1
ROXR	-(An)	14	2	1	18	3	1
ROXR	d(An)	16	2	1	20	4	1
ROXR	d(An,Dn/An)	18	3	1	22	4	1
ROXR	Abs.W	16	3	1	20	4	1
ROXR	Abs.L	20	4	1	24	5	1
RTE		20	5	0			
RTR		20	5	0			

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
RTS		16	4	0			
SBCD	Dm,Dn	06	1	0			
SBCD	-(Am),-(An)	18	3	1			
		Scc False			Scc True		
Scc	Dn	4	1	0	6	1	0
Scc	(An)	12	2	1			
Scc	(An)+	12	2	1			
Scc	-(An)	14	2	1			
Scc	d(An)	16	3	1			
Scc	d(An,Dn/An)	18	3	1			
Scc	Abs.W	16	3	1			
Scc	Abs.L	20	4	1			
STOP		04	0	0			
SUB	Dm,Dn	04	1	0	08	1	0
SUB	Am,Dn	04	1	0	08	1	0
SUB	(Am),Dn	08	2	0	14	3	0
SUB	(Am)+,Dn	08	2	0	14	3	0
SUB	-(Am),Dn	10	2	0	16	3	0
SUB	d(Am),Dn	12	3	0	18	4	0
SUB	d(Am,Dn/An),Dn	14	3	0	20	4	0
SUB	Abs.W,Dn	12	3	0	18	4	0
SUB	Abs.L,Dn	16	4	0	22	5	0
SUB	d(PC),Dn	12	3	0	18	4	0
SUB	d(PC,Dn/An),Dn	14	3	0	20	4	0
SUB	#Imm,Dn	08	2	0	14	3	0
SUB	Dm,(An)	12	2	1	20	3	2

Instruction		Byte/Word Cycle Time	Byte/Word Read	Byte/Word Write	Long Cycle Time	Long Read	Long Write
SUB	Dm,(An)+	12	2	1	20	3	2
SUB	Dm,-(An)	14	2	1	22	3	2
SUB	Dm,d(An)	16	3	1	24	4	2
SUB	Dm,d(An,Dn/An)	18	3	1	26	4	2
SUB	Dm,Abs.W	16	3	1	24	4	2
SUB	Dm,Abs.L	20	4	1	28	5	2
SUBA	Dm,An	08	1	0	08	1	0
SUBA	(Am),An	12	2	0	14	3	0
SUBA	(Am)+,An	12	2	0	14	3	0
SUBA	-(Am),An	14	2	0	16	3	0
SUBA	d(Am),An	16	3	0	18	4	0
SUBA	d(Am,Dn/An),An	18	3	0	20	4	0
SUBA	Abs.W,An	16	3	0	18	4	0
SUBA	Abs.L,An	20	4	0	22	5	0
SUBA	d(PC),An	16	3	0	18	4	0
SUBA	d(PC,Dn/An),An	18	3	0	20	4	0
SUBA	#Imm,An	12	2	0	14	3	0
SUBI	#Imm,Dn	08	2	0	16	3	0
SUBI	#Imm,(An)	16	3	1	28	5	2
SUBI	#Imm,(An)+	16	3	1	28	5	2
SUBI	#Imm,-(An)	18	3	1	30	5	2
SUBI	#Imm,d(An)	20	4	1	32	6	2
SUBI	#Imm,d(An,Dn/An)	22	4	1	34	6	2
SUBI	#Imm,Abs.W	20	4	1	32	6	2
SUBI	#Imm,Abs.L	24	5	1	36	7	2
SUBQ	#Imm,Dn	04	1	0	08	1	0

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
SUBQ	#Imm,An	08	1	0	08	1	0
SUBQ	#Imm,(An)	12	2	1	20	3	2
SUBQ	#Imm,(An)+	12	2	1	20	3	2
SUBQ	#Imm,-(An)	14	2	1	22	3	2
SUBQ	#Imm,d(An)	16	3	1	24	4	2
SUBQ	#Imm,d(An,Dn/An)	18	3	1	26	4	2
SUBQ	#Imm,Abs.W	16	3	1	24	4	2
SUBQ	#Imm,Abs.L	20	4	1	28	5	2
SUBX	Dm,Dn	04	1	0	08	1	0
SUBX	-(Am),-(An)	18	3	1	30	5	2
SWAP	Dn	04	1	0			
TAS	Dn	04	1	0			
TAS	(An)	14	2	1			
TAS	(An)+	14	2	1			
TAS	-(An)	16	2	1			
TAS	d(An)	18	3	1			
TAS	d(An,Dn/An)	20	3	1			
TAS	Abs.W	18	3	1			
TAS	Abs.L	22	4	1			
TRAP		34	4	3			
		TRAPV taken			TRAPV not taken		
TRAPV		34	5	3	04	1	0
TST	Dn	04	1	0	04	1	0
TST	(An)	08	2	0	12	3	0
TST	(An)+	08	2	0	12	3	0
TST	-(An)	10	2	0	14	3	0

Instruction		Byte/Word Cycle Time	Read	Write	Long Cycle Time	Read	Write
TST	d(An)	12	3	0	16	4	0
TST	d(An,Dn/An)	14	3	0	18	4	0
TST	Abs.W	12	3	0	16	4	0
TST	Abs.L	16	4	0	20	5	0
UNLK		12	3	0			

APPENDIX

Condition Codes

This appendix describes how each instruction affects the 5 condition code bits. These are abbreviated X for the extend flag, N for the negative flag, Z for the zero flag, V for the overflow flag, and C for the carry flag. X is the fifth or $10 bit, N is the fourth or $08 bit, Z is the third or $04 bit, V is the second or $02 bit, and C is the first or $01 bit of the condition codes.

Key

UNDEF = undefined (may be anything after this instruction)
NOCHG = unchanged by operation
NONCHG = no condition code changed by this operation
NORM = flag changed based on result the way you would expect as follows:

X = same as carry

N = if the result is negative (highest bit on) then this bit is set; if the result is not negative, this bit is cleared

Z = if the result is zero then this bit is set; if there is a non-zero result it is cleared

V = if an overflow occurs, this bit is set; it is cleared if no overflow occurs

C = if a carry (or borrow) is generated, this bit is set; it is cleared if no carry (or borrow) occurs (for ABCD and NBCD this will be a decimal carry)

CLEARED = bit always cleared by operation
SET = bit always set by operation
* = see description below

Co Code	X	N	Z	V	C
ABCD	NORM	UNDEF	*	UNDEF	NORM
* = if result non-zero, Z is cleared. No change if = 0.					
ADD	NORM	NORM	NORM	NORM	NORM
ADDA		NONCHG			
ADDI	NORM	NORM	NORM	NORM	NORM
ADDQ	NORM	NORM	NORM	NORM	NORM
ADDX	NORM	NORM	*	NORM	NORM
* = if result non-zero, Z is cleared. No change if = 0.					
AND	NOCHG	NORM	NORM	CLEARED	CLEARED
ANDI	NOCHG	NORM	NORM	CLEARED	CLEARED
ANDI CCR	cleared if appropriate bit of immediate operand is zero; no change if immediate operand bit is one.				
ANDI SR	cleared if appropriate bit of immediate operand is zero; no change if immediate operand bit is one.				
ASL	*	NORM	NORM	**	***
* = same as last bit shifted out. No change if shift count is zero.					
** = set if most significant bit changed at any time in this operation and cleared if no change in most significant bit					
*** = same as last bit shifted out of operand. Cleared for a shift count of zero.					
ASR	same as ASL				
Bcc		NONCHG			
BCHG	NOCHG	NOCHG	*	NOCHG	NOCHG
* = set if bit tested is zero; cleared if bit not zero.					
BCLR	same as BCHG				
BRA		NONCHG			
BSET	same as BCHG				
BSR		NONCHG			
BTST	same as BCHG				
CHK	NOCHG	*	UNDEF	UNDEF	UNDEF
* = set if Dn<0. clear if Dn > operand. UNDEF otherwise					
CLR	NOCHG	CLEARED	SET	CLEARED	CLEARED
CMP	NORM	NORM	NORM	NORM	NORM
CMPA	NOCHG	NORM	NORM	NORM	NORM
CMPI	NOCHG	NORM	NORM	NORM	NORM
CMPM	NOCHG	NORM	NORM	NORM	NORM
DBcc		NONCHG			
DIVS	NOCHG	*	**	***	CLEARED
* = if quotient is negative, set; cleared if not. undefined if overflow.					
** = if quotient is zero, set; cleared if not. undefined if overlow.					
*** = set if division overflows. cleared if no overflow.					

Co Code	X	N	Z	V	C
DIVU	same as DIVS				
EOR	NOCHG	NORM	NORM	CLEARED	CLEARED
EORI	NOCHG	NORM	NORM	CLEARED	CLEARED
EORI CCR	changed if corresponding bit of immediate operand is a one. No change if corresponding bit is zero.				
EORI SR	same as EORI CCR.				
EXG		NONCHG			
EXT	NOCHG	NORM	NORM	CLEARED	CLEARED
ILLEGAL		NONCHG			
JMP		NONCHG			
JSR		NONCHG			
LEA		NONCHG			
LINK		NONCHG			
LSL	same as ASL except that V is always cleared.				
LSR	same as LSL				
MOVE	NOCHG	NORM	NORM	CLEARED	CLEARED
MOVE CCR	corresponding bits set the same as the source operand				
MOVE TO SR	corresponding bits set the same as the source operand				
MOVE FROM SR		NONCHG			
MOVE USP		NONCHG			
MOVEA		NONCHG			
MOVEM		NONCHG			
MOVEP		NONCHG			
MOVEQ	NOCHG	NORM	NORM	CLEARED	CLEARED
MULS	NOCHG	NORM	NORM	CLEARED	CLEARED
MULU	NOCHG	NORM	NORM	CLEARED	CLEARED
NBCD	NORM	UNDEF	*	UNDEF	NORM
	* = cleared if result non-zero. No change if result zero				
NEG	NORM	NORM	NORM	NORM	*
	* = cleared if zero result; set if non-zero result.				
NEGX	NORM	NORM	*	NORM	**
	* = cleared if result non-zero. No change if result zero.				
	** = set if borrow. cleared if no borrow.				
NOP		NONCHG			
NOT	NOCHG	NORM	NORM	CLEARED	CLEARED
OR	NOCHG	NORM	NORM	CLEARED	CLEARED
ORI	NOCHG	NORM	NORM	CLEARED	CLEARED
ORI CCR	set if corresponding bit is one. No change if bit zero.				
ORI SR	set if corresponding bit is one. No change if bit zero.				
PEA		NONCHG			
RESET		NONCHG			

Co Code	X	N	Z	V	C
ROL	NOCHG	NORM	NORM	CLEARED	*
* = set according to last bit shifted out. Cleared if shift count is zero.					
ROR	same as ROL				
ROXL	*	NORM	NORM	CLEARED	**
* = same as last bit shifted out of operand. No change if shift count is zero.					
** = set according to last bit shifted out. Same as extend bit if shift count is zero.					
RTE	set according to status register pulled from stack				
RTR	set according to status register pulled from stack				
RTS		NONCHG			
SBCD	NORM	UNDEF	*	UNDEF	NORM
* = if result non-zero, Z is cleared. No change if = 0.					
Scc		NONCHG			
STOP	set according to immediate operand.				
SUB	NORM	NORM	NORM	NORM	NORM
SUBA		NONCHG			
SUBI	NORM	NORM	NORM	NORM	NORM
SUBQ	NORM	NORM	NORM	NORM	NORM
SUBX	NORM	NORM	*	NORM	NORM
* = if result non-zero, Z is cleared. No change if = 0.					
SWAP	NOCHG	NORM	NORM	CLEARED	CLEARED
TAS	NOCHG	NORM	NORM	CLEARED	CLEARED
TRAP		NONCHG			
TRAPV		NONCHG			
TST	NOCHG	NORM	NORM	CLEARED	CLEARED
UNLK		NONCHG			

APPENDIX

Error Messages

This MDS assembler file, from Apple Computer, gives all the error messages. The errors with positive numbers are what you see when there is a "bomb" alert box; the error number is in the lower right hand corner of the alert box as "ID=*xx*" where *xx* is the error number. Some errors are mildly misleading; for example, error 25, out of memory, sometimes means that a resource file was not found. After a while, you associate each system error with a particular type of error in your program. Using the same example, you will find that out of memory often occurs when you have a resource file that you should have unlocked when you were done with it.

```
; File: SysErr.Txt   (26 Apr 85) Version 1.1
;_____________________________________________________
;
; System Error Equates -- This file defines the equates for the Macintosh
;  return error codes  This is divided into two pieces for assembly
;  space and speed considerations.  The wholeErrors flag is used to include
;  the less common equates which realizes a complete set.
;
; Copyright 1984, Apple Computer, Inc.
;_____________________________________________________
```

```
; General System Errors (VBL Mgr, Queueing, Etc.)

noErr        EQU   0     ; 0 for success
qErr         EQU   -1    ; queue element not found during deletion
vTypErr      EQU   -2    ; invalid queue element
corErr       EQU   -3    ; core routine number out of range
unimpErr     EQU   -4    ; unimplemented core routine

; I/O System Errors

controlErr   EQU  -17
statusErr    EQU  -18
readErr      EQU  -19
writErr      EQU  -20
badUnitErr   EQU  -21
unitEmptyErr EQU  -22
openErr      EQU  -23
closErr      EQU  -24
dRemovErr    EQU  -25   ; tried to remove an open driver
dInstErr     EQU  -26   ; DrvrInstall couldn't find driver in resources
abortErr     EQU  -27   ; IO call aborted by KillIO
notOpenErr   EQU  -28    ; Couldn't rd/wr/ctl/sts cause driver not
opened

; File System error codes:

dirFulErr    EQU  -33   ; Directory full
dskFulErr    EQU  -34   ; disk full
nsvErr       EQU  -35   ; no such volume
ioErr        EQU  -36   ; I/O error (bummers)
```

```
bdNamErr    EQU   -37   ; there may be no bad names in the final
system!
fnOpnErr    EQU   -38   ; File not open
eofErr      EQU   -39   ; End of file
posErr      EQU   -40   ; tried to position to before start of file (r/w)
tmfoErr     EQU   -42   ; too many files open
fnfErr      EQU   -43   ; File not found

wPrErr      EQU   -44   ; diskette is write protected
fLckdErr    EQU   -45   ; file is locked
vLckdErr    EQU   -46   ; volume is locked
fBsyErr     EQU   -47   ; File is busy (delete)
dupFNErr    EQU   -48   ; duplicate filename (rename)
opWrErr     EQU   -49   ; file already open with with write permission
paramErr    EQU   -50   ; error in user parameter list
rfNumErr    EQU   -51   ; refnum error
gfpErr      EQU   -52   ; get file position error
volOffLinErr EQU  -53   ; volume not on line error (was Ejected)
permErr     EQU   -54   ; permissions error (on file open)
volOnLinErr EQU   -55   ; drive volume already on-line at MountVol
nsDrvErr    EQU   -56   ; no such drive (tried to mount a bad drive num)
noMacDskErr EQU   -57   ; not a mac diskette (sig bytes are wrong)
extFSErr    EQU   -58   ; volume in question belongs to an external fs
fsRnErr     EQU   -59   ; file system internal error:
                        ; during rename the old entry was deleted but could
                        ;  not be restored . . .
badMDBErr   EQU   -60   ; bad master directory block
wrPermErr   EQU   -61   ; write permissions error
```

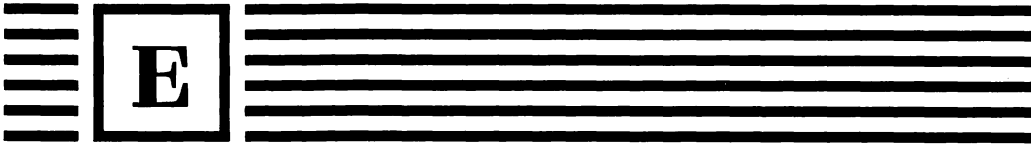
```
; Disk, Serial Ports, Clock Specific Errors

firstDskErr   EQU   -84
lastDskErr    EQU   -64

noDriveErr    EQU   -64   ; drive not installed
offLinErr     EQU   -65   ; r/w requested for an off-line drive

noNybErr      EQU   -66   ; couldn't find 5 nybbles in 200 tries
noAdrMkErr    EQU   -67   ; couldn't find valid addr mark
dataVerErr    EQU   -68   ; read verify compare failed
badCkSmErr    EQU   -69   ; addr mark checksum didn't check
badBtSlpErr   EQU   -70   ; bad addr mark bit slip nibbles
noDtaMkErr    EQU   -71   ; couldn't find a data mark header
badDCkSum     EQU   -72   ; bad data mark checksum
badDBtSlp     EQU   -73   ; bad data mark bit slip nibbles
wrUnderRun    EQU   -74   ; write underrun occurred

cantStepErr   EQU   -75   ; step handshake failed
tk0BadErr     EQU   -76   ; track 0 detect doesn't change
initIWMErr    EQU   -77   ; unable to initialize IWM
twoSideErr    EQU   -78   ; tried to read 2nd side on a 1-sided drive
spdAdjErr     EQU   -79   ; unable to correctly adjust disk speed
seekErr       EQU   -80   ; track number wrong on address mark
sectNFErr     EQU   -81   ; sector number never found on a track
clkRdErr      EQU   -85   ; unable to read same clock value twice
clkWrErr      EQU   -86   ; time written did not verify
prWrErr       EQU   -87   ; parameter ram written didn't read-verify
prInitErr     EQU   -88     ;  InitUtil found the parameter ram
uninitialized
```

```
rcvrErr         EQU   -89   ; SCC receiver error (framing, parity, OR)
breakRecd       EQU   -90   ; Break received (SCC)


; AppleTalk error codes

ddpSktErr       EQU   -91   ; error in socket number
ddpLenErr       EQU   -92   ; data length too big
noBridgeErr     EQU   -93   ; no network bridge for non-local send
lapProtErr      EQU   -94   ; error in attaching/detaching protocol
excessCollsns EQU   -95   ; excessive collisions on write
portInUse       EQU   -97   ; driver Open error code (port is in use)
portNotCf       EQU   -98   ; driver Open error code (parameter RAM not
                            ; configured for this connection)

; Storage allocator error codes

memFullErr      EQU   -108  ; Not enough room in heap zone
nilHandleErr    EQU   -109  ; Handle was NIL in HandleZone or other;
memWZErr        EQU   -111  ; WhichZone failed (applied to free block);
memPurErr       EQU   -112  ; trying to purge a locked or non-purgeable
block;

memAdrErr       EQU   -110  ; address was odd, or out of range;
memAZErr        EQU   -113  ; Address in zone check failed;
memPCErr        EQU   -114  ; Pointer Check failed;
memBCErr        EQU   -115  ; Block Check failed;
memSCErr        EQU   -116  ; Size Check failed;
memLockedErrEQU   -117  ; trying to move a locked block (MoveHHi)
```

```
; Resource Manager error codes (other than I/O errors)

resNotFound  EQU   -192 ; Resource not found
resFNotFound EQU   -193 ; Resource file not found
addResFailed EQU   -194 ; AddResource failed
rmvResFailed EQU   -196 ; RmveResource failed

; Scrap Manager error codes

noScrapErr   EQU   -100 ; No scrap exists error
noTypeErr    EQU   -102 ; No object of that type in scrap

; some miscellaneous result codes

evtNotEnb    EQU   1    ; event not enabled at PostEvent


; System Error Alert ID definitions. These are just for reference because
; one cannot intercept the calls and do anything programmatically...

dsSysErr     EQU   32767 ; general system error
dsBusError   EQU   1     ; bus error
dsAddressErr EQU   2     ; address error
dsIllInstErr EQU   3     ; illegal instruction error
dsZeroDivErr EQU   4     ; zero divide error
dsChkErr     EQU   5     ; check trap error
dsOvFlowErr  EQU   6     ; overflow trap error
dsPrivErr    EQU   7     ; privelege violation error
dsTraceErr   EQU   8     ; trace mode error
dsLineAErr   EQU   9     ; line 1010 trap error
```

```
dsLineFErr      EQU     10      ; line 1111 trap error
dsMiscErr       EQU     11      ; miscellaneous hardware exception error
dsCoreErr       EQU     12      ; unimplemented core routine error
dsIrqErr        EQU     13      ; uninstalled interrupt error

dsIOCoreErr     EQU     14      ; IO Core Error
dsLoadErr       EQU     15      ; Segment Loader Error
dsFPErr         EQU     16      ; Floating point error

dsNoPackErr     EQU     17      ; package 0 not present
dsNoPk1         EQU     18      ; package 1 not present
dsNoPk2         EQU     19      ; package 2 not present
dsNoPk3         EQU     20      ; package 3 not present
dsNoPk4         EQU     21      ; package 4 not present
dsNoPk5         EQU     22      ; package 5 not present
dsNoPk6         EQU     23      ; package 6 not present
dsNoPk7         EQU     24      ; package 7 not present

dsMemFullErr    EQU     25      ; out of memory!
dsBadLaunch     EQU     26      ; can't launch file

dsFSErr         EQU     27      ; file system map has been trashed
dsStknHeap      EQU     28      ; stack has moved into application heap
dsReinsert      EQU     30      ; request user to reinsert off-line volume
dsNotThe1       EQU     31      ; not the disk I wanted
```

APPENDIX



Using the Lisa WorkShop

This appendix shows how to create an assembly language program in the Lisa WorkShop. The assembly language is linked to a dummy Pascal program. You can easily expand this example to create assembly language procedures to augment a real Pascal program. It even shows how to define variables that are accessible both in Assembly language and Pascal. An exec file to run through all the steps of the process is also included.

The Lisa Exec File, SimpleCalcExec

The SimpleCalcExec file will create a Macintosh program on the Lisa WorkShop. This one file will assemble SimpleCalc, compile the small, top Pascal program, run the Linker, run the Resource Compiler, and finally run the MacCom program to make a Macintosh disk. You can easily change the program names to create other applications. To use this exec file, you need the short Pascal program, SimplePAS, which shows below.

```
$EXEC
$ {This is SimpleCalcExec, the exec file to assembly & link SimpleCalc}
$Writeln('SimpleCalcExec is now creating SimpleCalc')
A{ssemble}SimpleCalcASM  {Assemble Lisa version of SimpleCalc}
{this blank line is for the listing file}
SimpleCalcA    {Output Assembler file}
P{ascal}$M+  {Compile the dummy Pascal program to Macintosh format}
SimplePAS
{this blank line is for the listing file}
SimpleCalcP   {Pascal output file}
L{ink}?       {Link the programs and system libraries}
+X            {Link in Macintosh format}
```

```
SimpleCalcA   {Assembled object}
SimpleCalcP   {Compiled Pascal object}
obj/quickDraw
obj/tooltraps
obj/ostraps
obj/prlink
obj/packtraps
obj/sanelibasm
obj/PasLib
obj/PasLibasm
obj/PasInit
obj/RTLib
{No more files}
{this blank line is for the listing file}
SimpleCalcL.OBJ     {Linked object file}
$
R{un}RMaker.obj     {Start the Resource compiler}
SimpleCalcR         {Resource source file}
$
R{un}MacCom.obj     {Start MacCom which writes a Macintosh disk}
FYLSimpleCalc.RSRC  {Copy from file on Lisa disk}
SimpleCalc.RSRC     {Copy to file on Macintosh disk}
APPL                {set type to APPL}
CALC                {set creator to CALC}
Y{es bundle bit}E{ject}Q{uit}
$ENDEXEC
```

The Dummy Pascal Program, SimplePAS

This is the dummy Pascal program, SimplePas. When executed, it does nothing but call the SimpleCalc routines in Assembly language. SimplePas reserves global space for the assembly language procedure, SimpleCalc. The global space allocated by Pascal is addressed with register A5, just as though it were defined with a DS command. The dummy program defines DeskName which will correspond to DeskName in SimpleCalcAsm. It also reserves another 512 bytes for SimpleCalc with a Pascal array. When SimplePAS executes, it just calls SimpleCalc.

```
{$X-}
program SimplePAS;
            var
            {DeskName variable matches assembly language}
            DeskName :String[15];
            {Reserve 512 bytes for Assembler Globals}
            HalfK: ARRAY [1..256] OF INTEGER;
```

```
procedure SimpleCalc;   external;

begin
 SimpleCalc
end.
```

Lisa Version of the SimpleCalc, SimpleCalcASM

You have to modify the Macintosh Assembler version of SimpleCalcASM to use it in the Lisa WorkShop. These changes make SimpleCalc a procedure called from Pascal with data areas provided by a Pascal program.

```
;
; Make these changes to the SimpleCalcASM file to assemble the
; program under the Lisa Workshop.
;
;
;
; Make SimpleCalc a procedure called from Pascal
;       .PROC SimpleCal
;
; Place the rest of SimpleCalc here.
; Then make the changes shown below
;                 ........................................
;
; Change the Include files to Lisa Workshop names :
;
 ---------------------------------------INCLUDE ---------------------------------------
                  .INCLUDE   tlasm/SysEqu.Text
                  .INCLUDE   tlasm/SysTraps.Text
                  .INCLUDE   tlasm/QuickEqu.Text
                  .INCLUDE   tlasm/QuickTraps.Text
                  .INCLUDE   tlasm/ToolEqu.Text
                  .INCLUDE   tlasm/ToolTraps.Text


                 ........................................

; ---------------------------------GLOBALDATA ---------------------------------
; Assign the global data from the area reserved by Pascal
; System uses the first 4 bytes
FirstData       EQU    -4
; Desk accessory's name
DeskName        EQU    FirstData-16
; Storage for window
WindowStorage DS.B   DeskName-WindowSize
                 ..................................................
```

```
; Use .ASCII, .BLOCK, .WORD and .LONG to define data areas
OperTable
; 4 bytes per entry
;  byte 1=ASCII value of key
;  byte 2 not used
;  bytes 3&4 offset
                .ASCII   '+ '
                .WORD    AddOper-OperTable
                .ASCII   '- '
                .WORD    SubOper-OperTable
                .ASCII   '* '
                .WORD    MulOper-OperTable
                .ASCII   '/ '
                .WORD    DivOper-OperTable
                .ASCII   '= '
                .WORD    EqOper-OperTable
                .WORD    $0300      ;[Enter] key
                .WORD    Enter-OperTable
                .WORD    $0800      ;[BackSpace] key
                .WORD    Clear-OperTable
                .WORD    $1B00      ;[Clear] on 10-key pad
                .WORD    Clear-OperTable
                .LONG    0          ; End of table
                .....................................................

; ----------------------------------Data Storage ----------------------------------
CurrentEvent    ; Event record
What            .WORD       0               ; Type of Event
Message         .LONG       0               ; Info about event
When            .LONG       0               ; Tick when it happened
Where           ; Mouse location when it happened
WhereV          .WORD       0               ; Vertical coordinate
WhereH          .WORD       0               ; Horizontal coordinate
Modify          .WORD       0   ; Control keys down when it happened
EvtWind         .LONG       0               ; Window with event
Menu            .WORD       0               ; Menu that item is in
MenuItem        .WORD       0               ; Menu item selected
WindPntr        .LONG       0   ; Pointer to spread sheet window
DragLimit       ; Boundary rectangle for draging window
                .WORD       30              ; top
                .WORD       5               ; left
                .WORD       350             ; bottom
                .WORD       500             ; right
```

```
CellRect        .BLOCK      8,0         ; Rectangle enclosing selected cell
TxtPnt                                  ; Point in cell where text starts
                .LONG       0
                .END
```

Lisa Version of the SimpleCalcR File

Use this version of SimpleCalcR in the Lisa WorkShop. The resource compiler in the WorkShop has the type CODE, which puts the linked object file into the same resource file as the resource data. This is similar to the INCLUDE statement in the Macintosh resource file. When you use this version, SimpleCalcASM does not have to open a separate resource file. You can see this change in the Lisa version of SimpleCalcASM shown above.

```
*
* Resource file for SimpleCalc, Lisa version
*
* Resource file to be created.
SimpleCalc.RSRC

* Menu definitions
Type MENU
* Apple Menu is always ID 1. Desk accessories are added by the program
,1
\14
About SimpleCalc...
(-

* File Menu
  ,302
File
  Quit
* Edit Menu. This is not the standard edit menu
  ,303
Edit
   Cut/X
   Copy/C
   Paste/V
   Clear
   Negate/N
   (-
   Program/P
```

```
* Dialog for the About box
Type DLOG
 ,301
* Outside corners of the box, relative to the desk top
 100 100 200 400
* Type 1 window with reference number 0
Visible 1 NoGoAway 0
301

* Item list for the About box dialog
Type DITL
 ,301
* Number of items in list
4
* First item, the default item, is a rounded corner button
BtnItem Enabled
* Outside corners of the button relative to the edges of the dialog box
 70 220 90 280
* Inside of button reads "Try it!"
Try it \21

* Second item is the "Quit" button
BtnItem Enabled
70 20 90 80
Quit

* Information to display in a rectangle
StatText Disabled
* Use as many text lines as needed in this rectangle
15 50 35 280
* Text to display with no extra spaces
Simple Calc was written for fun

* More text
StatText Disabled
35 20 55 280
by Harland Harrison & Ed Rosenzweig

* Template for the window
Type WIND
 ,301
* Title the window "Simple Calc"
Simple Calc
* Outside corners of the window, relative to the desk top
64 32 320 480
```

```
Visible NoGoAway
0
0

* File reference
Type FREF
 ,128
 APPL 0

Type BNDL
 ,128
 CALC 0
 2
 ICN# 1
 0 128
 FREF 1
 0 128
Type ICN#
* Icon list for the program icon
 ,128
* There are two icons in the list. One to display and one for a mask
 2
* This is the SimpleCalc icon. There are 32 lines of icon data
00000000
00000000
00000000
1FFFFFFC
10000004
17FFFFF4
14081014
14C997D4
152A5054
14285094
14499114
14885214
150A5214
15E99214
14081014
17FFFFF4
14081FF4
15E89834
15099834
15089834
15C89834
14289834
```

```
14289834
15C9D834
14081FF4
17FFFFF4
10000004
1FFFFFFC
00000000
00000000
00000000
00000000
* This is the mask for the SimpleCalc icon. There are 32 lines of data.
00000000
00000000
00000000
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
00000000
00000000
00000000
00000000
```

```
* Signature for desktop file
Type CALC=STR
,0
SimpleCalc version 1.0, a fun spread sheet program

* Include the linked object file
Type CODE
SimpleCalcL,0
```

APPENDIX

F

Samples of Trap Calls into the ROM

Rules for Parameters in Pascal Definitions

1) If Pascal defines the parameter as a variable parameter using the word "VAR," pass a pointer to the address of the data to be modified.
2) If the data is longer than four bytes, always pass a pointer to the address of the data.
3) If the data is four bytes or less and the parameter is not a VAR, then pass the data itself on the stack.
4) Pass a two-byte integer for a CHAR, with the LSB holding the character value, and the MSB set to zero.
5) Pass a two-byte integer for a BOOLEAN, with the MSB holding 1 for "true." Pass $0100 for "true." Pass zero for "false." When receiving a Boolean value, test only the MSB. The low order byte might not be zero.
6) Pass a four-byte address for a pointer.
7) Pass a four-byte zero for NIL.
8) Pass a pointer to a subroutine for a ProcPtr, or a formal procedure.
9) Pass a pointer to the first, lowest-addressed byte of a record. Records must start on word boundaries.

The Most Common Pascal Types Used as Parameters

INTEGER parameter

```
MOVE.W    #-5,-(SP) ;two's complement negative 5
```

LONGINT parameter

```
            MOVE.L    #-8,-(SP) ;two's complement negative 8
```

CHAR parameter

```
            MOVEQ     # 'A',DO   ;letter "A"
            MOVE.W    DO,-(SP)
```

Boolean parameter

```
            MOVE.B    #1,-(SP)  ;Pass TRUE as a parameter.
            ...
            CLR.W     -(SP)     ;Pass FALSE as a parameter.
            ....
            BTST      #0,(SP)+  ;Test a function result. The
            BNE       TrueResult ;other bits are undefined!
```

Handle parameter

```
            LEA       Data,A0   ;Load pointer to data in A0.
                                ;Load the handle, a pointer to the
                                ;pointer, into A1.
            LEA       DataPntr(A5),A1
            MOVE.L    A0,(A1)   ;Set up the data pointer.
            MOVE.L    A1,-(SP)  ;Pass the handle parameter.
            ...
DataPntr    DS.B      4         ;four bytes to hold address
Data        DC        'Pass with handle'
```

ProcPtr parameter

```
            PEA       SubRout   ;Push entry point of procedure
            ...
                                ;The procedure starts here. It may
                                ;have parameters on the stack.
SubRout     NOP
            ...
```

NIL parameter

```
            CLR.L     -(SP)
```

String[n] or Str255 parameter

```
            PEA       'String Literal'
            ...
            PEA       String
            ...
            .ALIGN    2
```

```
String      DC.B      6          ;number of ASCII characters
            DC.B      'String'
            .ALIGN    2
```

Point parameter

```
            MOVE.L    Where,-(SP)
            ...
Where
WhereVert   DC.W      300        ;Vertical coordinate=300
WhereHoriz  DC.W      400        ;Horizontal coordinate=400
```

OSType parameter

```
            MOVE.L #'TEXT',-(SP) ;four ASCII chars without a length
                                 ;byte
```

The Structure of Common Record Types

RECORD TYPE	DESCRIPTION	PREDEFINE OFFSETS

QuickDraw Record Structures

```
Point
    DS.W    1     ;Vertical coordinate          v
    DS.W    1     ;Horizontal coordinate        h
Rectangle
    DS.W    1     ;Top coordinate               top,topleft
    DS.W    1     ;Left coordinate              left
    DS.W    1     ;Bottom coordinate            bottom,botright
    DS.W    1     ;Right coordinate             right
BitMap
    DS.L    1     ;Pointer to bit image         baseAddr
    DS.W    1     ;Number of bytes/row          rowBytes
    DS.W    4     ;Enclosing rectangle          bounds
Cursor
    DS.W    16    ;Visible bits (16 by 16)      data
    DS.W    16    ;Covered bits (16 by 16)      mask
    DS.W    2     ;Point of relative origin     hotSpot
```

Event Manager Record Structures

```
Event record
    DS.W    1     ;Type of event                evtNum
    DS.L    1     ;Information about event      evtMessage
    DS.L    1     ;Time when it happened
                  ;(ticks)                      evtTicks
                  ;Mouse point when it
                  ;happened
    DS.W    1     ;Mouse vertical               evtMouse
```

```
DS.W    1       ;Mouse horizontal
DS.B    1       ;Control keys held down        evtMeta
DS.B    1       ;Mouse down & deactivate       evtMBut
```

Some Common Calls Expanded

Window Manager

```
        function FrontWindow:WindowPtr
                                        ;Return a pointer to the top window
        CLR.L -(SP)                     ;Clear space for the window pointer
        __FrontWindow
        MOVE.L (SP)+,theWindow(A5)      ;Pointer to the top window

        procedure SelectWindow (theWindow:WindowPtr);
                                        ;Set the top, active window.
        MOVE.L theWindow(A5),-(SP)
        __SelectWindow

        function GetNewWindow
(windowID:integer;wStorage:Ptr;behind: WindowPtr):WindowPtr;
                                        ;Start up a new window, using window
                                        ;template 300.
        CLR.L       -(SP)               ;Clear space for the result
        MOVE        #300,-(SP)          ;Resource ID number of WIND
                                        ;Push the address of the window's
                                        ;storage area. You can push NIL instead
                                        ;to have the system allocate space for
                                        ;the Window Record
        PEA         wStorage
                                        ;Put this window over all the other
                                        ;windows. You can push zero
                                        ;instead to place this window behind all
                                        ;the others. You can use the
                                        ;pointer to another window to put the
                                        ;new window between
                                        ;that window and all of the windows
                                        ;below.
        MOVE.L      #-1,-(SP)
        __GetNewWindow                  ;Get The Window
        MOVE.L      (SP)+,theWindow(A5)
        ...
wStorage
                                        ;Storage area for the new window.
        DS.B        WindowSize          ;156 bytes
```

```
        procedure
DragWindow (theWindow:WindowPtr;startPt:Point;boundsRect:Rect);
                                                ;The Mouse went down in the Drag area.
                                                ;Let the operator move the window
                                                ;around the desktop. If the window is
                                                ;moved, update events will be
                                                ;generated for all windows whose
                                                ;appearance is affected
        MOVE.L      theWindow(A5),-(SP)         ;Point to window record area
        MOVE.L      startPt(A5),-(SP)           ;Point where mouse went down
        PEA         boundsRect                  ;Limits on operator's motion
        __DragWindow
startPt
                                                ;Mouse location of start of drag in global
                                                ;coordinates
        DS.W        1                           ;vertical location
        DS.W        1                           ;horizontal location
boundsrect
                                                ;Limits of window motion in global
                                                ;coordinates
        DC.W        30                          ;top
        DC.W        5                           ;left
        DC.W        330                         ;bottom
        DC.W        508                         ;right

        function FindWindow
        (thePt:Point; VAR whichWindow: WindowPtr): integer;
                                                ;Identify the window and region that
                                                ;contain a certain point.
        CLR         -(SP)                       ;Clear space for result
        MOVE.L      Where(A5),-(SP)             ;Point in Global Coordinates
        PEA         whichWindow(A5)             ;Area for pointer to window
        __FindWindow
        MOVE        (SP)+,D0                    ;Integer code for the region
                                                ;D0 tells where the point is in the
                                                ;window as follows:
                                                ;If D0 is: The point is:
                                                ;0          in the Desk area and not in
                                                           any application window.
                                                ;1          in the Menu Bar area.
                                                ;2          in a System window and not in
                                                           any application window.
                                                ;3          in the Content region of an
                                                           application window.
```

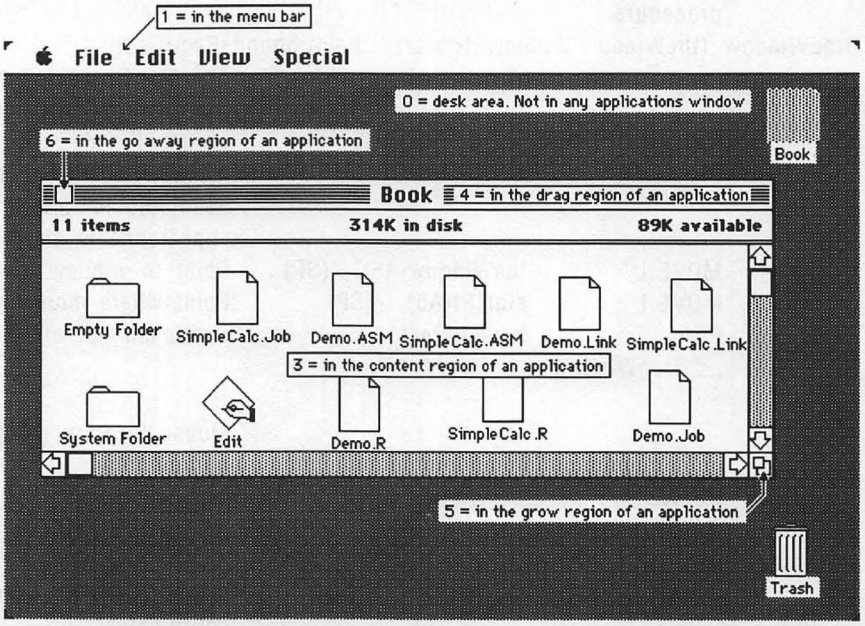


Figure F-1 Window Areas

```
                                    ;4      in the Drag region of an
                                            application window.
                                    ;5      in the Grow region of an
                                            application window.
                                    ;6      in the Go Away region of an
                                            application window.
```

Menu Manager

```
function GetMenu (menuID:integer): MenuHandle;
                                    ;Set up a new menu in memory.
CLR.L           -(SP)               ;Clear space for MenuHandle
                                    ;Load the ID of the menu resource, 257.
                                    ;The Apple menu should be
                                    ;ID number 1. All the other application
                                    ;menus should have ID numbers
                                    ;greater than 256.
MOVE.W          #257,-(SP)
__GetRMenu
MOVE.L          (SP)+,MenuHand(A5); Handle to new menu data

procedure InsertMenu (menu:MenuHandle;before ID: integer);
                                    ;Add a menu to the list of active menus,
                                    ;but don't draw the Menu Bar yet.
```

```
        MOVE.L      MenuHand(A5),-(SP)  ;Load handle to the menu date
                                        ;Put this menu to the left of menu #257
                                        ;in the Menu bar. To put it after
                                        ;all other menus, you would use a zero
                                        ;here instead.
        MOVE.W      #257,-(SP)
        _InsertMenu                     ;Insert the menu in the Menu List.

        procedure AddResMenu
        (theMenu: MenuHandle; theType: ResType);
                                        ;Add all resources of a certain type to a
                                        ;menu
        MOVE.L      AppleHandle(A5),-(SP) ;Handle for the Apple menu
        MOVE.L      # 'DRVR',-(SP)       ;Add the desk accessories.
        _AddResMenu

        function MenuKey (KeyChr:CHAR): LongInt;
                                        ;Identify the menu and menu item
                                        ;corresponding to a command key
        CLR.L       -(SP)               ;Make space for menu selection.
        MOVE.W      KeyChr(A5),-(SP)    ;Push the character.
        _MenuKey
        MOVE.W      (SP)+,MenuID(A5)    ;Get the ID of the chosen menu
                                        ;If a zero is returned for the Menu ID,
                                        ;the key does not correspond to an
                                        ;enabled menu item.
        BEQ         NullEvent           ;No menu item. Exit.
        MOVE.W      (SP)+,MenuItem(A5)  ;Get the chosen item.
        ...
KeyChr  DC.B        0, 'A'              ;Key character in lower byte

        function MenuSelect (MDownPt:Point):LongInt;
                                        ;The mouse has gone down in the Menu
                                        ;Bar. Track it until it
                                        ;is released. Then return the Menu and
                                        ;the item selected
        CLR.L       -(SP)               ;Space for menu choice
        MOVE.L      MDownPt,-(SP)       ;Where mouse went down in Menu Bar
        _MenuSelect
                                        ;_MenuSelect will show the menu until
                                        ;the mouse button is released.
        MOVE.W      (SP)+,MenuID(A5)    ;Get the ID of the chosen menu
                                        ;If a zero is returned for the Menu ID
                                        ;ignore the result. Either the item
                                        ;belonged to a Desk accessory or the
                                        ;operator did not release the mouse in
                                        ;an enabled menu item.
```

```
            BEQ             NullEvent               ;No menu item. Exit
            MOVE.W          (SP)+,MenuItem(A5)      ;Get the chosen item

            procedure HiLiteMenu (menuID: integer);
                                                    ;Mark a menu by darkening its title in
                                                    ;the Menu Bar. Only one
                                                    ;menu can be highlighted at a time. Use
                                                    ;zero for the menu ID to
                                                    ;unhighlight the menu bar.
            MOVE.W          #menuID,-(SP)
            _HiLiteMenu

            procedure GetItem
            (menu: MenuHandle;item: integer; VAR itemString: Str255);
                                                    ;Put the text of a menu item into a
                                                    ;Pascal string
            MOVE.L          MenuHand(A5),-(SP)      ;Menu handle
            MOVE.W          Item,-(SP)              ;Number of item
            PEA             itemString              ;Pointer to area for Pascal string
            _GetItem
                                                    ;Although the string must be 255 bytes
                                                    ;long in Pascal, only as many
                                                    ;bytes as are actually used by the name
                                                    ;of the item will be clobbered.
itemString  DS.B            40
```

Event Manager

```
            procedure FlushEvents (eventMask,stopMask: integer);
                                                    ;Remove events that match eventMask
                                                    ;from the Event Queue. Keep
                                                    ;checking and removing events until an
                                                    ;event matches stopMask.
            MOVE.W          #StopMask,D0            ;Event types to stop
            SWAP            D0
            MOVE.W          #EventMask,D0           ;Event types to remove
            _FlushEvents
                                                    ;to remove all events in the queue use:
StopMaskEQU 0                                       ;No event can stop it
EventMask EQU $FFFF                                 ;Remove all events

            function        GetNextEvent
(eventMask: integer; VAR Event: EventRecord):Boolean;
                                                    ;Take the next event from the queue
            CLR.W           -(SP)                   ;Space for true/false result
                                                    ;Pass negative one to accept any type of
                                                    ;event.
```

```
        MOVE.W      #-1,-(SP)       ;Event mask
        PEA         Event(A5)       ;Pointer to storage area for the event.
        _GetNextEvent
        BTST        #0,(SP)+        ;Was an event returned?
        BEQ         NullEvent       ;No. Return to the main program.
        ...
Event   DS.W        8               ;Storage for event record.
```

Desk Manager

```
function OpenDeskAcc (theAcc: Str255):integer;
                                    ;Start a desk accessory. The reference
                                    ;number returned could be used
                                    ;to close the desk accessory if that is
                                    ;ever necessary.
        CLR.W       -(SP)           ;Clear space for the reference number.
        PEA         'Clock'         ;Pass a Pascal string.
        _OpenDeskAcc
        MOVE.W      (SP)+,RefNo(A5) ;Reference number of accessory

function SystemEdit (editCmd:integer): Boolean;
                                    ;Let a desk accessory try to perform an
                                    ;edit action, if any is selected.
                                    ;Return True if the action was performed.
                                    ;Return False if the command
                                    ;is for the application.
        CLR.W       -(SP)           ;Space for result
        MOVE.W      editCmd(A5),D0  ;Get the action to perform
                                    ;The Edit Commands passed to the
                                    ;system are as follows:
                                    ;If D0 is: the command is:
                                    ;2          Cut.   Delete the selected data, but
                                                       save it on the Clipboard.
                                    ;3          Copy.  Copy the selected data to the
                                                       Clipboard.
                                    ;4          Paste. Copy the Clipboard data to
                                                       the selected area.
                                    ;5          Clear. Delete the selected data, but
                                                       don't alter the Clipboard.
                                    ;0          Undo.  Reverse the previous Cut,
                                                       Copy or Paste.
        MOVE.W      D0,-(SP)        ;Load the action.
        _SysEdit
        BTST        #0,(SP)+        ;Check the result.
        BNE         TRUE            ;The desk accessory performed the
                                    ;action.
        BEQ         FALSE           ;The command is for the application.
```

Dialog Manager

```
            procedure InitDialogs (restart:ProcPtr);
                                                ;Initialize the Dialog Manager. The restart
                                                ;procedure will be called in
                                                ;case of a serious system error, such as
                                                ;running out of memory. If you
                                                ;do not want a restart procedure, pass
                                                ;NIL instead.
            CLR.L           -(SP)               ;There is no Restart subroutine.
            __InitDialogs

            function        GetNewDialog
            (dialogID: integer, dStorage: Ptr; behind: WindowPtr);DialogPtr;
                                                ;Start a dialog box using the DLOG
                                                ;template #300.
            CLR.L           -(SP)               ;Space for the new dialog pointer.
            MOVE            #300,-(SP)          ;ID of DLOG in resource file
                                                ;Load a pointer to the space for the
                                                ;dialog record. Pass NIL to have the
                                                ;system allocate space for the dialog
                                                ;record.
            PEA             DStorage            ;Storage Area
                                                ;Pass-1 to make this dialog the top
                                                ;window. You can pass the pointer
                                                ;of another window or dialog instead, to
                                                ;create the new dialog below.
            MOVEQ           #-1,D0              ;Put dialog on top of all other windows.
            MOVE.L          D0,-(SP)            ;Pointer of window above the dialog
            __GetNewDialog
            MOVE.L          (SP)+,DiaPtr(A5)    ;new Dialog Pointer
            ...
DStorage
                                                ;Storage area for the new dialog.
            DS.B            dWindLen            ;170 bytes

            procedure ModalDialog
            (filterProc: ProcPtr; VAR itemHit: integer);
                                                ;Lock the Operator into a dialog box until
                                                ;he selects something or
                                                ;types a character into an Edit Field.
                                                ;Pass NIL to indicate that there is not
                                                ;filter routine. If you have a filter
                                                ;routine, pass its address instead.
            CLR.L           -(SP)
```

```
            PEA         ItemHit             ;Pointer to a word for the chosen item
            __ModalDialog
            ...
ItemHit
                                            ;Number of the chosen dialog item
            DC.W        1                   ;First item for default
```

Resource Manager

```
            function OpenResFile (fileName: str255):integer;
                                            ;Open a resource file. Resources will be
                                            ;taken from this file
                                            ;before any others.
            CLR         -(SP)               ;Clear space for ref num result
                                            ;Push a pointer to the file name in a
                                            ;Pascal string. The file name
                                            ;can have a volume name and folder
                                            ;names also.
            PEA         'SimpleCalc.Rsrc'
            __OpenResFile
            MOVE.W      (SP)+,RefNo(A5)     ;Save the reference number
```

QuickDraw

```
            procedure InitGraf (globalPtr:Ptr);
            PEA         -4(A5)
            __InitGraf

            procedure SetPort (gp:GrafPort);
                                            ;Make theWindow the current port. A
                                            ;grafport starts the window record
            MOVE.L      theWindow(A5),-(SP)
            __SetPort

            procedure SetOrigin (h,v:integer);
                                            ;Set the top left corner of the current
                                            ;window
            MOVE.W      h(A5),-(SP)
            MOVE.W      v(A5),-(SP)
            __SetOrigin

            procedure ClipRect (r:Rect);
            PEA         LimitBox
            __ClipRect
            ...
```

```
LimitBox
                                        ;All drawing calls will only affect what is
                                        ;inside this rectangle.
        DC.W    50                      ;top
        DC.W    100                     ;left
        DC.W    300                     ;bottom
        DC.W    400                     ;right

        procedure PenSize(width,height: integer);

                                        ;Set the pen shape to a rectangle, three
                                        ;pixels tall and one pixel wide.
        MOVE.W  #1,-(SP)                ;Pen width
        MOVE.W  #3,-(SP)                ;Pen height
        _PenSize

        procedure PenMode(mode: integer);

                                        ;Set the transfer mode for graphic
                                        ;drawing to "copy" mode.
        MOVE.W  #patCopy,-(SP)          ;Both black & white will cover
        _PenMode

        procedure PenPat(pat: Pattern);

                                        ;Set the pen pattern for graphic drawing
                                        ;to dark gray.
        MOVE.L  GrafGlobals(A5),A0      ;Pointer to QuickDraw globals
        PEA     DkGray(A0)              ;Dark-gray standard pattern
        _PenPat

        procedure PenNormal;

                                        ;Set the pen to the default state, black,
                                        ;copy mode, and one pixel square.
        _PenNormal

        procedure BackPat (pat:Pattern);

                                        ;Set the background pattern to light gray.
                                        ;Get the pointer to the QuickDraw globals
        MOVE.L  GrafGlobals(A5),A0

                                        ;Pass a pointer to the light-gray standard
                                        ;pattern.
        PEA     LtGray(A0)              ;Pointer to pattern
        _BackPat
```

Cursor Routines

```
        procedure InitCursor;

                                        ;Set the cursor to an arrow
        _InitCursor
```

```
        procedure HideCursor;
                                        ;Make the cursor invisible
        __HideCursor
        procedure ShowCursor;
                                        ;Make the cursor visible
        __ShowCursor
        procedure SetCursor(crsr:Cursor);
                                        ;Set the cursor to a hand pointing to the
                                        ;right.
        PEA         crsr
        __SetCursor
        ...

crsr                                    ;A cursor showing a hand pointing to the
                                        ;right.
                                        ;The first 16 words are the data
        DC.W        $0000               ;binary 0000000000000000
        DC.W        $00C0               ;       0000000011000000
        DC.W        $0320               ;       0000001100100000
        DC.W        $0C20               ;       0000110000100000
        DC.W        $1840               ;       0001100001000000
        DC.W        $60FF               ;       0110000011111111
        DC.W        $4181               ;       0100000110000001
        DC.W        $82FF               ;       1000001011111111
        DC.W        $8510               ;       1000010100010000
        DC.W        $81F0               ;       1000000111110000
        DC.W        $8110               ;       1000000100010000
        DC.W        $81F0               ;       1000000111110000
        DC.W        $8110               ;       1000000100010000
        DC.W        $81F0               ;       1000000111110000
        DC.W        $C100               ;       1100000100000000
        DC.W        $7F00               ;       0111111100000000
                                        ;The next 16 words are the mask
        DC.W        $0000               ;binary 0000000000000000
        DC.W        $00C0               ;       0000000011000000
        DC.W        $03E0               ;       0000001111100000
        DC.W        $0FE0               ;       0000111111100000
        DC.W        $1FC0               ;       0001111111000000
        DC.W        $7FFF               ;       0111111111111111
        DC.W        $7FFF               ;       0111111111111111
        DC.W        $FFFF               ;       1111111111111111
        DC.W        $FFF0               ;       1111111111110000
        DC.W        $FFF0               ;       1111111111110000
        DC.W        $FFF0               ;       1111111111110000
```

```
DC.W        $FFF0           ;       1111111111110000
DC.W        $FFF0           ;       1111111111110000
DC.W        $FFF0           ;       1111111111110000
DC.W        $FF00           ;       1111111100000000
DC.W        $7F00           ;       0111111100000000
                            ;The last two words are the hot-spot
                            ;point
DC.W        $0007           ;vertical coordinate
DC.W        $0015           ;horizontal coordinate
```

Line Routines

```
procedure MoveTo (h,v: integer);
                            ;Move the pen to the point, (400,300).
MOVE.W      #400,-(SP)      ;Load horizontal coordinate
MOVE.W      #300,-(SP)      ;Load vertical coordinate
_MoveTo

procedure Move (dh,dv: integer);
                            ;Move the pen 40 pixels right and 30
                            ;pixels down
MOVE.W      #40,-(SP)       ;Load horizontal offset
MOVE.W      #30,-(SP)       ;Load vertical offset
_Move

procedure LineTo (h,v: integer);
                            ;Draw a line from current location to the
                            ;point, (100,200).
MOVE.W      #100,-(SP)      ;Load horizontal coordinate
MOVE.W      #200,-(SP)      ;Load vertical coordinate
_LineTo

procedure Line (dh,dv: integer);
                            ;Draw a line left 20 and down 30
MOVE.W      #-20,-(SP)      ;Load horizontal offset
MOVE.W      #30,-(SP)       ;Load vertical offset
_Line
```

Text Routines

```
procedure TextFont (font: integer);
MOVE.W      #Cairo,-(SP)    ;Font with pictures
_TextFont

procedure TextFace (face: Style);
                            ;Set the bits in D0 for bold and italic
                            ;text.
```

```
        CLR.W       D0
        BSET        #boldBit,D0         ;Set $01 for bold.
        BSET        #italicBit,D0       ;Set $02 for italics.
        MOVE.W      D0,-(SP)            ;Bold & Italic
        __TextFace

        procedure TextMode(mode: integer);
                                        ;Set the mode for text drawing. Graphics
                                        ;drawing is not affected.
        MOVE.W      #srcXor,-(SP)
        __TextMode

        procedure TextSize (size: integer);
        MOVE.W      #12,-(SP)           ;Character height in pixels
        __TextSize

        procedure DrawChar(ch:char);
                                        ;Draw one character on the screen.
        MOVE.W      ch,-(SP)            ;Push character on the stack.
        __DrawChar
        ...
                                        ;The character is in the second byte of
                                        ;the word.
ch      DC.B        0, 'A'              ;Character "A" in lower byte.

        procedure DrawString (s: Str255);
        PEA         'Hello from Assembly language'
        __DrawString

        function CharWidth(ch: CHAR): integer;
        CLR.W       -(SP)               ;Clear space for result
        MOVE.W      ch,-(SP)            ;Character to measure
        __CharWidth
                                        ;Return the width of "A" in pixels in
                                        ;D0.
        MOVE.W      (SP)+,D0
        ...
ch      DC.B        0, 'A'              ;Character in lower byte.

        function StringWidth (s: Str255): integer;
        CLR.W       -(SP)               ;Clear space for result
        PEA         'Hello from Assembly language'
        __StringWidth
                                        ;Return the width of the string in D0.
        MOVE.W      (SP)+,D0
```

Coordinate System Conversions

```
procedure GlobalToLocal (VAR pt:Point);
                                ;Convert the screen coordinates of a
                                ;point to coordinates based on the
                                ;current window.
PEA         pt(A5)              ;Pointer to input and output
_GlobalToLocal                  ;Global to local

procedure LocalToGlobal (VAR pt: Point);
                                ;Convert the coordinates of a point in the
                                ;current window to
                                ;global coordinates.
PEA         pt(A5)              ;Pointer to input and output
_LocalToGlobal                  ;Local to global
```

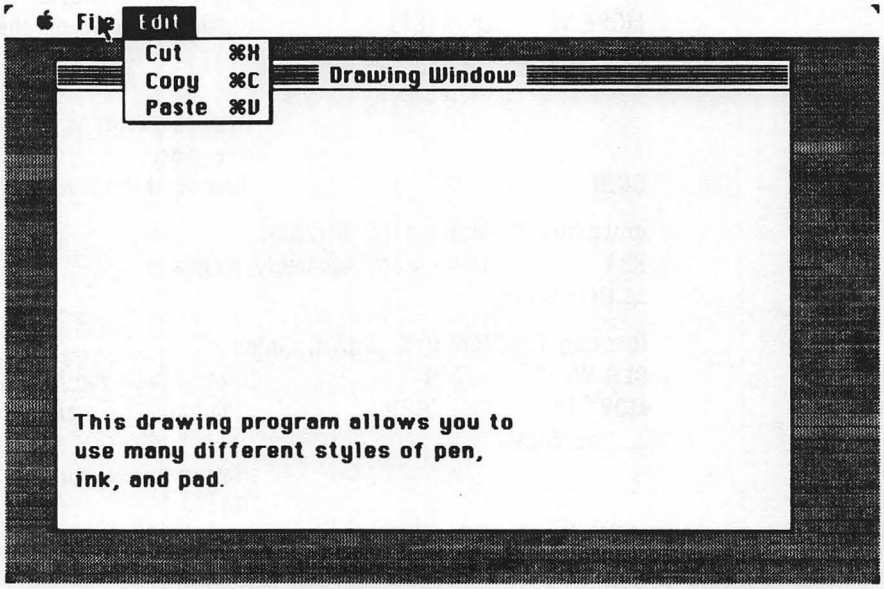


Figure F-2 Global versus Local Coordinates

Rectangle Routines

```
procedure FrameRect (r: Rect);
PEA         Box
_FrameRect
...
```

```
Box
        DC.W        50              ;top
        DC.W        100             ;left
        DC.W        300             ;bottom
        DC.W        400             ;right

        procedure PaintRect (r: Rect);
        PEA         r(A5)
        __PaintRect
```

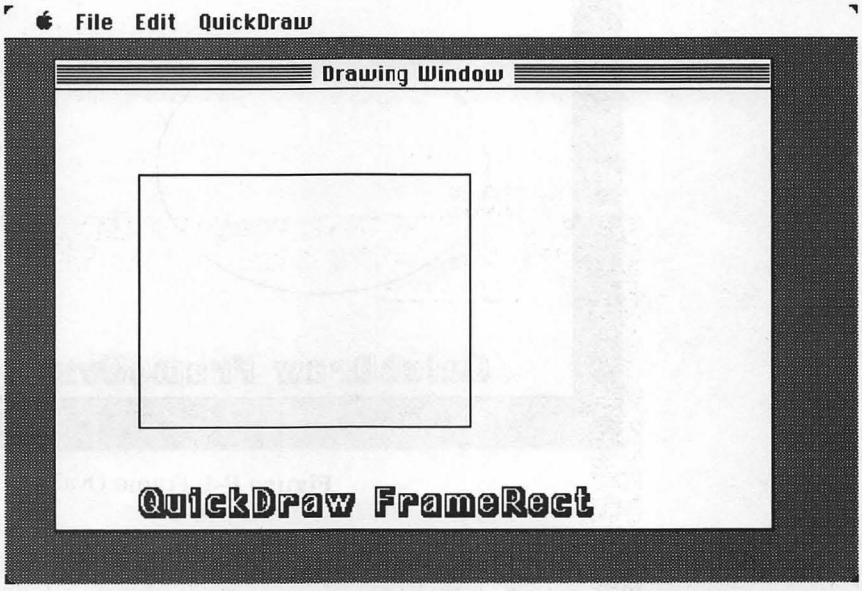


Figure F-3 Frame Rectangle

```
procedure EraseRect (r: Rect);
PEA         r(A5)
__EraseRect

procedure InvertRect (r: Rect);
PEA         r(A5)
__InverRect

procedure FillRect (r: Rect; pat: Pattern);
PEA         r(A5)
MOVE.L      GrafGlobals(A5),A0  ;Pointer to QuickDraw globals
PEA         Black(A0)           ;Pointer to standard black pattern
__FillRect                      ;Solid, black rectangle
```

Oval Routines

```
procedure FrameOval (r: Rect);
PEA         r(A5)
_FrameOval
```

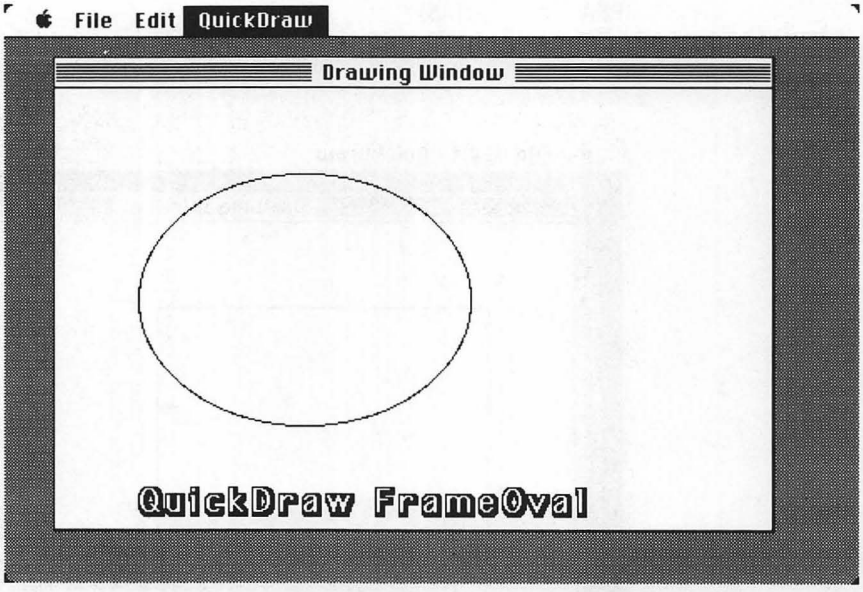


Figure F-4 Frame Oval

```
procedure PaintOval (r: Rect);
PEA         r(A5)
_PaintOval

procedure EraseOval (r: Rect);
PEA         r(A5)
_EraseOval

procedure InvertOval (r: Rect);
PEA         r(A5)
_InvertOval

procedure FillOval (r: Rect; pat: Pattern);
PEA         r(A5)
MOVE.L      GrafGlobals(A5),A0   ;Pointer to QuickDraw globals
PEA         Black(A0)            ;Pointer to standard black pattern
_FillOval; Draw a solid black ellipse
```

Round Rectangle Routines

```
procedure FrameRoundRect (r: Rect;ovWd,ovHt: integer);
PEA         r(A5)                ;Dimensions of smooth sides
MOVE.W      ovWd(A5),-(SP)       ;Width or corner ellipse
MOVE.W      ovHt(A5),-(SP)       ;Height of corner ellipse
__FrameRoundRect                 ;Outline a rectangle with rounded corners
```

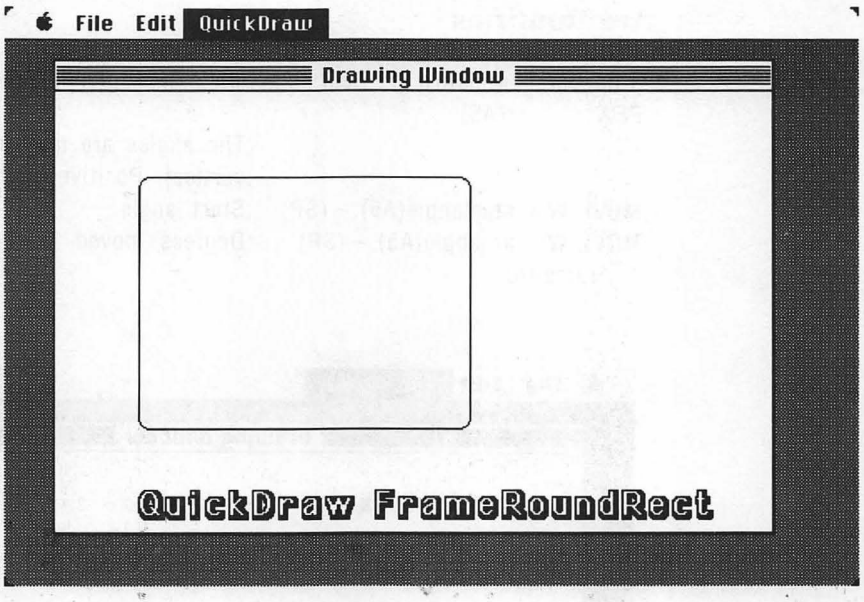


Figure F-5 Frame Round Rect

```
procedure PaintRoundRect        (r:Rect; ovWd,ovHt: integer);
PEA         r(A5)               ;Dimensions of smooth sides
MOVE.W      ovWd(A5),-(SP)      ;Width of corner ellipse
MOVE.W      ovHt(A5),-(SP)      ;Height of corner ellipse
__PaintRoundRect

procedure EraseRoundRect (r: Rect; ovWd,ovHt,ovHt: integer);
PEA         r(A5)               ;Dimensions of smooth sides
MOVE.W      ovWd(A5),-(SP)      ;Width of corner ellipse
MOVE.W      ovHt(A5),-(SP)      ;Height of corner ellipse
__EraseRoundRect

procedure InvertRoundRect (r: Rect; ovWd,ovHt: integer);
PEA         r(A5)               ;Dimensions of smooth sides
MOVE.W      ovWd(A5),-(SP)      ;Width of corner ellipse
MOVE.W      ovHt(A5),-(SP)      ;Height of corner ellipse
__InverRoundRect
```

```
procedure FillRoundRect
(r:Rect; ovWd,ovHt: integer; pat: Pattern);
PEA       r(A5)                  ;Dimensions of smooth sides
MOVE.W    ovWd(A5),-(SP)         ;Width of corner ellipse
MOVE.W    ovHt(A5),-(SP)         ;Height of corner ellipse
MOVE.L    GrafGlobals(A5),A0     ;Pointer to QuickDraw globals
PEA       Black(A0)              ;Pointer to standard black pattern
_FillRoundRect                   ;Solid, black, rounded-corner rectangle
```

Arc Routines

```
procedure FrameArc (r: Rect; startAngle,arcAngle: integer);
PEA       r(A5)
                                 ;The angles are measured from the
                                 ;vertical. Positive angles are clockwise.
MOVE.W    startangle(A5),-(SP)   ;Start angle
MOVE.W    arcangle(A5),-(SP)     ;Degrees moved
_FrameArc
```

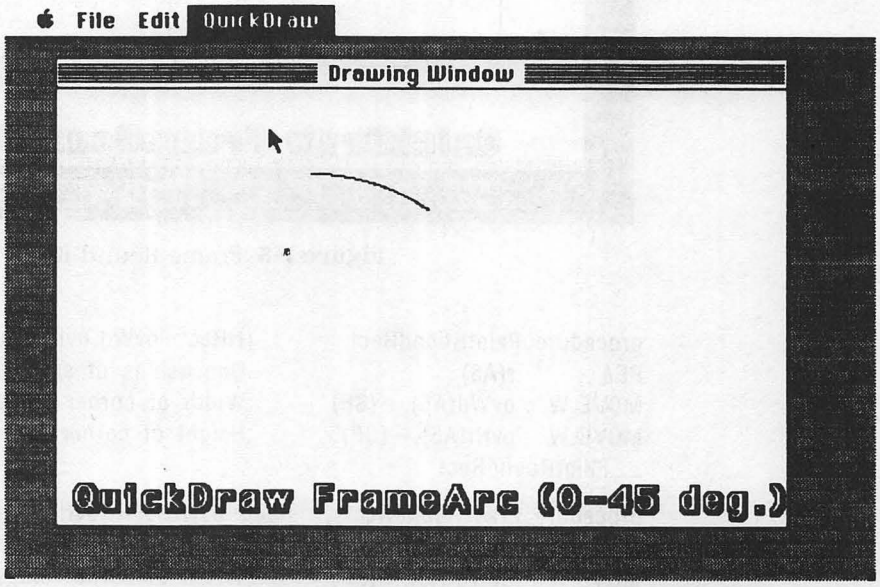


Figure F-6 Frame Arc

```
procedure PaintArc (r; Rect; startAngle,arcAngle: integer);
PEA       r(A5)
MOVE.W    startangle(A5),-(SP)
```

```
MOVE.W    arcangle(A5),-(SP)
_PaintArc

procedure EraseArc (r: Rect; startAngle,arcAngle: integer);
PEA       r(A5)
MOVE.W    startangle(A5),-(SP)
MOVE.W    arcangle(A5),-(SP)
_EraseArc

procedure InvertArc (r: Rect; startAngle,arcAngle: integer);
PEA       r(A5)
MOVE.W    startangle(A5),-(SP)
MOVE.W    arcangle(A5),-(SP)
_InvertArc

procedure FillArc
r: Rect; startAngle,arcAngle: integer;pat:Patern);
PEA       r(A5)
MOVE.W    startangle(A5),-(SP)
MOVE.W    arcangle(A5),-(SP)
MOVE.L    GrafGlobals(A5),A0      ;Pointer to QuickDraw globals
PEA       Black(A0)               ;Pointer to standard black pattern
_FillArc
```

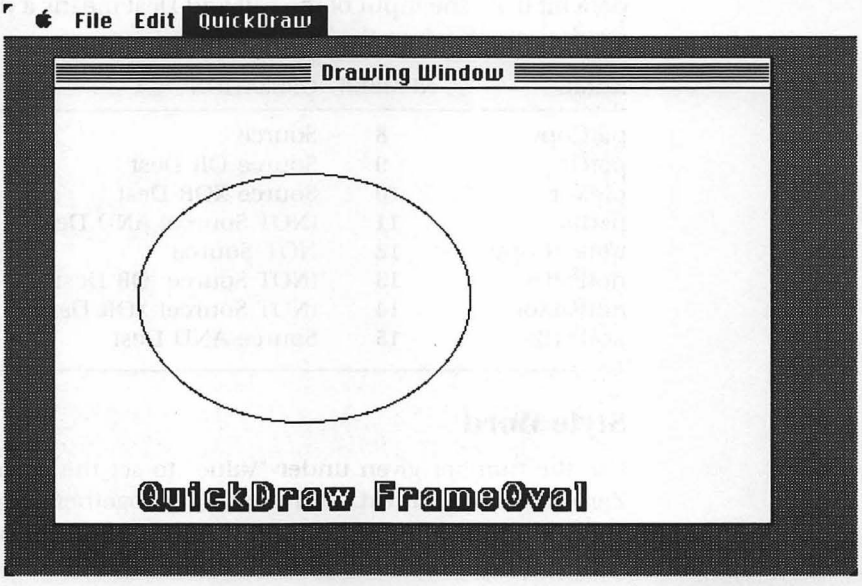


Figure F-7 Frame Oval

Most Common QuickDraw Data Definitions

Source Copy Modes

Use the Source copying modes for drawing text or transfering bit images. The effect of the operation is described using "Source" and "Dest," where Source means a data bit from the input of the call and Dest means a data bit that is already on the screen when the call is made.

NAME	NUMBER	OPERATION
srcCopy	0	Source
srcOr	1	Source OR Dest
srcXor	2	Source XOR Dest
srcBic	3	(NOT Source) AND Dest
notSrcCopy	4	NOT Source
notSrcOr	5	(NOT Source) OR Dest
notSrcXor	6	(NOT Source) XOR Dest
notSrcBic	7	Source AND Dest

Pattern Copy Modes

Use pattern copying modes for setting the pen pattern. The effect of the operation is described using "Source" and "Dest," where Source means a data bit from the input of the call and Dest means a data bit that is already on the screen when the call is made.

NAME	NUMBER	OPERATION
patCopy	8	Source
patOr	9	Source OR Dest
patXor	10	Source XOR Dest
patBic	11	(NOT Source) AND Dest
notPatCopy	12	NOT Source
notPatOr	13	(NOT Source) OR Dest
notPatXor	14	(NOT Source) XOR Dest
notPatBic	15	Source AND Dest

Style Word

Use the number given under "Value" to set the corresponding text style. Zero means plain text. Add the values together for combinations of text style. The symbols can be used with the BSET instruction.

SYMBOL	BIT NUMBER	VALUE	STYLE
boldBit	0	$01	bold
italicBit	1	$02	italicized
ulineBit	2	$04	underlined
outlineBit	3	$08	outlined
shadowBit	4	$10	shadowed
condenseBit	5	$20	condensed
extendBit	6	$40	expanded

Font Numbers

NAME	NUMBER	COMMENT
sysFont	0	Code for current system font
applFont	1	Code for current application font
NewYork	2	
Geneva	3	Default application font
Monaco	4	Monospaced
Venice	5	
London	6	Old English
Athens	7	
SanFran	8	Kidnapper's font
Toronto	9	
Cairo	11	Hieroglyphics. All pictures
LosAngel	12	

Offsets to QuickDraw Global Variables

NAME	OFFSET	TYPE
thePort	$0	GrafPtr
white	$FFFFFFF8	Pattern
black	$FFFFFFF0	Pattern
gray	$FFFFFFE8	Pattern
ltGray	$FFFFFFE0	Pattern
dkGray	$FFFFFFD8	Pattern
arrow	$FFFFFF94	Cursor
screenBits	$FFFFFF86	BitMap
randSeed	$FFFFFF82	long integer

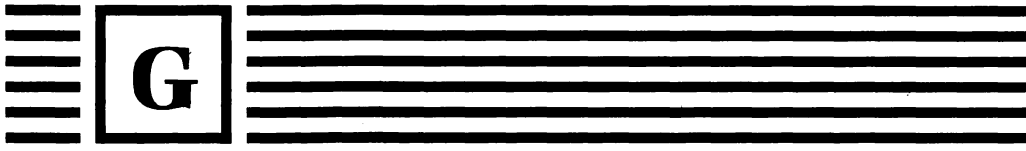
Event Types

NAME	NUMBER	CAUSE
nullEvt	0	null event or no events happened
mButDwnEvt	1	mouse button pushed down
mButUpEvt	2	mouse button released
keyDwnEvt	3	key pushed down
keyUpEvt	4	key released
autoKeyEvt	5	key held down to repeat
updatEvt	6	a window needs to be redrawn
diskInsertEvt	7	a disk was inserted
activateEvt	8	activate/deactive event
abortEvt	9	abort event
netWorkEvt	10	network event
ioDrvrEvt	11	a driver-defined event
app1Evt	12	an application-defined event
app2Evt	13	an application-defined event
app3Evt	14	an application-defined event
app4Evt	15	an application-defined event

Bits of the Modifier Word

SYMBOL	BIT NUMBER	VALUE	MEANING
activeFlag	0	$0001	activate
changeFlag	1	$0002	change to system window
btnState	7	$0080	mouse button up!
cmdKey	8	$0100	command key down
shiftKey	9	$0200	shift key down
alphaLock	10	$0400	alpha lock down
optionKey	11	$0800	option key down

APPENDIX



SimpleCalc Program Code

Assembler File, SimpleCalc.ASM

```
;---------------------------------------------------------------------------
; SimpleCalc  - A Simplified SpreadSheet Example by Harland Harrison
; and Ed Rosenzweig
; Other files needed are
;     SimpleCalc.R          Resource source
;     SimpleCalc.Job        Exec file
;     SimpleCalc.Link       Linkage list file
;------------------------------- INCLUDE ------------------------------
Include      MacTraps.D           ; Include equates and traps files
Include      ToolEqu.D
Include      QuickEqu.D
Include      SysEqu.D
;---------------------------- LOCAL DATA --------------------------------
DataSize   EQU     4110           ; Space needed for variables
Clip       EQU     4102           ; Clipboard for editing cells
ItemHit    EQU     4104           ; Dialog item chosen
AppleHand  EQU     4106           ; Handle for Apple menu
AccCell    EQU     127            ; Cell number of accumulator
; Stored spread sheet program format
;   30  bytes for each cell
;   $80 bit means select a cell. Otherwise arithmetic operation
;   $00 means end of program
Prg        EQU     2                   ; Offset to start of program in cell
ProgLast   EQU     29                  ; Maximum entries. Last byte must be 0
; Flag bits stored in Flag register, D5
FrontFlag  EQU     2                   ; Bit set if our window selected
Redraw     EQU     1                   ; Redraw cell bit. =  overflow in SR
QuitFlag   EQU     0                   ; Exit program bit. = carry in SR
;------------------------------ GLOBAL DATA ----------------------------
DeskName           DS     16          ; Desk accessory's name
WindowStorage       DS.B  WindowSize  ; Storage for window
;---------------------------- REGISTER USAGE ---------------------------
;
;          MainProgram            Inititalization
;     D4   Selected Cell          Menu ID
;     D5   Flag register          Menu handle
;     D6   Program byte counter
;     D7   Cell to draw or calculate
```

```
;
;
;     A3    Selected Cell
;     A4    Cell being calculated
;     A5    Global Variables
;     A6    Variable Base/ Accumulator
;
;------------------------------ MAIN PROGRAM -----------------------------

      LINK   A6,#-DataSize          ; Make space for spread sheet
      LEA    -DataSize(A6),A6       ; Address memory from low end
      BSR    InitMain               ; Initialize
MainLoop
      BSR    Calculate              ; Calculate values
      BSR    GetEvent
      BCS    MainExit               ; Carry Set is signal to quit
      BVC    MainLoop               ; Overflow is signal to redraw
      BSR    DrawSelect
      BRA    MainLoop
MainExit
      LEA    DataSize(A6),A6        ; Point to top of memory
      UNLK   A6                     ; Return memory to stack space
      RTS

;------------------------------ INITIALIZE -------------------------------

InitMain                            ; Main initialization
; Clear data area
      MOVE.L   A6,A0                ; Point to start of area
      MOVE.W   #DataSize,D0         ; Byte count to clear
      BRA      InitChk              ; DBRA counts to -1
InitLoop
      CLR.B    (A0)+                ; Clear one byte of memory
InitChk
      DBRA     D0,InitLoop          ; Repeat DataSize times
;Initialize Managers
      PEA      -4(A5)               ; System QD port
      _InitGraf                     ; Init Quickdraw, ^ Grafport
      _InitFonts                    ; Init Font Manager
      _TEInit                       ; Init Text Editor
      _InitWindows                  ; Init Window Manager
      CLR.L    -(SP)                ; No restart procedure
      _InitDialogs                  ; Init Dialog Manager
      _InitMenus                    ; Init Menu Manager
;Open the Resource File
      CLR      -(SP)                ; Clear space for reference number result
      PEA    'SimpleCalc.Rsrc'      ; [Volume: ]Resource Name
      _OpenResFile                  ; File-name string -> Reference Number
      MOVE     (SP)+,D0             ; Get rid of the Ref Num
;Set Up Apple Menu
      MOVEQ    #1,D4                ; Resource ID 1
      JSR      MakeMenu
      MOVE.L   D5,AppleHand(A6)     ; Save Apple menu handle
;Set Up File Menu
      MOVE.W   #303,D4              ; Resource ID 303
      JSR      MakeMenu
;Set Up Edit Menu
      MOVE.W   #302,D4              ; Resource ID 302
      JSR      MakeMenu
;Add Desk Accessories To Apple Menu
      MOVE.L   AppleHand(A6),-(SP); Apple menu handle
      MOVE.L   #'DRVR',-(SP)        ; Accessory resource type
      _AddResMenu                   ; MenuHandle,Type
;Draw the completed Menu Bar
      _DrawMenuBar
```

```
;Initialize the Window
      CLR.L     -(SP)               ; Make space for the window-pointer result
      MOVE      #301,-(SP)          ; Resource ID #301
      PEA       WindowStorage(A5)   ; Push address for window-data storage
      MOVE.L    #-1,-(SP)           ; Put window on top of any other windows
      _GetNewWindow                 ; ID,^ Storage, ^ window above -> ^ window
      LEA       WindowPointer,A0
      MOVE.L    (SP)+,(A0)          ; Save the window pointer in memory
      BSR       SelWindow           ; Make it the top window
; Empty event queue of old keystrokes and mouse clicks
      CLR.L     D0                  ; No type of event stops flush
      MOVE.W    #$FFFF,D0           ; Flush any type of event
      _FlushEvents                  ; Stop, Event in D0
      _InitCursor                   ; Initialize the cursor into an arrow
; Clear registers
      CLR.L     D6                  ; No Accumulator program
      CLR.L     D7                  ; Start update from cell 0
      CLR.L     D5                  ; Clear flags
      CLR.L     D4                  ; Select cell 0
; Draw spread sheet in window and exit InitMain
      BRA       DrawInside          ; Draw sheet of zeros

MakeMenu
;Install Menu
;   Input  D4 = Menu ID
;   Output D5 = Menu Handle
;          D4 = Menu ID
      CLR.L     -(SP)               ; Clear space for menu handle
      MOVE      D4,-(SP)            ; Resource ID input in D4
      _GetRMenu                     ; MenuID -> MenuHandle
      MOVE.L    (SP),D5             ; Return in D5 & leave on stack
      CLR.W     -(SP)               ; Put menu after all others
      _InsertMenu                   ; MenuHandle,BeforeID
      RTS

;------------------------- USER INTERFACE -------------------------

GetEvent                            ; Get next event
      _SystemTask                   ; Update desk accessories
      CLR       -(SP)               ; Clear space for result
      MOVE      #-1,-(SP)           ; Mask to accept all events
      PEA       CurrentEvent        ; Pointer to event record
      _GetNextEvent                 ; Mask, ^ event record -> TRUE if any events
      BTST      #0,(SP)+            ; Event returned?
      BEQ       NullEvent           ; No event.  Return to main program
      BSR       DoEvent             ; Respond to event
      BRA       GetEvent            ; Check for more events in queue

DoEvent                             ; Process event
      MOVE      What,D0             ; Type of event
      CMPI      #mButDwnEvt,D0      ; mouse button down is event 1
      BEQ       MouseDown
      CMPI      #keyDwnEvt,D0       ; key down is event 3
      BEQ       KeyDown
      CMPI      #autoKeyEvt,D0      ; auto-repeated key is event 5
      BEQ       KeyDown
      CMPI      #updatEvt,D0        ; update display is event 6
      BEQ       UpDate
      CMPI      #activateEvt,D0     ; activate/deactive is event 8
      BEQ       Activate
      CMPI      #9,D0               ; abort is event 9
      BEQ       Quit
NullEvent                           ; no event. check quit flag & exit
      MOVE      D5,CCR              ; Set carry if $01 set by
      RTS                           ;  quit command
```

```
;--------------------------- EVENT TYPES --------------------------

Activate
      BSR    WindChk              ; Is it our window?
      BNE    NullEvent            ; No. Ignore event
      BTST   #0,Modify+1          ; Activate or Deactivate event flag
      BEQ    Deactive             ; Activate if $01 set, else DeActivate
      BSET   #FrontFlag,D5        ; Remember window is active
      BRA    DrawSelect           ; Highlight selected cell
Deactive
      BCLR   #FrontFlag,D5        ; Remember window is inactive
      BRA    DrawSelect           ; Un Highlight selected cell

Update                            ; Our VISible ReGioN changed
      BSR    WindChk              ; Is it our window?
      BNE    NullEvent            ; No. Ignore event
      MOVE.L WindowPointer,-(SP)     ; Pointer to window being updated
      _BeginUpDate                ; Start drawing in Update Region
      BSR    DrawInside           ; Only part needing update actually drawn
      MOVE.L WindowPointer,-(SP) ; Pointer to window whose update is finished
      _EndUpdate                  ; Return to normal mode drawing
      RTS

KeyDown                           ; User pressed a key
      MOVE   Modify,D0            ; Modifier word maps shift keys
      BTST   #8,D0                ; If the Command key is down
      BNE    CommandKey           ;  it is a menu event
      BRA    KeyStroke            ; Spread-Sheet data

CommandKey                        ; Command key event. Short-cut menu choice
      CLR.L  -(SP)                ; Get space for menu choice
      MOVE   Message+2,-(SP)      ; Put message byte on stack
      _MenuKey                    ;Identify Key. ASCII -> MenuID,MenuItem
      BRA    MenuCommand          ; Menu ID & Item now on stack

MouseDown                         ; Find where mouse clicked
      CLR    -(SP)                ; Clear space for integer result
      MOVE.L Where,-(SP)          ; Mouse at time of GetNextEvent
      PEA    EvtWind              ; Window the event was in
      _FindWindow                 ; click point, ^ window -> window part code
      MOVE   (SP)+,D0             ; Result = section of window
      ASL.W      #1,D0            ; Window Part * 2 Bytes/Entry
      MOVE   WindowTable(D0),D0   ; Get offset from table
      JMP    WindowTable(D0)      ; Call subroutine

WindowTable

      DC.W   NullEvent-WindowTable    ; In Desk
      DC.W   InMenu-WindowTable       ; In Menu Bar
      DC.W   SystemEvent-WindowTable  ; System Window
      DC.W   Content-WindowTable      ; In Content
      DC.W   Drag-WindowTable         ; In Drag
      DC.W   NullEvent-WindowTable    ; In Grow
      DC.W   NullEvent-WindowTable    ; In Go Away

InMenu
      CLR.L      -(SP)                ; Get space for menu choice
      MOVE.L     Where,-(SP)          ; Mouse at time of event
      _MenuSelect                     ; Click Point -> MenuID,MenuItem
      BRA    MenuCommand              ; Menu choice now on stack

SystemEvent                           ; Action for desk accessory
      PEA    CurrentEvent             ; Pointer to event record
      MOVE.L     EvtWind,-(SP)        ; Load window pointer onto stack
      _SystemClick                    ; System takes care of own click
      RTS
```

```
Content                                 ; Click inside a window
      MOVE.L EvtWind,A0                 ; Event Window
      CMP.L  WindowPointer,A0           ; Is it our Window?
      BNE    NullEvent                  ; No. Ignore this event
      BTST   #FrontFlag,D5              ; Is our window top dog?
      BEQ    SelWindow                  ; No. Just select window for first click
      BRA    SelCell                    ; Yes. Change cell in top window

Drag                                    ; Let user move the window
      MOVE.L EvtWind,-(SP)              ; Window pointer
      MOVE.L Where,-(SP)                ; Current mouse location
      PEA    DragLimit                  ; Boundary rectangle
      _DragWindow                       ; ^ Window,Start point,Bounds rect
      RTS


WindChk
      MOVE.L Message,A0                 ; Find event window
      CMP.L  WindowPointer,A0           ; Is it our window?
      BNE    @10                        ; If not ignore it
      BSR    SelPort                    ; Select our port
      CMP    D0,D0                      ; Return Z set
@10   RTS

SelWindow
      MOVE.L WindowPointer,-(SP)        ; Window Pointer
      _SelectWindow                     ; Put the window in front

SelPort
      MOVE.L WindowPointer,-(SP)
      _SetPort                          ; Set the drawing port to the window
      RTS

;-------------------------- MENU EVENTS --------------------------

Menucommand
; Choose selections from Menu Bar
;   Input  (SP) = Menu ID
;             2(SP) = Menu Item
;
      LEA    Menu,A0            ; Point to menu variable
      MOVE.L    (SP)+,(A0)      ; Save menu and item
      BSR    ChooseMenu
      CLR    -(SP)              ; Menu 0 means unhighlight
      _HiLiteMenu               ;  menu highlighted now
      RTS

ChooseMenu                      ; Find menu & item
      MOVE.W     Menu,D0        ; Menu resource ID
      CMPI   #1,D0              ; Is it menu 1?
      BEQ    AppleCmd           ;  Apple menu command

      CMPI   #302,D0            ; Is it menu 302?
      BEQ    FileCmd            ;  File menu command

      CMPI   #303,D0            ; Is it menu 303?
      BEQ    EditCmd            ; Edit menu command
      RTS                       ; No item selected

FileCmd                         ; Only item is quit
Quit                            ; Exit to finder
      BSET   #QuitFlag,D5       ; Set quit bit in flag register
      RTS
```

```
EditCmd                         ; Could be for SimpleCalc or desk accessory
      CLR.L       -(SP)         ; Space for TRUE/FALSE result & item
      MOVE   MenuItem,D0        ; Action for system to try
      ADDQ   #1,D0              ; Adjust to desk accessory standard order
      CMPI   #6,D0              ; Make NEGATE = 0 for UNDO
      BCC    @10
      MOVE   D0,(SP)            ; Edit commands 2..5
@10   _SysEdit                  ; Menu Item -> TRUE if accessory used event
      BTST   #0,(SP)+           ; Check result. If system used the menu
      BNE    NullEvent          ;  item event we are all done
      MOVE   MenuItem,D0
      CMPI   #2,D0
      BEQ    Copy               ; Item 2?
      BCS    Cut                ; Item 1?
      CMPI   #4,D0
      BEQ    ClearCmd           ; Item 4?
      BCS    Paste              ; Item 3?
      CMPI   #7,D0
      BEQ    Program            ; Item 7?
      BRA    Invert             ; Must be item 5

AppleCmd

      MOVE   MenuItem,D0        ; Check item number
      CMP    #1,D0              ; Item 1?
      BEQ    About              ;  Yes. Do About...

; Desk accesory. The name in the menu is the same as the program file name

      MOVE.L AppleHand(A6),-(SP) ; Apple-Menu handle
      MOVE.W D0,-(SP)           ; Number of chosen item
      PEA    DeskName(A5)       ; Pointer to place for name
      _GetItem                  ; Get string for item
      CLR    -(SP)              ; Space for reference number
      PEA    DeskName(A5)       ; Open desk acc
      _OpenDeskAcc              ; Name -> Reference number
      MOVE   (SP)+,D0           ; Discard result
      BRA    SetPort            ; Restore our graph port

About                           ; Display "about" box
      CLR.L  -(SP)              ; Clear space for dialog pointer
      MOVE   #301,-(SP)         ; Dialog resource ID 301
      CLR.L  -(SP)              ; Let Dialog Mgr provide storage area
      MOVE.L #-1,-(SP)          ; Place Dialog box above all other windows
      _GetNewDialog       ; DialogID, ^ Storage, ^ Window above->Dialog pointer
      CLR.L  -(SP)              ; No filter procedure
      PEA    ItemHit(A6)        ; ^ Area for Item Hit
      _ModalDialog              ; filter procedure, ^ item chosen
      _DisposDialog             ; Dialog pointer still on stack
      CMPI.W #2,ItemHit(A6)     ; End program if button 2 chosen
      BEQ    Quit
      RTS

;------------------------ SPREADSHEET FUNCTIONS ----------------------------

SelCell                         ; User clicked to select a cell
      PEA    Where              ; Adjust Mouse location to window coordinates
      _GlobalToLocal            ; pointer to the point for input and output
      BSR    DrawCell           ; Un Highlight current selection
      BSR    CalcNum            ; Which cell to select
      MOVE.W D2,D4              ; Save the selected cell
      BSR    ProgSel            ; Record the action
      BRA    DrawSelect         ; Outline cell darkly
```

```
KeyStroke                       ; User pressed a key
      BSET   #Redraw,D5         ; Redraw selected cell when done
      MOVE.L Message,D2         ; Get Character record
; Vector to operation or put digit in cell value
      CMPI.B #'0',D2
      BCS    NotDigit           ; Not a digit
      CMPI.B #'9',D2
      BLS    DigiKey

NotDigit
; Check table for operation
      BSR    OperVect           ; Check operation table for key
      BEQ    BadKey             ; Ignore keystroke if not in table
; Save address of operation. Store operation in program. Then perform it.
      PEA    (A0)               ; Push address on stack
      BRA    ProgOper           ; Save operation in accumulator program

BadKey
; Not in table so ignore key
      RTS

DigiKey
;  Digit typed in D2.  Add it onto end of number in (A3)
      MOVEQ  #$0F,D0
      AND.L  D0,D2              ; Clear upper nibble & junk
      MOVEQ  #10,D1             ; number base
      MULS   (A3),D1            ; Current value times 10
      BPL    DigiAdd            ; Is the cell number negative?
      NEG.L  D2                 ;  then increment is negative too
DigiAdd
      ADD.L  D2,D1              ; plus key stroke
      MOVE   D1,(A3)            ; Save value
; now check for overflow
; bits 15 through 30 must be the same
      LSR.L  D0,D1              ; shift down 15 bits
      ADDQ.W #1,D1
      BEQ    DigiOK             ; OK negative number
      SUBQ.W #1,D1
      BEQ    DigiOK             ; OK positive number
; overflow. clear to zero and start over
      CLR.W  (A3)
DigiOK                          ; New cell value in range
; fall through to CellConst

CellConst                       ; Cell set to constant value
      BSET   #Redraw,D5         ; Redraw selected cell after editing
      CLR.B  Prg(A3)            ; No program for cell
      RTS

OperVect
; Return vector to operation from table
;       INPUT     D2 = Character to match
;       OUTPUT    A0 = Vector address
;                 D2 = Character matched
;             Z flag -> character not found
;
      LEA    Opertable,A0
OpVecLoop
      CMP.B (A0),D2             ; Compare key stroke to table
      BNE    NextEntry
; Found It
      MOVE.W 2(A0),D0           ; Vector to operation
      LEA    Opertable(D0),A0   ; Actually LEA Optable-*(PC,D0)
      RTS                       ; Return NZ
```

```
NextEntry
; Check for end of table & advance pointer
      TST.L  (A0)+
      BNE    OpVecLoop
      RTS                          ; Not found. Return Z flag set

OperTable
;  4 bytes per entry
;    byte 1 = ascii value of key
;    byte 2 not used
;    bytes 3&4 offset

      DC     '+ '
      DC     AddOper-OperTable
      DC     '- '
      DC     SubOper-OperTable
      DC     '* '
      DC     MulOper-OperTable
      DC     '/ '
      DC     DivOper-OperTable
      DC     '= '
      DC     EqOper-OperTable
      DC     $0300                 ; [Enter] key
      DC     Enter-OperTable
      DC     $0800                 ; [BackSpace] key
      DC     ClearCmd-OperTable
      DC     $1B00                 ; [Clear] on 10-key pad
      DC     ClearCmd-OperTable
      DC.L   0                     ; End of table

AddOper
;  Add the selected cell into the accumulator
      MOVE.W (A3),D0
      ADD.W  D0,(A6)
      RTS

SubOper
;  Subtract the selected cell from the accumulator
      MOVE.W (A3),D0
      SUB.W  D0,(A6)
      RTS

MulOper
;  Multiply the accumulator by the selected cell
      MOVE.W (A3),D0
      MULS   (A6),D0
      MOVE.W D0,(A6)
      RTS

DivOper
;  Divide the accumulator by the selected cell
      MOVE.W (A6),D1
      EXT.L  D1
      MOVE.W (A3),D0
      BEQ    DivErr
      DIVS   D0,D1
      MOVE.W D1,(A6)
      RTS

DivErr                             ; Divide by zero
      SWAP   D1                    ; Return largest magnitude possible
      EORI.W #$7FFF,D1
      MOVE.W D1,(A6)
      RTS
```

```
Enter
;  Set the accumulator to the value in the selected cell
      MOVE.W (A3),(A6)          ; Set value
      RTS

EQOper
;  Set the selected cell to the value of the accumulator
;  Clear the program in the cell
;  delete the "="from the accumulator program
      CLR    Prg(A3)
      MOVE.W (A6),(A3)
      SUBQ.W #1,D6              ; Delete the = operator
      RTS

;  User can select these Edit Menu items

Cut                             ; CUT / X
      MOVE.W (A3),Clip(A6)      ; ClipBoard = Cell
      CLR.W  (A3)               ; Cell = 0
      BRA    CellConst

Copy                            ; COPY / C
      MOVE.W (A3),Clip(A6)      ; ClipBoard = Cell
      RTS

Paste                           ; PASTE / V
      MOVE.W Clip(A6),(A3)      ; Cell = ClipBoard
      BRA    CellConst

ClearCmd                        ; CLEAR  or Clear key
      CLR    (A3)               ; Cell = 0
      CLR    (A6)               ; accumulator  = 0
      CLR    D6                 ; Clear accumulator equation
      BSR    ProgSel            ; Accumulator equation = select
      BRA    CellConst

Invert                          ; NEGATE / N
      NEG    (A3)               ; cell = -cell
      BRA    CellConst

Program                         ; PROGRAM / P
; Copy the accumulator program into the selected cell
      LEA    Prg(A3),A1         ; Point to cell program area
      LEA    Prg(A6),A0         ; Point to accumulator program area
      MOVE   D6,D0              ; Count of valid bytes
CopyPLoop
      MOVE.B (A0)+,(A1)+
      DBRA   D0,CopyPLoop       ; Move D0+1 bytes
      CLR.B  -(A1)              ; Program ends with 0
      RTS
;-------------------------- DRAWING ROUTINES --------------------------

;
DrawCell
;  Put numbers into selected cell. Erase existing numbers
      PEA    CellRect
      _ERASERECT
      BSR    FrameCell          ; Draw gray box
;  Go to start of cell
      MOVE.L TxtPnt,-(SP)       ; X,Y of start of cell
      _MOVETO                   ; horiz, vertical
;  Set up to draw a value
      CLR.L  D0
      MOVE.W (A3),D0            ; First integer is value
      BPL    DrawValue
```

```
;  Precede negative number with a minus sign
      MOVE.W #$2D,-(SP)        ; minus sign
      _DRAWCHAR                ; Draw the -
      CLR.L  D0
      MOVE   (A3),D0           ; Get value again
      NEG.W  D0                ;  and make it positive
DrawValue
      DIVU   #10,D0            ; Draw in base 10
      SWAP   D0                ; Get remainder
      ORI.B  #'0',D0           ; Make digit a character by oring in zero
      MOVE.W D0,-(SP)          ; Save digit to draw
      SWAP   D0                ; Restore quotient
      EXT.L  D0                ; Leading digits?
      BEQ    DrawDigit         ; No more digits. Draw them all now
      BSR    DrawValue         ; Calculate next higher digit
DrawDigit
      _DRAWCHAR                ; Draw a byte from stack
      RTS                      ; Return to draw next byte or exit

MarkCell
;  Draw Selected cell border Dark so we know it is selected
      BTST   #FrontFlag,D5     ; Is our window active?
      BEQ    FrameCell         ;  No. Don't highlight
      _PENNORMAL               ; Normal Width,Mode,Black Color
      MOVE.L #$00030003,-(SP)  ; Width,Height=3
      _PENSIZE
      PEA    CellRect          ; pointer to separating rectangle
      _FRAMERECT               ; Draws box INSIDE CellRect
@10   RTS

FrameCell
;  Outline cell with light border
      _PENNORMAL               ; Normal Width,Mode,Black Color
      MOVE.L GRAFGLOBALS(A5),A0 ; Pointer to QD globals
      PEA    GRAY(A0)          ; Standard pattern
      _PENPAT
      PEA    CellRect          ; pointer to separating rectangle
      _FRAMERECT               ; Draws box INSIDE CellRect
      RTS

DrawInside
; Draw the entire spreadsheet
      MOVE   D7,-(SP)          ; Save cell to recalculate
      MOVEQ  #$7F,D7           ; Highest number cell
InsideLoop
      MOVE   D7,D2
      BSR    CalcCellRect
      BSR    DrawCell          ; Draw numbers
      DBRA   D7,InsideLoop
      MOVE   (SP)+,D7          ; Restore cell to recalculate
; Draw the selected cell and Accumulator.  Restore pointers
;    Fall through to DrawSelect

DrawSelect
; Draw the selected cell and Acc. Exit with pointers set up for selected cell
      BCLR   #Redraw,D5        ; Clear update needed flag
; Redraw accumulator
      MOVEQ  #AccCell,D2       ; Cell num of accumulator
      BSR    CalcCellRect
      BSR    DrawCell
      BSR    MarkCell          ; Mark accumulator
; Redraw Selected cell
      MOVE   D4,D2             ; Selected cell
      BSR    CalcCellRect
```

```
        BSR     DrawCell
        BRA     MarkCell            ; Highlight cell

;------------------------ SPREAD SHEET UPDATING --------------------------

Calculate
        MOVE    (A6),D0             ; Save accumulator
        MOVEM.L D0/D6/A3/A4,-(A7) ; Save registers
        ADDQ.B #1,D7                ; Next cell to update
        BPL     CalcNext
        CLR     D7                  ; Start with cell 0
CalcNext
        MOVE.W D7,D2
        BSR     CalcPntr            ; Point to cell data
        BEQ     CalcExit            ; No need to calc accumulator
        TST.B   Prg(A3)             ; Any program ?
        BEQ     CalcExit
        MOVE.L A3,A4                ; Pointer to update cell
        CLR     D6                  ; Start of program
        CLR.W   (A6)                ; Start with clear acc
CalcLoop
        MOVE.B Prg(A4,D6),D2        ; Get Code byte
        BEQ     CalcDone
        ADDQ    #1,D6               ; Point to next byte
        BCLR    #7,D2               ; Is it Select or arithmetic?
        BEQ     CalcCheck
; select cell for input to calculator
        BSR     CalcPntr
        BRA     CalcLoop

CalcCheck
; Find operation in table
        MOVE.B D2,D0
        BSR     OperVect
        BEQ     CalcOper            ; Found it
; Not in table. Impossible !!
        STOP    #$2000              ; No privilege violation
CalcOper
        JSR     (A0)                ; Execute operation
        BRA     CalcLoop

CalcDone
;  end of prog. assign value. Redraw cell if it changed
        MOVE    (A6),D0             ; Get result
        CMP.W   (A4),D0             ; Compare to old value
        BEQ     CalcExit            ; No change
        MOVE.W D0,(A4)              ; Assign new value
;  new value. redraw cell
        CMP     D7,D4               ; Calculating selected cell?
        BNE     CalcUpdate
        BSET    #ReDraw,D5          ; Set flag to do later
        BRA     CalcExit
;
CalcUpdate                          ; Redraw the re calculated cell
        MOVE    D7,D2
        BSR     CalcCellRect
        BSR     DrawCell
        MOVE    D4,D2                       ; Selected cell
        BSR     CalcCellRect

CalcExit
        MOVEM.L (A7)+,D0/D6/A3/A4 ; Restore saved registers
        MOVE    D0,(A6)             ; Restore accumulator
        RTS
```

```
;------------------------ SPREAD SHEET SUBROUTINES --------------------------


CalcNum
;  Calculate the cell number that contains a point
;  Input   Where = point in local coordinates
;  Output  D2    = cell number
      CLR.L  D0
      MOVE.W WhereH,D0
      DIVU   #56,D0              ; Horiz pixels/cell
      MOVE.W WhereV,D2
      LSR.W  #4,D2               ; 16 Vert pixels/cell
      ASL.W  #3,D2               ;  8 cells  Horiz per line
      ADD.W  D0,D2               ;  Cell number = 8 * y + x
      RTS

CalcCellRect
;  Calculate a cell Rectangle from  a cell number
;  Input   D2       = cell number
;  Output  D2       = cell number
;          A3       = pointer to cell data
;          TxtPnt   = first text pixel of cell
;          CellRect = rectangle separating cell
;
      MOVE.W D2,D1
      EXT.L  D1                  ; Divide always 32 bits
      DIVU   #8,D1               ; 8 columns per row
      MOVE.L D1,D0
      SWAP   D0                  ; Remainder in upper word
      LEA    CellRect,A0         ; Point to cell rectangle
      LEA    TxtPnt,A1           ; Point to start of text
      MULU   #56,D0              ; 56 pixels per column
      MOVE   D0,left(A0)         ; Left separator
      ADDQ   #4,D0               ; Margin before text
      MOVE.W D0,h(A1)            ; First text location
      ADD.W  #52,D0              ; Advance to next column
      MOVE.W D0,right(A0)        ; Right separator
      ASL.W  #4,D1               ; 16 pixels per row
      MOVE.W D1,top(A0)          ; Top separator
      ADD.W  #12,D1              ; Ascent of text + top margin
      MOVE.W D1,v(A1)            ; First text location
      ADDQ   #4,D1               ; Advance to next row
      MOVE.W D1,bottom(A0)       ; Bottom separator
; Fall through to calculate cell data pointer

CalcPntp
;  Calculate the pointer to a cell value
;  Input   D2   = cell number
;  Output  D2   = cell number
;          A3   = pointer to cell data
;          Z flag -> cell number is accumulator
      MOVEQ #AccCell,D0          ; Make accumulator = cell number 127
      EOR.W D2,D0                ; Calc relative location
      ASL   #5,D0                ; 32 bytes / cell
      LEA   0(A6,D0.W),A3        ; Does not affect Condition codes
      RTS

ProgSel                          ; Store selection operation in program
; Put selected cell, D4, into program with $80 flag
      MOVE   D4,D2
      BSET   #7,D2
; Fall thru to ProgOper to put byte into program
```

```
ProgOper                        ; Store arithmetic operation in program
; Put byte in D2 into accumulator program
      MOVE.B D2,Prg(A6,D6.W)    ; Store byte in program list
      CMPI.B #ProgLast,D6       ; More program space?
      BEQ    ProgOVF
      ADDQ.B #1,D6              ; Advance program pointer
      RTS
ProgOVF                         ; Program too big
      RTS


; ---------------------------- Data Storage ------------------------

CurrentEvent                    ; Event record
What            DC    0         ;  Type of event
Message         DC.L  0         ;  Info about event
When            DC.L  0         ;  Tick when it happened
Where                           ;  Mouse location when it happened
WhereV          DC    0         ;   Vertical coordinate
WhereH          DC    0         ;   Horizontal coordinate
Modify          DC    0         ;  Control keys down when it happened

EvtWind         DC.L  0         ; Window with event

Menu            DC    0         ; Menu that item is in
MenuItem        DC    0         ; Menu item selected

WindowPointer   DC.L  0         ; Pointer to spread sheet window

DragLimit                       ; Boundary rectangle for dragging window
                DC    30        ;  top
                DC    5         ;  left
                DC    350       ;  bottom
                DC    500       ;  right

CellRect        DCB.W 4,0       ; Rectangle enclosing selected cell
TxtPnt          DC.L  0         ; Point in cell where text starts

END
```

R.Maker File, SimpleCalc.R

```
* Resource file for SimpleCalc
*
* Compiled object file name
SimpleCalc
*
* Type and signature
APPLCALC
*
* Linked object file name
INCLUDE SimpleCalcLinked
*
* You can also create a separate resource file without code,
*  by omitting all of the lines above and adding the compiled
*  resource file name shown on the next comment line below.
*SimpleCalc.Rsrc
```

```
*
*
* Signature for desktop file
Type CALC = GNRL
 ,0
 .P
SimpleCalc version 1.0, a fun spread sheet program

Type MENU
  ,1
\14
 About Simple Calc...
 (-

  ,302
File
  Quit

  ,303
Edit
  Cut/X
  Copy/C
  Paste/V
  Clear
  Negate/N
  (-
  Program/P

Type DLOG
  ,301
  100 100 200 400
  Visible  NoGoAway
  1
  0
  301


  Type DITL
    ,301
  4

  Button
  70 220 90 280
  Try it \21
```

```
Button
70 20 90 80
Quit

StaticText
15 50 35 280
Simple Calc was written for fun

StaticText
35 20 55 280
by Harland Harrison & Ed Rosenzweig

Type WIND
 ,301
Simple Calc
64 32 320 480
Visible NoGoAway
0
0

* File reference
Type FREF
  ,128
  APPL 0

Type BNDL
  ,128
  CALC 0
  ICN#
  0  128
  FREF
  0  128

Type ICN# = GNRL
* Icon list for the program icon
* There are two icons in the list. One to display and one for a mask.
  ,128
  .H
* This is the SimpleCalc icon. There are 32 lines of icon data
00000000
00000000
00000000
1FFFFFFC
10000004
```

```
17FFFFF4
14081014
14C997D4
152A5054
14285094
14499114
14885214
150A5214
15E99214
14081014
17FFFFF4
14081FF4
15E89834
15099834
15089834
15C89834
14289834
14289834
15C9D834
14081FF4
17FFFFF4
10000004
1FFFFFFC
00000000
00000000
00000000
00000000
* This is the mask for the SimpleCalc icon. There are 32 lines of data.
00000000
00000000
00000000
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
```

```
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
1FFFFFFC
00000000
00000000
00000000
00000000
```

Linker File, SimpleCalc.LINK

```
/Output SimpleCalcLinked     ; Output file
SimpleCalc.Rel               ; Input file
$
```

Exec File, SimpleCalc.Job

```
Asm         SimpleCalc.Asm      Exec    Edit
Link        SimpleCalc.Link     Exec    Edit
RMaker      SimpleCalc.R        Edit    Edit
```

Index

HAYDEN Macintosh Library

6310-5

Programming the 68000

Macintosh Assembly Language

A prime resource for programmers who want to use assembly language programs to get the most out of the Mac. *Programming the 68000* builds on your awareness of assembly language and experience with high-level languages like BASIC, Pascal, or C. This book fully covers and carefully examines the power of the Motorola 68000 microprocessor.

The authors detail the assembly language process of coding, editing, compiling, linking, and resource compiling, with thorough explanations of the 68000 instruction set and addressing modes. Also featured are many example routines that show the mechanics of 68000 code and provide a first step for learning to examine and analyze assembly language programs to create the fastest, most efficient code possible.

The text explains the advantages of the Macintosh programming environment, its user interface toolbox, and specialized ROM calls. As an introduction to writing worthwhile programs, the book shows how to code a simple electronic spreadsheet called *SimpleCalc,* a medium-size Macintosh assembly language program. The book provides not only a conceptual understanding of 68000 assembly language, but also a first look at many types of routines that serve as the building blocks for serious assembly language programming as well.

photo: Ed Kashi

About the Authors...

Edwin Rosenzweig is a graduate of the University of California. He has ten years' programming experience in BASIC, Pascal, C, FORTRAN, COBOL, and 6502 and 68000 assembly languages. He developed a census data program for the Macintosh, called "People in Places," and is the publisher of "PCMacBASIC," a Macintosh BASIC Compiler.

Harland Harrison has ten years' programming experience developing business and systems software. Among his many programs are "Quik-Circuit," a PCB CAD/CAM system for designing printed circuit boards; "PCMacBASIC," and the "PTD 6502" debugger for the Apple II. He is now a consultant specializing in Macintosh applications.

Hayden Book Company
A DIVISION OF HAYDEN PUBLISHING COMPANY, INC.
HASBROUCK HEIGHTS, NEW JERSEY
ISBN 0-8104-6310-5