Brady

GARY B. LITTLE
Author of *Inside the Apple IIe* and *Apple Pro DOS: Advanced Features for Programmers*

MAC ASSEMBLY LANGUAGE

A GUIDE FOR PROGRAMMERS

COVERS MDS 2.0

MAC ASSEMBLY LANGUAGE

A Guide for Programmers

Other Brady Books by Gary B. Little

Inside the Apple IIc
Inside the Apple IIe
Apple ProDOS: Advanced Features for Programmers

MAC ASSEMBLY LANGUAGE

A Guide for Programmers

Gary B. Little

A Brady Book
Published by Prentice Hall Press
New York, NY 10023

Copyright © 1986 by Gary B. Little
All rights reserved,
including the right of reproduction
in whole or in part in any form

A Brady Book
Published by Prentice Hall Press
A Division of Simon & Schuster, Inc.
Gulf + Western Building
One Gulf + Western Plaza
New York, New York 10023

PRENTICE HALL PRESS is a trademark of Simon & Schuster, Inc.

Manufactured in the United States of America

1 2 3 4 5 6 7 8 9 10

Library of Congress Cataloging-in-Publication Data

Little, Gary B., 1954—
Mac Assembly Language.

"A Brady book."
Bibliography: p.
Includes index.
1. Macintosh (Computer)—Programming.
2. Assembler language (Computer program language)
I. Title.
QA76.8.M3L58 1986 005.265 86-25980

ISBN 0-13-541434-2

Dedication

This book is dedicated to my mother,
Barbara Hope Little

Contents

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Registered Trademarks

Apple, the Apple logo, the Macintosh logo, MacWrite, and MacPaint are trademarks of Apple Computer, Inc.
Macintosh is a trademark of McIntosh Laboratory, Inc., and is licensed to Apple Computer, Inc.
CompuServe is a trademark of CompuServe Incorporated.
Delphi is a trademark of General Videotex Corporation.
MC68000 is a trademark of Motorola Inc.
IBM PC is a trademark of International Business Machines Corporation.

Preface

Nothing gives me more pleasure than programming the Macintosh in 68000 assembly language. Why? Primarily because it's much more challenging than using a high-level language like BASIC or Pascal, I suppose; and I do enjoy a good challenge. But it's more than that: It's knowing that when the program is finished it will run faster and more efficiently than its high-level language counterpart. It's knowing I can exercise complete control over the microprocessor and that the program isn't doing anything "behind my back" because I control its every move.

Assembly language programming is not for everyone, of course. It is far too demanding for the less exacting programmers among us. If you don't want to concern yourself with such finicky, but important, details as avoiding register conflicts, selecting addressing modes, or allocating safe data areas, then I suggest you stick to a language like BASIC, which handles these details for you.

If, on the other hand, you're intrigued by the possibility of writing blindingly fast programs and learning about the internal structure of the Macintosh and the Macintosh Plus along the way, by all means give assembly language a try. It may take a while to fully understand the language, but the rewards are worth the effort.

The purpose of this book is to show you how to develop assembly language programs on the Macintosh using Apple's Macintosh 68000 Development System (MDS). The discussion relates to both the original MDS and version 2.0, which was released in the summer of 1986. If you're using another assembler, you will still find the book useful, although the specific examples will probably have to be adapted to suit the syntactical requirements of your assembler.

I've assumed throughout the book that you are no stranger to programming or computerese, so I've avoided lengthy explanations of such fundamental concepts as byte, bit, hexadecimal, and so on. I have, of course, emphasized those points that might be confusing to those who have programming experience, but not necessarily in assembly language. Knowledge of any other assembly language will be helpful, however.

This book is divided into two parts. The first part—Chapters 1, 2, and 3—primarily contains reference material describing 68000 assembly language in general and the programming tools you must master to create usable programs. The second part—Chapters 4

through 9—shows specific examples of how to create assembly language programs that implement the standard Macintosh user interface.

In Chapter 1 I describe the 68000 microprocessor itself. This includes a preliminary look at the instructions it supports and an analysis of its internal registers. Also described is the 68000 stack and the addressing modes that instructions use to locate data.

Chapter 2 is a detailed description of how to use the programming tools that come with MDS: the editor, assembler, linker, resource compiler, and executive program. I also include step-by-step instructions for creating a typical application program.

Chapter 3 takes a close look at each of the 68000's instructions, highlighting the important characteristics of each. This chapter includes several reference tables indicating the permitted addressing modes for each instruction and showing how each instruction affects the 68000's internal condition code flags. You will refer to these tables quite often when you're developing assembly language programs.

Part 2 shows how to develop 68000 programs in the unique Macintosh environment. I analyze most of the important subroutines contained in the Macintosh ROM and show how to use them from assembly language. By taking advantage of these subroutines, you can implement the standard Macintosh user-interface in your own programs with a minimum of effort. The specification for this interface is found in *Inside Macintosh*, Apple's programming guide to the Macintosh, a book that every programmer should obtain and review.

Chapter 4 shows how memory is used on the Macintosh and describes a group of ROM subroutines, called the Memory Manager, that applications can use to allocate and deallocate memory space. Chapter 5 covers the Event Manager, the portion of the ROM that deals with input/output activity like key presses and mouse clicks.

Chapters 6, 7, and 8 deal with three of the most important groups of ROM subroutines: the Window Manager, the Menu Manager, and the Dialog Manager. After mastering them, you should have no trouble handling multiple windows on the screen, implementing pull-down menus, and using dialog and alert boxes.

I wrap things up in Chapter 9 by showing how to write an assembly language program in such a way that it will work properly with

desk accessories. This is an important consideration for all Macintosh programmers.

By the end of the book I hope you will have learned enough to develop serious assembly language applications on the Macintosh. Keep in mind, however, that I have only scratched the surface of the Macintosh ROM and there are many more subroutines in it that are available to the assembly language programmer. The definitive description of these subroutines is in *Inside Macintosh*. I've included a list of other useful books and periodicals for Macintosh software developers in the Bibliography.

You should also consider becoming a member of the new Apple Programmer's and Developer's Association (APDA), since it is a convenient source of official Apple technical material. Contact APDA at 290 SW 43rd Street, Renton, WA 98055, (206) 251–6548.

Gary B. Little
Vancouver, British Columbia, Canada

Chapter 1

Inside the 68000 Microprocessor

The Motorola MC68000 microprocessor (the 68000 to its friends) is the brain that controls the Apple Macintosh. Its primary function, of course, is to run all those programs that make the Macintosh such a delightful computer to work with. It is also responsible for controlling the various input/output (I/O) devices attached to the Macintosh: the two serial ports (usually used to connect a printer and a modem), the external disk drive port, the mouse, the keyboard, and, on the Macintosh Plus, the SCSI (Small Computer Standard Interface) port.

In this chapter I'll examine those aspects of the 68000 microprocessor important to programmers. This will include an overview of the instructions the 68000 supports (more detailed descriptions appear in Chapter 3), the internal registers it uses to store addresses and data, and the methods it uses to locate data to work with. I definitely won't be discussing anything that requires a degree in electrical engineering. If you're interested in such topics as hardware interfacing or timing diagrams, you won't find them here. Instead, refer to Motorola's specification booklet for the MC68000 microprocessor.

First, a few general words about the operational characteristics of the 68000. The 68000 is usually referred to as a 16/32-bit microprocessor by computer designers. This is because, although it has a 16-bit data bus, (it grabs data from memory 16 bits—one word, two bytes—at a time), it has several 32-bit internal registers in which to store data. Despite this apparent size mismatch, you can tell the 68000 to fill an entire register using just one instruction; when you

do this, the 68000 automatically fetches two consecutive words of data. A true 32-bit version of the 68000, called the 68020, sports a 32-bit data bus, so it can fill a register more quickly, in just one fetch. Macintosh-like computers using the faster 68020 are rumored to be on the drawing board at Apple Computer, Inc.

The 68000 uses a 24-bit address bus, which means it can directly access up to 16 megabytes (actually 16,777,216 bytes) of memory! You can calculate this number for yourself by realizing that each of the 24 bits placed on the address bus can be either on or off; that means there are 2^{24} (16,777,216) unique address combinations that can be formed. Addresses are usually referred to by six-digit hexadecimal numbers ranging from $000000 to $FFFFFF (the leading dollar sign indicates the number is hexadecimal, not decimal).

Compare the 68000 with the 6502 microprocessor used by Apple's first product, the Apple II. The 6502 is limited to a 64K address space (1K = 1,024 bytes) because its address bus is only 16 bits wide (and 2^{16} = 65,536). Additional 64K banks of memory can be added to a 6502 system, but you can use only one of them at a time. To select a bank, or go from one bank to another, you must use complex bank-switching techniques to ensure that no two banks become active at the same time. This is not a very straightforward way of accessing memory and makes software development very difficult.

Most of the 68000's address space is unused on current versions of the Macintosh, but future releases will probably use much more. We'll look at how the memory space is used in a Macintosh in Chapter 4.

The 68000 Instruction Set

As shown in Table 1-1, there are 55 basic types of **instructions** the 68000 understands. Instructions are commands to the 68000, telling it what to do: Move data from place to place, fill an internal register with a number, call a subroutine, add two numbers together, and so on. A few of the 68000

instructions are similar in nature to commands in a higher-level language, such as BASIC: JSR (GOSUB—call a subroutine), RTS (RETURN—return from a subroutine), and JMP (GOTO—jump to a specific location) are the most obvious. Most, however, involve direct manipulation of internal registers and status bits, operations that are usually not easily done from higher-level languages like BASIC and Pascal.

Table 1-1. The 68000 Instruction Set.

Instruction	*Description*
ABCD	Add two BCD numbers with extend bit (X)
ADD	Add two binary numbers
AND	Logical "and"
ASL	Arithmetic shift left
ASR	Arithmetic shift right
Bcc	Branch if condition (cc) is true
BCHG	Test a bit and change it
BCLR	Test a bit and clear it
BRA	Branch relative always
BSET	Test a bit and set it
BSR	Branch to a subroutine
BTST	Test a bit and set flags
CHK	Check a data register against bounds
CLR	Clear to zero
CMP	Compare
DBcc	Decrement, Test, Branch until condition is true
DIVS	Signed division
DIVU	Unsigned division
EOR	Logical "exclusive or"
EXG	Exchange two registers
EXT	Sign extension
ILLEGAL	Illegal instruction exception
JMP	Jump
JSR	Jump to a subroutine
LEA	Load effective address
LINK	Allocate a stack frame
LSL	Logical shift left
LSR	Logical shift right
MOVE	Move
MULS	Signed multiplication

Table 1-1. *continued*

Instruction	*Description*
MULU	Unsigned multiplication
NBCD	Negate a BCD number
NEG	Negate a binary number
NOP	No operation
NOT	One's complement
OR	Logical "or"
PEA	Push effective address
RESET	Reset external devices
ROL	Rotate bits left
ROR	Rotate bits right
ROXL	Rotate bits through extend bit (X) left
ROXR	Rotate bits through extend bit (X) right
RTE	Return from exception
RTR	Return and restore status
RTS	Return from subroutine
SBCD	Subtract two BCD numbers
Scc	Set bits conditionally
STOP	Stop execution until interrupt
SUB	Subtract two binary numbers
SWAP	Exchange halves of data register
TAS	Test and set a bit
TRAP	Trap exception
TRAPV	Trap if overflow flag (V) is set
TST	Test
UNLK	Deallocate a stack frame

The instructions in Table 1-1 are the ones you will use to write source code for a 68000 assembly language program. Since the 68000 processor (like all processors) understands only binary numbers, you must convert any assembly language program you write to this computer-readable binary form before you can run it. One way to do this is to use the assembler and linker that come with Apple's Macintosh 68000 Development System (MDS), which we'll be using in this book. The executable code generated by MDS (or any other assembler/linker) is called **object code** or **machine code**.

You'll rarely need to know the binary equivalent of a 68000 instruction unless you're hand-patching machine code during a debugging operation. For instance, you may sometimes want to remove a portion of code before rerunning a program you're debugging. You can do this by storing words containing $4E71, the binary equivalent of the "do-nothing" NOP (No OPeration) instruction, over top of the code.

Even though you will probably never have to concern yourself with the binary form of an instruction or how an instruction is stored in memory, it's still interesting to know something about what's involved.

First of all, an instruction is always an even number of bytes in length. Since two bytes are referred to as a **word**, this means an instruction is an integral number of words in length. The shortest instruction is one word long and the longest is five words.

The first word of an instruction is called the **operation word**. (Refer to Figure 1-1.) It tells the 68000 the type of instruction involved in the operation (MOVE, MULU, LEA, and so on) and some, perhaps all, information relating to the **operands** of the instruction. The operands are the portions of the instruction that tell the 68000 where to find the data it is to manipulate, and where to store the result. Most instructions have at least one operand and many have two, one called a **source operand** and the other called a **destination operand**.

If there isn't enough room in the operation word to store all the information needed to describe the operands fully, the next one to four words in memory are used as **extension words** to complete the operand definition. If there are two operands, the extension word or words for the source operand comes first. I'll discuss this in greater detail later on when we look at the various addressing modes the 68000 operands rely on to access data.

It is instructive at this point to consider the general form of a two-operand instruction, such as the ADD instruction:

```
ADD  source_operand,dest_operand
```

As you can see, the two operands are entered right after the instruction mnemonic and are separated by a comma. For the ADD instruction, the operands tell the 68000 how to locate the two numbers to be added together; the result is stored at the location described by the destination operand. We'll be looking at the precise form of the operands later on in this chapter.

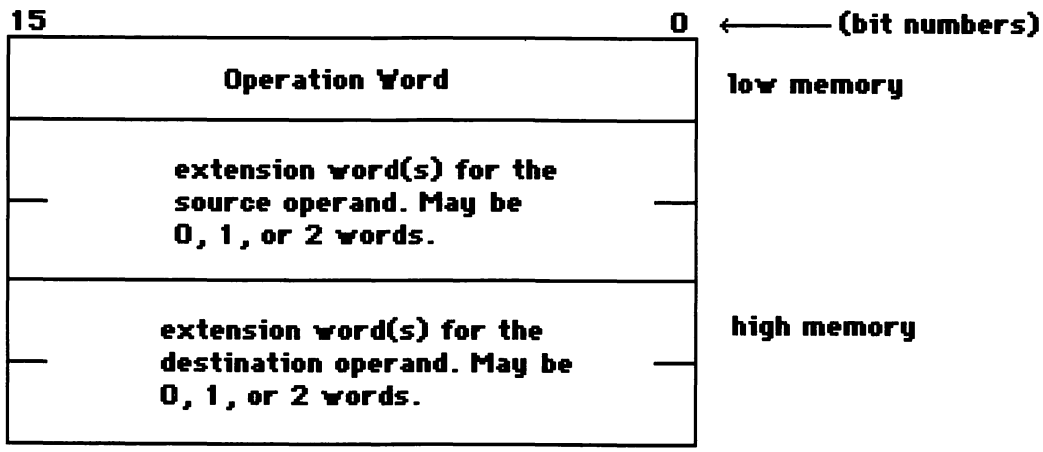


Figure 1-1. The Format of a 68000 Instruction.

By the way, you must be careful to specify the operands in a two-operand instruction in the correct order: The source operand always comes first. Assemblers for some other microprocessors, notably 8088 assemblers, insist that operands be specified in the opposite order!

Programmer's Model

One measure of the power of a microprocessor is the number of internal registers it uses to manipulate and transfer data. This is because operations involving data stored in registers are handled much faster than the corresponding operations with data stored somewhere in the random-

access memory (RAM) space available to the 68000. With plenty of registers at your disposal, it's easy to crank up the speed of a program by using as many of them as possible for storage of data or intermediate results before storing the final result in memory.

As shown in Figure 1-2, the 68000 supports several registers: eight 32-bit **data registers**, nine 32-bit **address registers**, one 32-bit **program counter register**, and one 16-bit **status register**. Keep in mind that these registers are *not* RAM or read-only memory (ROM) locations; they form part of the internal structure of the 68000 microprocessor itself.

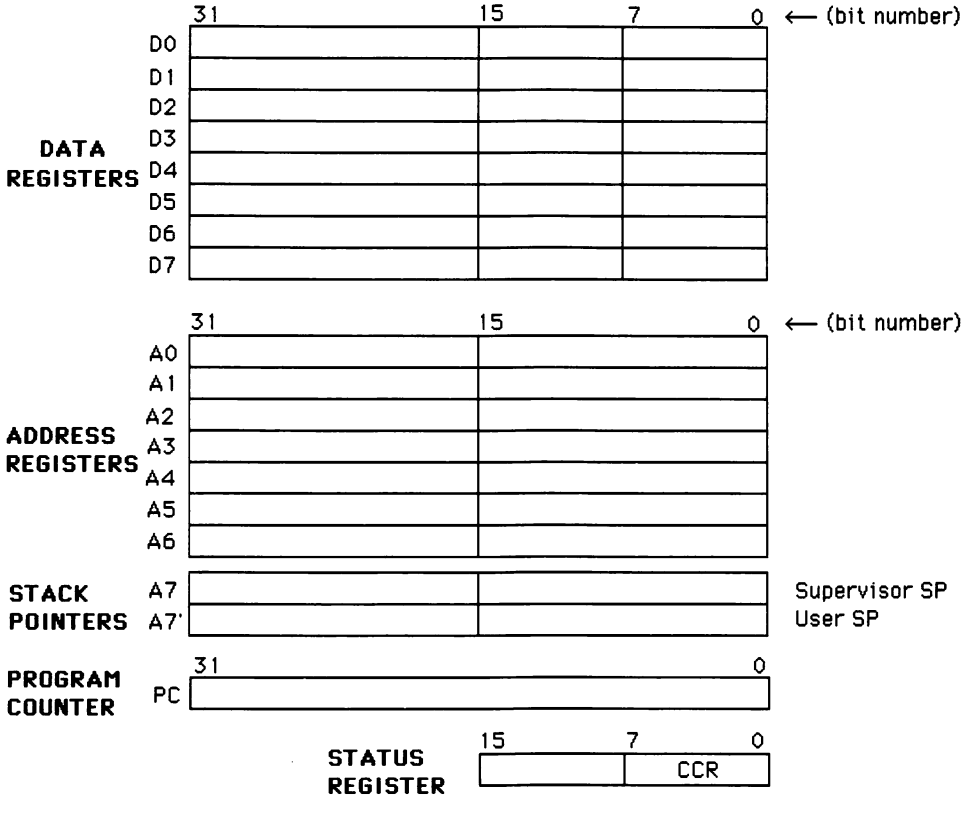


Figure 1-2. The 68000 Register Set.

The bits in a register are numbered from 0 to 31 (or 15 in the case of the status register), where bit 0 is the least-significant bit and bit 31 is the most-significant bit.

> A bit is said to be more significant than another if its **binary weight** is higher. The binary weight is the numeric value associated with the bit, that is, its contribution to the magnitude of the number. For bit number n, the binary weight is 2^n. When numbers are written down, they are written with the most-significant digit on the left and the least-significant on the right.

The right-to-left numbering scheme used to identify bits in a register is the same as the one used to identify bits in a byte (eight bits), a word (two consecutive bytes; 16 bits), and a long word (four consecutive bytes; 32 bits).

Now let's look at each of the 68000's 19 registers.

Address Registers

Although the 68000 has nine **address registers**, only eight are active at any given time. The conventional names for these eight address registers are A0, A1, A2, and so on, up to A7. The ninth address register is an alternate A7 register.

Except for the A7 register, the address registers can be used interchangeably, subject only to restrictions on their use dictated by the Macintosh's **operating system**. (The operating system is made up of a program called **System** and the subroutines in ROM it uses.) The A7 register is special because the 68000 uses it as a pointer to an important data structure called a **stack**. As you will see below, certain 68000 instructions, and even your own programs, use the stack for temporary data storage. Not surprisingly, another name for the A7 register is **SP** (which stands for stack pointer).

You can use the 68000 in one of two operating modes, **user** or **supervisor**. You select the appropriate mode by adjusting

the supervisor state bit in the 68000 status register. Only one of the A7 registers (the **supervisor stack pointer**) is normally used on the Macintosh because you're always in the supervisor mode. If you enter the user mode, the **user stack pointer** (the alternate A7 register) becomes active. This means it is possible to set up two independent stacks in memory, one for the operating system and one for the application. Another difference between the two operating modes is that in the user mode there are certain instructions you are not permitted to execute; if you're in supervisor mode, you can execute any instruction you want. The Macintosh always operates in supervisor mode, so you should not worry about what instructions are valid.

An address register may hold any data you care to store in it. But, as the name suggests, it is usually used to hold the address of something, perhaps a data structure or variable used by your program. You can also use it as an index into a data structure when either of the 68000's two indexed addressing modes are used (more on these addressing modes later).

You can store either a word (two bytes) or a long word (four bytes) in an address register, but not just one byte. You have to be careful if you store a word in an address register, however. The word, as you might expect, occupies the lower half (bits 0 to 15) of the register. What you might not expect is that the upper 16 bits of the register are also affected because of automatic **sign extension**. The sign bit of a word is bit 15, the most significant bit. If it is a one, the number is considered to be negative; if it is zero, it's positive. When you store a word in an address register, the contents of bit 15 are copied (**extended**) to bits 16 through 31.

This means that a word-sized address can only describe an address in the first 32K of the 68000's 16-megabyte address space, or the last 32K. This is because bit 15 of the address words from $0000 to $7FFF (the first 32K addresses) is zero, so the top 16 bits of the address register will also be zero. From $8000 to $FFFF, however, bit 15 is one, and sign extension means that the address register will contain $FFFF8000

to $FFFFFFFF. Since only the first 24 bits of an address are significant to the 68000, this corresponds to an address range from $FF8000 to $FFFFFF, the 32K area of memory just below the 16-megabyte upper limit of the address space.

You should avoid moving word quantities into address registers on the Macintosh unless you are absolutely sure the address is less than $8000. Even if you are sure, it's probably best to use long words instead, to ensure your program will be compatible with future versions of the Macintosh, where data areas may be relocated. The only "penalty" is that the instruction is slightly longer and takes a little longer to execute.

Data Registers

There are eight **data registers** and their names are D0, D1, D2, and so on, up to D7. They are logically equivalent, so you can use any one of them in exactly the same way you would use any other, subject, again, to any restrictions of the operating system.

Although the data registers are the same size as the address registers (32 bits), the 68000 behaves differently when you store numbers in them. First of all, you can store long words, words, or bytes in them (not just long words or words). Bytes occupy bits 0 to 7, words occupy bits 0 to 15, and long words occupy the entire register, bits 0 to 31. Second, no sign extension occurs when you store a word (or a byte) in a data register. Data registers are usually used for the storage of numeric data or indexes into data structures.

The Status Register

The **status register** (called the **SR**) is a 16-bit register that reflects the operational mode of the 68000. It is actually made up of two parts, the **system byte** and the **user byte** (or condition code register), as shown in Figure 1-3.

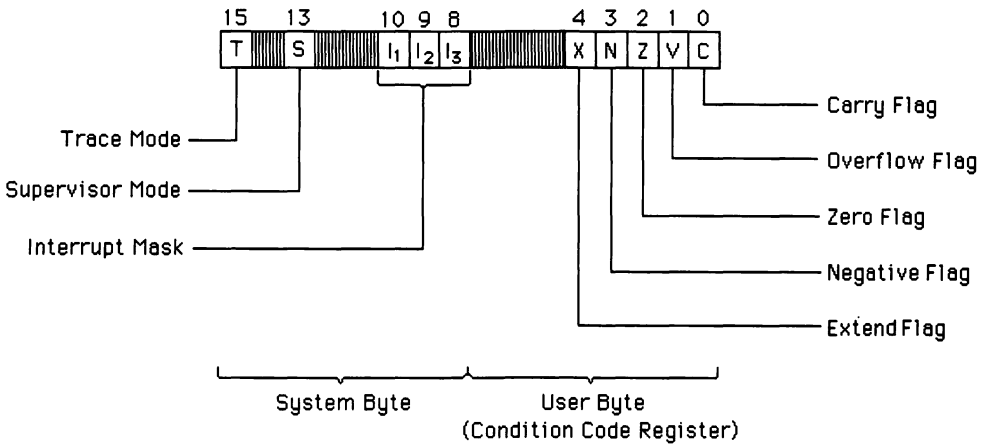


Figure 1-3. The 68000 Status Register.

The System Byte

The bits in the system byte reflect the fundamental operating state of the 68000: whether it's in supervisor or user mode, what hardware interrupts are enabled, and whether instruction tracing is enabled. The Macintosh operating system initializes the system byte when you first turn on the Macintosh. Few applications need to change this initial setting, so you can usually ignore it altogether.

In any event, let's look at the meaning of the five bits in the system byte the 68000 uses (the other three bits have no meaning). The bit numbers given are relative to the entire 16-bit status register.

T—Trace Mode

If the trace mode bit (bit 15) is zero, the 68000 executes a program in the normal way. If it is one, however, the 68000 interrupts the program after the execution of each instruction and passes control to a program whose starting address is stored at location $000024. (This is exception vector #9;

more on exceptions at the end of this chapter.) You are responsible for installing this program before activating the trace mode; if you don't, the Mac will die a horrible death as it tries to execute a program that doesn't exist.

The only programs you're likely to come across that fiddle with the trace mode bit are programming utilities called **debuggers**. The most well-known is the MacsBug program that comes with MDS (see Appendix C). When you activate its trace feature, the contents of all the registers are displayed every time an instruction in the main program is executed. This makes it easy to check that a program is performing as it should.

S—Supervisor State

The supervisor state bit (bit 13) is initially set to one when you start up the Macintosh, meaning that the 68000 is operating in supervisor mode. In this mode, you can execute any 68000 instruction and the supervisor stack pointer is active.

If the S bit is off, you're in user mode and the user stack pointer (the alternate A7 register) is active. In this mode there are several 68000 instructions you can't use (all noted in Chapter 3), most of which involve direct modification of the system byte in the status register. If you try to use them anyway, you'll generate a privilege violation exception.

I—Interrupt Mask

The interrupt mask (bits 10, 9, 8) lets you enable or disable hardware interrupt processing. As you will see later on, **interrupts** are events that peripherals use to signal they are ready to receive or send data or that some special condition has taken place that needs immediate attention. Interrupts are usually enabled so the 68000 can execute its programs without having to periodically check (or **poll**) for I/O operations. Programs that use polling techniques for I/O operations are

understandably less efficient than those using interrupt techniques.

Each hardware interrupt is associated with a particular priority level fixed by the hardware design. Any interrupt having a level at or below the value of the interrupt mask is ignored by the 68000. There is an exception: If the mask value is 7 (binary 111), a level 7 interrupt is permitted.

You should never change the value of the interrupt mask unless you really know what you're doing. Applications requiring you to change it are rare, indeed.

The User Byte

The user byte is of much more interest to a programmer. It contains five one-bit condition code **flags**, the settings of which are referred to, or affected by, most 68000 instructions. For example, the 68000 supports a group of branch conditionally instructions (like BCC, BCS, and BVC) you can use to change the order of execution of a program based on the state of a condition code flag.

The user byte is also called the **Condition Code Register**, or CCR for short.

The Extend Flag

The extend flag (X) is primarily used to indicate the carry or borrow status in an addition or subtraction operation. If it is set to one, a carry out of the most significant bit of the operand occurred during an addition operation, or a borrow occurred during a subtraction operation.

The existence of the extend flag makes it possible to carry out multiword mathematical operations. For example, if you want to add two numbers that each occupy three words in memory, you would first add the lowest-order words using the standard ADD instruction, and then use the ADDX (add with extend) instruction for the middle- and high-order words so that any carry would be included in the total.

The extend flag also participates in many of the 68000's bit shifting instructions that we'll look at in Chapter 3.

The Negative Flag

The negative flag (N) indicates the sign of the result of the last mathematical operation or of the data last moved into a data register.

The negative flag is set to one if the sign bit (the most-significant bit) of the result or data is one; otherwise, it will be zero. The sign bit is bit 7 for a byte, bit 15 for a word, and bit 31 for a long word.

The Zero Flag

The zero flag (Z) indicates whether the result of the last mathematical operation was zero. If it was, the zero flag is set to one; if not, the zero flag is cleared to zero.

You should note that the zero flag is not just adjusted after mathematical operations. Any time you move a zero value into a data register, the zero flag is set to one. If some other value is involved, the zero flag is cleared to zero.

The Overflow Flag

The overflow flag (V) is only important if you're working with signed numbers. Signed numbers are those where the most significant bit of the operand (bit 7 for a byte, bit 15 for a word, or bit 31 for a long word) is used to hold the sign of the number (one means negative, zero means positive). The rest of the bits hold the magnitude of the number in a **two's complement form**. The two's complement form of a number is formed by taking the binary form of the absolute value of the number, complementing all the bits (changing ones to zeros and vice versa), and then adding one. This form is used to facilitate internal addition and subtraction operations.

If the result of a mathematical operation on two signed numbers is outside the range of numbers that can be represented in this format, the overflow flag is set to one; if all goes well and the number is in range, it is cleared to zero. You can use the branch conditionally instructions BVS (branch on overflow set) and BVC (branch on overflow clear) to pass program control anywhere you want in the program when an overflow condition occurs. Another instruction that checks the V flag is TRAPV; this instruction causes a 68000 exception if the V flag is one.

The overflow flag is also set if the divisor in a DIVU or DIVS instruction is one or if the state of the sign bit changes as the result of one of the 68000's bit shifting instructions.

The Carry Flag

The carry flag (C) is very similar to the extend flag. In fact, every instruction that sets or clears the extend flag also sets or clears the carry flag. The converse is not true, however, as only the carry flag is affected by certain operations. For example, most data movement operations clear the carry flag only, and only the carry bit participates in the "rotate left" (ROL) and "rotate right" (ROR) bit shifting operations.

The Program Counter

Another important register in the 68000 is the **program counter**. It always holds the address of the position in a program where the 68000 is currently operating. Without the program counter, the 68000 would have no idea of what instruction to execute next.

Although the program counter is a 32-bit register, only the first 24 bits are significant, since the 68000 only has a 24-bit address bus. The program counter can hold any address in the 68000's 16-megabyte range.

Instructions are usually processed in the order in which they appear in a program. After an instruction has been dealt with, the program counter will be pointing to the next in-line instruction. There are ways to skip around a program, of course, such as by using the branch conditionally instructions (Bcc) and the jump (JMP) and jump to subroutine (JSR) instructions. When the program flow changes with a branch or a jump, the program counter is automatically set equal to the target address specified in the instruction.

The program counter is also affected by the 68000 return from subroutine instructions RTS, RTR, and RTE. In each case, an address is removed from the 68000 stack and placed in the program counter so that processing can continue with the instruction following the one that called the subroutine and placed the address on the stack in the first place.

The Addressing Modes

You've already seen that a complete 68000 instruction is made up of an instruction mnemonic and up to two operands that tell the 68000 where to find data to work with and where to store the results. A location to be dealt with by an instruction is called an **effective address** and could be a memory location or an internal register specifically referred to in the instruction, or a memory location calculated by adding together as many as three different quantities. The availability of so many different ways to access data permits you to write very efficient and powerful programs.

You tell the 68000 how to calculate an effective address for an instruction by specifying an addressing mode for each operand the instruction uses. Actually, as you will see, for some instructions you don't have to specify an addressing mode at all if the mode is implicit.

There are 12 fundamental addressing modes you can use with the 68000. Their MDS assembler formats are summarized in Table 1-2, as are the effective address calculations for each

mode. Most other assemblers use these same formats.

Be aware that not all instructions can use each addressing mode. To determine which are permitted, you'll have to consult the detailed description for the instruction in question. In Chapter 3, I'll summarize the addressing modes each instruction can use.

Table 1-2. The 68000 Addressing Modes.

Name of Addressing Mode		*Effective Address Calculation*	*Assembler Syntax*
Data Register Direct		EA = Dn	Dn
Address Register Direct		EA = An	An
Register Indirect		EA = (An)	(An)
Register Indirect with Post-increment		EA = (An), An = An+N	(An)+
Register Indirect with Pre-decrement		An = An−N, EA = (An)	−(An)
Register Indirect with Displacement		EA = (An)+d16	d16(An)
Register Indirect with Index		EA = (An)+(Rn)+d8	d8(An,Rn)
Absolute	short:	EA = (next word)	xxxx
	long:	EA = (next long word)	xxxxxxxx
Program Counter with Displacement		EA = (PC)+d16	d16(PC)
Program Counter Relative with Index		EA = (PC)+(Rn)+d8	d8(PC,Rn)
Immediate	standard:	EA = next word	#xxxx
			or
		EA = next long word	#xxxxxxxx
	quick:	EA = operation word	#xx
Implicit		EA = SR, SP, PC, USP	

Abbreviations:
EA = effective address
Dn = data register (n=0 to 7)
An = address register (n=0 to 7)
d16 = 16-bit signed displacement
d8 = 8-bit signed displacement
Rn = address or data register used as index (n=0 to 7)
PC = program counter register
SR = status register

Table 1-2. *continued*

Abbreviations	SP = stack pointer (same as A7) USP = user stack pointer N = 1 for bytes, 2 for words, 4 for long words ( ) = contents of xx refers to a number from 1 to 8 (ADDQ, SUBQ) or to a number from −128 to +127 (MOVEQ). xxxx and xxxxxxxx refer to numbers of size word and long word, respectively.

Let's look at each of the addressing modes.

Implicit Mode

There are a few 68000 instructions that don't require you to specify all operands because the "missing" operands are implicit to the instruction. The implicit operand usually involves the program counter, the user or supervisor stack pointer, or the status register. For example, when you use the JMP (jump) or Bcc (branch conditionally) instructions, the only operand you have to specify represents the target address of the jump or branch. You don't have to indicate that this address is to be transferred into the program counter because it is implicit. Similarly, the JSR (jump to subroutine) and BSR (branch to subroutine) instructions always save a return address on the stack, so there's no need to spell it out.

Instructions that act on the 68000 status register and the condition code register also use the implicit addressing mode, even though the MDS assembler actually requires you to explicitly refer to SR or CCR in the operand field of the instruction. These are MOVE to/from CCR and MOVE to/from SR, as well as certain logical instructions used to modify bits in the status register (ANDI to SR, EORI to SR, and ORI to SR).

Immediate Mode

An **immediate operand** is a numeric constant (a specific number) that is stored in the one to five words each 68000 instructions occupies. Only a **source** operand can use the immediate addressing mode.

Byte and word operations require one extension word to hold the number; long word operations require two extension words. As we saw earlier in this chapter, these extension words immediately follow the operation word for the instruction.

When you use the immediate addressing mode to refer to a constant, the 68000 simply reads it directly from the extension word or words. That means the effective address is simply the address of the instruction's extension word. The assembler format for the immediate addressing mode is:

```
#number
```

or

```
#symbol
```

where number represents a number within the range allowed by the operand, and symbol represents a symbolic name for a numeric constant. A number can be decimal or hexadecimal; hexadecimal numbers are preceded by a $ sign. Symbols are defined in source code using the EQU or SET assembler directives (see Chapter 2).

The # symbol preceding the number or symbol is very important; if you don't include it, the instruction is assembled as if the operand were an address, rather than a constant, so the program won't work as expected.

Here are examples of some immediate operands:

```
#45          decimal 45
#$2D         hexadecimal 2D (decimal 45)
#mySize      "mySize" is the EQU symbol for a number
```

There is a special "quick" form of the immediate addressing mode you can use if the number is small enough to fit within the operation word itself. For example, if you're adding or subtracting an immediate quantity from one to eight, you can use the ADDQ and SUBQ instructions. To move an immediate quantity from −128 to 127 to a destination, you can use the MOVEQ instruction.

The MDS assembler automatically optimizes your code by converting any ADD, SUB, and MOVE instructions to the quick equivalent if the immediate operand is in the range permitted by the quick form.

Data Register Direct Mode

When you use this addressing mode the operand is simply one of the eight data registers, D0–D7. To select this mode when you write a program, refer to the data register as Dn (n = 0 to 7). Here are some simple examples of how to use the data register direct addressing mode:

```
MOVE  #4,D0       ;put a 4 into D0.W
CLR.L D4          ;clear D4.L to zero
```

(Notice that the semicolon indicates the beginning of a comment in an assembler source statement.)

When you use the data register direct addressing mode, you can deal with operands that are three different sizes: byte, word, and long word. To indicate the size of the operand, attach one of the following suffixes to the standard instruction mnemonic:

.B (byte)
.W (word)
.L (long word)

When you do this, only the first eight bits (byte), 16 bits (word), or 32 bits (long word) of the data register are used by the instruction.

If you don't specify a suffix, the MDS assembler assumes you're dealing with an operand size of word. The MOVE example we just looked at, for instance, only acts on the lower 16 bits of the data register.

Address Register Direct Mode

The operand here is one of the eight address registers, A0–A7. The assembler format parallels that used for the data register direct mode: You refer to the address register by An (n = 0 to 7). (You can also refer to A7 as SP, if you wish.) You cannot, however, move byte quantities into an address register and, as explained earlier in this chapter, word quantities are sign-extended across the upper part of the register.

Here's an example of how to load the address of the data area whose **label** is "Record" into the address register A0:

```
LEA          Record,A0        ;Move the address of "Record" into A0
```

(A **label** is a symbolic name associated with an instruction or a data area within a program.)

Although you don't have to store addresses in an address register, you will often use them for that purpose so you can take advantage of the 68000's powerful address register indirect addressing modes. These modes are typically used after first storing the base address of a data structure in an address register. They cannot be used with data registers.

Address Register Indirect Mode

The operand here is one of the eight address registers, but the effective address that is acted on is the address stored in the register, not the register itself (see Figure 1-4). To indicate this addressing mode, enclose the name of the address register in parentheses. The assembler format for this addressing mode is:

```
(An)
```

Here are some examples:

```
MOVE.L  #4,(A3)    ;Store 4 (long word form) at the address contained in A3
MOVE    D0,(A2)    ;Store what's in D0 (word) at the address contained in A2
```

You should be careful not to confuse this indirect mode with the direct mode, where data is read from or written to the register itself. Whenever you see the parentheses, think "contents of".

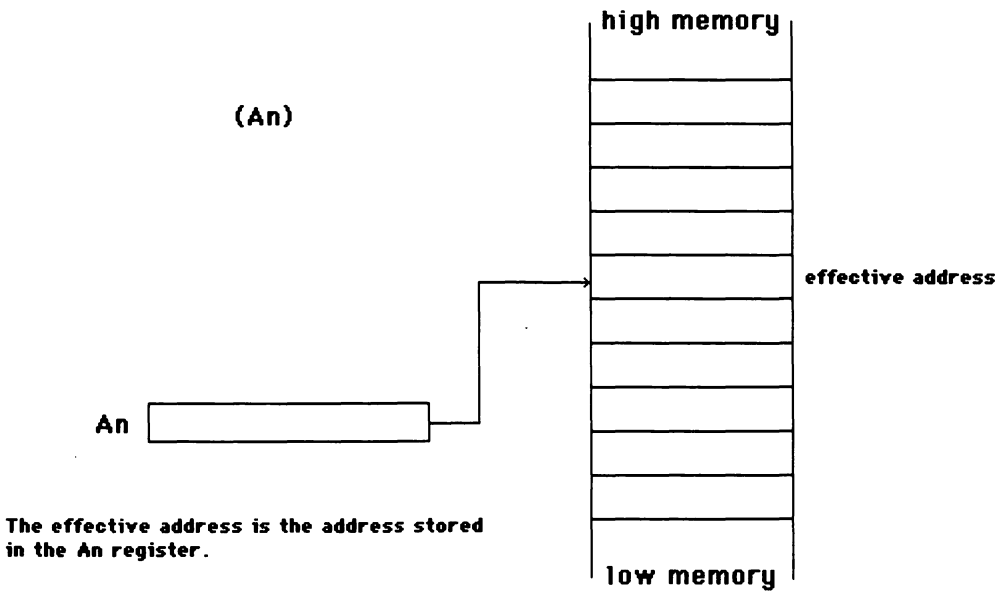


Figure 1-4. Address Register Indirect Addressing Mode.

Also be careful that the address register does not contain an odd address when the operand size is a word or long word. When you try to execute such an instruction, the 68000 generates an address error exception and you'll see the infamous Macintosh bomb box. I'll discuss exceptions in detail later in this chapter.

Address Register Indirect with Post-Increment Mode

This addressing mode is quite similar to the address register indirect mode, except after the effective address is used, the number in the address register is incremented by one (for a byte operation), two (for a word operation), or four (for a long word operation). (See Figure 1-5.) The addressing mode is designated by placing parentheses around the name of the address register, followed by a plus sign to indicate the post-increment operation:

```
(An)+
```

The post-increment addressing mode is very useful when you're moving ranges of memory from one place to another because you don't have to follow each move with extra instructions to increment the base pointers for the two blocks. For example, consider the instruction:

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

If you execute this 10 times in a row, the 10 words beginning at the initial address stored in A0 will be moved to a block beginning at the initial address stored in A1. After each word is moved, the address registers are incremented by two (it's a word operation) so that you're ready to move the next word.

Post-increment addressing is also handy for removing (**popping**) data that is passed on the 68000's stack. To do this, you would execute an instruction such as:

```
MOVE    (SP)+,D0        ;Take a word off the stack
```

As you'll see later on, the **stack** is a data structure that grows down in memory: When you add data to it, the stack pointer is decreased, and when you remove data from it, the stack pointer is increased.

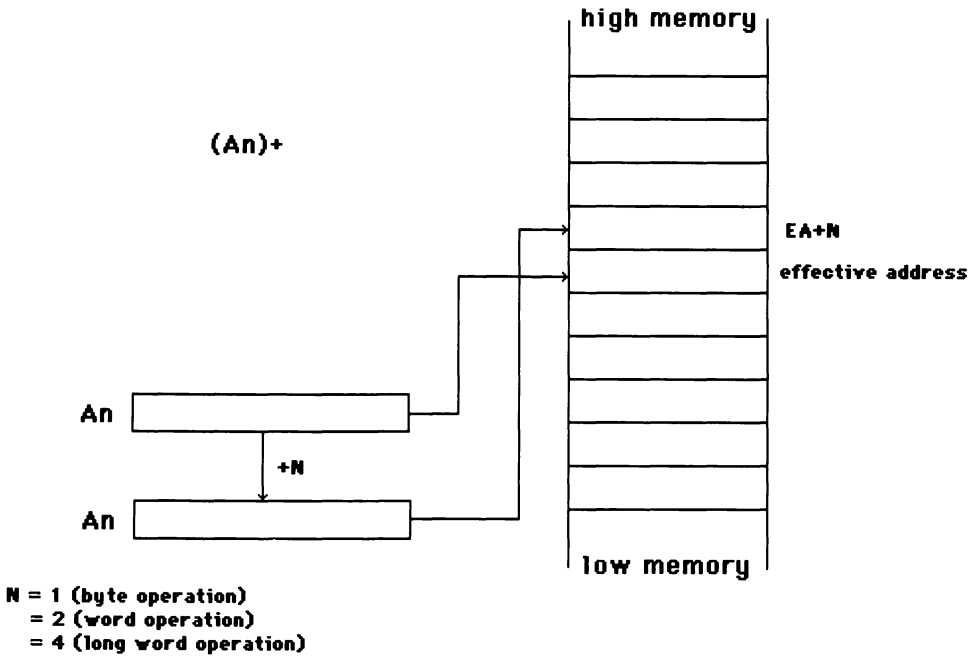


Figure 1-5. The Address Register Indirect With Post-Increment Addressing Mode.

Address Register Indirect with Pre-Decrement Mode

This addressing mode is primarly used to add items to the stack. It is designated by placing a minus sign before the name of an address register enclosed in parentheses:

```
-(An)
```

The effective address is calculated by first decrementing the contents of the address register by one (for a byte operation), by two (for a word operation), or by four (for a long word operation). The address acted on is the new address in the address register. (See Figure 1-6.)

For example, to place a long word operand on the stack, use an instruction like

```
MOVE.L  D0,-(SP)
```

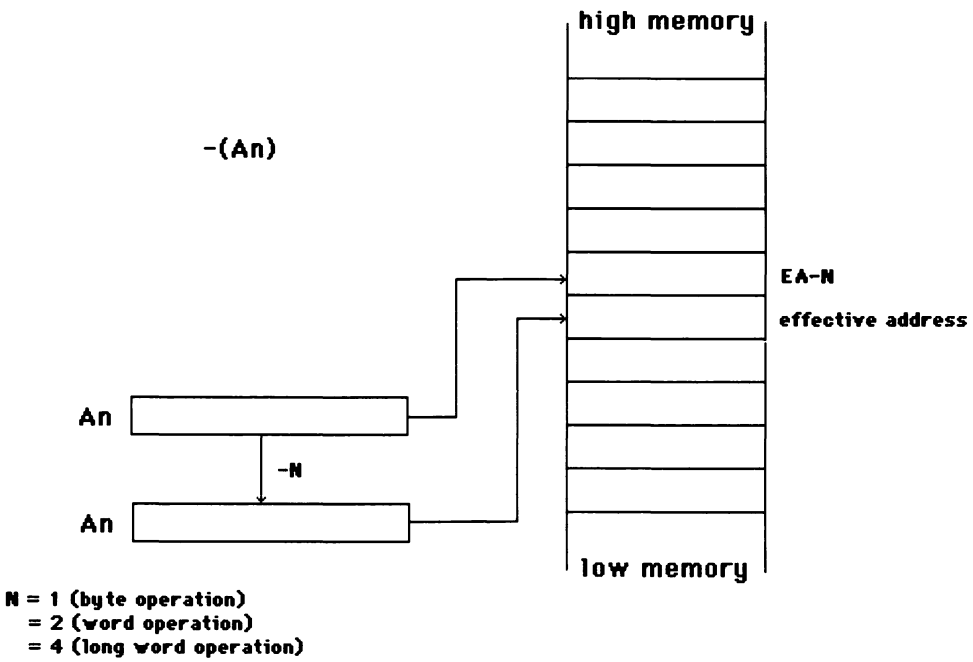


Figure 1-6. Address Register Indirect With Pre-Decrement Addressing Mode.

When this instruction is executed, the stack pointer is first decremented by four and then the contents of D0 are placed at the address stored in the stack pointer register.

Address Register Indirect with Displacement Mode

The effective address here is calculated by adding a 16-bit sign-extended word to the address stored in an address register. (See Figure 1-7.) The assembler format for this addressing mode is:

```
d16(An)
```

or

```
label(An)
```

where d16 represents a number from −32768 to 32767 and label represents the EQU symbol for such a number, or the label for an instruction or data area. If you use a number or label, do not precede it with #, as you would if using the immediate addressing mode.

This addressing mode is typically used when you want to access a specific item in a data structure whose base address is stored in the address register. For example, if there's a data structure beginning at location "Addresses" in your program, and you want to read the third word (which occupies bytes 4 and 5; numbering begins with 0) in the structure, you would use the instruction:

```
LEA     Addresses,A0    ;Load base address into A0
MOVE    4(A0),D0        ;Third word starts at offset 4
```

You'll be using this addressing mode quite often on the Macintosh because any variables you define within the program (using the DS assembler directive) must always be ref-

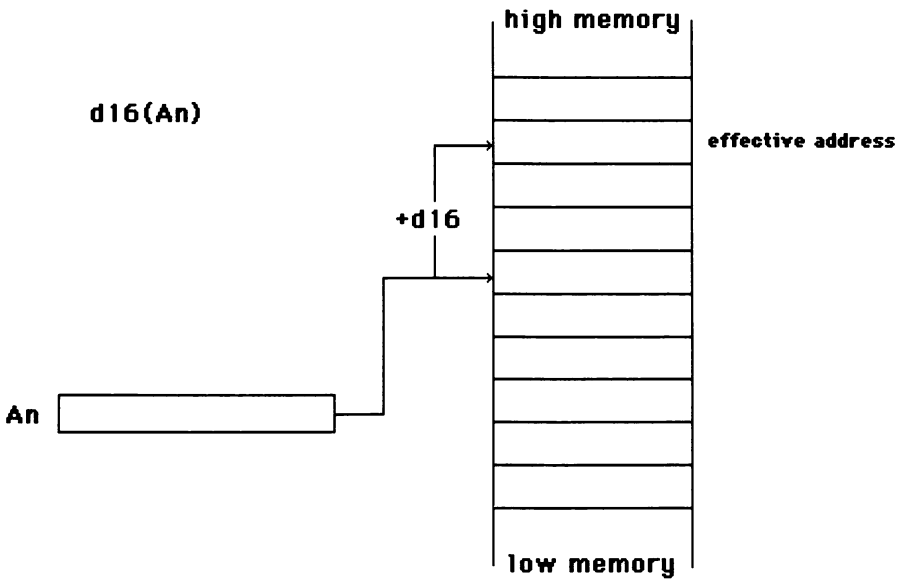


Figure 1-7. Address Register Indirect With Displacement Addressing Mode.

erenced as displacements from the address stored in the A5 register. So, for example, if you call your variable "MyRecord", its base address is given by a "MyRecord(A5)" operand and the address of the third word in the data structure is "MyRecord+4(A5)". The assembler calculates the (A5) offset by adding MyRecord and 4 at assembly time.

Address Register Indirect with Index Mode

The effective address for this addressing mode is calculated by adding together the number stored in an address register, an index that is stored in another address register or a data register (sign-extended word or long word), and an 8-bit, sign-extended, displacement byte (see Figure 1-8). The assembler format for this mode is:

```
dd(An,Rn)
```

or

```
label(An,Rn)
```

where `Rn` represents the data or address register being used as the index register, `dd` represents a number, and `label` represents the symbolic name for a number.

To indicate whether the index register is a word or a long word, add a .W or .L suffix to its name; some examples are D3.L, A0.W, and D2.W. If no suffix is specified, a word index is assumed.

This addressing mode requires one extension word. The first byte contains information relating to the index register and its size. The second byte contains the 8-bit displacement. Note that since the displacement byte is only 8 bits long, the displacement range is from −128 to +127. (The base point is the word following the operation word.)

This indexed addressing mode is useful when you're working with a group of fixed-length records. To access any record in the group, first store the address of the first record

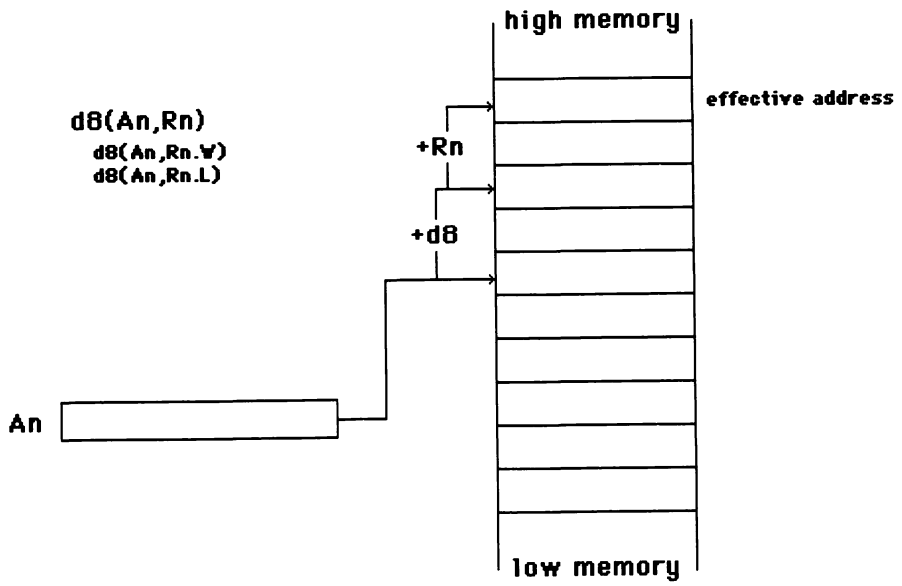


Figure 1-8. Address Register Indirect With Index Addressing Mode.

(record 0) in an address register, then multiply the record number you want by the number of bytes in a record, and store the result in the index register. When you access the record, you can use the displacement byte to step to the field in the record you are interested in.

Absolute Modes

The effective address for the **absolute long** addressing mode is stored in two extension words for the operand. It represents a specific address in the 68000's 16-megabyte memory space that the operand is to use.

The assembler format for this addressing mode is simply:

```
$xyzzzz          ;xy > 00   z = 0..F
```

or

```
label
```

where `label` is a symbolic label for a fixed memory location defined using the EQU or SET assembler directive. (See Chapter 2.)

If label refers to a position within your application program, the MDS assembler uses the program counter with displacement addressing mode, label (PC), instead.

There is one other variant of the absolute addressing mode: **absolute short**. In this case, the effective address is formed by sign extending the 16-bit address stored in the extension word for the operand. The assembler format is:

```
$00xxxx          ;x = 0..F
```

or

```
label
```

Since the address is sign-extended, this addressing mode can only be used if the address is in the first 32K or last 32K of memory.

Program Counter with Displacement Mode

The effective address in this mode is the sum of the address in the program counter register and a sign-extended, 16-bit displacement word stored in an extension word after the operation word. (See Figure 1-9.)

The assembler format is:

```
label(PC)
```

or

```
label
```

where `label` represents the symbolic label for a position within the program. The assembler automatically determines what the displacement between the program counter and the

labeled position is and puts it into the executable object code for the program. You should never have to calculate it yourself.

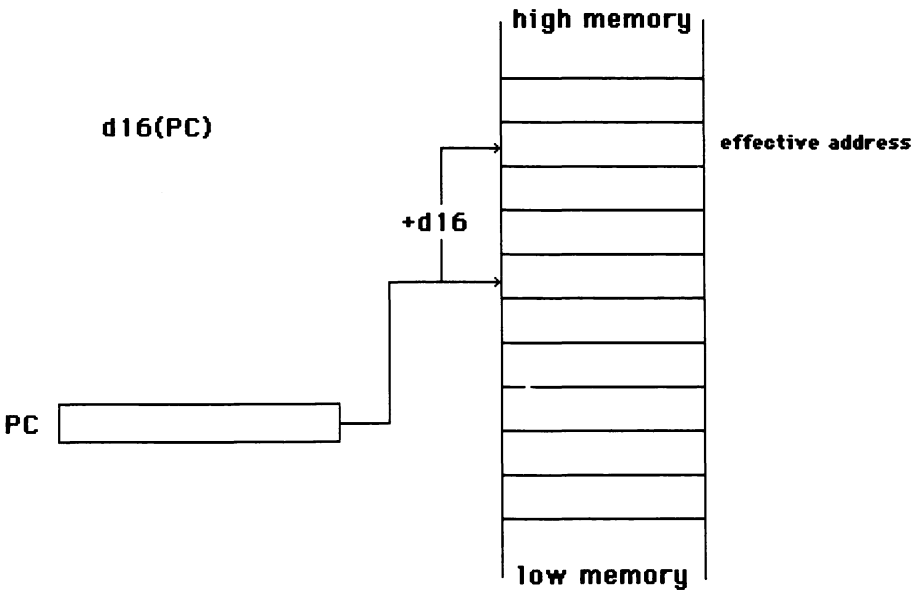


Figure 1-9. Program Counter With Displacement Addressing Mode.

Program counter with displacement is a very important addressing mode on the Macintosh because all the programs you write must be **relocatable**, able to run at any position in memory. This means that no part of the program must refer to an absolute location if that location is within the body of the program. By referring to addresses in a relative way, the operating system can load and run the program anywhere it wants. In fact, the MDS makes it very difficult to write non-relocatable code because all references to labels are relative to the program counter; the alternative mode, absolute long addressing, is not used in these circumstances.

Program Counter with Index Mode

The effective address for this mode is the sum of the address in the program counter, the long word or sign-extended word in a data or address register used as an index, and a sign-extended 8-bit displacement byte. (See Figure 1-10.) The assembler format is:

```
label(PC,Rn)
```

where `Rn` represents the data or address register being used as the index and label represents the location of another instruction in the program. The assembler takes care of converting this location into an 8-bit offset from the value of the program counter.

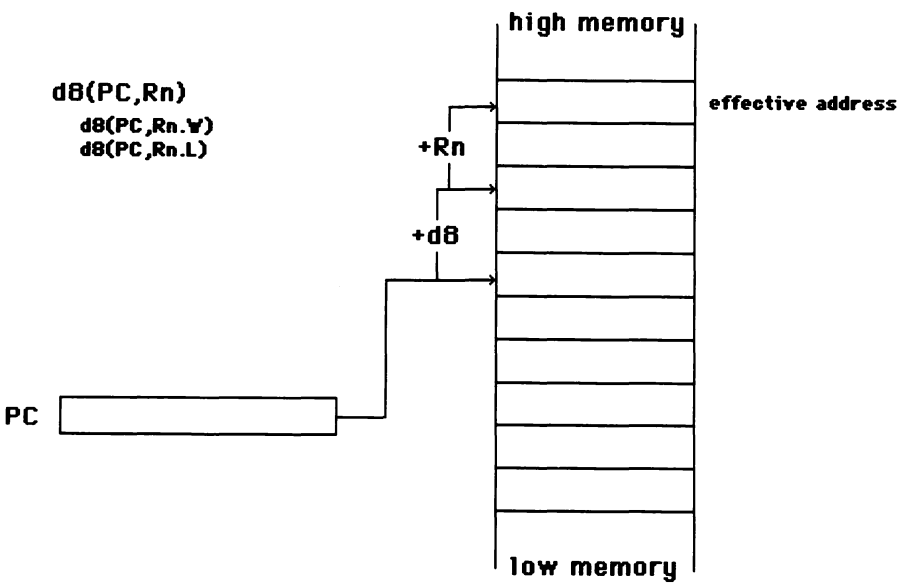


Figure 1-10. Program Counter With Index Addressing Mode.

As with the address register indirect with index mode, the index register can be a word or a long word. Use the .W or .L suffix to identify its size.

Note that if the index register is a data register, you can also use the following assembler operand format:

```
label(Dn)
```

You don't have to specifically refer to the PC register. You can't omit the reference to PC if the index register is an address register because "label (An)" is used to indicate that the address register indirect with displacement addressing mode is to be used.

The program counter with index addressing mode requires one extension word. The first byte contains information relating to the index register and its size. The second byte contains the 8-bit displacement. Note that since the displacement byte is only 8 bits long, the displacement range is from −128 to +127. (The base point is the word following the operation word.) This means that the data structure you're indexing into must be quite close to the instruction.

The program counter with index addressing mode is quite similar to the address register indirect with index we looked at earlier. In fact, it's possible to write code to access records in a data structure that uses either addressing mode. The advantage of using the program counter with index mode is that you don't "waste" an address register for storage of the base address. The disadvantage is that the base address must be within the range covered by the 8-bit displacement byte.

The Stack

The **stack** is a last-in, first-out (LIFO) data area in RAM that is implicitly used by several 68000 instructions for storage or retrieval of data. The LIFO characteristic means that the last item placed (or **pushed**) on the stack will be the first item removed (or **pulled**) from the stack. Figure 1-11 shows, symbolically, what a stack looks like and where data is pushed on it and pulled from it.

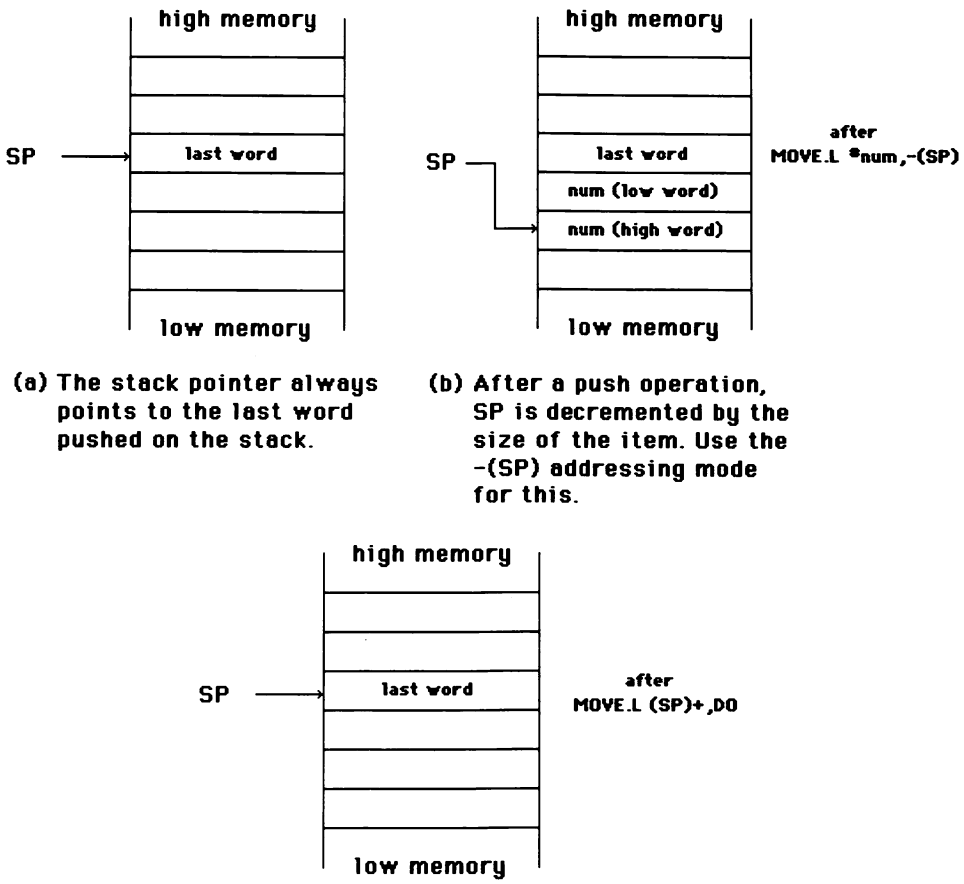


Figure 1-11a, b, c. The 68000 Stack and Stack Pointer.

The address of the top of the stack (the address of the last item pushed on the stack) is stored in the current stack pointer register, SP, the active A7 register. (The Macintosh operating system initializes its value when you start up.)

Remember that SP refers to either the supervisor stack pointer (A7) or the user stack pointer (A7'), depending on whether the supervisor state flag (S) in the status register is on or off.

When data is pushed on the stack, the stack pointer is first *decremented* by the size of the data and then the data is stored at the address stored in the stack pointer. Be aware, however, that if you try to push an odd number of bytes on the stack, the 68000 first pushes one extra byte on the stack to ensure that the stack pointer always contains an even address.

The instructions that implicitly push data on the stack are JSR, BSR, and PEA (all of which push long word addresses); and LINK, which pushes the contents of a 32-bit address register and then creates a data "frame" within the stack by decrementing the stack pointer by the number of bytes specified in the operand.

Since the stack pointer is decremented when something is placed on the stack, the stack grows downward in memory.

When you pop data from the stack, the stack pointer is incremented by the size of the data, or by the size of the data plus one if you try to pop a single byte.

The instructions that implicitly pop data from the stack are RTS, which pops a long word address pushed there by a JSR instruction and puts it in the program counter; RTR and RTE, both of which pop a word into the status register and a long word address into the program counter; and UNLK, which restores the initial value of the stack pointer to deallocate a data frame created by LINK and then pops a long word into an address register.

You can also explicitly use the stack for temporary data storage by using the A7 indirect with pre-decrement addressing mode, −(SP), to place data on the stack. When you want to remove data, you can use the corresponding post-increment addressing mode, (SP)+. You'll see in later chapters that most of the internal ROM subroutines used on the Macintosh to perform standard operations are accessed by passing parameters on the stack using the pre-decrement

addressing mode; function results are retrieved from the stack using the post-increment addressing mode.

The stack can be positioned anywhere in memory simply by adjusting the value of SP. It's usually placed near the top of the available RAM space so that as large a space as possible will be left for a program to use. You'll see where it's positioned on the Macintosh in Chapter 4.

Exceptions

Under normal circumstances, the 68000 keeps busy by executing program instructions in the order dictated by the program logic. There are several special events called **exceptions**, however, that can temporarily interrupt the natural flow of a program, and force the 68000 to enter an **exception processing** state. As shown in Table 1-3, these events are generated by external input/output devices, internal errors, or certain 68000 instructions.

Table 1-3. The 68000 Exception Vectors.

Vector Number	*Vector Address (hexadecimal)*	*Description of Exception*
0	$000	Reset: initial SSP
	$004	Reset: initial PC
2	$008	Bus Error
3	$00C	Address Error
4	$010	Illegal Instruction
5	$014	Divide by Zero
6	$018	CHK Instruction
7	$01C	TRAPV Instruction
8	$020	Privilege Violation
9	$024	Trace
10	$028	Line "A" Emulator
11	$02C	Line "F" Emulator
12	$030	[reserved]
13	$034	[reserved]
14	$038	[reserved]

Table 1-3. *continued*

Vector Number	*Vector Address (hexadecimal)*	*Description of Exception*
15	$03C	Uninitialized interrupt vector
16–23	$040–$05F	[reserved]
24	$060	Spurious Interrupt
25	$064	Level 1 autovector (VIA)
26	$068	Level 2 autovector (SCC)
27	$06C	Level 3 autovector (VIA+SCC)
28	$070	Level 4 autovector (switch)
29	$074	Level 5 autovector (VIA+switch)
30	$078	Level 6 autovector (SCC+switch)
31	$07C	Level 7 autovector (VIA+SCC+switch)
32–47	$080–$0BF	TRAP #n Instruction Vectors
48–63	$0C0–$0FF	[reserved]
64–255	$100–$3FF	User Interrupt Vectors (*)

(*) The user interrupt vectors are actually used for storage of global variables on the Macintosh.

Each of the events that can trigger exception processing is associated with a **vector** number from #0 to #255. The vectors themselves are long words, stored in the first 1024 bytes of the 68000 memory space, and which hold the address of the program to which control is to pass when the exception occurs. To calculate the location of the vector from a given vector number, simply multiply the vector number by four. This means that the vectors occupy the address space from $000000 to $0003FF.

The exception-handling subroutine is responsible for properly servicing the exception request before returning control to the main program at the point where the exception occurred. The operating system automatically installs a default set of exception subroutines when the Macintosh first starts up. You normally don't have to change these unless you want to take advantage of an exception that the

operating system doesn't normally use, or if you want to change what happens when a given exception occurs.

Before we begin looking at each type of exception in detail, let's review exactly what happens when an exception occurs. It's a four-step process:

- The 68000 makes an internal copy of the status register and then sets the supervisor state flag to one and clears the trace mode flag to zero. This means the 68000 will commence operating in supervisor mode (if it wasn't already) and instruction tracing will be turned off. If the exception was caused by reset or by a hardware interrupt, the interrupt level mask in the status register is also changed, as explained below.
- The 68000 determines the vector number for the interrupt and uses it to calculate the address of the vector (by multiplying by four).
- The 68000 pushes on the supervisor stack the program counter (it contains the address of the next instruction to be executed in the interrupted program), followed by the previously saved copy of the status register. These two pushes aren't made if the exception is caused by a reset.
- The new program counter value is loaded from the exception vector.

Once these steps are completed, the 68000 starts executing the exception handling subroutine. Such a subroutine finishes with an RTE (return from exception) instruction that restores the original values of the program counter and the status register from the stack so that execution of the main program will continue on as if nothing had happened.

Let's take a more detailed look at the types of exceptions handled by the 68000.

The Reset Exception

A reset exception (vector #0) occurs when you first turn on the Macintosh or when you press the front part of the programmer's switch on the side of the Macintosh. The interrupt mask bits in the status register are set to 111 by the 68000 when a reset occurs. On the Macintosh, the subroutine that

handles the reset exception boots the system from disk.

Unlike any other exception vector, the reset vector requires two long words of storage so it actually occupies the space you'd think would be used by exception vector #1 (such an exception can't occur with the 68000). The first long word contains the new value of the supervisor stack pointer and the second long word contains the new value of the program counter.

If you look at the values stored in the reset vector after you've turned on the Macintosh, you'll become very confused because the second long word doesn't point to the standard disk boot subroutine. In fact, it doesn't point anywhere in particular. This is because the Macintosh has been designed to temporarily remap its memory space when a reset exception occurs so that the address of the base of the ROM space, which is normally $400000, becomes $000000. Thus, it is the long words at $400000 and $400004 that are used to fill the stack pointer and the program counter when a reset occurs. The ROM is remapped to $400000 during the reset sequence. If resets weren't handled this way, you wouldn't be able to start up the Mac because a RAM-based reset vector would contain random data on power up.

The Internally Generated Exceptions

BUS ERROR (vector #2). If implemented by the system hardware, this exception occurs if you try to address an area of memory that doesn't exist. Bus errors cannot occur on the Macintosh.

ADDRESS ERROR (vector #3). This exception occurs when the 68000 tries to access a word or long word operand that begins at an odd address or when it tries to execute an instruction that begins at an odd address. This type of error is very serious and causes a fatal bomb box to appear on the Macintosh screen.

PRIVILEGE VIOLATION (vector #8). This exception occurs if you're in user mode and you try to execute an instruction

that is valid in supervisor mode only. Since you're always in supervisor mode on the Macintosh (or should be!) this exception should never occur.

TRACE (vector #9). This exception occurs after every instruction is executed if the trace flag (T) in the system byte of the status register is one. Debugging programs such as MacsBug typically handle trace exceptions by displaying the contents of all the registers, the stack, an area of memory, or other information that may assist a programmer in determining whether a program is executing as expected.

The Externally Generated Exceptions

An exception caused by a peripheral device is called an **interrupt**. An interrupt is a simply an electrical signal from the peripheral port, such as one of the Mac's serial ports, indicating that an event has just occurred that should be dealt with immediately, such as the arrival of data from a modem. The 68000 normally responds to an interrupt by halting execution of the main program in memory, servicing the source of the interrupt, and then returning to the main program. By using interrupts to signal events, the 68000 does not have to worry about missing events that might occur while it's performing time-consuming operations. If you couldn't use interrupts, you'd have to do periodic status checks for incoming information and this would slow down your program.

A different priority level, from 1 to 7, is assigned to each source of interrupts when a 68000-based system is designed. Here are the priority levels for the interrupts possible on a Macintosh:

- level 1: VIA interrupts
- level 2: SCC interrupts
- level 3: VIA and SCC together
- level 4: interrupt button on programmer's switch
- level 5: interrupt button and VIA together
- level 6: interrupt button and SCC together
- level 7: interrupt button and VIA and SCC together

The VIA is the 6522 Versatile Interface Adapter used to control the mouse and clock. The SCC is the 8530 Serial Communications Controller that controls the two serial ports on the Macintosh.

The interrupt mask in the status register lets you prevent certain interrupts from occurring. Any interrupt having a level at or below the level of the mask is ignored by the 68000. The exception is a level 7 interrupt; it is always permitted and is referred to as a **non-maskable** interrupt.

All interrupts can be enabled by storing 000 in the mask, and this is the value the Macintosh operating system uses.

The interrupt mask is automatically changed by the 68000 when an interrupt of a certain level is dealt with. As part of the interrupt handling process, the mask is changed to the current level being processed so that lower- or equal-priority events can't interfere with the handling of the current interrupt. The normal mask is restored after interrupt processing is finished.

AUTOVECTOR INTERRUPT (vectors #25 to #31). A peripheral device can use one of two techniques to interrupt the 68000, depending on how the hardware interface has been configured. One alternative is to have the peripheral provide a vector number between 64 and 255 to the 68000 when the interrupt occurs. The other technique, the one used on the Macintosh, is for the 68000 to use one of the seven **autovector** exception vectors. The one used will depend on the interrupt priority level: vectors #25 to #31 correspond to priority levels 1 to 7, respectively.

USER INTERRUPT (vectors #64 to #255). Since all interrupts are handled by autovectors on the Macintosh, these vectors aren't used for storage of interrupt vectors. The Macintosh operating system, however, uses this space (from $100 to $3FF) for the storage of global system variables, so you can't use this area for your own purposes.

SPURIOUS INTERRUPT (vector #24). This exception occurs only if a bus error condition is detected during the handling of another exception. Since bus errors can't occur on the Macintosh, this exception can't occur either.

The Exceptions Caused by Instructions

An exception caused by a 68000 instruction is called a **trap**. There are two classes of traps: unconditional and conditional.

Unconditional Traps

The following trap instructions unconditionally cause exception processing to begin:

```
TRAP #n          ;n = 0 to 15 (trap number)
ILLEGAL          ;special "illegal" instruction
$Fxxx            ;any instruction whose operation
                 word begins with 1111 ($F)
$Axxx            ;any instruction whose operation
                 word begins with 1010 ($A)
```

TRAP (vectors #32 to #47). When one of the 16 TRAP #n instructions is encountered, control passes to the address stored in exception vector n+32. On some 68000-based systems, the trap instructions are used to invoke the fundamental I/O operations supported by the computer's operating system. The Macintosh operating system does not use these vectors, however, so they're free for use by your own application programs. The MacsBug debugger, for example, uses the TRAP #15 instruction to implant software **breakpoints** in a program. (A breakpoint is a position in a program where you want processing to stop so you can examine registers to verify that all is going well.)

ILLEGAL (vector #4). The ILLEGAL instruction is one that does not correspond to any other documented instruction. All other undocumented instructions are reserved for future extensions to the 68000 instruction set, so you should not use them to generate an illegal instruction exception.

The exception vector for the ILLEGAL instruction is not used by the Macintosh operating system, so it's available for your own use.

$Fxxx and $Axxx (vectors #11 and #10). Any instruction whose operation word is of the form $Fxxx or $Axxx (xxx represents three hexadecimal digits) will also cause an exception. The **line F emulator** exception ($Fxxx) is not used by the Macintosh operating system, so you can install your own code to handle these types of instructions. Such code would typically examine the unused 12 bits of the operation word to determine the exact nature of the instruction, much like the 68000 does when it interprets a standard instruction. Data for the instruction could also be passed on the stack and results could be returned on the stack, in accordance with a pre-defined software protocol.

The **line A emulator** trap ($Axxx), on the other hand, is extensively used by programs running on the Macintosh to access a group of about 500 standard subroutines contained in the Macintosh ROM area. These subroutines are logically divided into two groups:

- operating system calls
- user-interface toolbox calls

The **operating system calls** perform fundamental low-level operations, such as communicating with peripheral devices or allocating and deallocating blocks of memory. The **user-interface toolbox calls** include subroutines for implementing pull-down menus, windows, scroll bars, and most other features defined by the standard Macintosh human-interface guidelines.

Apple has assigned standard mnemonic names to each of the $Axxx trap instructions; they are easily identified because they all start with the underscore symbol. The names are defined in standard trap files that come with the MDS assembler. Each line in these files is of the form:

```
.TRAP  _SubroutineName $Axxx    ;xxx = hex digits
```

.TRAP is an assembler directive that assigns the name following it to the line A emulator instruction on the right. (See Chapter 2.)

The Macintosh operating system stores the address of a standard trap handler in the $Axxx vector when the system starts up. This handler takes the last nine bits (for a toolbox instruction) or eight bits (for an operating system) of the $Axxx word, multiplies it by four, and uses the result as an index into a trap dispatch table extending from $400 to $7FF. The entry in the table is the address of the subroutine to be called (in slightly encoded form on the 64K ROM version of the Macintosh). When the subroutine ends, control returns to the instruction following the trap in the main program.

Conditional Traps

There are also four instructions that may cause an exception to occur, depending on the state of the condition codes or the results of a calculation:

```
DIVU              if division by zero
DIVS              if division by zero
TRAPV             if the overflow flag (V) is 1
CHK <ea>Dn        if data register is out of range
```

ZERO DIVIDE (vector #5). If the divisor you specify for a DIVU (unsigned divide) or DIVS (signed divide) instruction is zero, a zero divide exception occurs.

TRAPV (vector #7). This exception occurs if you execute the TRAPV instruction when the overflow flag in the status register is set to 1.

CHK INSTRUCTION (vector #6). The CHK instruction is always of the form:

```
CHK  <ea>,Dn
```

where <ea> designates any valid addressing mode yielding an effective address. If the signed number stored in Dn is less than zero or greater than the value specified by the source operand, an exception occurs.

Chapter 2
Assembler Tools

In this chapter we're going to take a look at most of the programs that make up the Macintosh 68000 Development System (MDS) published by Apple. The MDS contains all the tools you'll need to develop assembly language programs on the Macintosh, including:

- Edit, an **editor** for creating 68000 assembly language source code files and MDS control files;
- Asm, an **assembler** for converting source code files into object code modules;
- Link, a **linker** for combining one or more object code modules into a single application;
- RMaker, a **resource compiler** for converting source code defining standard data structures such as menus, windows, and icons into object code modules, and for adding resources to applications; and
- Exec, an **executive program** for automating the entire assembly, linking, and resource compilation process.

A flowchart of the usual assembly/linking/compilation procedure is shown in Figure 2-1. Variations are possible, but I'll be following this basic procedure to develop most of the programs in the book. The usual procedure goes something like this (the numbers refer to the steps shown in Figure 2-1):

(1,2,3) The editor is used to produce source code for the assembler, a linker control file for the linker, and resource source code for the resource compiler.

(4) The assembler is used to convert program source code files into object code (.Rel) modules.

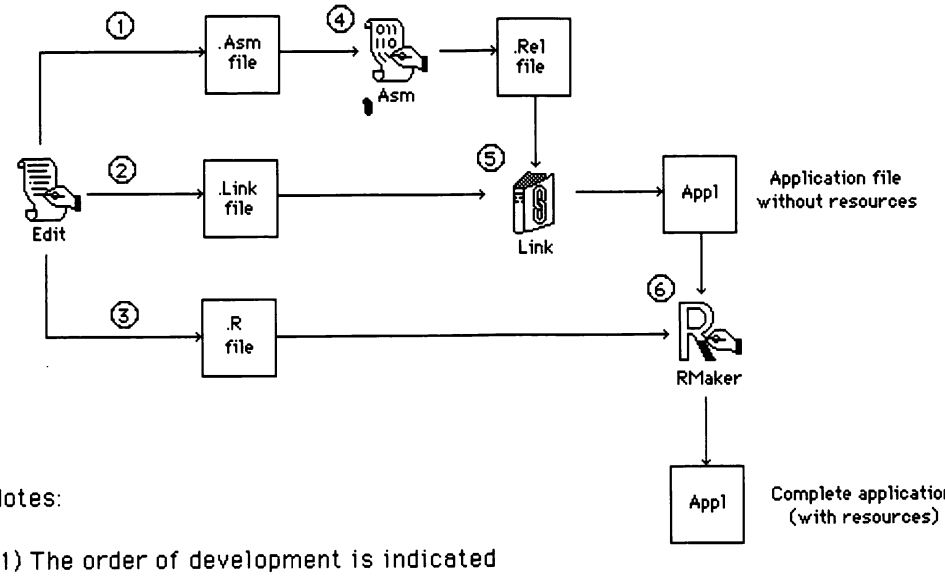
(5) The linker is used to process a linker control file that tells it how to combine object code modules into an application

file. After linking, the application's resources are usually not yet available.

(6) The resource compiler is used to create the resources used by the program and either append them to the application file or store them in a separate resource file.

(Later in this chapter, you'll see how to combine steps 4, 5, and 6 by executing a .Job file with the Exec program.)

Once these steps are completed, the application can be run by double-clicking its icon on the Finder's desktop.



Notes:

(1) The order of development is indicated by the circled numbers.

(2) Steps 4, 5, and 6 can be combined by executing a .Job file with the Exec program.

Figure 2-1. A Flowchart Showing One Way to Develop an Application with MDS.

In this chapter we'll also look at some of the standard symbol definition files that come with MDS, explain what they contain, and show how to use them to your advantage.

Finally, you'll see how to translate calls to standard toolbox trap subroutines in the Macintosh ROM, which are usually documented in terms of Pascal procedures and functions, into the equivalent assembly language calling sequences.

At the end of the chapter, I'll put everything together and show you how to use the MDS tools to create a simple application that will serve as the shell for some of the short programs and subroutines developed in later chapters.

The Editor

The MDS editor is called **Edit**. With it you can create the source documents used by the assembler, linker, resource compiler, and executive. These documents are stored as standard text files, so you can also modify or create them with any other editor or word processor that supports such files.

To invoke the editor from the Finder you can either double-click its icon or double-click one of the text files you've previously created with it.

There is nothing really exciting about the MDS editor; it's basically a vanilla editor with few of the frills you might find in a serious word processing program like MacWrite or Microsoft Word. For example, you can't change the size or style of a portion of text in the file, only the entire file. It is, however, all you need to quickly and easily develop source code for the other MDS programs.

The Assembler

The MDS assembler is called **Asm**. Use it to convert a 68000 assembly language program from source code form to a relocatable object code form suitable for subsequent processing with the linker. You invoke it by double-clicking the Asm icon and selecting a file name or by passing control to it

directly from the linker or the resource compiler Transfer menus. The object code files created by the assembler have .Rel suffixes (which stands for relocatable).

Listing 2-1. A Simple Program Showing the Formats of Lines of 68000 Source Code.

```
* Format.Asm
*
* This simple program shows the formats of typical lines
* of 68000 assembly language code.
* An asterisk in column 1 means the entire line is a comment
; (A semicolon works, too!)


        .TRAP   _SysBeep        $A9C8
        .TRAP   _Button         $A974

Start   NOP                     ;"Start" is a regular label

@1      BSR     GetDur          ;Comment field begins with a ;
        BSR     Beep

        CLR.B   -(SP)           ;Space for result
        _Button
        TST.B   (SP)+           ;Is mouse button down?
        BNE     Start           ;Yes, so keep beeping

        RTS                     ;@1 is a local label
Beep    TST     Duration(A5)    ;Is Duration zero?
        BEQ     @1              ;If so, branch

        MOVE    Duration(A5),-(SP)
        _SysBeep                ;Toolbox traps begins with "_"

@1:     RTS                     ;Not the same as the other @1
                                ;Indented, so followed by ":"

GetDur  MOVE    #30,Duration(A5)   ;Initialize beep length
        RTS

Duration        DS      1       ;Define space for variable
```

Source Code Format

The source code for an assembly language program is stored in a standard text file, and is created with the MDS editor. The program source code must adhere to certain rules of syntax dictated by the assembler, some of which were referred to in Chapter 1 when we looked at instruction names and addressing mode formats. Each line in the source file is composed of four **fields**, each separated from the next by one or more tab characters or blank spaces. (See Listing 2-1.) These fields are the **label field**, the **instruction field**, the **operand field**, and the **comment field**.

The Label Field

A **label** is a symbolic name for a position in a program or a piece of data. It must begin with a letter (A..Z or a..z), a period (.), or an underscore (_); subsequent characters can also include digits (0..9) and dollar signs ($). If the label does not begin in the first column of the line, it must end with a colon (:) so that it will not be mistaken for an instruction mnemonic.

A special form of label, called a **local label** begins with an at sign (@) and is followed by a single digit (0..9). (If it is indented you must also include a colon.) A given local label has meaning only between two standard labels and can only be referred to by instructions within that range. This means you can use the same local label in another area of a program without causing an assembly error. Local labels are handy for identifying instructions not referred to in other parts of a program.

Labels are primarily used for two purposes: to provide a symbolic name for a piece of data or for the target address of a branch or jump instruction. By using labels, you never have to worry about calculating offsets when you're using program counter relative addressing modes; it's done for you

automatically by the assembler. The use of labels also makes a program listing much easier to understand.

The Instruction Field

An instruction is either a mnemonic for a 68000 instruction or one of several assembler directives supported by the MDS assembler. The standard names for the 68000 instructions were given in Chapter 1. Assembler directives will be described later in this chapter.

The Operand Field

This field contains the operand or operands for the instruction or assembler directive in the instruction field. As we saw in Chapter 1, operands for instructions indicate the addressing modes to be used to form effective addresses. If there are two operands for the instruction, the source operand comes first and is separated from the destination operand with a comma (,).

The assembler formats for the standard 68000 operands were also given in Chapter 1. There are two special forms of operands involving strings of characters supported by the MDS assembler that we haven't seen yet, however. The first is the **string immediate operand**, #'WXYZ', where the immediate values used are the ASCII codes for the characters in the string. (The ASCII coding scheme is described in Chapter 5.) The number of characters used is one (byte operation), two (word operation), or four (long word) operation.

You can also specify a string as the operand of a PEA or LEA instruction:

```
PEA  'GARY'        ;Push address of string
LEA  'GARY',A1     ;Load address of string
```

In these cases, the string can be any length and the assembler will store it as a constant at the end of the code

space during the assembly process. What is actually pushed or loaded by these instructions is the address of the string, not the contents of the string. The method used to store the string is dictated by the setting of the STRING_FORMAT directive (see below); the default format is a length byte followed by the ASCII codes for the characters in the string.

You should also be aware that MDS lets you use mathematical expressions in operands, using operators such as: + (add), − (subtract or negate), * (multiply), and / (divide). The add operator is especially useful for identifying an offset into a data structure. If the base address of the structure is in A0 and you want to access the word that begins bottom bytes into the PortRect field, for example, use an operand of the form PortRect+bottom(A0).

I'll be discussing all the mathematical operators and how they're evaluated later on in this chapter.

The Comment Field

The comment field begins with a semicolon (;). The semicolon, and everything after it on the line, is ignored by the assembler, so you are free to enter any text you want, typically an explanation of what the program is doing. If the entire line is a comment, put a semicolon or an asterisk (*) in column 1.

Assembler Directives

The instruction field in an assembly language program usually holds a 68000 instruction, but it can also hold a "pseudo-instruction", or **assembler directive**, which controls the assembly process in some way, defines symbols, or allocates space for data storage.

In this section, I'll summarize all the common assembler directives you will use with MDS.

Symbol Definition Directives

The symbol definition directives assign names to certain values or expressions. (See Table 2-1.) These values can represent absolute memory locations, trap instructions, numeric constants, or arbitrary sequences of characters. By making liberal use of symbols, you can improve the readability of your programs, thus making them easier to modify and debug.

Table 2-1. Symbol Definition Directives Used by Asm.

Directive	*Meaning*
.TRAP	Assigns a symbol to a $Axxx trap instruction
EQU	Assigns a symbol to an expression (can't reassign)
SET	Assigns a symbol to an expression (can reassign)
REG	Assigns a symbol to a register list

There are four symbol definition directives. Let's look at them in the order of their popularity.

.TRAP The .TRAP (Define Trap Instruction) directive assigns a name to a particular line A emulator trap instruction. As you saw in Chapter 1, these instructions are used to access the user interface toolbox and operating system subroutines in the Macintosh ROM. Once you've assigned a name to a trap, the name can be used in the instruction field just like any standard 68000 instruction; you don't have to memorize its $Annn numeric form. The format for the .TRAP directive is:

```
.TRAP  name  $Annn
```

where `name` is the name to be given to the trap instruction.

The standard trap names used on the Macintosh are contained in a file called Traps.txt on the MDS disk (or in a packed symbol file called Traps.D). In most cases, you will incorporate these files into every program using the

INCLUDE directive (see page 56), so you probably won't need to explicitly use the .TRAP directive in your own programs. If you're short on memory space, however, (and you might be if you're still using a 128K Macintosh) you may not have enough room to include an entire trap file, so use the .TRAP directive to define the subset of trap instructions used by your program.

EQU The EQU (Symbol Equate, Permanent) directive assigns a symbolic name to a particular expression. Once the name is assigned, it cannot be reassigned later in the program. The format for the EQU directive is:

```
label  EQU  expression
```

where `expression` is usually a numeric constant representing data or an address, but it can also be a mathematical formula or even an operand such as (A3)+. Label represents a symbolic name adhering to the naming guidelines described above for entries in the label field in a line of source code.

Here are some example of how to use the EQU directive:

```
NumberOne EQU    1              ;A numeric equate
PUSH      EQU    -(SP)          ;An operand equate
BuffSize  EQU    1024*6         ;A formula equate
Time      EQU    $20C           ;An address equate
```

SET The SET (Symbol Equate, Temporary) directive is just like the EQU directive, except that the symbol can be redefined later in the program using another SET directive. The assembler format is:

```
label  SET  expression
```

where `label` and `expression` have the same meaning as for the EQU directive.

The SET directive is often used within a macro definition to assign symbolic names to variables used only by the macro. You can't use EQU because the next time the macro is

invoked, the same symbols are defined once again. I'll be looking at macros later in this chapter.

REG The REG (Register Equate) directive assigns a symbol to a register list used by the MOVEM (move multiple registers) instruction. The format for the directive is:

```
label  REG  register list
```

where `register list` refers to a group of one or more ranges of consecutive registers. The starting and ending registers in a range are separated by a dash (–) and each range group is separated from the next by a slash (/). For example, the register list for registers D0, D1, D2, D7, and A0, A1, and A2 would be D0–D2/D7/A0–A2.

Data Allocation Directives

The data allocation directives are used to allocate space for any constants, variables, and data structures that the program uses. (See Table 2-2.) The data space so allocated can be part of the code space for the program or it can be located in the application global variable space in the upper end of memory, depending on the directive used.

Table 2-2. Data Allocation Directives Used by Asm.

Directive	*Meaning*
DC	**Reserves space for a constant**
DCB	**Reserves space for a block of constants**
DS	**Reserves space for a variable**

As you will see, each of the data allocation directives has a byte, word, and long word form that you select by using a .B, .W, or .L extension. If no extension is specified, the word form is used. If you use the word or long word forms, and the next free location is an odd address, a byte of padding is first

allocated by the assembler. The label associated with a data allocation directive always refers to the address after any padding byte, however.

Remember that word alignment is absolutely necessary if you want to access long or long word data. If the alignment isn't correct, an address error exception will crash the system. Fortunately, the MDS assembler does all it can to help you avoid such situations.

DC The DC (Define Constant) directive stores data within the code space of the program. The assembler formats are:

```
[label]  DC   value      ;store word data
[label]  DC.B value      ;store byte data
[label]  DC.W value      ;store word data
[label]  DC.L value      ;store long word data
```

where `value` represents the numeric value of the data to be stored or a string of characters enclosed in single quotation marks. If a string is specified, the ASCII code for each character is stored by the assembler. The brackets around label indicate that a label (a symbolic name for the constant) is optional.

Multiple values can be stored by specifying a series of values in the operand field separated by commas. For example,

```
TwoBytes  DC.L    $45,63         ;TwoBytes is the label
```

causes the values 00 00 00 45 00 00 00 3F to be stored in the object code. The first four bytes are for $00000045 and the next four are for $0000003F (decimal 63).

DCB The DCB (Define Constant Block) directive allocates a block of data within the code space of the program and stores a specific value in each byte in that block. The assembler formats for the DCB directive are:

```
[label]  DCB   size,value       ;store block of words
[label]  DCB.B size,value       ;store block of bytes
[label]  DCB.W size,value       ;store block of words
[label]  DCB.L size,value       ;store block of long words
```

where size represents the number of data units (bytes, words, or long words) to be allocated, and value represents the number to be stored in each data unit in the block.

To reserve 32 bytes of memory, each initialized to the value $7F, use the following directive:

```
OurBlock  DCB.B   32,$7F
```

Be sure not to reverse the order of the size and value parameters.

DS You can also allocate data space in the application global variable space, which begins at the address pointed to by the A5 register and grows downward in memory. (The A5 register is properly initialized by the operating system when your program starts to run.) This is where you should reserve space for a number if you are going to change its value during program execution.

Unlike the DC directive, DS (Define Storage) does not initialize the values stored in this data space. The assembler formats are:

```
.
[label]  DS    size            ;reserve words
[label]  DS.B  size            ;reserve bytes
[label]  DS.L  size            ;reserve long words
[label]  DS.W  size            ;reserve words
```

where size represents the number of data units (bytes, words, or long words) to be allocated.

When you read from or write to a piece of data in the application global area, you must always use the A5 register indirect with offset addressing mode. That is, if you have allocated a word with a directive like:

```
MyData  DS.W    1
```

then you must use a MyData(A5) operand to access it. One of the most common sources of bugs in an assembly lan-

guage program is caused by forgetting to tack on (A5) to the name of a variable allocated with the DS directive. Be careful.

Assembly Control Directives

The assembly control directives dictate the exact manner in which the source file is to be assembled. (See Table 2-3.) In particular, they dictate what parts of the file are to be assembled, how strings are to be handled, and how macros are to be handled.

Table 2-3. Assembly Control Directives Used by Asm.

Directive	*Meaning*
INCLUDE	Reads in another source file during assembly
STRING_FORMAT	Sets the string storage format
IF..ELSE..ENDIF	Assembles code according to conditions
MACRO	Assigns an instruction sequence to a name
END	Marks the end of the source code
.DUMP	Saves symbols in a packed symbol file (.Sym)

INCLUDE If you have a standard chunk of assembly source code that is common to many programs, you may want to save it in a separate file on disk. Then, when you want to incorporate it into another source document, all you need to do is specify the file name as the argument of an INCLUDE (Include a Source File) directive in your program. This makes it easy to incorporate a standard sequence of source statements in any program you develop.

The assembler format for the INCLUDE directive is:

```
[label]  INCLUDE  filename
```

where the name of the file to be included is `filename` or `filename.asm`.

As you'll see later on, INCLUDE is most commonly used for including the standard trap and symbol definition files that come with MDS.

STRING_FORMAT The STRING_FORMAT (Set String Storage Format) directive sets the method the assembler is to use when it stores character strings in memory. The format for the directive is:

```
STRING_FORMAT  value
```

where `value` represents a number from zero to three, inclusive.

There are two types of strings in a program that STRING_FORMAT affects, those used as arguments of PEA and LEA instructions, and those allocated using the DC data allocation directive. In each case, a string is represented as a sequence of characters enclosed by single quotation marks:

```
'Test String'
```

If your string includes the single quotation mark, enter two of them in a row; the first one is ignored and the second forms part of the string:

```
'Gary''s String'
```

Bit 0 of the value assigned to STRING_FORMAT controls how strings used with PEA and LEA are handled: if zero, the string is stored as a group of ASCII characters followed by a 0 byte; if one, the string is preceded by a length byte. In either case, space for the string is allocated after the end of the program code.

Bit 1 of STRING_FORMAT controls the format of DC strings. If the bit is zero, the string is stored without a preceding length byte or a trailing 0 byte. If it is one, however, the string is preceded by a length byte. This means the following values for STRING_FORMAT are permitted:

```
STRING_FORMAT 0        ;DC (text only) , PEA/LEA (0 trailer)
STRING_FORMAT 1        ;DC (text only) , PEA/LEA (length)
STRING_FORMAT 2        ;DC (length) , PEA/LEA (0 trailer)
STRING_FORMAT 3        ;All strings preceded by length byte
```

You'll probably find the most convenient STRING_FORMAT to use is 3 because the standard Macintosh trap instructions expect strings preceded by length bytes. Unfortunately, the default value is 1, so be careful that you set STRING_FORMAT to 3 if you want to define a string preceded by a length byte with the DC directive.

IF..ELSE..ENDIF The IF..ELSE..ENDIF (Conditional Assembly) directives let you develop programs that can be assembled in different ways depending on the state of a condition you specify. For example, your program may contain extra code that should only be assembled if you are testing the program, not if you are releasing the program to the public. Rather than removing this extra code for good and then assembling the program, you can simply leave it in and assemble it only if you enable a debug condition. That way, you don't have to maintain two versions of the same program.

The format of the conditional assembly directive is as follows:

```
IF   condition
insert lines to be assembled if the condition is true
[ ELSE
insert lines to be assembled if the condition is false
       ]
ENDIF
```

where `condition` is a mathematical or logical expression that evaluates to a true (non-zero) or false (zero) result. If the full IF..ELSE..ENDIF structure is used, everything between the IF and ELSE lines is assembled if the condition is true and assembly continues after the ENDIF line; otherwise, everything between ELSE and ENDIF is assembled instead.

Notice, however, that the ELSE directive is optional (as indicated by the brackets). If it's not used, all the lines between IF and ENDIF are assembled only if the condition is true.

When specifying a condition, you can use several arithmetic, logical, and shifting operators to form a mathematical expression. Here are those operators, in decreasing order of precedence of evaluation:

negation	–
shift one bit right	>>
shift one bit left	<<
logical and	&
logical or	!
multiplication	*
division	/
addition	+
subtraction	–

To alter the standard order of evaluation, use parentheses to enclose the expressions to be evaluated first.

Logical conditions can also be formed using comparison operators: > (greater than), < (less than), >= (greater than or equal), <= (less than or equal), = (equal), and <> (not equal). Note, however, that you can only compare strings for equality or non-equality.

Here are some valid expressions for condition:

```
DEBUG = 1        ; true if DEBUG is 1
MYFLAG           ; true if MYFLAG is non-zero
(MYFLAG+3)/8     ; true if the expression is non-zero
MYFLAG <> 45     ; true if MYFLAG is not 45
```

MACRO A macro is a shorthand representation for a commonly used group of instructions. When the assembler encounters a reference to a macro, it automatically expands the reference by placing the group of instructions the macro defines into the object code.

The format for a MACRO (Macro Definition) directive is:

```
MACRO  name  [arguments] =
[body of macro]
|
```

where `name` is the symbolic name for a macro (the syntax rules for a name are the same as for labels) and `arguments` represents an optional list of variables used within the body of the macro. Each argument is separated from the next by a comma.

Each line in the body of the macro can be anything you want, as long as it follows the syntax rules dictated by the assembler. That is, just as with any normal line, it can include labels, instructions, operands, and comments. (If you use labels, they should be local ones so that global label names won't be duplicated if you use the macro more than once. Symbols should be defined with the SET directive rather than the EQU directive.) The only difference is that a line can also contain references to the arguments of the macro; this is done by enclosing the argument name in braces ({ }). When the macro is invoked, these arguments are replaced by the actual parameters specified when the macro is invoked.

To invoke a macro, use its name as if it were an instruction; the operands for the instruction are actually the parameters for the macro. For example, suppose there are several places in your program where you're executing a portion of code that takes the form:

```
CLR.L   -(SP)
MOVE    x(A5),-(SP)
MOVE    y(A5),-(SP)
PEA     Address
```

Your program might become more understandable, and easier to develop, if you define a macro called PushPos, as follows:

```
MACRO  PushPos M1,M2,M3 =
CLR.L  -(SP)
MOVE   {M1}(A5),-(SP)
MOVE   {M2}(A5),-(SP)
PEA    {M3}
|
```

If you do this, you can invoke the macro by including a line like this in the source file:

```
PushPos hpos,vpos,MyRecord
```

This will generate the following code when the file is assembled:

```
CLR.L  -(SP)
MOVE    hpos(A5),-(SP)
MOVE    vpos(A5),-(SP)
PEA     MyRecord
```

Macros are often used to define push and pop stack operations. Here are two macros, POP and PUSH, you can use for this:

```
MACRO   POP  Dest =
        MOVE.W  (SP)+,{Dest}    ;Pop into Dest
        |
MACRO   PUSH Item =
        MOVE.W  {Item},-(SP)    ;Push from Item
        |
```

The POP and PUSH macros are word-sized. You should also define macros that can handle long word operations (and call them POP.L and PUSH.L).

END The END (End of Source Statements) directive tells the assembler to terminate the assembly process. Anything that appears in the source file after the END directive is ignored by the assembler.

.DUMP The format of the .DUMP (Dump Symbols to File) directive is:

```
.DUMP   filename
```

This command tells the assembler to store its list of symbols (defined using EQU or SET) in a file having the name filename.Sym. Such a file can be converted to a **packed symbol file** using the PackSyms program on the MDS disk.

A packed symbol file has an extension of .D and can be incorporated in a program with the INCLUDE directive, just as if it was a standard text file. The advantage of using a packed symbol file is that it takes up less space on the disk than a standard symbol file and it can be assembled faster than a standard text file.

Linker Control Directives

Linker control directives provide information the linker requires in order to properly combine more than one assembled code file into a single, executable application. (See Table 2-4.) Such information includes lists of program symbols that may be accessed from other code modules, and symbols that are defined externally in other modules.

Table 2-4. Linker Control Directives Used by Asm.

Directive	*Meaning*
XDEF	**Identifies symbols that can be accessed by code in another .Rel file**
XREF	**Identifies symbols that are defined in another .Rel file**

XDEF The XDEF (External Definition) directive identifies any label or symbol in one code module that might be referred to in another code module it is linked with. Without this information, the linker cannot resolve references between modules and will generate an error.

The assembler form for the XDEF command is:

```
XDEF    symbol_list
```

where `symbol_list` is a list of all symbols in the code module that can be referred to in other code modules. Each symbol in the list is separated from the next by a comma.

XREF The XREF (External Reference) directive tells the linker that a particular group of symbols used in the program is defined in another code module. The assembler form for XREF is:

```
XREF    symbol_list
```

As with XDEF, each symbol in `symbol_list` is separated from the next with a comma. Each symbol must be named in an XDEF directive in another code module.

Printing Control Directives

The printing control directives control the way in which output generated by the assembler is handled. This output consists of a listing of the program that includes the source code statements and the object code they generate.

Table 2-5. Printer Control Directives Used by Asm.

Directive	*Meaning*
.NoList	Turns off the listing of the Asm output
.ListToFile	Lists the Asm output to a file
.ListToDisp	Lists the Asm output on the screen

.NoList The .NoList (Don't List Assembly Output) directives turns off the assembler listing entirely. This is the default condition.

.ListToFile The .ListToFile (List Assembly Output in File) directive tells the assembler to store its output in the file

whose name is stored in the operand field following it. For example, the command:

```
.ListToFile Assembly.Txt
```

causes the listing to be stored in a file called Assembly.Txt.

.ListToDisp The .ListToDisp (List Assembly Output on Screen) directive tells the assembler to display its output on the screen while the file is assembled.

You can also indicate what's to happen to an assembler listing by pulling down Edit's Options menu before you begin to assemble and selecting either No Listing, List to File, or List to Display.

The Linker

The linker is used to combine .Rel files (**relocatable object code files**), usually created by the assembler or resource compiler, into an application that can be launched from the Finder with a double-click. Less frequently, it is used to create other types of files, not necessarily directly executable applications.

The activities of the linker are controlled by a linker control file that has a file name extension of .Link. This extension is mandatory. The control file contains commands that tell the linker such things as what .Rel files are to be combined, what the name of the output file is to be, what the file type and creator codes for the output file are to be, and what the starting location in an application is to be.

The simplest linker control file contains just two lines and is all you need for linking most simple applications:

```
MyFile.Rel
$
```

These lines tell the linker to deal with one relocatable file only, MyFile.Rel, and to create an output file called MyFile

that will contain the final application. MyFile.Rel is typically created by using Asm to assemble your 68000 source code file. It could, however, be created by compiling a program written in another language, such as Pascal or C, that supports the MDS .Rel file format. The $ sign following the name of the .Rel file signifies the end of the linker control file and is required. If you are using MDS 2.0, you can also use the /END command.

If you're linking several separate .Rel files together, you'll include all their names before the final $ sign. The name of the application file created will be the name of the first .Rel file specified, without the .Rel extension. If you don't place a disk prefix in front of the name of a .Rel file, Link expects to find it on the same disk volume that holds the linker control file.

Comment lines within a linker control file begin with a semicolon. Unlike Asm (or RMaker), you cannot use an asterisk instead. Another important difference is that comments cannot be tacked onto the ends of lines containing linker control statements.

There are only a few linker commands you need to master. A discussion of these commands follows.

Linker Code Modules

The names of all .Rel files containing the application's program code must appear at the beginning of the control file. Each name must appear on a separate line and you can omit the .Rel extension if you wish. For example, if you want to link filename1.rel to filename2.rel, place the following two lines at the beginning of the linker control file:

```
filename1.Rel
filename2
```

The name given to the application created by this linker control file will be filename1 unless you choose another using the /OUTPUT command. It is the same as the name of the first .Rel file, but without the .Rel extension.

The linker combines separate code modules into a single resource of type CODE, called a **segment**. Its resource identification code is 1. (Resources are described later in this chapter.) The maximum size of a code segment is 32K.

If you want to create another code segment, enter a line in the linker control file that contains just the < symbol, and follow it with lines containing the .Rel files to be linked into the new code segment.

The main reason for breaking a program into several code segments is to save memory space. Only when code in one segment calls code in another segment is that other segment loaded into memory. At the same time, the space occupied by the calling segment is normally freed up using the _UnLoadSeg trap instruction. Unless you're short of memory space, or your program is larger than 32K, you won't have to bother with creating multiple code segments.

File Type and Creator Code

The Macintosh operating system uses two four-character sequences called the **file type code** and the **creator code** (or **signature**) to identify a file.

The file type code identifies the type of information stored in a file so that an application can interpret it properly. Some common file type codes are PNTG (a MacPaint document), TEXT (a file containing lines of text), and APPL (a file containing code for an application).

The creator code serves to mate a file to the application that created it in the first place. When a non-application file is opened, the Finder reads its creator code, launches the application with the same creator code (its file type code is APPL), and passes the name of the non-application file to it so it can be opened. MacPaint, for example, stores a creator code of MPNT in its application file and all data files it creates. When you double-click a MacPaint document, the MacPaint application is launched and the document is opened.

File type codes and creator codes for commercial products must be approved by the Macintosh Technical Support Division of Apple Computer, Inc. to ensure uniqueness. That doesn't mean you can't use file type codes that are already taken. It just means the internal structure of any file you create with that file type code should be the same as for standard files of that type. For example, you can use a file type code of TEXT as long as your file contains lines of text in ASCII-encoded form.

For a file that defines an application, the file type code is always APPL and this is the default used for Link's output files. The creator code can be anything you like (as long as no other application uses it). If you don't specify one, the linker chooses a null creator code (four zero bytes).

Use the /TYPE command to override these defaults. For example, if you are linking files to create an application that has a creator code of DEMO, place the command:

```
/TYPE 'APPL' 'DEMO'
```

in the linker control file. Note that the first code after /TYPE is the file type code and is followed by the creator code. They must both be exactly four characters long and enclosed in single quotation marks; shorter strings must include padding blanks.

Output File

The name of the file created by the linker is usually the same as the name of the first .Rel file linked, but without the .Rel extension. If you want to use a different name, use the /OUTPUT command. For example, to select a file name of OurDemo, place the command:

```
/OUTPUT OurDemo
```

in the linker control file before the $ terminator.

Bundle Bit

The bundle bit must be set for those applications containing ICN#, FREF, and BNDL resources that define custom desktop icons to be used by the Finder. The bundle bit is part of an attribute byte in the file's directory entry on disk.

To set the bundle bit, use the command:

```
/BUNDLE
```

You'll see how to create custom icons for applications and their documents at the end of this chapter.

Starting Location

The starting location of a program is usually the first instruction in the first .Rel file specified. You can, however, override this default using a linker command of the form:

```
!FirstLoc
```

where `FirstLoc` is the label of the instruction in the program that is to be called when the program is launched. There must be no spaces between the exclamation mark and the name of the label. With MDS 2.0 you can also use the /START command to achieve the same result.

The only labels you can use as starting locations are those that are defined as external in the source program. Use the XDEF assembler directive to do this.

Linker Resource Modules

You can also link .Rel files containing nothing but resources created with the RMaker resource compiler. To do this, insert the line:

```
/RESOURCES
```

in the linker control file after the names of all the code files

being linked, then list the names of all the RMaker .Rel files, one per line.

End of File

The $ sign (or, for MDS 2.0 only, the /END command) is used to signify the end of a linker control file. Place it on a line by itself at the end of the linker control file. It is required.

The Resource Compiler

Each file on a Macintosh disk is actually made up of two logical parts called the **data fork** and the **resource fork**. The data fork contains anything you care to store in it, such as parameters for an application, the text for a word processor or editor, and so on. The data is placed there using a group of operating system instructions making up the Macintosh File Manager. Refer to *Inside Macintosh* for a description of those instructions.

The resource fork contains one or more **resources** used by an application. A resource is a chunk of data or program code that defines such data structures as a character string, the bit image for an icon, a character font, a cursor, the code for a desk accessory, and the templates the toolbox uses to build windows, menus, and dialog boxes. Even the code for your application program is a resource.

The main advantage of using resources for the storage of such items is that it makes your program more modular and, hence, easier to debug. It's also very easy for someone else to change the visual interface of your program without having to rewrite the program code itself.

The Macintosh supports several general classes of resources, each identified by a unique four-character name. Some of the more common ones are summarized in Table 2-6. Note that the operating system distinguishes between uppercase and lowercase characters; that means a MENU resource, for example, is not the same as a Menu resource.

Table 2-6. Common Resource Types Used by the Macintosh.

	Resource Type	*Meaning*
*	ALRT	Alert box template
*	BNDL	Application bundle
	CDEF	Control definition
*	CNTL	Control template
	CODE	Assembly language code segment
	CURS	Cursor
*	DITL	List of items in a dialog or alert box
*	DLOG	Dialog box template
	DRVR	Device driver or desk accessory
	DSAT	System startup alert table
	EFNT	Font selection for MDS Edit
	ETAB	Tab settings for MDS Edit
	FCMT	"Get Info" comments
	FKEY	Function key routine (command-shift-number)
	FOBJ	Folder information
	FONT	Character font
*	FREF	File reference
	FRSV	Reserved font
	FWID	Font widths
	ICN#	Icon list
	ICON	Icon
	INIT	Initialization routine
	INTL	International utilities
	KEYC	Keyboard configuration
	MBAR	Menu bar
	MDEF	Menu definition routine
*	MENU	Menu
	PACK	Package of routines
	PAT	Pattern
	PAT#	Pattern list
	PDEF	Printing routines
	PICT	Picture
	PREC	Printing record
	SERD	Serial drivers
*	STR	String
*	STR#	String list
	WDEF	Window definition routine
*	WIND	Window template

* These resource types are explicitly supported by RMaker.

Each resource is associated with a resource identification code, a number from −32768 to +32767. All codes from −32768 to +127 are reserved for use by the operating system, so you shouldn't use them for application resources unless they are intended to replace system resources. The ID code you assign to a resource must be unique within the group of resource files of a certain type open at any given time. If there is duplication, only the first resource located with that number will be available.

As soon as an application starts running, two resource files are automatically opened: the operating system's (located in the resource fork of a file called System on your startup disk) and the application's. When the application requests a resource (this is done by specifying the resource type and identification code) its own resource file is searched first and, if the resource isn't found, the operating system resource file is searched.

The application can also explicitly open other resource files, such as those associated with an open data file. If it does this, the search for a given resource begins with the last file opened and continues down through to the System resource file until the resource is found.

As shown in Figure 2-2, each resource has an attribute byte that reflects some of its properties. The property associated with a particular bit in the attribute byte is asserted if the bit is set to one. The attribute byte is usually set to zero when you create the resource, but you can adjust it to suit your requirements.

The only two attributes you're likely to deal with are ResPreload (bit 2) and ResPurgeable (bit 5). If the ResPreload bit is set to 1, the resource is loaded into memory as soon as the resource file in which it is stored is opened. This means if your resources are part of the application file itself (the usual case), they will all be loaded into memory as soon as you run (or launch) your application from the Finder. If you don't set the ResPreload bit, the disk will be accessed each time you use a resource for the first time, and your application may appear to run more slowly.

If the ResPurgeable bit is set to 1, the resource may be

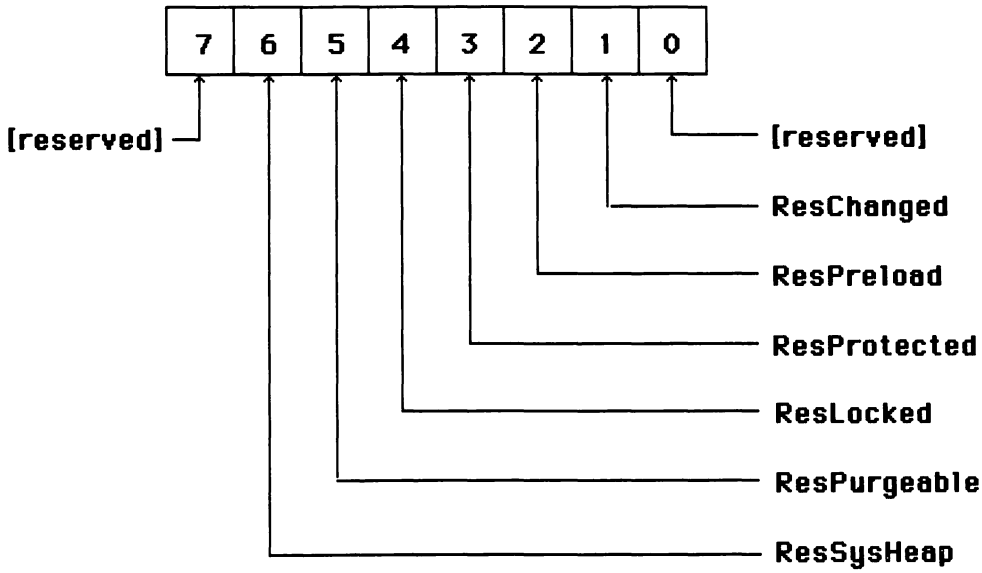


Figure 2-2. The Format of the Attribute Byte for a Resource.

removed, that is, **purged**, from memory if the operating system runs out of space. If the resource is purged, it can't be used until it's loaded into memory again. The Macintosh trap instructions that use resources automatically detect when a resource has been purged and will reload the resource as required. If your application program needs plenty of memory, mark your resources as purgeable so that space can be freed up when required. The only penalty is that you will encounter more disk activity when purged resources are later reloaded.

Using the RMaker Resource Compiler

In this section you'll learn how to use the MDS resource compiler, **RMaker**. This program is primarily used to append the resources defined in an RMaker source code document to an application file. It can also be used to create a separate resource file for use by other applications, or to create a .Rel file suitable for linking by Link using the /RESOURCES command.

Name of Output File

The first non-comment and non-blank line in a RMaker source file (we'll call this file the **input file**) must contain the name of the file in which the resources are to be stored. The line that follows must either be blank or contain a sequence of eight characters defining the file type code (first four characters) and the creator code (second four characters). (Comment lines begin with an asterisk.)

There are three general forms of names you can specify, each causing a slightly different result:

!filename. If you place an exclamation mark in front of a file name, the resources are appended to the file with that name. Use this form of name to add your resources to the application file created by the linker so your program code and all its resources will be in the same file. This is the form you'll probably use most often.

filename or filename.xxx. If you specify a file name with or without an extension (other than .Rel), RMaker stores the resources in a standard resource file. By convention, you should use a name extension of .Rsrc to identify such a file.

filename.Rel. If you specify a .Rel file name extension, RMaker saves the resources in the file in the same format the assembler uses to save relocatable object code. This means the file can can be linked with Linker by specifying its name after the line containing the linker's /RESOURCES command.

No other information may appear on the line containing the file name. In particular, comment fields like the ones used with TYPE commands are not permitted. (See page 74.)

Including Other Resource Files

You can direct the resource compiler to combine the resources in other files with the current file by using the INCLUDE statement. The format is:

```
INCLUDE filename
```

where `filename` is the name of the file to use. The included file name can be any file containing resources, including an application file created by Link. The code for an application is stored in a series of two or more CODE resources.

TYPE Statements

Most of an RMaker input file is made up of several TYPE statements that define the data in the resources. The format of a TYPE statement is as follows:

```
TYPE XXXX              ;;XXXX = resource type code
[name],ID [(aa)]       ;;resource name, ID, attribute
data for resource      ;;the resource data goes here
```

The brackets enclose optional parameters (don't include them in your file!), which means you don't have to assign a name to a resource and you don't have to specify the value of the resource's attribute byte (but you do have to specify a resource ID). The default value for the attribute byte is zero.

In general, each TYPE statement must be entered on one line. You can indicate a continuation to the next line, however, by typing in + + at the end of the line. Comments follow two successive semicolons (;;) and are ignored by RMaker. For example, to define a STR# (string list) resource, use a TYPE statement like the following:

```
TYPE STR#              ;;type code is 'STR#'
MyStrings,128 (32)     ;;Name, ID, attribute
2                      ;;Data: number of strings
The first line ++      ;;1st resource string
is a long one          ;;(continuation of 1st string)
The last line is short ;;2nd resource string
```

Notice that all numbers used in RMaker source statements must be decimal numbers, with a few exceptions that I'll point out as we encounter them. Strings are entered without quotation marks and are converted to ASCII codes by RMaker. To enter the ASCII code for a character that can't

be entered directly from the keyboard, use the command \xx where xx represents the two hexadecimal digits of the ASCII code. For example, use \14 to enter the "Apple" symbol (ASCII $14).

You can define multiple resources of the same type with a single TYPE command. Do this by following the TYPE statements for a resource definition with a blank line and beginning the next resource definition with the name, ID, attribute line.

Of the many resources used on the Macintosh, only 12 are directly supported by RMaker (they are marked with asterisks in Table 2-6). You'll see how to use most of these later in this book.

It is possible, however, to create any resource by equating it to the special PROC (procedure) or GNRL (general) resource types supported by RMaker. This is useful if the resource is simply an assembly language program or if you know the internal structure of the resource you're trying to create. The structure of standard resources not directly supported by RMaker is available in *Inside Macintosh*.

Let's look at how to use the special PROC and GNRL resource types.

PROC A PROC (Procedure) resource contains an executable assembly language program. Its RMaker format is as follows:

```
Type PROC
,128 (32)           ;;ID followed by attribute
AProgram            ;;Name of program containing code
```

Notice that the attribute value in this example is 32 (that is, bit 5 of the attribute byte is 1). This means the resource is purgeable.

The PROC resource is made up of the entire contents of the first code segment in the specified program, except the first four bytes. (These bytes are used by the operating system only and are not part of the code.) This segment has a CODE resource ID of 1, is created by the linker, and contains

the application's 68000 assembly language instructions. Other code segments are ignored by RMaker.

Other Macintosh resources that contain executable code can be created with RMaker by equating their names to PROC as follows:

```
Type PACK = PROC      ;; "PACK is like a PROC resource"
,128
MyPackage
```

This resource definition creates a PACK resource (a resource containing a group of related assembly language subroutines) and stores the code in MyPackage in it. Another type of resource that contains executable code is DRVR (a device driver or a desk accessory).

GNRL The GNRL (General Resource) resource type provides another way to define resources that RMaker doesn't explicitly support. These could be standard system resources like ICON, FONT, or MBAR, or your own custom resources.

The first part of a TYPE statement using GNRL looks like this:

```
Type MINE = GNRL
,128                  ;;Resource ID
```

This means we're about to define a new resource type called MINE. After the line containing the resource ID, you can use six **element type designators** to indicate the format of subsequent data lines in the definition. They are:

.H The following numbers are hexadecimal
.I The following numbers are decimal words (integers)
.L The following numbers are decimal long words
.P The following string is preceded by a length byte
.R Read the following resource from a file (the format of the following line is: filename type ID)
.S The following string has no length byte

An element type designator must appear on a line by itself, although comments can be included.

Let's use these designators to continue our definition of the MINE resource:

```
.H                      ;;hexadecimals follow
0FFF 7FFF
.I                      ;;decimal words follow
123 343
.L                      ;;decimal long words follow
70000 83423
.P                      ;;length+string
Your name here
.R                      ;;read in ICON resource #244
Custom.Rsrc ICON 244    ;;from a file called Custom.Rsrc
.S                      ;;string without length
No length string
```

Of course, since this is a custom resource, the meaning of the contents are completely up to you. If you assign a reserved resource type to GNRL, you will have to make sure you store data in the form and order described in *Inside Macintosh*.

In later chapters you'll see how to use GNRL to create MBAR (menu bar), ICON, and ICN# resources.

The Executive Program

You use the Exec program to combine the many chores associated with the assembly process into one simple step. To do this, first use Edit to create an executive control file whose file name has a .Job extension. Each line in this file is made up of the following parts in this order:

- the name of an application to run
- the name of the file to be opened by the application when it starts up
- the name of the application to run if the first application ends normally
- the name of the application to run if the first application ends with an error

Each part of the line is separated from the next by a single tab character. For example, when the executive encounters the following line:

```
Asm    OurCode.Asm    Exec    Edit
```

it loads the assembler and begins assembling the file called OurCode.Asm. If the assembly ends with no error, control returns to Exec so the next line in the .Job control file will be dealt with; otherwise, Edit is called up to enable you to fix your error. In a typical scenario, the next lines in the .Job file would be of the form:

```
Link    OurCode.Link   Exec    Edit
RMaker  OurCode.R      Exec    Edit
```

This causes the final application to be linked after assembly and the resources to be created.

Search Paths

A Macintosh with 128K ROM supports a hierarchical disk directory structure called **HFS**. In an HFS system, several directories may be set up on one disk, each containing many files, and directories may be created within other directories. To uniquely identify a file, you must specify its name and the sequence of directories to pass through to reach its directory. The identifying string, called a **pathname**, is made up of the directory names, each separated from the next by a colon, followed by the file name.

For example, suppose you have a disk called MyDisk. MyDisk is also the name of the first directory on the disk, called the root directory. If you have a directory called Work within the root directory and a file called Demo.Asm within Work, the pathname describing the file is MyDisk:Work:Demo.Asm.

A problem arises when using MDS 2.0 with an HFS system: how to tell the assembler tools where to find input files and

where to store output files. One solution is to specify a complete pathname for each file, but this is awkward. The recommended solution is to specify a simple file name and let the assembler tool read the file from, or store it to, a default directory.

Each class of file an MDS 2.0 assembler tool might use is associated with a default **search path**, as described in the MDS users manual. The search path is a list of the directories a tool will search to locate an input file (or a directory for an output file), in search order. Each directory is referred to in relative terms, usually using the source file directory, the root directory, or the launch directory (the directory in which the assembler tool is stored) as a reference point.

For example, the first three directories in the search path for Asm INCLUDE files are *L:MDS Includes:, *S::MDS Includes, and *S:MDS Includes. *L symbolizes the launch directory and *S the source directory. Two colons in a row, as in the second directory, means "back up one directory". When you use the INCLUDE command in an assembler source file, MDS first looks for the file in a directory called MDS Includes within the launch directory. If it's not there, the search continues with a similarly named directory within the directory in which the source directory is defined. The third choice is the MDS Includes directory within the source directory. The search continues until the file is found or the search path is exhausted.

The default search paths can be changed using an application called the **Path Manager**. You should resist tampering with the standard defaults, however.

MDS 1.0 does not support search paths because it works with 'flat' disks (called **MFS disks**) only, disks which have only one directory. To tell MDS to use a certain disk when reading or writing a file, precede the file name with a "DiskName:" disk name prefix. If you don't specify a disk name, all the assembler tools, with one exception, will use the disk on which the source file is stored when it looks for input files or creates output files.

The exception is RMaker. It expects **input files** (INCLUDE

files and files to which it is appending) to be on the same disk as RMaker itself, and it stores output files on the RMaker disk. This can cause problems if your assembler tools are on a disk in one drive and your source files are in a second drive because RMaker will not find its input files unless you use disk names to override the default. To avoid having to use disk names, put a copy of RMaker on your data disk and run it from that disk.

Equate, Trap, and Macro Files

The MDS disk contains several definition files containing standard symbolic names for various items often used in assembly language programs: addresses of system global variables, offsets into data structures, bit flags, data masks, and other numeric quantities. There are also files defining the standard toolbox and operating system trap instructions, and useful sets of macros.

You should always use these symbolic names in your own programs instead of absolute numbers or addresses because the programs will be easier to understand and debug. And if you need to use a number describing a system parameter, don't assume it will be a specific value; always read its value from the system variable given in the equate file. For example, don't assume the base address of the screen memory is $7A700—read it from the ScrnBase variable instead, so the same program will work properly on a 128K Macintosh or a Macintosh Plus.

The names, and general contents, of the major symbol definition files that come with MDS, are as follows:

```
ATalkEqu.txt      ;AppleTalk equates
SysEqu.txt        ;Operating system equates
ToolEqu.txt       ;Toolbox equates
QuickEqu.txt      ;Quickdraw equates
FixMath.txt       ;Fixed point math traps/macros
FSEqu.txt         ;Filing system equates
PackMacs.txt      ;Macros for standard packages
```

```
PrEqu.txt          ;Printing system equates
SysErr.txt         ;System error numbers
TimeEqu.txt        ;Time Manager equates
Traps.txt          ;Toolbox/operating system/QuickDraw traps
SANEMacs.txt       ;Standard Apple Numeric Environment macros
MacDefs.txt        ;Macros for translating Lisa macros
```

> Note: The file Traps.txt was broken into three files, called SysTraps.txt, ToolTraps.txt, and QuickTraps.txt, in version 1.0 of MDS.

You should make a point of printing out the contents of each definition file as soon as you buy your MDS assembler because you're going to use the symbols they contain again and again. To assemble the pre-defined symbol definition files with your application source code, use the INCLUDE directive in your source code.

The MDS also contains several packed symbol files, identifiable by their .D file name extensions:

```
ATalkEqu.D         ;AppleTalk equates
FSEqu.D            ;File system equates
QuickEqu.D         ;Common Quickdraw equates
QuickEquX.D        ;All Quickdraw equates
SysEqu.D           ;Common operating system equates
SysEquX.D          ;All operating system equates
SysErr.D           ;Common error number equates
SysErrX.D          ;All error number equates
TimeEqu.D          ;Time Manager equates
ToolEqu.D          ;Common toolbox equates
ToolEquX.D         ;All toolbox equates
Traps.D            ;All trap instructions
```

> Note: The file Traps.D was called **MacTraps.D** in version 1.0 of MDS.

These files contain the same information as their unpacked (.txt) counterparts, but are more convenient to use because they take up less disk space and assemble faster. They can-

not, however, be viewed using the editor. Remember that you can create your own packed symbol files using the .DUMP directive and the PackSyms program.

The most useful definition files are the ones containing the trap instructions (Traps.txt) and the standard equate files (SysEqu.txt, ToolEqu.txt, and QuickEqu.txt). You should make it a point to always include the packed form of these files (Traps.D, SysEqu.D, ToolEqu.D, and QuickEqu.D) at the beginning of every program you write so they're always there when you need them.

If you do include standard definition files in your programs, don't redefine the symbols they contain, or use the symbols for instruction labels. (It's easy to do this accidentally if you're not familiar with the contents of the included files.) If you do, you will see either a "Multiply defined symbol" or "Illegal line" error when the program is assembled.

The Pascal Connection

In the early days of the Macintosh, most applications were written in the Pascal language; alternative development tools simply were not available at the time. It should not come as a surprise, therefore, to learn that most of the hundreds of subroutines in the Macintosh ROM are designed to receive parameters and return results in accordance with Pascal specifications. In fact, Apple's standard Macintosh software reference manual, *Inside Macintosh*, documents calls to these subroutines in terms of two Pascal constructs, **procedures** and **functions**, and the parameters are defined in terms of Pascal data types.

Much of the preliminary work in developing an assembly language program on the Macintosh is determining how to emulate the effect of a Pascal procedure or function call when calling a ROM subroutine with a $Axxx trap instruction. In this section you'll see how to approach this problem.

First of all, let's look at the general form of a Pascal procedure and function. A Pascal **procedure** is simply a call to a subroutine that performs some action but which does not return a separate result. (If parameters are passed by address, rather than by value, a result can be returned through a specific parameter.) The general form for a procedure call is:

```
PROCEDURE name (parm1 : type1 ;
                parm2 : type2 ;
                parmN : typeN );
```

where `name` is a character string identifying the procedure, `parmX` (X = 1, 2, ..., N) are the names of the parameters for the procedure, and `typeX` are the data types for the parameters. The standard data types are integers, characters, real numbers, and so on. Other, more complex data types can be created from these standard data types, as you'll see below.

A Pascal **function** is a call to a subroutine that returns a result. Its form is similar to that for a procedure:

```
FUNCTION name (parm1 : type1 ;
               parm2 : type2 ;
               parmN : typeN ) : type;
```

The `type` on the far right represents the data type of the **result** generated by the function.

The function and procedure parameters are usually specific values. If a VAR identifier is placed in front of any parameter name in the Pascal description, however, the **address** of the location containing the parameter must be passed to the subroutine, not its value. Addresses are also passed if the parameter data structure is longer than four bytes or if a result is to be returned through the parameter.

The fundamental data types supported by Pascal are shown in Table 2-7.

Table 2-7. The Fundamental Data Types Supported by Pascal.

Data Type	*Stack Size (bytes)*	*Description*
INTEGER	2	two's complement integer
LONGINT	4	two's complement long integer
BOOLEAN	2	Boolean (true/false) value (bit 0 of the high-order byte contains the value; 1 = true, 0 = false)
CHAR	2	ASCII-encoded character (in the low-order byte)
STRING[n]	4	the address of a sequence of bytes preceded by a length byte
SignedByte	2	one-byte signed number in the low-order byte
Byte	2	one-byte unsigned number in the low-order byte
Ptr	4	a pointer to (the address of) a data structure
Handle	4	a handle to (the address of the pointer to) a data structure
Record	2 or 4	if the data structure (the record) is 4 bytes or less, the value itself; if longer, a pointer to the data structure
VAR parameter	4	the address of the parameter

Pascal procedure and function calls to the subroutines in the Macintosh ROM are handled in one of two ways: they are either **stack-based** or **register-based**. The exact method used is important in determining the equivalent assembly language instructions.

Stack-Based Subroutines

Of two classes of subroutines in the Macintosh ROM, user interface toolbox subroutines and operating system subroutines, it is the toolbox subroutines that are usually stack-based. This means the ROM subroutine expects to find its parameters on the stack when it takes control. (You must push them on the stack in the order they occur in the rou-

tine's Pascal definition.) It also means that any function results are returned on the stack. The space each standard Pascal data type occupies on the stack is shown in Table 2-7.

Let's look at a Pascal procedure call to a stack-based ROM subroutine so you can see how to convert it into the equivalent assembly language code:

```
PROCEDURE DragWindow (theWindow : WindowPtr;
                      startPt : Point;
                      boundsRect : Rect);
```

The first thing to determine is the name of the assembly language trap instruction corresponding to this procedure. In almost all cases, this name is the same as the Pascal procedure name, except it is preceded by an underscore character (the names are defined in the Traps.D file). Thus, the trap instruction for the above example is _DragWindow. Unfortunately, this simple rule is not always followed; a few Pascal and trap instruction names are slightly different. These differences are noted in *Inside Macintosh*.

The DragWindow procedure has three parameters: one of type WindowPtr, one of type Point, and one of type Rectangle. WindowPtr is really of type Ptr (by convention, so is any other parameter whose name ends in Ptr), so it simply represents the address of a data structure. The other two data types are not fundamental Pascal data types referred to in the previous section. They are, however, defined in terms of these data types; it's just a question of knowing the definition. Table 2-8 shows the definitions of some of the common custom data types used with the Macintosh trap instructions.

Table 2-8. Examples of Some Custom Data Types Used by the Macintosh Trap Instructions.

Point:	**Integer**	**(vertical position)**
	Integer	**(horizontal position)**
Rectangle:	**Integer**	**(top position)**
	Integer	**(left position)**

Table 2-8. *continued*

	Integer	(bottom position)
	Integer	(right position)
BitMap:	Ptr	(pointer to a bit image)
	Integer	(width of bit image in bytes)
	Rectangle	(boundary rectangle)
Pattern:	Integer	(rows 1,2 of pattern)
	Integer	(rows 3,4 of pattern)
	Integer	(rows 5,6 of pattern)
	Integer	(rows 7,8 of pattern)
PenState:	Point	(pen location)
	Point	(pen size)
	Integer	(pen mode)
	Pattern	(pen pattern)

The three pushes needed for DragWindow are:

```
MOVE.L  theWindow,-(SP)     ;pointer (long word)
MOVE.L  startPt,-(SP)       ;point (long word)
PEA     boundsRect          ;address of rectangle coordinates
```

(In this example, theWindow, startPt, and boundsRect are constants defined using the DC assembler directive.)

The first parameter is a **pointer**, so we have to push a long word on the stack. The second parameter is a defined data type called point; it is made up of two words (see Table 2-8), the first for the vertical coordinate and the second for the horizontal coordinate, so we push another long word containing the values. The third parameter is of type rectangle, a data structure longer than four bytes. **Such data structures are always passed by address rather than value**; this means we have to push an address to this structure rather than the coordinates of two opposite corners of the rectangle itself. This is done using the PEA (push effective address) instruction.

Finally, to make the call, use the trap instruction:

```
_DragWindow
```

This trap instruction is defined with the .TRAP directive in one of the system trap files. Its definition can be incorporated in your program using an INCLUDE directive at the start of your source file.

Pascal functions are handled quite similarly to procedures. The key difference is that Pascal functions make room for a result on the stack by decrementing the stack pointer before calling a ROM subroutine, therefore your assembly language program must do the same. This is done before its parameters are pushed on the stack. The stack size of the result is always one word or two, depending on the ROM subroutine, so you will usually use CLR.L -(SP) or CLR -(SP) to do this.

After calling a ROM subroutine that returns a result, you must remember to remove the result from the stack. If you don't do this before executing a return from subroutine (RTS) instruction, you will almost certainly crash the system. (RTS expects the last word on the stack to be the address of the code following the JSR or BSR instruction that called the subroutine.)

For example, the assembly language equivalent of the function:

```
FUNCTION GetNewWindow (windowID : INTEGER ;
                       wStorage : Ptr ;
                       behind : WindowPtr) : WindowPtr;
```

is:

```
CLR.L   -(SP)           ;Push long word for ptr result
MOVE    windowID,-(SP)  ;Push integer
MOVE.L  wStorage,-(SP)  ;Push pointer
MOVE.L  behind,-(SP)    ;Push pointer
_GetNewWindow           ;Call ROM
MOVE.L  (SP)+,A0        ;Pop the result into A0
```

In this example, `windowID`, `wStorage`, and `behind` are constants that were defined using the DC assembler directive.

The size of the result of a Pascal function call is indicated by the data type referred to at the end of the FUNCTION decla-

ration. In the case of GetNewWindow, we are dealing with a pointer that has a size of four bytes. Thus, the first thing to do is clear space for it on the stack with a CLR.L -(SP) instruction. You could have also done this using a SUBQ.L #4,SP or a MOVE.L #0,-(SP) instruction. The parameters are then stacked just as they are for a procedure call, before executing the trap instruction. Finally, pop the result off the stack into the A0 register.

> Space for a word result can be allocated with a CLR -(SP) instruction. Some programmers allocate space for a Boolean (true/false) result with a CLR.B -(SP) instruction, but this actually causes the 68000 to reduce SP by two bytes to ensure that SP contains an even address. Why use the CLR.B notation? It emphasizes the fact that a true Pascal Boolean parameter uses only the least-significant bit in the high-order byte of a word.

For both procedures and functions it is critically important to ensure that parameters of the proper size are pushed on the stack and results of the proper size are popped from the stack. If you use incorrect sizes, the stack will soon become damaged and the system will crash.

Stack-based calls preserve all registers except A0, A1, and A7 (the stack pointer), and D0, D1, and D2. This means you can safely store intermediate results in A2-A5 or D3-D7 before calling a trap instruction.

Register-Based Subroutines

Most of the calls to the **operating system** subroutines in the Macintosh ROM are not made by passing parameters and results on the stack. Rather, they are passed using certain 68000 registers. These subroutines typically handle low-level system chores such as memory management. (See Chapter 4 for a discussion.)

In cases where only one or two parameters are involved, the A0 and D0 registers are used to pass the parameters. A0 is used for addresses and D0 for data. If you're dealing with more than two parameters, the address of a parameter block is passed in A0 instead. In either case, all other registers (except A7) are preserved by a register-based subroutine; there's no need to save and restore them yourself.

On exit from a register-based subroutine, the D0 register contains the returned result, which is usually an error code. If it is zero, no error occurred. Since each such subroutine executes a TST.W D0 (test for D0 zero) instruction, a BEQ (branch on zero) instruction can be used to transfer control if no error occurred.

Putting It All Together

The purpose of this section is to walk you through the process of developing an actual application program using the MDS. In so doing, you'll produce a program that creates a large window on the screen and a menu bar with an Apple menu and a File menu. The File menu contains a Quit command you can use to exit the program and return to the Finder. Use this program as a shell for some of the programming examples presented in later chapters where all you need is a window in which to display a result.

The first step, of course, is to use Edit to create the program source code file for Asm, the control file for Link, the source code for the RMaker resource compiler, and the executive control file for Exec. The program we're going to examine is shown in Listing 2-2, the linker control file in Listing 2-3, the RMaker source file in Listing 2-4, and the executive control file in Listing 2-5. They are called MainDemo.Asm, MainDemo.Link, MainDemo.R, and MainDemo.Job, respectively. At this point, it's not important that you understand exactly what the assembly language code for the program does, although the comments should help you.

Listing 2-2. The Assembly Language Source File for MainDemo.

```
; MainDemo.Asm
;
; This is a shell for a simple one-window application.
; It does not support desk accessories.

WindID          EQU     128     ;Window ID
AppleID         EQU     1       ;Menu ID for Apple menu
FileID          EQU     2       ;Menu ID for File menu

; START OF STANDARD HEADER...

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QuickDraw globals
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want "arrow" cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; END OF STANDARD HEADER...

;       CLR     -(SP)           ;Use these instructions
;       PEA     'MainDemo.Rsrc' ; if you create
;       _OpenResFile            ; a separate resource file.
;       MOVE    (SP)+,D0

; Create and draw a window on the screen:
```

Listing 2-2. *continued*

```
        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file


; The next step is very important. It ensures that our new
; window is the active port so that we can draw in it. The
; pointer to the window is already on the stack.

        _SetPort                ;Make window the active GrafPort

; Create two standard menus:

        CLR.L   -(SP)           ;Space for handle
        MOVE    #AppleID,-(SP)  ;Menu ID number
        _GetRMenu               ;Get Menu from resource file

        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        CLR.L   -(SP)           ;Space for handle
        MOVE    #FileID,-(SP)   ;Menu ID number
        _GetRMenu               ;Get menu from resource file

        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        _DrawMenuBar            ;Display menu bar


; [insert your application code here]


GetEvent
        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #$FFFF,-(SP)    ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result flag
        BEQ     GetEvent        ;Branch if no pending event
```

Listing 2-2. *continued*

```
        MOVE    EventRecord+evtNum,D0   ;Get event type code
        CMP     #mButDwnEvt,D0  ;Is it a button-down event?
        BNE     GetEvent        ;No, so branch

        CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)      ;Where info
        PEA     ClickWindow     ;VAR window involved
        _FindWindow             ;Where was button pressed?

        MOVE    (SP)+,D0        ;Get result
        CMP     #InMenuBar,D0   ;Pressed in menu bar?
        BNE     GetEvent        ;No, so ignore

; See if "QUIT" was selected from File menu:

        CLR.L   -(SP)           ;space for result
        PEA     EventRecord+evtMouse   ;Where
        _MenuSelect             ;Get menu selection
        MOVE    (SP)+,D6        ;Save menu number in D6
        MOVE    (SP)+,D0        ;Discard item number

        MOVE    #0,-(SP)
        _HiliteMenu             ;Highlight from menu title

        CMP     #FileID,D6      ;In the FILE menu?
        BNE     GetEvent        ;No, so branch

; must have selected QUIT command:

        RTS                     ;Return to Finder


; The application constants:

EventRecord     DCB.B   EvtBlkSize,0   ;Space for event record

ClickWindow     DC.L    0       ;Pointer to window
```

Listing 2-3. The Linker Control File for MainDemo.

```
; MainDemo.Link
;
; Link this file to create an application
;  (without resources).

  MainDemo
  ; Insert "/Bundle" to set the bundle bit
  $
```

Listing 2-4. The RMaker Source File for MainDemo.

```
* MainDemo.R
*
* Compile this after assembling and linking MainDemo.Asm
*
* The next command appends the resources to the application:
!MainDemo

Type MENU
,1                      ;;Resource ID
\14                     ;;Title is the Apple symbol (ASCII $14)
About this demo...      ;;About box

,2                      ;;Resource ID
File                    ;;Menu Title
  Quit                  ;;Only item is "Quit"

Type WIND
,128                    ;;Resource ID
Development             ;;Title for Window
40 5 332 502            ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)
```

Listing 2-5. The Executive Control File for MainDemo.

```
Asm     MainDemo.Asm    Exec    Edit
Link    MainDemo.Link   Exec    Edit
RMaker  MainDemo.R      Exec    Edit
```

What is important is to know how to convert this group of source files into an executable program. To begin, start up the MDS assembler. This can be done in several ways, depending on where you are in the MDS system:

- You can double-click the Asm icon from the Finder's desktop, or click it and select Open from the Finder's File menu.
- If you're using Edit, Link, RMaker, or Exec, you can select the Asm command from the Transfer menu. If you do this from Edit, the current file being edited is assembled.

When the assembler takes over, you must specify the name of the file to be assembled by selecting the Open... command from the File menu. (This isn't necessary if you're editing the file and you transfer to Asm directly.) Select the file called MainDemo.Asm.

The assembly process then begins. As it proceeds, the name of the file being acted on is displayed in a box at the top of the screen. This is normally the name of the main source file, but it will change to that of an included file when necessary.

The assembly process creates a relocatable object code file called MainDemo.Rel. The next step is to convert this file into an application using Link. To do this, go from the assembler to the linker by pulling down the Transfer menu and selecting LINK. Then select the linker control file called MainDemo.Link and wait for the linking procedure to end. When it does, an application file called MainDemo will have been created.

At this point, you're still not done because the application makes use of resources for a window and two menus that have not yet been added to its file. Add them by moving to the resource compiler by selecting the RMAKER command from the Transfer menu, and then selecting the MainDemo.R file to work with.

The MainDemo.R file tells the resource compiler to append the menu and window resources to the application file created with the linker (MainDemo). The RMaker command for this is !MainDemo. If you're using MDS 1.0, precede the file name with the name of the disk on which it resides; if you

don't specify the disk prefix, and RMaker is on a different disk than MainDemo, you will see a "Can't create the output file!" error message.

When RMaker finishes, the application contains all the resources it needs to operate, and you're done. Return to the Finder by selecting the Quit command from the File menu and then double-click the MainDemo icon to run the application. When you do this, you'll see a large window entitled Development covering most of the screen. To leave the application, select the Quit command from its File menu.

Of course, you can also use the Exec program to automatically perform all the steps needed to create the complete application. To do this, double-click the Exec icon, choose the Open Job File command from its File menu, then select the MainDemo.Job file to act on. If all goes well, you'll eventually return to Exec where you can choose the Quit command from the File menu to return to the Finder so that you can launch MainDemo.

Alternative Application Development Techniques

There are, of course, several other ways to create a complete application. One alternative is to create .Rel modules with both Asm and RMaker, then link the two modules together into a final application with a linker control file of the form:

```
MainDemo.Rel
/RESOURCES
Resources.Rel
$
```

To do this, change the !MainDemo output file name in the RMaker source program to Resources.Rel before running RMaker.

Another alternative is to use RMaker to store the resources in an APPL file, and use an INCLUDE command to incorporate the code resources in the output file created by

Link. To do this, replace the RMaker line containing !MainDemo and the blank line following it with:

```
OurApplication
APPL????
```

and put the line:

```
INCLUDE MainDemo
```

at the end of the RMaker source file. This creates a complete application called OurApplication that has a creator code of ???? (which means undefined).

If you use the latter technique, you might want to use Link's /TYPE command to specify a file type code other than APPL and a creator code of ???? for the MainDemo output file. If you don't, the system will crash if you try to launch the resourceless MainDemo application. The Finder will not try to launch a non-application file that has a creator code of ????.

Table 2-9 contains a summary of the various types of files that can be handled by Asm, Link, and RMaker, and the types of output files they can create.

Table 2-9. Input and Output Files for the MDS Assembler Tools.

	Input Files	*Output Files*
Asm	* start with any text file containing 68000 source code (a file name extension of .Asm is optional). * the main source code file can include any textfile containing 68000 source code (a file name extension of .Asm is optional). Use INCLUDE.	* filename.Rel

Table 2-9. *continued*

	Input Files	*Output Files*
	* the main source code file can include any files containing packed symbols (a file name extension of .D is optional). Use INCLUDE.	
Link	* start with the FileName.Link linker control file. * the filename.Rel modules to be linked are named within the control file.	* filename (root name of first .Rel module). * as specified by the /OUTPUT FILENAME command.
RMaker	* start with the filename.R file containing RMaker source commands. * the main .R source file can include any previously created resource	* filename.Rel (a resource file in linkable form). * filename or filename.Rsrc (a general resource file). * !filename (append to existing resource file).

Creating a Separate Resource File

In the early stages of program development you may find it more convenient to compile your resources into a file other than the one containing the application code. That way, if you modify your resource definitions you won't have to waste time assembling and linking the program source code again (and vice versa); all you have to do is recompile the RMaker source file and run the same application file again. This speeds up the development process considerably.

To create a separate resource file in our example, replace the !MainDemo statement in the RMaker file with MainDemo.Rsrc. This tells RMaker to store the resources in a file called MainDemo.Rsrc on the disk.

You must also add the following four lines of code after the initialization instructions in MainDemo.Asm:

```
CLR     -(SP)               ;Space for result
PEA     'MainDemo.Rsrc'     ;Name of resource file
_OpenResFile                ;Open the resource file
MOVE    (SP)+,D0            ;Pop result (refnum)
```

This specifically opens the MainDemo.Rsrc resource file so that its resources are available to the application. You don't have to explicitly open a resource file if the resources are stored in the same file as the application, because the resource fork of the application file is automatically opened when the application is launched.

There is also a _CloseResFile instruction you can use to close a resource file. You rarely have to use it, however, since all resource files are closed when the application ends and returns to the Finder.

The Standard Program Header

The first part of the demonstration program contains a sequence of instructions designed to initialize various groups of toolbox and operating system instructions, called **Managers**, used by the program. (We've actually initialized managers we don't use so that the same header can be used with any program you might write.) It also flushes any pending input/output (I/O) operations and turns on the standard arrow cursor.

A similar header should be inserted at the beginning of every application you write for the Macintosh. If you don't do this, the

program will fail when you try to use a manager that has not been initialized. The easiest way to insert it is to create the header with the Editor and save it to a file called Header.Asm. Then, when you write a program, just put the line:

```
INCLUDE Header.Asm
```

at the beginning of the file and the header will be loaded and assembled when you assemble the file.

Applications and the Finder

Unless you include some special resources in an application file, the Finder uses the generic icons shown in Figure 2-3 when it displays the application, or its data files, on the desktop.

Figure 2-3. The Generic Icons for an Application and a Document File.

It is possible to define alternative icons for the Finder to use, however. To do this, you must first add four types of resources to the application file: BNDL, FREF, ICN#, and a resource whose type code is the same as the signature of the application. You must also set the bundle bit in the application's disk directory entry using the /BUNDLE linker instruction.

Let's go through the steps to follow to create the resources the Finder needs to display the icon shown in Figure 2-4 for our example application.

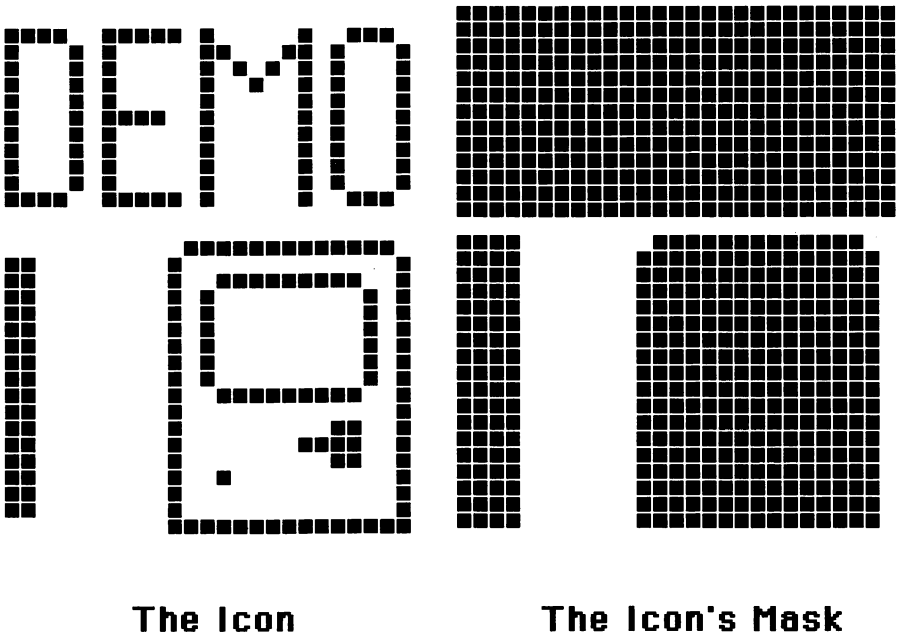


Figure 2-4. A Custom Icon and its Mask.

Version Data Resource

The first step is to pick a four-character signature for the application. We'll choose DEMO: we won't bother seeking approval of the signature from Apple's Macintosh Technical Support Division. You *should* get approval if you're developing a commercial application. The first resource type we need to define has the name DEMO; it is called the **version data** (or **autograph**) of the application.

By convention, the resource ID of the version data is zero. The data it contains can be anything you like, so we'll use a standard Pascal-style string (one preceded by a length byte) containing the title of the application:

```
TYPE DEMO = GNRL        ;;DEMO is not pre-defined
,0                      ;;Resource ID (0 by convention)
.P                      ;;Pascal-type string follows
Demo by Gary Little     ;;The string itself!
```

Icon List Resource

Next, we have to design the icon for our application and any files it creates. As you will see in Chapter 7, an icon is represented by a series of 32 long words; each bit in a long word reflects the state of a pixel in a row of the icon (1 = black, 0 = white).

The Finder also requires a **mask** for each icon—the Finder uses it to determine the appearance of the icon when it is selected. For a selected icon, pixels in the icon that correspond to white pixels in the mask are displayed as usual; those that correspond to black pixels are inverted. (This is an exclusive-or operation.) The mask is usually the same shape as the standard icon, but is filled with black.

The icon definition and its mask must be stored in an ICN# (icon list) resource, with the standard icon coming first. (See Listing 2-6.) If you need to define more than one ICN# resource, be sure to give each a unique resource ID.

File Reference Resource

The next resource to create is FREF (file reference). It contains a list of each file type used by the application and the **local ID** of the ICN# resource containing its icon definition. This local ID is not the same as the resource ID; the mapping of a local ID to an actual resource ID is defined within the BNDL resource.

Here is the format of the RMaker source code for a FREF resource:

```
Type FREF
,128                ;;Resource ID for FREF
APPL 0              ;;File type APPL, local ID of ICN# = 0
```

If you've defined a special icon for a data file used by the application, you must create another FREF resource containing the file type code for the file and its local ID code. Its local ID must be different from the one used for the application's icon.

Bundle Resource

The last resource to define is BNDL (bundle). It contains three types of items:

- The application's signature and the resource ID of its version data (usually zero);
- a mapping of the local IDs for the FREF resources used by the application to the actual resource IDs; and
- a mapping of the local IDs for all ICN# resources referred to in the FREF resources to the actual resource IDs.

Here is the RMaker format of the BNDL resource file:

```
Type BNDL
,128                    ;; resource ID for BNDL
DEMO 0                  ;; Version data resource ID
ICN#                    ;; resource type for next line
0 128 1 129             ;; local to absolute ID mapping
FREF                    ;; resource type for next line
0 128 1 129             ;; local to absolute ID mapping
```

This example presumes we've created two ICN# and FREF resources (one for the application and one for a data file it uses).

Notice how the mapping scheme works: The line after the one containing the name of the resource type is made up of consecutive pairs of numbers. The first number in a pair represents the local ID and the second represents the actual resource ID.

There is one last step you must take to ensure that your application and its icons will integrate smoothly with the Finder: You must set the application's **bundle bit**. To do this, put the /BUNDLE command in the linker control file.

When the Finder first encounters an application whose bundle bit is set, it copies the version data, BNDL, ICN#, and FREF resources from the application's resource file and puts them in an invisible file called DeskTop. This is the file that the Finder inspects to determine what icon to display for a file associated with a given creator-type code (signature).

When the Finder transfers resources to DeskTop it checks to see if the resource IDs of the application's ICN# and FREF resource IDs are already in use. If they are, it renumbers them and changes the absolute IDs in the BNDL resource to reflect the changes. Since the data in other Finder-related resources use local IDs, they do not have to be modified—this is why local IDs are used in the first place.

You can use the resource definitions in Listing 2-6 to create a custom icon for our example program. Add these statements to the RMaker source file in Listing 2-4, and add the commands:

```
/BUNDLE
/TYPE 'APPL' 'DEMO'
```

to the linker control file in Listing 2-3 before assembling and linking the application. The /TYPE commands sets the application's signature to DEMO, the one referred to in the BNDL resource.

Listing 2-6. The RMaker Resource Definitions for a Custom Icon.

```
* NewIcon.R
*
* These resources allow the Finder to display
* a custom icon for an application.

Type DEMO = GNRL        ;;Version data (signature) resource
,0                      ;;Resource ID (0 by convention)
.P                      ;;A Pascal string follows
Demo by Gary Little

Type ICN# = GNRL        ;;An icon list resource
```

Listing 2-6. *continued*

```
,128
.H                      ;;hexadecimals follow
00000000                ;;This is the icon definition
1E7D04E0
11418D10
11415510
11412510
11410510
11790510
11410510
11410510
11410510
11410510
1E7D04E0
00000000
00000000
0003FFE0
18040010
1804FF90
18050050
18050050
18050050
18050050
18050050
18050050
1804FF90
18040010
18040190
18040790
18040190
18050010
18040010
18040010
0007FFF0
3FFFFFF8                ;;This is the icon's mask
3FFFFFF8
3FFFFFF8
3FFFFFF8
3FFFFFF8
3FFFFFF8
3FFFFFF8
3FFFFFF8
```

Listing 2-6. *continued*

```
3FFFFFF8
3FFFFFF8
3FFFFFF8
3FFFFFF8
3FFFFFF8
00000000
1E01FFF0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
1E03FFE0
00000000

Type FREF
,128
APPL 0                  ;;APPL file type uses ICN# 0

Type BNDL               ;;Bundle resource
,128
DEMO 0                  ;;Version data resource + ID
ICN#
0 128                   ;;Local 0 = resource ID 128
FREF
0 128                   ;;Local 0 = resource ID 128
```

Once you've defined an icon for an application, it's a bit tricky changing it. That's because the Finder doesn't transfer the application's icon to the DeskTop file every time it encounters the application, only the first time it encounters it. To force the Finder to use your redefined icon, you must rebuild the DeskTop file from scratch by holding the Option and Command keys when the application's disk is inserted.

Chapter 3

The 68000 Instruction Set

In this chapter we're going to take a close look at the complete 68000 instruction set to determine exactly what each instruction does. In so doing, you'll see what addressing modes can be used with what instructions and how instructions affect the five status flags in the condition code register. Once you've mastered this information, you'll be ready to develop assembly language programs on the Macintosh.

For the purpose of analysis, the 68000 instruction set will be separated into seven logical groups:

Data Movement Instructions. These instructions move data from place to place.

Program Control Instructions. These instructions control the order in which the 68000 executes a program.

Arithmetic Instructions. These instructions add, subtract, multiply, divide, and negate binary numbers, or add, subtract, and negate binary-coded decimal (BCD) numbers.

Bit Manipulation Instructions. These instructions adjust or test the settings of individual bits in an operand.

Logical Instructions. These instructions perform logical operations (and, or, exclusive or, not) according to the rules of Boolean algebra.

Shift and Rotate Instructions. These instructions move bits in an operand to the left or right, or in a circle formed by logically "connecting" the least- and most-significant bits directly or through the X flag.

System Control Instructions. These instructions perform a variety of system control operations, such as manipulating the status

register, the user stack pointer, and forcing exception processing.

We'll look at each of these groups in separate sections.

The tables of instructions in this chapter indicate the addressing modes permitted for the source and destination operands of each 68000 instruction. These modes are described in Table 1-2 in Chapter 1. For a two-operand instruction, any source mode marked with a given symbol (x or o) may be associated with any destination mode marked with the same symbol. One-operand instructions can be used with any of the marked addressing modes, of course.

Note that the word **Address** in an operand table refers to the absolute addressing mode (long or short) and the word **#Immediate** refers to the immediate addressing mode.

The tables also indicate how the settings of the condition code flags change after an instruction is executed. The following symbols are used to represent the changes:

*	the flag changes to 0 or 1, depending on the result
1	the flag is always set to 1
0	the flag is always cleared to 0
-	the flag is not affected
U	the flag is undefined and meaningless

Data Movement Instructions

The data movement instructions are summarized in Table 3-1, which begins on page 149 at the end of this chapter.

The main 68000 instruction for moving data from place to place is MOVE. With it you can move data between registers, between memory locations, or between a memory location and a register. You can also use it to store a specific number in a register or a memory location.

If a multibyte number is moved to an area of memory beginning at a particular location, the most-significant bytes of the number are stored first (at the lower addresses). Some

microprocessors, notably the Apple II's 6502, store such numbers in the opposite order.

Here are some examples of how to use MOVE:

```
MOVE.L D1,D0            ;Move D1 to D0 (entire register)
MOVE #345,D3            ;Move decimal 345 to D3 (word)
MOVE MyConstant,D1      ;Move word stored at MyConstant
                        ; into D1
```

In the last example, MyConstant is the label for a data area reserved using the DC (define constant) assembler directive. The MDS assembler always converts a reference to this type of label as a reference to label(PC) to make the program relocatable, as required by the Macintosh operating system. If MyConstant is a symbol assigned to an absolute memory location using the EQU or SET directive, however, the absolute addressing mode is used for the source operand. If a symbol represents an immediate quantity rather than an address, it must be preceded by #.

If space for a variable is reserved using the DS directive, any reference to the variable must use A5 address register indirect addressing:

```
MOVE D6,MyVariable(A5)  ;Move the word in D6 into
                        ; MyVariable + (A5)
```

If you will be moving a value to a data area in memory, you should reserve the area with the DS directive. If you use DC instead, you'll run into difficulties, because program counter indirect addressing is not permitted for destination operands. This means you cannot use an instruction of the form:

```
MOVE D1,MyData        ;Illegal where MyData is a constant
```

Instead, you must use code like this:

```
LEA  MyData,A0          ;Move EA of MyData into A0
MOVE D1,(A0)            ;Store D1 at MyData
```

LEA is another common data movement instruction. It moves the **effective address** (EA) of the source operand into the destination operand, *not* the value stored at that address. It is often used to move the base address of a data structure into an address register so items in the structure can be accessed using an indirect addressing mode with or without index. A related instruction, PEA, pushes the effective address of its operand on the stack; this is often used for passing the address of a data structure to a Macintosh toolbox subroutine.

Clearing to Zero

The 68000 has a special instruction for storing a zero in a particular operand: CLR (CLeaR). This instruction is preferable to a MOVE #0,<EA> instruction because it executes more quickly.

Moving to Address Registers

The 68000 has a separate instruction for moving a word or long word quantity into an address register: MOVEA. If you use it to move a word, the **sign bit** (bit 15) is automatically extended through the high-order 16 bits of the address register. The other major difference between MOVEA and MOVE is that MOVEA does not affect the status flags in the condition code register.

Quick Moves

MOVEQ (MOVE Quick) is a special form of the standard MOVE instruction you can use when the source operand is a small immediate quantity between −128 to +127 and the destination operand is a data register. The advantage of using it instead of a standard MOVE instruction is that it is faster and takes up less space.

If you use a MOVE instruction when you could have used a MOVEQ instruction, don't worry. The MDS assembler auto-

matically optomizes the code by substituting the MOVEQ instruction during the assembly process.

Moving Multiple Registers

MOVEM is a very convenient instruction. It moves the contents of a group of data and address registers to a temporary storage area in memory or vice versa.

MOVEM is most often used to save the contents of registers before calling a subroutine that might change the values in those registers. On return from the subroutine the original values can be restored by another MOVEM in the opposite direction. You would not, of course, restore (or save) any registers that may be used to return results.

The two forms of the MOVEM instruction are:

```
MOVE    register_list,<EA>      ;save registers
MOVE    <EA>,register_list
```

where `register_list` represents the names of the registers to be transferred. Each individual register in the list is separated from the next by a /. In addition, you can specify a group of consecutive address or data registers by using a minus sign to separate the first and last register in the range.

For example, if you wanted to save D0, D1, D2, D4, A2, and A3 on the stack, you could use the instruction:

```
MOVEM D0/D1/D2/D4/A2/A3,-(SP)
```

or you could use:

```
MOVEM D0-D2/D4/A2-A3,-(SP)
```

Use the MDS assembler's REG directive to assign a symbolic name to a register list.

The order of transfer of registers with MOVEM is D0 through D7, followed by A0 through A7, unless you are using the −(An) addressing mode where the order is A7 through

A0 then D7 through D0. This means that no matter what addressing mode is used, the register values are arranged in memory in the same order.

Swapping Data Register Halves

SWAP exchanges the upper word of a data register with the lower word, which then can be accessed with a word operation. It cannot be used with address registers.

Exchanging Registers

EXG (EXchanGe) exchanges the contents of two address registers, an address register and a data register, or two data registers. Using EXG is a convenient way to save the contents of a register when you have to use the register for something else (perhaps for passing data to a subroutine). For example, the following subroutine could be used to preserve D0:

```
EXG     D0,D6           ;Save D0 in D6
LEA     Data,D0
JSR     Subroutine
EXG     D0,D6           ;Restore value of D0
```

Note that this technique works only if the register you're exchanging with D0 (D6 in the example) is not altered by the subroutine.

Linking and Unlinking the Stack

The LINK and UNLK instructions facilitate the development of **re-entrant** and **recursive subroutines**. A re-entrant subroutine is one that can be interrupted, called by the interrupt handler, and then completed without any adverse effects. A recursive subroutine is one that can call itself without causing spurious results.

Most subroutines are not re-entrant or recursive because they use a fixed area for storage of their own variables (called **local variables**) and temporary results. This area is overwritten if you call the subroutine while you're already in it.

To avoid this problem, you can set up a data area on the stack relative to the stack pointer (called a **stack frame**), and access the data elements as offsets from the stack pointer. When the subroutine is called recursively, a similar frame is created, but it will be below the old one, so there will be no interference with the data used during the first subroutine call.

The LINK instruction sets up such a stack frame. It is of the form:

```
LINK    An,#-num
```

where `-num` represents the number of bytes in the frame. This number *must* be negative and even. When the 68000 executes the LINK instruction it first pushes the address register specified in its operand on the stack and then places the resulting stack pointer into the address register. The number in the operand is then added to the stack pointer to make room for the frame on the stack. Since the frame size is a negative number, the frame is, in effect, pushed on the stack.

Once the frame has been created, the stack pointer will point to its base, so you can access the data elements in the frame using the address register indirect with displacement addressing mode:

```
MOVE    2(SP),D0        ;Move 2nd word in frame into D0
MOVE    D1,0(SP)        ;Store D1 into 1st word in frame
```

These examples assume, of course, that you haven't pushed anything else on the stack after the LINK instruc-

tion. If you have, you'll have to increase the SP displacements accordingly.

To remove a stack frame, use the UNLK (UNLinK) instruction. It is of the form:

```
UNLK    An
```

where `An` is the same address register used by the LINK instruction. UNLK transfers the contents of the address register into the stack pointer, and pops a long word from the stack into the address register. As a result, the stack pointer and the address register are restored to the values they held just before the LINK instruction was executed.

In order for UNLK to work properly, the address register must contain the same value stored in it by the LINK instruction.

Moving Data to and from Peripherals

There is one final data movement instruction, but you'll rarely use it on the Macintosh. It is the move peripheral data instruction, MOVEP, and it transfers information between a data register and peripheral devices such as the Macintosh's two serial ports. These chores are usually performed by calling low-level I/O subroutines that form part of the Macintosh operating system.

When you send data to a peripheral device with MOVEP, each byte of the operand (which is a word or a long word in a data register) is stored at every second memory location beginning at the effective address of the destination operand. The high-order byte or bytes of the operand are sent first and the effective address can describe an odd or even address.

Similarly, when you're reading data from the peripheral

device, the data register is filled, high-order byte first, from every second memory location starting with the base address.

Communication with peripherals is handled in this strange way so that peripherals that can handle only a byte of data at a time can be interfaced to the 68000.

Program Control Instructions

In the normal course of events, the 68000 executes an instruction, increments the program counter by the size of the instruction, then executes the next instruction in memory. From time to time, however, it becomes necessary to alter this linear flow so you can skip to parts of a program dictated by the results of a calculation or the behavior of the user. This is done with 68000 branch and jump instructions that implicitly change the address stored in the program counter register. See Table 3-2, page 159, for a complete listing of the 68000 program control instructions.

In this section we're going to look at the instructions you can use to move around in a program. These instructions can be categorized as unconditional branch, conditional branch, and looping instructions. We'll also look at some conditional instructions that can be used to set and clear memory locations, and registers that may be used as flags.

Unconditional Jumps and Branches

The 68000 has four instructions you can use to force a transfer of control to a particular target address: JMP (JuMP), BRA (Branch Relative Always), JSR (Jump to SubRoutine) and BSR (Branch to SubRoutine).

The first two instructions, JMP and BRA, are straightforward. In each case, the effective address of the operand is placed in the program counter, causing execution to continue

at that new address. JMP is usually used with a program label as an operand; BRA must be:

```
JMP     MoreCode
BRA     SkipNext
```

The difference between the two is that the operand for BRA is always a 16-bit signed offset (or 8-bit if the BRA.S short form is used) to the target address, whereas the operand for a JMP instruction could represent a number of addressing modes, including program counter relative, as shown in the example. This means the target of a branch is more limited since it must be in the range −32768 to +32767 (16-bit) or −128 to +127 (8-bit) from the position immediately following the operation word. Since code segments on the Macintosh cannot exceed 32K bytes, this does not pose a problem.

The other two unconditional branch instructions, JSR and BSR, are a bit more complex. They pass control to a target address like their JMP and BRA counterparts, but they also push the address of the instruction that follows them in memory on the stack. By doing this, the program at the target address, called a **subroutine**, can return control to this instruction by popping the address from the stack into the program counter using the RTS (ReTurn from Subroutine) instruction.

Any portion of code that may have to be executed in various parts of a program should be made into a subroutine and called with a JSR or BSR instruction. This not only makes the program more modular and easy to read, it reduces the amount of memory needed by the program.

There is another return instruction with which you should become familiar: RTR (ReTurn and Restore condition codes). Like RTS, this instruction pops a return address from the stack, but before it does, it pops a word and places the low-order byte into the condition code register. RTR is useful where you want to preserve the condition code flags across a subroutine call. To use it you have to push the contents of the CCR as soon as you enter the subroutine:

```
        JSR     MySub           ;Call the subroutine
        .
        .
MySub   MOVE    SR,-(SP)        ;Save flags on stack
        .
        [subroutine code here]
        .
        RTR                     ;Restore flags and return
```

The RTR instruction eliminates the need to execute an explicit MOVE (SP)+,CCR pop instruction before ending the subroutine with an RTS.

Conditional Branches

One of the most common things you will do in a program is alter the program flow conditionally. That is, you will make a decision on what part of a program to execute based on the result of a calculation or comparison, the number in a register, or the value of the condition code flags. The instructions you use to make such decisions are Bcc (branch conditionally) instructions of the form:

```
Bcc     TargetAddr
```

where `TargetAddr` represents the label for the instruction in the program where control is to pass if the flag settings associated with the cc condition are in effect. If the flags are not properly set, the next instruction in line is executed instead.

The cc in Bcc represents a one- or two-character mnemonic for a condition that is true when the condition code flags in the status register have a particular set of values. Table 3-3 shows what each cc mnemonic is, what it means, and the settings of the flags if the cc condition is true. For example, LS means lower or same and is true if the carry and zero flags are both set to 1.

Table 3-3. Conditional Tests Used with the Bcc, DBcc, and Scc Instructions.

	cc Name	Condition	Flag Setting for Condition=true
(1)	CC	carry clear	C=0
(2)	CS	carry set	C=1
	EQ	equal	Z=1
(3)	F	false	(always false)
(4)	GE	greater or equal	(N=1 V=1) or (N=0 V=0)
(4)	GT	greater than	(N=1 V=1 Z=0) or (N=0 V=0 Z=0)
	HI	higher	(C=0 Z=0)
(4)	LE	less or equal	Z=1 or (N=1 V=0) or (N=0 V=1)
	LS	lower or same	C=1 or Z=1
(4)	LT	less or same	(N=1 V=0) or (N=0 V=1)
(4)	MI	minus	N=1
	NE	not equal	Z=0
(4)	PL	plus	N=0
(3)	T	true	(always true)
(4)	VC	overflow clear	V=0
(4)	VS	overflow set	V=1

(1) CC is equivalent to HS (higher or same).
(2) CS is equivalent to LO (lower).
(3) F and T cannot be used with the Bcc instructions. BF, if it were permitted, would be the same as NOP. BT, if it were permitted, would be the same as BRA.
(4) These conditions are useful when your operation uses two's complement signed arithmetic.
Note: Higher and lower refer to unsigned numbers. Greater and less refer to signed two's complement numbers.

Most 68000 instructions affect the condition code flags in some way. For example, if you move a zero into a data register, the zero flag becomes 1. If you compare two operands with a CMP instruction, all flags except extend are affected, depending on the result of the subtraction operation that CMP performs: destination minus source.

Bcc instructions are most often used after a CMP instruction so that you can easily change the program flow if a result

is the same as, higher, or lower, than another number. In fact, most of the cc mnemonics stand for a phrase that reflects the relative magnitudes of two numbers, thus it's easy to remember which Bcc instruction to use.

Consider the following comparison:

```
CMP     #4,D1
```

When this instruction is executed, the 68000 first subtracts 4 from the word in the D1 register, then sets the flags based on the result. The result itself is not stored anywhere (use SUB for that).

If the two operands are unsigned binary numbers, you can use the following Bcc instructions to transfer control to another position in the program according to the result of the comparison:

- BEQ : branch if D1 is equal to 4
- BNE : branch if D1 is not equal to 4
- BHS : branch if D1 is higher than or the same as 4
- BHI : branch if D1 is higher than 4
- BLS : branch if D1 is lower than or the same as 4
- BLO : branch if D1 is lower than 4

Note: You can use BCC (carry clear) instead of BHS, and BCS (carry set) instead of BLO.

For signed binary numbers, the following branch instructions should be used instead:

- BGE : branch if D1 is greater than or equal to 4
- BGT : branch if D1 is greater than 4
- BLE : branch if D1 is less than or equal to 4
- BLT : branch if D1 is less than 4
- BVC : branch if no overflow occurred
- BVS : branch if overflow occurred
- BPL : branch if result is positive
- BMI : branch if result is negative

Notice that in a comparison operation, you are always comparing the destination operand to the source operand, not the source operand to the destination operand.

The default form of a Bcc instruction is the **long form**. This means that the word following the operation word in memory contains the 16-bit **offset** to the target destination (the offset is measured from the location after the operation word). This is a signed number from −32768 to +32767 and is calculated for you by the assembler; all you have to do is specify a label.

The alternate form is the **short form**; you tell the assembler to use it by adding a .S suffix to the Bcc instruction mnemonic. For these branches, eight bits in the operation word itself are used to hold the offset. This means the range of the branch is restricted to −128 to +127, but you do save two bytes of memory.

There is another set of conditional instructions of the form Scc (Set conditionally). These instructions don't directly affect the program flow but can be used to store ones in each bit of a byte operand if the condition defined by cc is true, or to store zeros if the condition is false. The target operands used by Scc instructions are called **flags** because their contents are usually status indicators that a program can check when making decisions on what part of a program to execute next.

Looping

Two of the most common high-level programming constructs are DO...UNTIL and FOR...NEXT loops where a portion of code is repeatedly executed until a counter is exhausted or a particular terminating condition occurs. You can build such loops in assembly language using the 68000 DBcc (test condition Decrement and Branch until condition true) instructions. As with the Bcc instructions, the cc refers to one of the 16 conditional tests supported by the 68000.

The operation of a DBcc instruction is shown in Figure 3-1. Before entering the loop, a data register is loaded with a word that contains the maximum number of loops to perform, *minus one*. The start of the loop is identified by a labeled instruction.

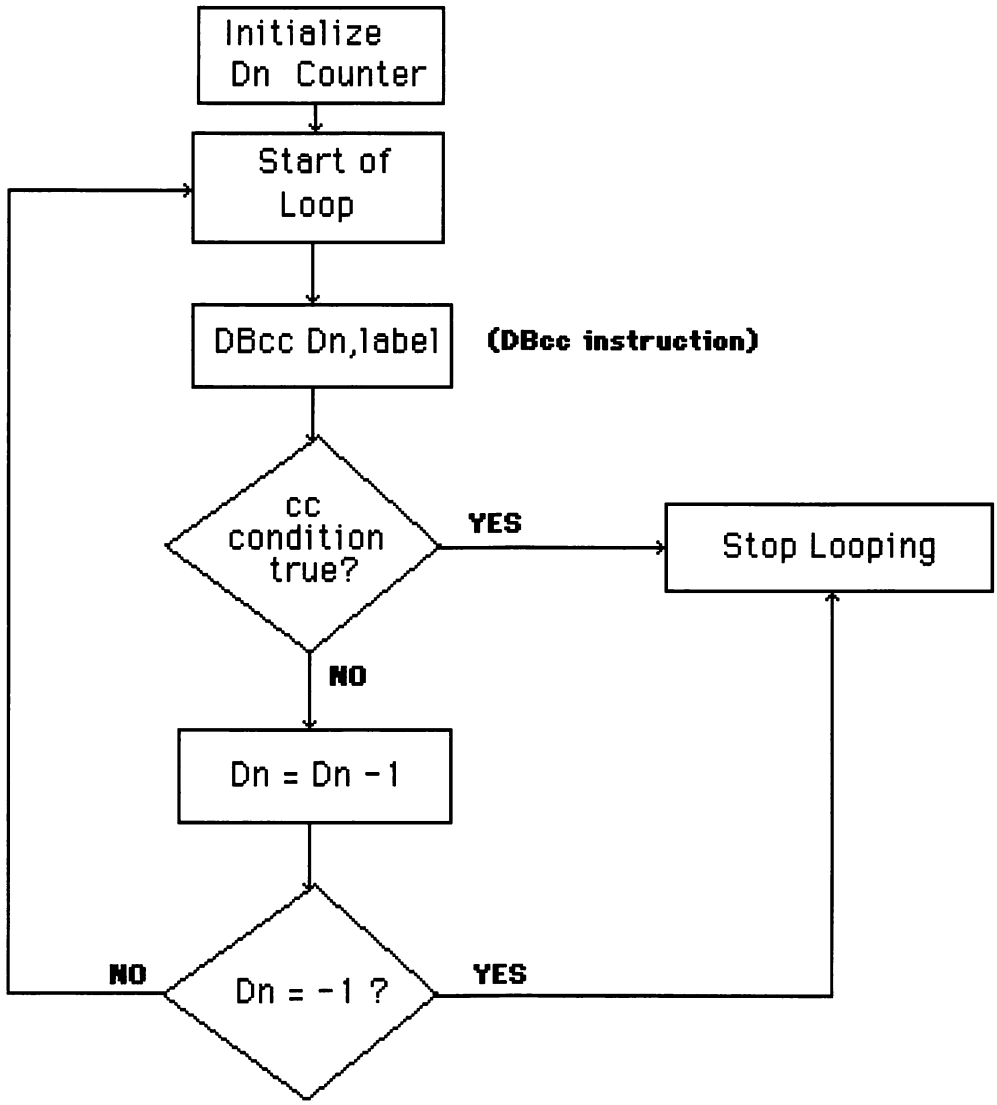


Figure 3-1. The 68000 DBcc Instruction.

At the bottom of the loop is a `DBcc Dn,label` instruction. When this instruction is encountered, the 68000 first checks to see if the condition is true; if it is, control passes to the following instruction and looping ends. If it's not, the word in the data register is decremented and, if the result is not −1, you will loop to the labeled instruction. When the counter reaches −1, looping ends.

Since the termination condition is Dn = −1, the initial value of the loop counter must be one less than the number of times you want to loop. For example, if you want to loop a maximum of 10 times, set Dn equal to 9.

As you have seen, you normally exit a loop in one of two ways: when the condition becomes true or the counter reaches −1, whichever comes first. You can easily design a loop governed by the counter only, by using the DBF version of DBcc. Since the condition code F means false or never true, looping will never end before the counter reaches −1. DBF is useful when you must perform a task a fixed number of times. Another name for DBF is DBRA.

Arithmetic Instructions

The 68000 has several instructions you can use to perform the basic arithmetic functions: addition, subtraction, multiplication, division, and negation. All these instructions, except the negation instructions, require two operands. Negation involves one operand only. Table 3-4 on page 166 summarizes the arithmetic instructions supported by the 68000.

The operands for addition, subtraction, and negation instructions can be either simple **binary numbers** (signed or unsigned) or **binary-coded-decimal** (BCD) numbers. The multiplication and division instructions work with binary numbers only. Before discussing the arithmetic instructions, let's look at the differences between binary and BCD numbers. (See Figure 3-2.)

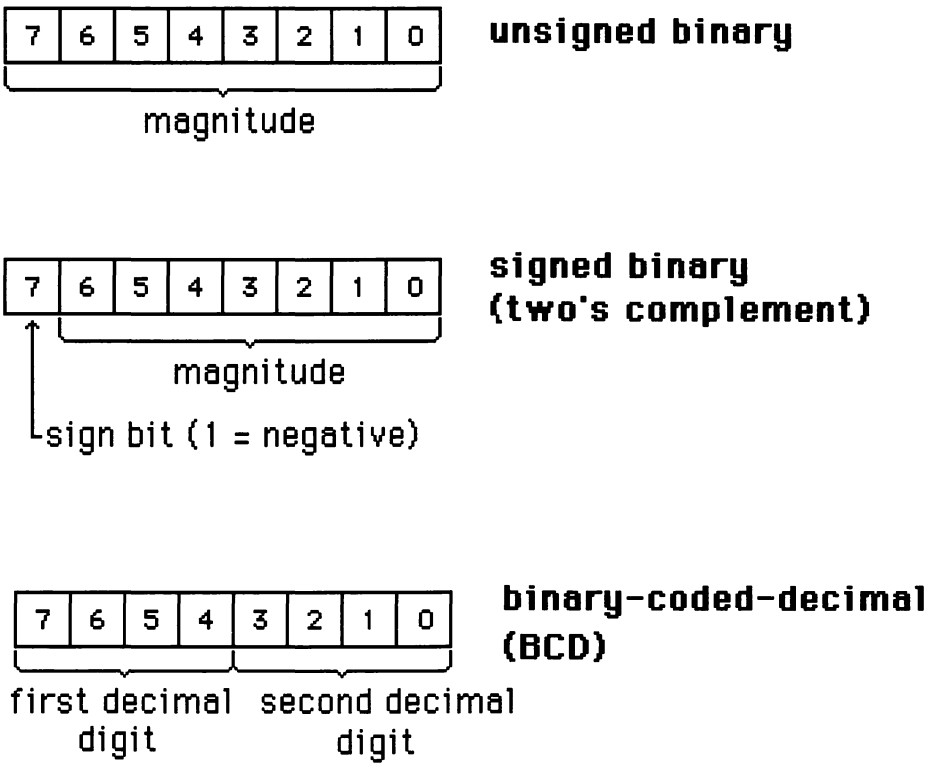


Figure 3-2. Binary and BCD numbers.

Unsigned and Signed Binary Numbers

Binary numbers can be **unsigned** or **signed**. An unsigned binary number is one where every bit of the byte, word, or long word describing the number contributes to the number's magnitude. The number is always positive, of course. The contribution, or weight, of a particular bit is given by the decimal number 2^n, where n is the bit number (0 to 7 for a byte, 0 to 15 for a word, and 0 to 31 for a long word). To calculate the decimal equivalent value of a binary number, simply add together the weights of every bit that is one. For instance, binary 10001001 is the same as decimal 137 ($2^7 + 2^3 + 2^0$).

The 68000 expects a signed binary number to be in **two's complement form**. This means the most-significant bit of the number (bit 7 for a byte, bit 15 for a word, and bit 31 for a long word) holds the sign of the number: one for negative, zero for positive. For a positive number, the remaining bits reflect the magnitude of the number in the same way they do for an unsigned binary number.

To calculate the two's complement binary representation of a negative number, take the binary form of the absolute value of the number, complement it by changing all ones to zeros and zeros to ones, then add binary one to the result. Let's see how this works by considering how to convert decimal −43 into binary two's complement form:

```
00101011 (+43)
11010100 (complement)
       1 (add 1)

--------
11010101 (-43 in two's complement)
```

The 68000 uses the two's complement form for signed numbers because it simplifies binary arithmetic operations: The numbers can be dealt with just as if they were unsigned binary numbers and the signed result will still be correct. No explicit adjustments have to be made by your program to account for the different signs of the numbers being manipulated.

An overflow condition occurs if the result of an operation is too large or too small to be represented in the two's complement form. The ranges of allowed values are as follows:

- byte −128 to +127
- word −32768 to +32767
- long word $-(2^{31})$ to $(2^{31})-1$

When an overflow occurs, the overflow flag in the 68000 status register is set.

BCD Numbers

A **binary-coded-decimal** number is a decimal number whose digits are stored in consecutive half-bytes. Each digit is stored as a binary number from 0000 to 1001. The binary patterns from 1010 to 1111 are not used since they don't correspond to decimal digits.

Consider the BCD form of decimal 83, namely 10000011: The first half-byte, 1000, is the 8 digit, and the second half-byte, 0011, is the 3 digit. Compare this with the standard binary representation of the same number, 01010011.

Decimal numbers entered by a user from the keyboard are often stored in BCD form. It is easy to do because the low-order four bits of the ASCII code for a digit turn out to be the digit's BCD representation.

Binary Addition, Subtraction, and Negation

You can add two binary numbers together with the ADD and ADDX instructions; the result is stored in the destination operand. The only difference between these two instructions is that ADDX also adds the value of the extend flag to the result.

After an addition operation, the extend flag indicates whether the result of the addition was larger than the largest unsigned number the operand can hold. For example, if you're working with byte operands and you add 245 to 10, a carry is generated, and the extend flag is set, because the result is larger than the largest byte quantity, 255.

Addition affects the state of the overflow flag as well, a fact that is important if you are dealing with two's complement signed numbers. The overflow flag is set if the result of the addition is out of the range of signed numbers permitted by the operand size.

If you're simply adding two numbers, each of which fits in one operand, you will use the ADD instruction, or the ADDQ instruction if the source operand is an immediate number from one to eight, to combine them:

```
ADDQ    #5,D0           ;Add 5 to D0
ADD     D0,D1           ;Add D0 to D1
ADD.L   #60000,(A0)     ;Add 60000 to (A0)
```

If you're dealing with numbers that won't fit in one operand, you must add the low-order part of the numbers with the ADD instruction, and then add the higher-order parts with the ADDX instruction to ensure that any carry generated is included in the total. For example, suppose you want to add the number $00FEC200 stored at (A0) to (A0)+3 to the number $00353500 stored at (A1) to (A1)+3 using word-sized operands. Here's how you'd do it:

```
ADD.W   2(A0),2(A1)     ;Add low-order, generate carry
ADDX.W  (A0),(A1)       ;Add high-order with carry
```

If the extend flag is set after these two instructions, the number is larger than $FFFFFFFF. The overflow flag is set if the result is greater than $7FFFFFFF or less than $80000000.

The SUB (Subtract) and SUBX (Subtract with eXtend) instructions are the basic subtraction operations and they behave similarly to ADD and ADDX. For subtractions, however, the extend flag is set if a borrow condition occurs.

NEG (Negate) and NEGX (Negate with eXtend) change the sign of a number by subtracting the operand from zero. If NEGX is used, the extend bit is also subtracted.

BCD Addition, Subtraction, and Negation

Three instructions use BCD numbers: ABCD (Add BCD with extend), SBCD (Subtract BCD with extend), NBCD (Negate BCD with extend). They use byte operands only and all include the extend flag in their operations.

There are two basic forms for the BCD addition (ABCD) and subtraction (SBCD) instructions:

```
ABCD    Dx,Dy           ;Add Dx to Dy
SBCD    Dx,Dy           ;Subtract Dx from Dy
```

and:

```
ABCD    -(Ax),-(Ay)     ;Decrement, add (Ax) to (Ay)
SBCD    -(Ax),-(Ay)     ;Decrement, subtract (Ax) from (Ay)
```

In each case, the source operand is added to or subtracted from the destination operand, as is the extend flag, and the result is stored in the destination operand. If the result of the addition is nonzero, the zero flag is cleared to zero. If it isn't, the state of the zero flag does not change. If a decimal carry is generated, the carry and extend flags are set to one. Because of the way flags are handled, you should always set the zero flag to one and the extend flag to zero before a BCD operation. This can be done with a MOVE #4,CCR instruction.

The second form of the ABCD instruction is handy for quickly adding together two sequences of BCD digits stored in memory. To do this, first load one address register with the address following the last digit of the first number and another with the address following the last digit of the second. Then, if you have six digits to add together, use the following code:

```
MOVE    #4,CCR          ;Z=1, X=0
ABCD    -(A0),-(A1)     ;Add low-order digits
ABCD    -(A0),-(A1)     ;Add mid-order digits
ABCD    -(A0),-(A1)     ;Add high-order digits
```

Because the address registers are pre-decremented, successive ABCD operations always access the next two digits in the BCD string of digits. In a more general program, you would create a program loop using a DBRA instruction to repeat a single ABCD instruction a given number of times.

The last BCD instruction is NBCD (Negate BCD with extend). This instruction subtracts from zero the sum of the operand (a byte containing two BCD digits) and the extend bit and stores the result in the operand. As with the ABCD and SBCD instructions, the zero flag is cleared if the result is non-zero, but is not changed otherwise.

Multiplication and Division

The 68000 has two powerful multiplication instructions, MULU (Unsigned MULtiply) and MULS (Signed MULtiply). As their names suggest, MULU acts on unsigned binary numbers and MULS acts on signed binary numbers.

The general forms of these instructions are:

```
MULU    <EA>,Dn
MULS    <EA>,Dn
```

The two operands are always words and the result is a long word. The result is stored in the data register specified in the destination operand.

There are also two division instructions, DIVU (Unsigned DIVide) and DIVS (Signed DIVide). The general forms are the same as the multiplication instructions:

```
DIVU    <EA>,Dn
DIVS    <EA>,Dn
```

A division operation is performed by dividing the destination operand (a 32-bit data register) by the 16-bit source operand. The quotient is a 16-bit number stored in the lower word of the data register and the remainder is a 16-bit number stored in the upper word of the data register. If you attempt to divide by zero, a division by zero exception occurs. This exception uses exception vector #5.

Sign Extension

EXT (sign EXTend) converts a signed byte to a signed word (EXT.W) or a signed word to a signed long word (EXT.L). This is done by extending the sign bit of the byte (bit 7) or word (bit 15) through bits 8 to 15 (for a word operation) or bits 16 through 31 (for a long word operation). The number acted on must be in a data register.

EXT is useful when you've loaded a byte or word into a data register but you are about to use it in a word or long word operation. If you don't extend the operand first, the operation will not behave as expected because the high order part of the data register is not properly set up.

Here is an example of how to use EXT with the Macintosh's built-in number-to-string conversion instruction, _Pack7:

```
LEA    theString,A0        ;A0 = pointer to string space
MOVE   theNumber,D0        ;D0.L = number to convert
EXT.L  D0                  ;Adjust upper word
MOVE   #0,-(SP)
_Pack7                     ;Convert number to string
```

The _Pack7 instruction expects the number to be in the D0.L register. In this example, theNumber is a word quantity, so the upper word of D0 is undefined after the MOVE theNumber,D0 instruction. An EXT.L D0 instruction extends the word while maintaining its sign and magnitude.

Comparing

The CMP (CoMPare) instruction compares the value specified by the destination operand with the number specified by the source operand. The comparison is made by subtracting the source operand from the destination operand, setting the flags according to the result, but not storing the result. The Bcc instructions can then be used to transfer control to various parts of the program, based on the state of the condition code flags. See the section on Program Control Instructions for more information on how to use these branch instructions.

There is one special form of the compare instruction, CMPM, which is usually used as part of a subroutine that compares one block of memory to another. Both its operands use the (An)+ addressing mode only. Here is how you would use it to check whether two 128-byte areas of memory, initially pointed to by A0 and A1, are the same:

```
CompareMem  MOVE   #32-1,D0        ;32 long words to compare
@1          CMPM.L (A0)+,(A1)+     ;Compare the two areas
            DBNE   D0,@1           ;Loop until all done or
                                   ;until a mismatch.
            SNE    D0              ;Set according to result
            RTS
```

On exit, D0 = 0 if the two areas are the same; otherwise D0 = −1.

The DBNE loop can terminate in two ways: if the loop counter (D0) reaches −1 or if the condition associated with BNE becomes true. (The condition becomes true if the two long words being compared with CMPM are not the same.) Since the (An)+ addressing mode means the data block pointers are incremented by four after each comparison, the next comparison always checks the next long word in the block.

The SNE instruction places ones in every bit of D0 if the comparison fails (NE is true), or zeros if it succeeds.

Testing

The TST (TeST) instruction works just like a CMP #0,<EA> instruction. That is, the operand is compared with zero and the condition code flags are set according to the result. In particular, if the operand is zero, the zero flag is set to one and if the high-order bit (the sign bit) of the operand is one, the negative flag is set to one. Here is the code to use to check if the D0 register contains a zero:

```
TST.L   D0      ;Is D0 zero?
BEQ     ToZero  ;Branch if it is
```

The TAS (Test And Set) instruction is similar to the TST instruction, but it only works with byte operands. In addition, it always causes bit 7 of the byte at the effective address to be set to one. TAS is an **indivisible** operation, that is, one that cannot be interrupted. This means that if the bit is used as a busy flag to indicate that a certain data area, device, or other

resource is unavailable, no two co-processors will ever think the flag is not busy at the same time. It's not possible for one processor to interrupt another after the N flag has been cleared (to indicate that bit 7 is zero) but before the bit 7 flag is set to one. If it could, the second processor would think that the resource was available when it wasn't and chaos would ensue. Since the Macintosh is a single processor system, you should never need to use the TAS instruction.

Bit Manipulation Instructions

The bit manipulation instructions, BTST, BSET, BCLR, and BCHG act on single bits within an operand. If the destination operand is a data register, you can act on any bit from 0 to 31. For other operands, you can only act on bits 0 to 7. The bit number to be acted on is either stored in a data register or is an immediate quantity. Table 3-5 shows the bit manipulation instructions and describes their actions.

Table 3-5. The 68000 Bit Manipulation Instructions.

Instructions	BCHG (test a Bit and CHanGe) BCLR (test a Bit and CLeaR) BSET (test a Bit and SET) BTST (TeST a Bit)
Descriptions	BCHG tests a bit in the destination operand, sets the Z flag in the status register to reflect the result, then complements the bit in the destination operand whose number is given by the source operand. The bit number can be 0 to 31 if the destination operand is a data register, or 0 to 7 if it is not. BCLR tests a bit in the destination operand, sets the Z flag in the status register to reflect the result, then clears to zero the bit in the destination operand whose number is given by the source operand. The bit number can be 0 to 31 if the destination operand is a data register, or 0 to 7 if it is not. BSET tests a bit in the destination operand, sets the Z flag in the status register to reflect the result, then sets to 1 the bit in the destination operand whose number is given by the source operand. The bit number can be 0 to 31 if the destination operand is a data register, or 0 to 7 if it is not. BTST tests the bit in the destination operand whose number is given by the source operand and sets the Z flag in the status register to one if the bit is zero; if it isn't, the Z flag is cleared. The bit number can be 0 to 31 if the destination operand is a data register, or 0 to 7 if it is not.
Operand Size	.B, .L

Table 3-5. *continued*

Srce	Dest	Modes
xo	x	Dn
		An
	o	(An)
	o	(An)+
	o	−(An)
	o	d16(An)
	o	d8(An,Rn)
	o	Address
		d16(PC)
		d8(PC,Rn)
xo		#Immediate

For x operations, the operand size is always long. For o operations the operand size is always byte.

X	N	Z	V	C
-	-	*	-	-

Z=1 if the bit tested is 0; Z=0 otherwise.

BTST (Bit TeST) tests the state of a bit in the operand and sets the zero flag in the status register to one if the bit is zero; otherwise, it clears it to zero.

BSET (Bit SET) forces a bit in the operand to one. For example, suppose you want to set bit 2 of a long word operand to one. You could use either:

```
BSET.L  #2,<EA>
```

or:

```
MOVEQ.L #2,Dn           ;Dn = any data register
BSET.L  Dn,<EA>
```

BCLR (Bit CLeaR) works just like BSET except that it clears the bit to zero.

BCHG (Bit CHanGe) complements a given bit in the operand. If the bit is one, it is changed to zero, and vice versa.

Logical Instructions

There are four basic groups of logical instructions supported by the 68000: complement (NOT), logical and (AND), inclusive-or (OR), and exclusive-or (EOR). They are summarized in Table 3-6. There are also varieties of AND, OR, and EOR called ANDI, ORI, and EORI that can be used if the source operand is an immediate quantity.

Table 3-6. The 68000 Logical Instructions.

Instructions AND (AND logical)
OR (OR logical)

Descriptions AND combines two operands by clearing to zero all bits in the destination operand that correspond to 0 bits in the source operand. Bits corresponding to 1 bits in the source operand are not affected. The result is stored in the destination operand.

OR combines two operands by setting to one all bits in the destination operand that correspond to 1 bits in the source operand. Bits corresponding to 0 bits in the source operand are not affected. The result is stored in the destination operand.

Operand Size .B, .W, .L

Srce	Dest	Modes
xo	x	Dn
		An
x	o	(An)
x	o	(An)+
x	o	−(An)
x	o	d16(An)
x	o	d8(An,Rn)
x	o	Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if the most-significant bit of the result is 1; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.

Table 3-6. *continued*

Instruction EOR (Exclusive OR logical)

Description EOR combines two operands by complementing all bits in the destination operand that correspond to 1 bits in the source operand. All other bits are unaffected. The result is stored in the destination operand.

Operand Size .B, .W, .L

Srce	Dest	Modes
x	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if the most-significant bit of the result is 1; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.

Instructions ANDI (AND logical Immediate)
ORI (OR logical Immediate)
EORI (Exclusive OR Immediate)

Descriptions ANDI works just like AND, but the source operand is always an immediate number.

ORI works just like OR, but the source operand is always an immediate number.

EORI works just like EOR, but the source operand is always an immediate number.

Operand Size .B, .W, .L

Table 3-6. *continued*

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if the most-significant bit of the result is 1; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.

Note: There are also word forms of ANDI, ORI, and EORI that implicitly use the CCR (not privileged) or SR (privileged) as the destination operand: ANDI to CCR, ORI to CCR, EORI to CCR, ANDI to SR, ORI to SR, and EORI to SR. All condition code flags can change after these operations.

Instruction NOT (logical complement)

Description Calculates the one's complement of the destination operand and stores it in the destination location. The one's complement is calculated by converting all 1's to 0's and vice versa.

Operand Size .B, .W, .L

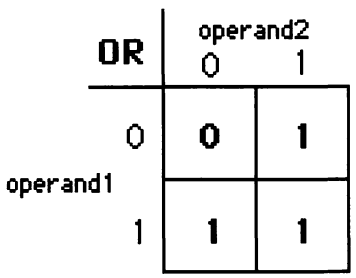
Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
-	*	*	0	0

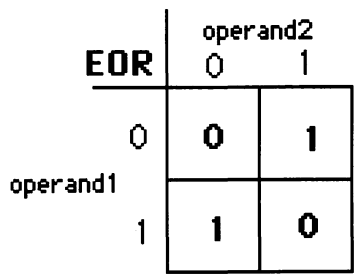
N=1 if the most-significant bit of the result is 1; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.

A logical operation involves the bit by bit combination of the source operand with the destination operand. The result of a particular bit combination (either 1 or 0) is dictated by the rules of Boolean algebra outlined in Figure 3-3, and is stored in the same bit of the destination operand. As you can see, the combination rules are different for OR, EOR, AND, and NOT operations.

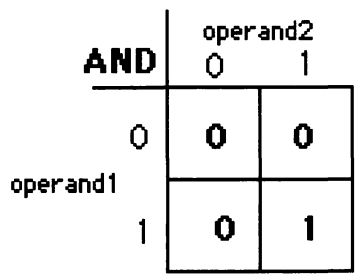
The NOT instruction involves only one operand. As a result of the operation, any one bits are converted to zero and zero



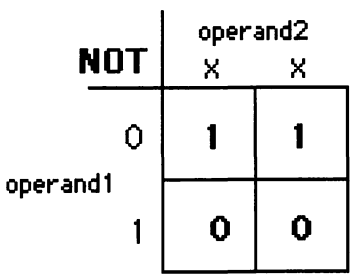
inclusive OR



exclusive OR



logical AND



logical NOT

(x = don't care)

Figure 3-3. Boolean Algebra Logic Tables.

bits to one bits. The operand is said to have been complemented.

The AND instruction clears certain bits of the destination operand to zero (if they are not already zero) and leaves others unaffected. The bits cleared are those that are zero in the source operand. For example, if D0 contains $00001232 and you execute an AND.L #$F0 instruction, D0 changes to $00001230.

If you'd rather set bits to one, use the OR instruction instead. This instruction forces every bit that is one in the source operand to one in the destination operand. All other bits are unaffected.

The EOR instruction complements those bits in the destination operand that correspond to one bits in the source operand. Any bit that is zero is changed to one and vice versa. EOR is useful for flipping bits used as software flags or switches, although you can also use the BCHG instruction for that.

Shift and Rotate Instructions

The 68000 has several instructions you can use to move an operand's data bits one or more positions to the left or right. (See Table 3-7.) The bit shift count is either an immediate quantity or is stored in a data register.

Table 3-7. The 68000 Shift and Rotate Instructions.

Instructions	ROL (ROtate Left) ROR (ROtate Right) ROXL (ROtate Left with eXtend) ROXR (ROtate Right with eXtend)
Descriptions	ROL shifts the bits in the operand to the left. Bits shifted out of the high-order bit are placed in the carry flag of the condition code register and into bit 0 of the destination operand. ROR shifts the bits in the operand to the right. Bits shifted out of bit 0 are placed in the carry flag of the condition code register and into the high-order bit of the destination operand. ROXL shifts the bits in the operand to the left. Bits shifted out of the high-order bit are placed in the carry and extend flags of the condition code register. The previous contents of the extend flag are shifted into bit 0 of the destination operand. ROXR shifts the bits in the operand to the right. Bits shifted out of bit 0 are placed in the carry and extend flags of the condition code register. The previous contents of the extend flag are shifted into the high-order bit of the destination operand.
Operand Sizes	.B, .W, .L

Table 3-7. *continued*

ROL, ROR, ROXL, and ROXR ***continued***

Srce	Dest	Modes
x	x	Dn
		An
	o	(An)
	o	(An)+
	o	−(An)
	o	d16(An)
	o	d8(An,Rn)
	o	Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

The shift count is 1 for the single-operand form of the instruction (o). In this case the operand size is always word.

X	N	Z	V	C
*	*	*	0	*

N = 1 if the most significant bit of the result is set; N = 0 otherwise.

Z = 1 if the result is zero; Z = 0 otherwise.

C = 1 if the last bit shifted out of the operand is 1; C = 0 otherwise. For ROL and ROR, C = 0 if the shift count is 0.

For ROXL and ROXR, C is set equal to the value of the X bit if the shift count is 0.

For ROL and ROR, X is unaffected. For ROXL and ROXR, X = 1 if the last bit shifted out of the operand is 1; it is unaffected if the shift count is 0.

Instructions

LSL (Logical Shift Left)
LSR (Logical Shift Right)
ASL (Arithmetic Shift Left)
ASR (Arithmetic Shift Right)

Descriptions

LSL shifts the bits in the operand to the left. Bits shifted out of the high-order bit are placed in the carry and extend flags of the condition code register and a 0 is placed in bit 0 of the destination operand.

LSR shifts the bits in the operand to the right. Bits shifted out of bit 0 are placed in the carry and extend flags of the condition code register and a 0 is placed in the high-order bit of the destination operand.

Table 3-7. *continued*

Descriptions ASL shifts the bits in the operand to the left. Bits shifted out of the high-order bit are placed in the carry and extend flags of the condition code register and a 0 is placed in bit 0 of the destination operand. If the high-order bit of the operand changes, the overflow flag is set.

ASR shifts the bits in the destination operand to the right. Bits shifted out of bit 0 are placed in the carry and extend flags of the condition code register. The high-order bit of the operand remains as it was before the shift.

Operand Size .B, .W, .L

Srce	Dest	Modes
x	x	Dn
		An
	o	(An)
	o	(An)+
	o	−(An)
	o	d16(An)
	o	d8(An,Rn)
	o	Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

The shift count is 1 for the single-operand form of the instruction (o). In this case the operand size is always word.

X	N	Z	V	C
*	*	*	*	*

N=1 if the most significant bit of the result is set; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.
C=1 if the last bit shifted out of the operand is 1; C=0 otherwise. C=0 if the shift count is 0.
X=1 if the last bit shifted out of the operand is 1; it is unaffected if the shift count is 0.
For LSL and LSR, V is always 0. For ASL and ASR, V=1 if the most-significant bit changes any time during the shift operation; V=0 otherwise.

Bit shifting operations have two main purposes. First, they can be used to multiply or divide operands by powers of two. This is because every shift right halves the value in the oper-

and and every shift left doubles it. Second, they can be used to transfer any eight consecutive bits in a register to the low order eight bits so that they can be dealt with by byte operations. The general form of the shifting and rotating instructions is shown in Figure 3-4.

Arithmetic Shift Instructions

Signed numbers can be shifted with the arithmetic shift instructions, ASR (Arithmetic Shift Right) and ASL (Arithmetic Shift Left). ASR shifts the bits one position to the right while preserving the status of the sign bit; the least-significant bit is moved into the carry and extend flags in the status register. ASL shifts bits to the left and clears bit 0 to zero but does not preserve the sign bit; it does, however, set the overflow flag if the sign bit of the number changes as a result of the shift.

Logical Shift Instructions

The logical shifts, LSR (Logical Shift Right) and LSL (Logical Shift Left), are similar to arithmetic shifts except the sign bit is not preserved (LSR) and the overflow flag is not affected (LSL). For LSR, a 0 bit is always moved into the most-significant bit. For LSL, a 0 bit is always moved into the least-significant bit. LSR and LSL should be used if you are working with unsigned numbers.

Rotate Instructions

The rotate instructions move bits through an operand in a circular path including only the bits in the operand (ROL and ROR), or the bits in the operand and the extend bit (ROXL and ROXR). As a bit passes through one end of the operand (or operand plus extend bit) it reappears at the other end. The carry flag is set according to the state of the bit shifted out of an operand. The extend flag is only affected by the ROXL and ROXR instructions where it is a member of the circle through which the bits are rotated.

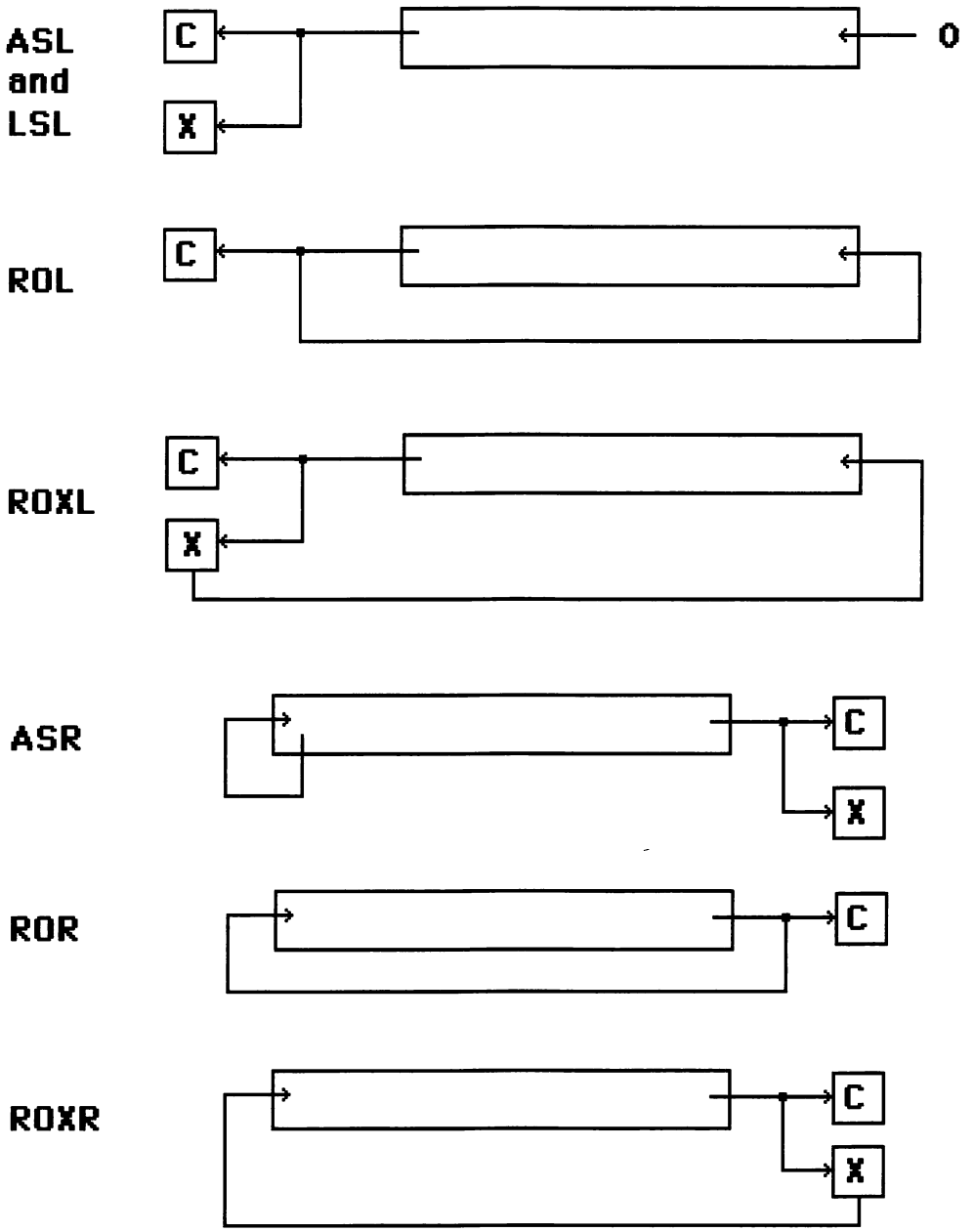


Figure 3-4. The 68000 Shift and Rotate Instructions.

System Control Instructions

The system control instructions are made up of a hodgepodge of rarely used instructions. (See Table 3-8.) They are usually used in low-level operating system programs only. Most of them are used to read from or write to the 68000 status register, and many are privileged instructions that can only be executed in supervisor mode.

Table 3-8. The 68000 System Control Instructions.

Instruction CHK (CHecK register against bounds)

Description Compares the signed number in the data register specified by the destination operand with the signed number in the source operand. If the number in the data register is less than zero or greater than the number in the source operand, exception #6 is generated.

Operand Size .W

Srce	Dest	Modes
x	x	Dn
		An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	U	U	U

N=1 if the the destination operand is negative. N=0 if the destination operand is greater than the source operand. If neither of these two conditions is true, N is undefined.

Table 3-8. *continued*

Instruction ILLEGAL (ILLEGAL instruction)

Description Causes exception #4 to occur.

Operand Size not applicable

No flags are affected.

Instruction RESET (RESET external devices) [privileged]

Description Causes all external devices to be reset to their startup states.

Operand Size not applicable

No flags are affected.

Instruction STOP (load status register and STOP) [privileged]

Description Loads the immediate value specified in the operand into the 68000 status register. Processing then stops until a hardware interrupt exception occurs that has a priority greater than the one set by the interrupt mask in the status register.

Operand Size not applicable

Srce	Dest	Modes
		Dn An (An) (An)+ −(An) d16(An) d8(An,Rn) Address d16(PC) d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
*	*	*	*	*

The flags are set according to the number stored in the immediate source operand.

Table 3-8. *continued*

Instruction TRAP (TRAP)

Description Causes an exception to occur. The number of the exception vector involved is 32 plus the number specified in the immediate operand (0 to 15).

Operand Size not applicable

Srce	Dest	Modes
		Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

No flags are affected.

Instruction TRAPV (TRAP on oVerflow)

Description Causes exception #7 to occur if the overflow flag in the condition code register is set to 1.

Operand Size not applicable

No flags are affected.

Status Register Control Instructions

The system control instructions you'll probably use most often are the ones that let you read from and write to the condition code register (recall that the CCR is low-order byte of the 16-bit status register). They are:

```
MOVE    SR,<EA>         ;Read entire status register
ANDI.B  #num,SR         ;AND CCR portion of SR
EORI.B  #num,SR         ;EOR CCR portion of SR
ORI.B   #num,SR         ;OR CCR portion of SR
MOVE    <EA>,CCR        ;(always byte size)
```

With these instructions you can set or clear any of the condition code flags in several different ways, as well as read the values of the flags. For example, you could set the carry flag of the CCR as follows:

```
ORI.B   #1,SR           ;Force bit 0 of SR (C) to 1
```

The .B extensions of the instruction mnemonics for the logical instructions are important. They indicate you are dealing with the lower half of the status register only—the condition code register.

There are also word forms of ANDI, EORI, and ORI that can be used to modify the entire status register, not just the condition code half. In addition, you can use MOVE <EA>,SR to put a certain number in the entire register. All of these are privileged instructions that you will probably never use. There are two other privileged instructions that deal with registers: MOVE USP,An and MOVE An,USP. They are used to read from or write to the user stack pointer.

Trap Instructions

There are three system control instructions that generate traps. Traps are exceptions caused by these instructions: TRAP, TRAPV, and CHK. The TRAP instruction is of the form

```
TRAP    #num
```

where num represents a trap number from 0 to 15. The exception vectors for these traps are #32 to #47, respectively. The TRAP instruction is a convenient way to pass control to a subroutine you've previously installed by placing its address in the appropriate exception vector. The Macintosh operating system does not use any of the TRAP vectors but the MDS MacsBug debugger uses TRAP #15 as a software breakpoint instruction.

The TRAPV instruction causes an exception only if the overflow flag in the CCR is 1. The exception vector used is

#7. You probably won't use this instruction very often as it's usually more convenient to handle an overflow condition by using a BVS instruction to direct an application to your own error handler.

The CHK instruction is of the form

```
CHK     <EA>,Dn          ;Dn = any data register
```

and it checks to see if the word in the data register is either negative or greater than the word stored at the effective address. If it is, an exception occurs that uses exception vector #8. You cannot use CHK with a byte or long word operand.

If you're writing a subroutine to handle exceptions, it must end with another system control instruction, RTE (ReTurn from Exception. This instruction pops the three words stored on the stack by the 68000 when an exception occurs and places them in the status register (one word) and the program counter (two words).

Processor Control Instructions

The RESET instruction causes a hardware reset condition to be sent to all peripheral ports connected to the 68000 and is usually used to force these ports to their power-on states. On the Macintosh, however, a RESET instruction causes the system to shut down and start up just as if the power had been turned on.

The STOP instruction loads its 16-bit immediate operand into the status register, then causes the 68000 to stop executing instructions until an interrupt occurs that has a priority higher than that stored in the status register's interrupt mask. When execution continues, it begins with the instruction following the STOP instruction. You will probably never have to use this instruction on the Macintosh.

Table 3-1. The 68000 Data Movement Instructions.

Instruction MOVE (MOVE data)

Description Moves the value at the source location to the destination location.

Operand Size .B, .W, .L

Srce	Dest	Modes
x	x	Dn
x		* An
x	x	(An)
x	x	(An)+
x	x	−(An)
x	x	d16(An)
x	x	d8(An,Rn)
x	x	Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

* A source operand of An is not permitted for byte-sized operations.

X	N	Z	V	C
-	*	*	0	0

N=1 if a negative number is moved; N=0 otherwise.
Z=1 if a zero is moved; Z=0 otherwise.

Instruction MOVE from CCR (MOVE data from the CCR)

Description Transfers the contents of the condition code register to the destination location.

Operand Size .W (only the low-order byte is significant)

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

The source operand is always CCR.

No flags are affected.

Table 3-1. *continued*

Instruction MOVE to CCR (MOVE data to the CCR)

Description Transfers the contents of the condition code register to the destination location.

Operand Size .W (only the low-order byte is significant)

Srce	Dest	Modes
x		Dn
		An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

The destination operand is always CCR.

X	N	Z	V	C
*	*	*	*	*

The flags are set in accordance with the number in the source operand.

Instruction MOVE to SR (MOVE data to the SR) [privileged]

Description Moves the value at the source location into the full 16-bit status register.

Operand Size .W

Srce	Dest	Modes
x		Dn
		An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

The destination operand is always SR.

X	N	Z	V	C
*	*	*	*	*

The flags are set in accordance with the number in the source operand.

Table 3-1. *continued*

Instruction MOVE from SR (MOVE data from the SR) [privileged]

Description Transfers the contents of the status register to the destination location.

Operand Size .W

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

The source operand is always SR.

Instruction MOVE USP (MOVE User Stack Pointer) [privileged]

Description Transfers the contents of the user stack pointer (A7') to the destination register, or vice-versa.

Operand Size .L

Srce	Dest	Modes
		Dn
x	o	An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

The source operand is always USP if moving to An. The destination operand is always USP if moving from An.

Table 3-1. *continued*

Instruction MOVEA (MOVE Address)

Description Moves the value at the source location to an address register. If the value is word-sized, it is first sign-extended to a 32-bit value.

Operand Size .W, .L

Srce	Dest	Modes
x		Dn
x	x	An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

No flags are affected.

Instruction MOVEM (MOVE Multiple registers)

Description Moves the contents of a group of registers to an area of memory, or vice versa. The order of transfer is always D0 through D7, then A0 through A7, unless -(An) addressing mode is used, in which case the standard order is reversed.

Operand Size .W, .L

Table 3-1. *continued*

MOVEM ***continued***

Srce	Dest	Modes
		Dn
		An
x	o	(An)
x		(An)+
	o	−(An)
x	o	d16(An)
x	o	d8(An,Rn)
x	o	Address
x		d16(PC)
x		d8(PC,Rn)
		#Immediate

No flags are affected.

The destination operand used with an operand marked with x is a register list. The source operand used with an operand marked with o is a register list. (See page 111 for a description of a register list.)

Instruction MOVEP (MOVE Peripheral data)

Description Moves bytes of data between a data register and alternate bytes of memory. The transfer begins with the highest-order byte and ends with the lowest-order byte.

Operand Size .W, .L

Srce	Dest	Modes
x	o	Dn
		An
		(An)
		(An)+
		−(An)
o	x	d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Table 3-1. *continued*

Instruction MOVEQ (MOVE Quick)

Description Moves an immediate quantity from −128 to −127 into a data register.

Operand Size .L

Srce	Dest	Modes
	x	Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if a negative number is moved; N=0 otherwise.
Z=1 if a zero is moved; Z=0 otherwise.

Instruction CLR (CLeaR an operand)

Description Moves a zero into the location specified by the operand.

Operand Size .B, .W, .L

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

An immediate source operand of #0 is implicit.

X	N	Z	V	C
-	0	1	0	0

Table 3-1. *continued*

Instruction EXG (EXchanGe registers)

Description Exchanges the contents of the source and destination registers. An entire register (all 32 bits) is acted on by this operation.

Operand Size .L

Srce	Dest	Modes
x	x	Dn
x	x	An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Instruction LEA (Load Effective Address)

Description Transfers the effective address of the source operand into the address register specified by the destination operand.

Operand Size .L

Srce	Dest	Modes
		Dn
	x	An
x		(An)
		(An)+
		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
		#Immediate

No flags are affected.

Table 3-1. *continued*

Instruction LINK (LINK and allocate)

Description Pushes the value of the address register in the operand on to the stack, transfers the new value of the stack pointer to the address register, then adds the immediate quantity specified by the destination operand (a sign-extended word) to the stack pointer. The immediate quantity must be negative to allocate stack space in the normal way.

Operand Size not applicable

Srce	Dest	Modes
		Dn
x		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
	x	#Immediate

No flags are affected.

Instruction PEA (Push Effective Address)

Description Decrements the stack pointer by four, then places the effective address of the operand at the location pointed to by SP.

Operand Size .L

Table 3-1. *continued*

PEA ***continued***

Srce	Dest	Modes
		Dn
		An
x		(An)
		(An)+
		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
		#Immediate

A destination addressing mode of −(SP) is implicit.

No flags are affected.

Instruction SWAP (SWAP register halves)

Description Exchanges the upper 16 bits of a data register with the lower 16 bits.

Operand Size .W

Srce	Dest	Modes
x		Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if bit 31 is set after the swap; N=0 otherwise.
Z=1 if the entire register is zero; Z=0 otherwise.

Table 3-1. *continued*

Instruction UNLK (UNLinK and deallocate)

Description Deallocates the stack space allocated with LINK by transferring the value in the address register operand to the stack pointer and then popping the long word on the stack into the address register.

Operand Size not applicable

Srce	Dest	Modes
x		Dn An (An) (An)+ −(An) d16(An) d8(An,Rn) Address d16(PC) d8(PC,Rn) #Immediate

No flags are affected.

Table 3-2. The 68000 Program Control Instructions.

Instruction Bcc (Branch conditionally) The cc stands for one of the two-character mnemonics shown in Table 3-3.

Description Causes program execution to continue at a position in the program relative to the program counter if the conditions associated with cc are true. The branch instructions are BCC (carry clear), BCS (carry set), BEQ (equal), BGE (greater or equal), BGT (greater than), BHI (higher), BLE (less or equal), BLS (lower or same), BLT (less than), BMI (minus), BNE (not equal), BPL (plus), BVC (overflow clear), and BVS (overflow set).

Operand Size .S (short branch) or .L (long branch) If the size is .S, the branching range is −128 to +127. If the size is .L, the range is −32768 to +32767.

Srce	Dest	Modes
		Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
x		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Instruction BRA (Branch Relative Always)

Description Causes program execution to continue at a position in the program relative to the program counter. The branch is always taken.

Operand Size .S (short branch) or .L (long branch) If the size is .S, the branching range is −128 to +127. If the size is .L, the range is −32768 to +32767.

Table 3-2. *continued*

BRA ***continued***

Srce	Dest	Modes
		Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
x		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Instruction BSR (Branch to SubRoutine)

Description Pushes the current value of the program counter on the stack (the address of the next instruction in the program), then causes program execution to continue at a position in the program relative to the program counter.

Operand Size .S (short branch) or .L (long branch) If the size is .S, the branching range is −128 to +127. If the size is .L, the range is −32768 to +32767.

Srce	Dest	Modes
		Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
x		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Table 3-2. *continued*

Instruction DBcc (Test condition, Decrement, and Branch)

Description This is a looping instruction that first tests to see if the condition referred to by Bcc is true; if it is, execution continues with the next in-line instruction. If not, the contents of the data register (always a word) are decremented and, if the result is not −1, control passes to the position in the program specified by the destination operand. If the result is −1, execution continues with the next in-line instruction. Any of the 16 conditions in Table 3-3 can be used with DBcc.

Operand Size .W

Srce	Dest	Modes
x		Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
	x	d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Instruction JMP (JuMP)

Description Causes program execution to continue at the address stored at the effective address specified in the operand.

Operand Size not applicable

Table 3-2. *continued*

JMP ***continued***

Srce	Dest	Modes
		Dn
		An
x		(An)
		(An)+
		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
		#Immediate

No flags are affected.

Instruction JSR (Jump to SubRoutine)

Description Pushes the current value of the program counter on the stack (the address of the next instruction in the program), then causes program execution to the effective address specified in the operand.

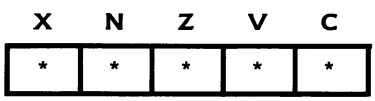
Operand Size not applicable

Srce	Dest	Modes
		Dn
		An
x		(An)
		(An)+
		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
		#Immediate

No flags are affected.

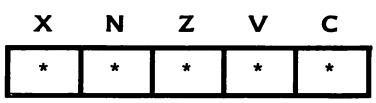
Table 3-2. *continued*

Instruction	NOP (No OPeration)
Description	Does nothing at all except kill some time and advance the program counter by one word.
Operand Size	Not applicable, there are no operands. No flags are affected.
Instruction	RTE (ReTurn from Exception) [privileged]
Description	Terminates execution of an exception-handling subroutine. It pops the status register and program counter from the system stack, and control resumes at the point where the exception occurred. (The SR and PC are automatically placed on the stack when the exception first occurs.)
Operand Size	Not applicable, there are no operands.



The flags are set according to the contents of the word popped from the stack.

Instruction	RTR (ReTurn and Restore condition codes)
Description	Pops the first word on the stack and places the low-order byte in the condition code register. It then pops the next long word on the stack into the program counter so that control resumes at the point where the subroutine was called.
Operand Size	Not applicable, there are no operands.



The flags are set according to the contents of the word popped from the stack.

Table 3-2. *continued*

Instruction RTS (ReTurn from Subroutine)

Description Pops a long word on the stack into the program counter, causing execution to continue at the address stored in the long word. RTS is used to terminate a subroutine called with JSR or BSR.

Operand Size Not applicable, there are no operands. No flags are affected.

Instruction Scc (Set conditionally)

Description Causes the byte operand to be set to all ones if the condition given by cc is true, or to all zeros if the condition is false. The conditions corresponding to cc are shown in Table 3-3.

Operand Size .B

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

No flags are affected.

Instruction TAS (Test and Set an operand)

Description Tests bit 7 of the byte operand, sets the N and Z flags according to the result, then sets the bit to one. This operation cannot be interrupted, therefore it's ideal for setting busy flags or other semaphores used in multitasking environments.

Operand Size .B

Table 3-2. *continued*

TAS ***continued***

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if bit 7 of the operand is 1; N=0 otherwise.
Z=1 if the operand is zero; Z=0 otherwise.

Instruction TST (Test an operand)

Description Compares the operand with zero and sets the condition code flags according to the result. The result itself is not saved.

Operand Size .B, .W, .L

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if the operand is negative; N=0 otherwise.
Z=1 if the operand is zero; Z=0 otherwise.

Table 3-4. The 68000 Arithmetic Instructions.

Instructions ABCD (Add Binary-Coded Decimal with extend)
SBCD (Subtract Binary-Coded Decimal with extend)

Descriptions ABCD adds the BCD number in the source operand and the extend bit to the BCD number in the destination operand, then stores the result in the destination operand.

SBCD subtracts the BCD number in the source operand and the extend bit from the BCD number in the destination operand, and stores the result in the destination operand.

Operand Size .B

Srce	Dest	Modes
x	x	Dn An (An) (An)+
o	o	−(An) d16(An) d8(An,Rn) Address d16(PC) d8(PC,Rn) #Immediate

X	N	Z	V	C
*	U	*	U	*

Z=1 if the result is zero; otherwise, Z is unchanged.
C=1 if a decimal carry (for ABCD) or borrow (for SBCD) was generated; C=0 otherwise.
X=1 if a decimal carry (for ABCD) or borrow (for SBCD) was generated; X=0 otherwise.

Instructions ADD (Add Binary)
SUB (Subtract Binary)

Descriptions ADD adds the binary number in the source operand to the binary number in the destination operand, then stores the result in the destination operand.

SUB subtracts the binary number in the source operand from the binary number in the destination operand, then stores the result in the destination operand.

Operand Size .B, .W, .L

Table 3-4. *continued*

ADD and SUB *continued*

Srce	Dest	Modes
xo	x	Dn
x		* An
x	o	(An)
x	o	(An)+
x	o	−(An)
x	o	d16(An)
x	o	d8(An,Rn)
x	o	Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

* A source operand of An is not permitted for byte-sized operations.

X	N	Z	V	C
*	*	*	*	*

N=1 if the result is negative; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.
V=1 if an overflow resulted; V=0 otherwise.
C=1 if a carry (ADD) or borrow (SUB) was generated; C=0 otherwise.
X=1 if a carry (ADD) or borrow (SUB) was generated; X=0 otherwise.

Instructions ADDA (Add Address)
SUBA (Subtract Address)

Descriptions ADDA adds the binary number in the source operand to the address in the destination operand, then stores the result in the destination operand.

SUBA subtracts the binary number in the source operand from the address in the destination operand, then stores the result in the destination operand.

Operand Size .W .L

Srce	Dest	Modes
x		Dn
x	x	An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

No flags are affected.

Unlike normal add and subtract instructions, ADDA and SUBA do not affect the status flags.

Table 3-4. *continued*

Instructions ADDI (Add Immediate)
SUBI (Subtract Immediate)

Descriptions ADDI adds the immediate binary number in the source operand to the binary number in the destination operand, then stores the result in the destination operand.

SUBI subtracts the immediate binary number in the source operand from the binary number in the destination operand, then stores the result in the destination operand.

Operand Size .B, .W, .L

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
*	*	*	*	*

N=1 if the result is negative; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.
V=1 if an overflow resulted; V=0 otherwise.
C=1 if a carry (for ADDI) or borrow (for SUBI) was generated; C=0 otherwise.
X=1 if a carry (for ADDI) or borrow (for SUBI) was generated; X=0 otherwise.

Instructions ADDQ (Add Quick)
SUBQ (Subtract Quick)

Descriptions ADDQ adds the immediate binary number in the source operand (a number from 1 to 8) to the binary number in the destination operand, then stores the result in the destination operand.

Table 3-4. *continued*

Description SUBQ subtracts the immediate binary number in the source operand (a number from 1 to 8) from the binary number in the destination operand, then stores the result in the destination operand.

Operand Size .B, .W, .L

Srce	Dest	Modes
	x	Dn
	x	* An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

* A destination operand of An is not permitted for byte-sized operations.

X	N	Z	V	C
*	*	*	*	*

N=1 if the result is negative; N=0 if the result is positive.
Z=1 if the result is zero; Z=0 if it isn't.
V=1 if an overflow resulted; V=0 if it didn't.
C=1 if a carry (ADDQ) or borrow (SUBQ) was generated; C=0 otherwise.
X=1 if a carry (ADDQ) or borrow (SUBQ) was generated; X=0 otherwise.

Instructions ADDX (Add eXtended)
SUBX (Subtract eXtended)

Descriptions ADDX adds the binary number in the source operand to the extend bit to the binary number in the destination operand, then stores the result in the destination operand.

SUBX subtracts the binary number in the source operand and the extend bit from the binary number in the destination operand, then stores the result in the destination operand.

Operand Size .B, .W, .L

Table 3-4. *continued*

ADDX and SUBX ***continued***

Srce	Dest	Modes
x	x	Dn
		* An
		(An)
		(An)+
o	o	−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
*	*	*	*	*

N=1 if the result is negative; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.
V=1 if an overflow resulted; V=0 otherwise.
C=1 if a carry (ADDX) or borrow (SUBX) was generated; C=0 otherwise.
X=1 if a carry (ADDX) or borrow (SUBX) was generated; X=0 otherwise.

Instruction CMP (Compare)

Description Subtracts the source operand from the destination operand, and sets the condition code flags according to the result. The result itself is not stored. The destination operand *must* be a data register.

Operand Size .B, .W, .L

Srce	Dest	Modes
x	x	Dn
x		* An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

* A source operand of An is not permitted for byte-sized operations.

X	N	Z	V	C
-	*	*	*	*

N=1 if the source operand is less than the destination operand; N=0 otherwise.
Z=1 if the source operand is the same as the destination operand; Z=0 otherwise.
V=1 if the subtraction caused an overflow; V=0 otherwise.
C=1 if the destination operand is less than the source operand; C=0 otherwise.

Table 3-4. *continued*

Instruction CMPA (Compare Address)

Description Subtracts the address in the source operand from the address in the destination operand, and sets the condition code flags according to the result. The result itself is not stored. If the word form of CMPA is used, the addresses are sign-extended before the comparison is made.

Operand Size .W .L

Srce	Dest	Modes
x		Dn
x	x	An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	*	*	*

N=1 if the source operand is less than the destination operand; N=0 otherwise.

Z=1 if the source operand is the same as the destination operand; Z=0 otherwise.

V=1 if the subtraction caused an overflow; V=0 otherwise.

C=1 if the destination operand is less than the source operand; C=0 otherwise.

Instruction CMPI (Compare Immediate)

Description Subtracts the immediate source operand from the destination operand, then sets the condition code flags according to the result. The result is not stored.

Operand Size .B, .W, .L

Table 3-4. *continued*

CMPI ***continued***

Srce	Dest	Modes
	x	Dn
		An
	x	(An)
	x	(An)+
	x	−(An)
	x	d16(An)
	x	d8(An,Rn)
	x	Address
		d16(PC)
		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	*	*	*

N=1 if the source operand is less than the destination operand; N=0 otherwise.

Z=1 if the source operand is the same as the destination operand; Z=0 otherwise.

V=1 if the subtraction caused an overflow; V=0 otherwise.

C=1 if the destination operand is less than the source operand; C=0 otherwise.

Instruction CMPM (Compare Memory)

Description Subtracts the source operand from the destination operand, and sets the condition code flags according to the result. The result is not stored. Only the (An)+ addressing mode can be used with this instruction, so it's ideal for comparing one area of memory with another.

Operand Size .B, .W, .L

Table 3-4. *continued*

CMPM ***continued***

Srce	Dest	Modes
x	x	Dn An (An) (An)+ −(An) d16(An) d8(An,Rn) Address d16(PC) d8(PC,Rn) #Immediate

X	N	Z	V	C
-	*	*	*	*

N=1 if the source operand is less than the destination operand; N=0 otherwise.

Z=1 if the source operand is the same as the destination operand; Z=0 otherwise.

V=1 if the subtraction caused an overflow; V=0 otherwise.

C=1 if the destination operand is less than the source operand; C=0 otherwise.

Instructions

DIVS (Signed DIVide)
DIVU (Unsigned DIVide)
MULS (Signed MULtiply)
MULU (Unsigned MULtiply)

Descriptions

DIVS divides the destination operand (a long word) by the source operand (a word) and stores the result in the destination operand. The low-order 16 bits of the result is the quotient and the upper 16 bits is the remainder. The division operation is performed using two's complement signed arithmetic.

DIVU divides the destination operand (a long word) by the source operand (a word) and stores the result in the destination operand. The low-order 16 bits of the result is the quotient and the upper 16 bits is the remainder. The division operation is performed using unsigned arithmetic.

MULS multiplies the destination operand (a word) by the source operand (a word) and stores the result (a long word) in the destination operand. The multiplication is performed using two's complement signed arithmetic.

Table 3-4. *continued*

Descriptions MULU multiplies the destination operand (a word) by the source operand (a word) and stores the result (a long word) in the destination operand. The multiplication operation is performed using unsigned arithmetic.

Operand Size .W

Srce	Dest	Modes
x	x	Dn
		An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
x		d16(PC)
x		d8(PC,Rn)
x		#Immediate

X	N	Z	V	C
-	*	*	*	0

N=1 if the dividend for DIVU or DIVS is negative; N=0 otherwise.
Z=1 if the dividend for DIVU or DIVS is zero; Z=0 otherwise.
For DIVU and DIVS, V=1 if a division overflow occurred; V=0 otherwise. For MULU and MULS, V=0.

Instruction EXT (sign EXTend)

Description Copies bit 7 (word form) or bit 15 (long word form) of a data register through bits 8–15 or bits 16–31 of the register.

Operand Size .W .L

Srce	Dest	Modes
	x	Dn
		An
		(An)
		(An)+
		−(An)
		d16(An)
		d8(An,Rn)
		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
-	*	*	0	0

N=1 if the result is negative; N=0 otherwise.
Z=1 if the result is zero; Z=0 otherwise.

Table 3-4. *continued*

Instructions NEG (NEGate)
NBCD (Negate Binary-Coded Decimal with extend)
NEGX (NEGate with eXtend)

Descriptions NEG subtracts the operand from zero and stores the result at the operand location. Standard binary arithmetic is used.

NBCD subtracts the operand and the extend bit from zero and stores the result at the operand location. BCD arithmetic is used.

NEGX subtracts the operand and the extend bit from zero and stores the result at the operand location. Standard binary arithmetic is used.

Operand Size .B, .W, .L

Srce	Dest	Modes
x		Dn
		An
x		(An)
x		(An)+
x		−(An)
x		d16(An)
x		d8(An,Rn)
x		Address
		d16(PC)
		d8(PC,Rn)
		#Immediate

X	N	Z	V	C
*	*	*	*	*

For NEG and NEGX, N=1 if the result is negative; N=0 otherwise. For NBCD, N is undefined.

For NEG and NEGX, Z=1 if the result is zero; Z=0 otherwise. For NBCD, Z=0 if the result is non-zero; otherwise, Z is unchanged.

For NEG and NEGX, V=1 if an overflow occurred; V=0 otherwise. For NBCD, V is undefined.

For NEG, C=0 if the result is 0; C=1 otherwise. For NEGX, C=1 if a borrow occurred; C=0 otherwise. For NBCD, C=1 if a decimal borrow occurred; C=0 otherwise.

The X flag is set the same as the C flag.

Chapter 4
Memory Management

In this chapter I'll begin by explaining how the Macintosh operating system uses the 16-megabyte address space supported by the 68000. This will include a discussion of the usage of the RAM space (128K or 512K for the Macintosh, 1M for the Macintosh Plus), the ROM space (64K or 128K), and the memory-mapped I/O space in the high end of memory.

When we finish the guided tour of the memory space, you'll see how a 68000 assembly language program makes use of different areas in the RAM memory space for the storage of the program itself and the data it uses. In so doing, you'll see how and where space for constants, variables, and other data structures used by the program can be allocated without interfering with the smooth operation of the system.

Macintosh Memory Map

Figure 4-1 shows the allocation of the 68000's 16M address space on a Macintosh. As you can see, much of the space is unused, so there's plenty of room for future RAM or ROM expansion. Global variables containing the addresses of the key areas in the memory space are also shown in Figure 4-1.

Let's traverse the memory space from bottom to top to see how it's used on the Macintosh.

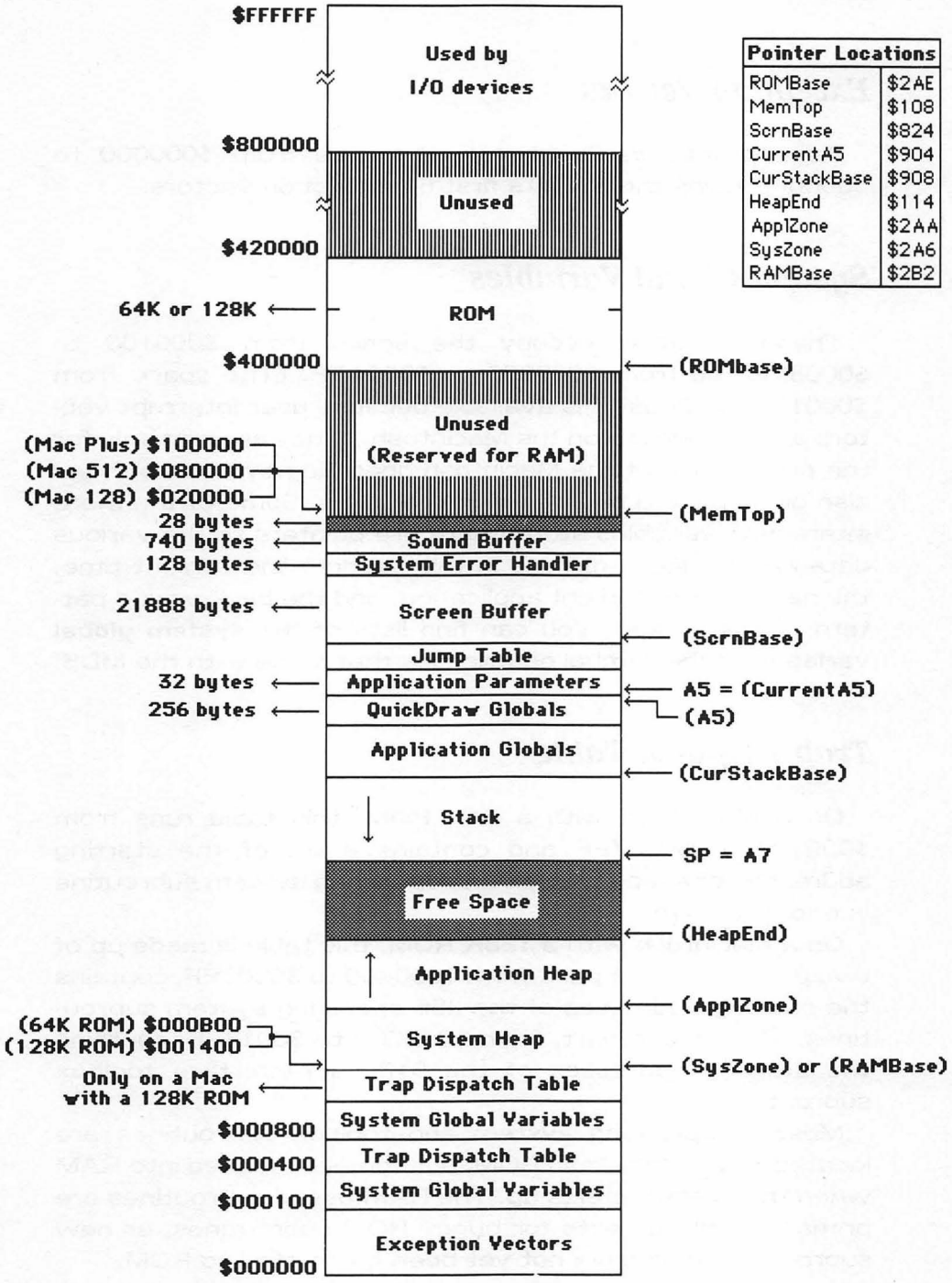


Figure 4-1. A Macintosh Memory Map.

Exception Vectors

As explained in Chapter 1, the area from $000000 to $0000FF holds the 68000's first 64 exception vectors.

System Global Variables

These variables occupy the space from $000100 to $0003FF and from $000800 to $000AFF. (The space from $000100 to $0003FF is available because user interrupt vectors are not needed on the Macintosh.) They are primarily for the private use of the Macintosh operating system, but can also be inspected by your own programs. Some of the more interesting variables stored here are pointers to the various data areas that I'll discuss in this section: the current time, the name of the current application, and the background pattern of the screen. You can find lists of the **system global variables** in the symbol equate files that come with the MDS.

Trap Dispatch Table

On a Macintosh with a 64K ROM, this table runs from $000400 to $0007FF and contains a list of the starting addresses of every toolbox and operating system subroutine in encoded form.

On a Macintosh with a 128K ROM, this table is made up of two parts. The first part, from $000400 to $0007FF, contains the starting addresses of the 256 operating system subroutines. The second part, from $000C00 to $0013FF, contains the starting addresses of the 512 user-interface toolbox subroutines.

Most of operating system and toolbox subroutines are located in the Macintosh ROM, but some are loaded into RAM when the system starts up. The RAM-based subroutines are primarily replacements for buggy ROM subroutines, or new subroutines that have not yet been committed to ROM.

System Heap

The **system heap** is an area reserved for the storage of the resources and data structures used by the operating system. The contents of the system heap are not removed when a new program is started up (or **launched**) from the Finder.

A pointer to the base of the system heap is stored in a system global variable called RAMBase. For a Macintosh with a 64K ROM, the heap base is $000B00; for a Macintosh with a 128K ROM, it is $001400.

Application Heap

The **application heap** is an area used by programs for storage of data, program constants, resources, and even the program itself. The application heap is initially 6K long, but expands to fill the free space above the heap as more space is requested by your program. The items in the application heap are released whenever a new application is launched.

Stack

This is the stack described in Chapter 1. Recall from the previous discussion that when data is added to it, the stack grows down in memory; this means it shares the same free space the application heap can move up into. If the top of the stack and the top of the heap ever collide, an out-of-memory bomb appears on the screen.

Application Global Space

This space is made up of global variables, application parameters, and a jump table. The size of this space is dictated by the specific application you are using; it is automatically reserved when the application is launched. The **global variables** are those defined in your program with the DS assembler directive, and also those used by QuickDraw (the

set of toolbox subroutines responsible for drawing images on the screen). Variables are placed just below the memory location whose address is stored in the A5 register, in reverse order of declaration; the first variable defined is stored in the highest part of the space.

The long word stored at the address in A5 is the first entry in the 32-byte **application parameter table**. It contains the address of the first QuickDraw global variable, normally given by the effective address of −4(A5). The only other active entry in the table, located at offset 16, is a handle to a Finder startup information record. (A **handle** is the address of a pointer to the record.) This record tells the application which documents it is to open or print as it starts up, if any.

The **jump table** is present only if the application is made up of more than one code segment. Calls from one segment to the other are made through the jump table.

Screen Buffer

The **screen buffer** is exactly 21,888 bytes long and begins at $1A700 for a 128K Macintosh or at $7A700 for the 512K model. If you do the mathematics, you'll find this corresponds to 175,104 bits, exactly the number in a grid measuring 512 wide by 342 high. Not surprisingly, these are the dimensions of the Macintosh screen display, in pixels. (**Pixel** stands for "picture element," a dot on the screen display.)

Each bit in the screen buffer controls the appearance of a different pixel on the screen. If the bit is one, its pixel is black; if it's zero, its pixel is white.

There is a linear relationship between the screen buffer and the pixels on the screen. To calculate the byte and bit number within the screen buffer that corresponds to a particular pixel, you first multiply the row number (0 to 341) by 512 (the screen width in pixels) and add the result to the column number (0 to 511). If you then divide this number by eight, the byte number you want is the dividend; subtract the remainder from seven to determine the relevant bit number within that byte.

System Error Handler Buffer

The **System Error Handler**, the part of the operating system that takes over when a fatal system error occurs, uses this area for data storage. It is the System Error Handler that displays the infamous bomb alert box.

Sound Buffer

The operating system uses this buffer to control the sound generated by the Macintosh's built-in speaker.

Expansion RAM

This is the space that is occupied as you add more RAM memory to your Macintosh.

ROM

The Macintosh ROM space begins at $400000 (4 megabytes) and is either 64K or 128K bytes long, depending on which version of the Macintosh you are using. This ROM contains the hundreds of fundamental subroutines making up the Macintosh operating system and the user interface toolbox. It is these subroutines that are accessed through the use of the line A emulator trap instructions ($Axxx) discussed in Chapter 1.

Memory-Mapped I/O Space

The upper eight megabytes of the 16M address space are reserved for control of Macintosh input/output devices even though only a few locations are actually used. There's really no RAM or ROM up here, it's just that the I/O devices are wired into the system in such a way that you can communicate with them by reading from or writing to certain loca-

tions. This method of handling I/O operations is called memory-mapped I/O.

The I/O locations provide support for the control of the following peripheral interfaces:

- Intel 8530 Serial Communications Controller (SCC).
- Synertek 6522 Versatile Interface Adapter (VIA).
- Integrated Woz Machine (IWM) disk controller.

You should never have to refer to I/O locations in your own programs because the Macintosh operating system includes the low-level drivers that perform most I/O operations you'd ever need.

Data Storage in the Application Heap

Now that you've seen how the various areas of memory are used on the Macintosh, it's time to discover where your own programs can store data safely. The operating system trap instructions we'll be looking at are summarized in Table 4-1.

Table 4-1. Memory Manager Trap Instructions.

Pointers	
_DisposPtr	**Releases a nonrelocatable block.**
`MOVE.L thePtr,A0`	`;A0.L = pointer to the block`
`_DisposPtr`	`;Error code returned in D0.W`
_NewPtr	**Allocates a nonrelocatable block.**
`MOVE.L #blockSize,D0`	`;D0.L = size of block in bytes`
`_NewPtr`	`;The pointer is returned in A0.L`
	`;D0.L contains the error code`
Handles	
_DisposHandle	**Releases a relocatable block.**
`MOVE.L thePtr,A0`	`;A0.L = pointer to the block`
`_DisposHandle`	`;Error code returned in D0.W`

Table 4-1. *continued*

Handles	
_HLock	**Locks a relocatable block in place.**
`MOVE.L theHandle,A0`	`;A0.L = handle to the block`
`_HLock`	`;Error code returned in D0.W`
_HUnlock	**Unlocks a relocatable block.**
`MOVE.L theHandle,A0`	`;A0.L = handle to the block`
`_HUnlock`	`;Error code returned in D0.Wt`
_NewHandle	**Allocates a relocatable block.**
`MOVE.L #blockSize,D0`	`;D0.L = size of block in bytes`
`_NewHandle`	`;The handle is returned in A0.L`
	`;D0.L contains the error code`

Error Reporting

Memory management error codes are returned in the D0.W register.

Here is a list of errors codes and their meanings:

Symbolic Name	*Value*	*Meaning*
NoErr	**0**	**No error occurred**
MemFullErr	**−108**	**No room for block**
NilHandleErr	**−109**	**Illegal operation on nil handle**
MemWZErr	**−111**	**Illegal operation on free block**
MemPurErr	**−112**	**Illegal operation on locked block**

There are four areas of RAM we'll be looking at:

- the application heap
- the stack
- the application global space
- the program storage area (within the heap)

Let's begin by considering the **application heap**. When a 68000 program is launched, it is loaded into an area of memory called the application heap, located in the low end of memory, just above the system heap used by the operating system. The exact position the program occupies in the heap is not important as long as the program is **relocatable** (capa-

ble of running at any position in memory). Because of the way the MDS assembler assembles code, it is difficult to write a program that is not relocatable.

When a new application is launched from the Finder, the application heap is cleared of all information to prevent uncontrolled growth. The system heap is not affected, however.

The application heap is simply a general-purpose data storage area. It is used to hold not only program code, but also resources used by the program: data buffers, and other data structures used either by the toolbox and operating system subroutines or directly by the program. It is obviously not a static structure; as items are added to it, it grows upward in memory toward the top of the stack, which grows downward.

There are four trap instructions you can use to dynamically allocate and deallocate blocks of data on the application heap. These blocks are referred to by either **pointers** or **handles**, depending on how they were first allocated.

Pointers

A **pointer** is a long word containing the address of a block of data in the heap. (See Figure 4-2.) This block is **nonrelocatable**: the operating system never tries to relocate it when it compacts the heap. (**Heap compaction** is the packing together of relocatable blocks in the heap; it is performed periodically to ensure there will be minimal dead space between blocks. To reserve a data block on the heap referenced by a pointer, use the _NewPtr trap instruction:

```
MOVE.L  #size,D0        ;D0 = size of block in bytes
_NewPtr
```

The pointer (a long word) is returned in the A0 register if no error occurred. An error code (a word) is returned in the D0 register; if it is zero, no error occurred. The only error that can occur for _NewPtr has the symbolic name MemFullErr (code -108), which means that there is not enough space in

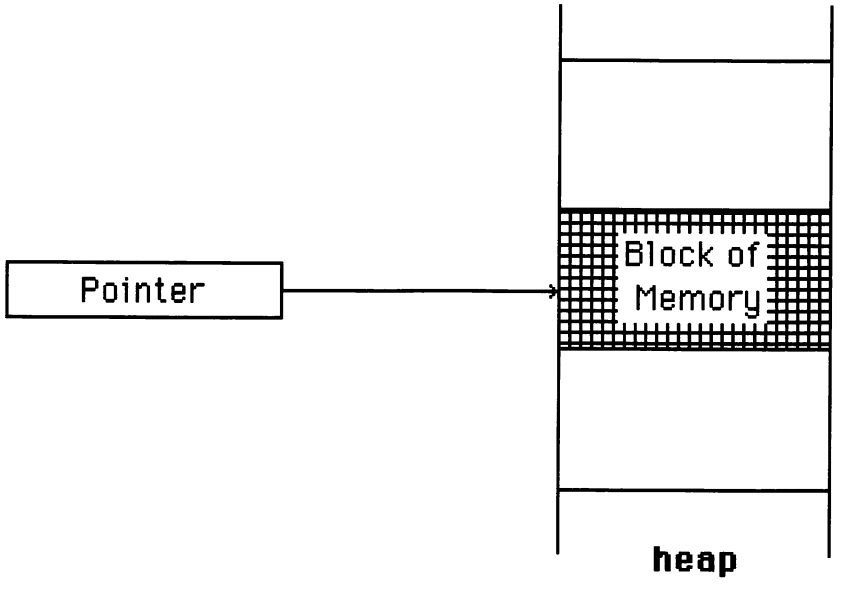


Figure 4-2. A Pointer to a Block of Memory.

the application heap for a nonrelocatable block of the size requested.

You can add the ",CLEAR" suffix to _NewPtr to zero the allocated block after allocating it. Another suffix, "SYS", lets you allocate space in the system heap rather than the application heap.

Handles

A **handle** is a long word containing the address, not of the block of data itself, but of the location of a **master pointer** that contains the current address of the block of data. (See Figure 4-3.) If necessary, the operating system may relocate the data block in memory but if it does, the handle to it remains valid (only the address stored in the master pointer changes). This means you can use your handles without having to worry about whether your data block has moved, at least until the data block is finally disposed of (see below). To

reserve heap space referenced by a handle, use the _NewHandle trap:

```
MOVE.L  #size,D0        ;D0 = size of block in bytes
_NewHandle
```

The handle (a long word) is returned in the A0 register. As with _NewPtr, an error code is returned as a word in D0.

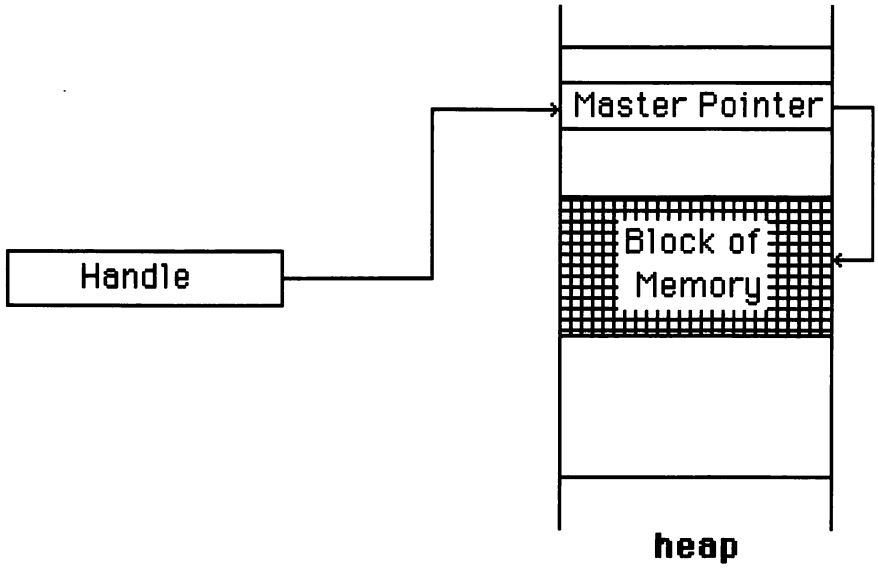


Figure 4-3. A Handle to a Block of Memory.

The choice of whether to reference a block of data by handle or by pointer is often dictated by the requirements of the toolbox trap instruction that uses the data block: Some instructions require you to pass pointers to data areas; others require handles.

If the space is for a data structure of your own design, it's easiest to deal with pointers because the elements of the data block can be easily accessed using the address register indirect with offset addressing mode, d16(An). You don't

have to worry about the block of data moving around in memory if you use a pointer. For example, if A0 contains the pointer to the data block, position 60 in the block would be accessed by using an operand of the form "60(A0)".

If, on the other hand, you have a handle to a block, you must "de-reference" the handle so you can access the block as you would if you were dealing with a pointer:

```
MOVE.L  MyHndl(A5),A0   ;MyHndl is a long word variable
MOVE.L  (A0),A0         ;Convert handle to pointer
```

Notice what the last instruction does: It takes the long word to which the handle points and puts it in A0. Since the handle points to a pointer to the data block, you can now use the A0 address register indirect addressing modes to access the block in the same way you would if you were dealing with a pointer directly.

But be careful! The de-referenced handle used to access the data in a block is valid only if the block has not been relocated since you did the de-referencing. Calling some trap instructions can result in block relocation, so you should lock the data block in place before doing so. Do this by calling the _HLock trap instruction:

```
        MOVE.L MyHndl(A5),A0    ;A0 = handle to block
        _HLock
MyHndl  DS.L    1               ;Handle stored here
```

When you've finished fiddling with the data block, you should unlock the data block with the _HUnlock trap to make it relocatable again:

```
MOVE.L  MyHndl(A5),A0   ;A0 = handle to block
_HUnlock
```

If you don't do this, you'll create islands of immovable data blocks in the heap space that can interfere with efficient

heap compaction; these may make it impossible to allocate a large space in the heap without running out of memory.

Deallocation

One of the nice things about allocating blocks in the heap is that you can easily deallocate them when you're through with them. Thus, you don't have to waste memory space for data areas you only use once. The two trap instructions for removing data blocks are _DisposPtr and _DisposHandle. The first is for blocks referenced by pointer and the other is for those referenced by handle. On entry to either trap, the A0 register contains the pointer or handle to the data block being disposed of. Here's an example using _DisposPtr:

```
MOVE.L  MyPtr(A5),A0    ;Load pointer into A0
_DisposPtr              ;Release the data block
```

It's always a good idea to dispose of unused data blocks because it frees up valuable memory space.

Allocation Tips

If you're dealing with both pointers and handles, you should try to allocate all your pointer areas first, if possible, to avoid excessive heap fragmentation. Suppose you don't and you allocate a handle area first, followed by some pointer areas. If you subsequently dispose of the handle area, you'll be left with a hole in the heap that can't be filled by compaction because the pointer areas can't be moved. If you have several holes like this, shielded by pointer areas, and they're not big enough to hold later-defined handle areas, the heap will soon become very large and you may run out of memory.

If the pointer areas are defined first, they cannot interfere with movement of handle areas during heap compaction.

Data Storage on the Stack

We saw in Chapter 3 that several 68000 instructions implicitly use the 68000 stack for data storage or retrieval, notably the JSR, BSR, and RTS instructions used in connection with subroutine calls.

You can also explicitly push data on the stack using the -(SP) addressing mode. You might want to do this, for example, to save the contents of a register that is temporarily required for something else. You can restore the original contents by later popping it from the stack using the (SP)+ addressing mode.

More commonly, you'll use the stack for passing parameters to the user interface toolbox subroutines. For example, to set the active drawing location to position (8,50) on the screen, you push the two coordinates on the stack, then call the _MoveTo trap instruction:

```
MOVE    #8,-(SP)        ;Horizontal coordinate
MOVE    #50,-(SP)       ;Vertical coordinate
_MoveTo                 ;Position the cursor
```

The subroutines to which trap instructions pass control are designed in such a way that you don't have to pop the data from the stack after the call; this is done for you automatically. If a result is returned on the stack, however, you will have to pop it off to prevent uncontrolled stack growth and to ensure that you'll return to the correct location when the next RTS or RTR instruction is encountered.

An example of a trap instruction that returns a result on the stack is _MenuSelect. (See Chapter 7.) Here's how to handle a call to this trap:

```
CLR.L   -(SP)           ;Clear space for result
MOVE.L  Point,-(SP)     ;Position of mouse
_MenuSelect
MOVE.L  (SP)+,D0        ;Pop the result
```

Since _MenuSelect returns a result on the stack, you have to clear a space for it before calling it. That's the purpose of the CLR.L -(SP) instruction. The instruction after _MenuSelect transfers the result (a long word) from the stack to the D0 register.

LINK and UNLK

Recall from Chapter 3 that a data area can also be allocated on the stack using the LINK and UNLK instructions. You must use these instructions for re-entrant or recursive subroutines. They may also be useful to you for allocating and deallocating temporary data structures so you don't have to deal with the more cumbersome _NewPtr and _DisposPtr (and _NewHandle and _DisposHandle) traps. It also means you don't have to allocate permanent storage space for the record using the DS or DC directives.

Suppose you want to use the Macintosh _GetFontInfo trap instruction to determine the dimensions of the active font. As you will see in Chapter 6, _GetFontInfo returns data in an 8-byte font information record. The height of a text line can then be calculated by adding the three integer fields in the record that are stored at offsets 0 (ascent), 2 (descent), and 6 (leading) bytes from the start of the record.

Here is a subroutine using LINK and UNLK that you can use to create space for the font information record on the stack and return the height in D0. It also shows how to access the parameters in the record using the d16(SP) addressing mode:

```
GetHeight LINK   A6,#-8          ;Push A6, create stack frame
          MOVE.L SP,-(SP)        ;Push pointer to stack frame
          _GetFontInfo           ;On exit, SP points to record
          MOVE   0(SP),D0        ;Height of text is
          ADD    2(SP),D0        ; ascent+descent+leading
          ADD    6(SP),D0
          UNLK   A6              ;Restore original SP, A6
          RTS
```

After allocation of space for the record with LINK (notice the size is negative, as required), SP points to the base address of the font information record in the stack area. Since _GetFontInfo requires a pointer to this record, SP is pushed on the stack with a MOVE.L SP,-(SP) instruction. When _GetFontInfo finishes, SP again points to the base of the record, and the fields are accessed with operands of 0(SP), 2(SP), and 6(SP). The UNLK A6 instruction at the end of the subroutine restores the stack to its state on entry.

Data Storage Within the Application Global Space

The traditional definition of a **variable** is a memory location (or locations) that holds data of byte, word, or long word size. This data can be altered at any time by your program to change its value. In the MDS environment, the word variable usually means a memory location allocated using the DS (Define Storage) assembler directive.

When a program is launched, its variables are set up in an application global space in the upper end of RAM memory, just below the screen buffer. (See Figure 4-4.)

The application global space not only holds your program's variables, but also variables used by QuickDraw screen drawing primitives (the group of subroutines in the Macintosh ROM responsible for managing the screen display), and parameters the Macintosh operating system uses to transfer control to other code segments. The 68000 stack is set up just below the variable space.

After launch, the A5 register points to the first long word past the end of the variables; this is the first entry in the application parameters table. Variables are stored between this address and the bottom of the stack, in reverse order of declaration.

The first variables in the space (at the upper end of the space) are usually the QuickDraw global variables, although

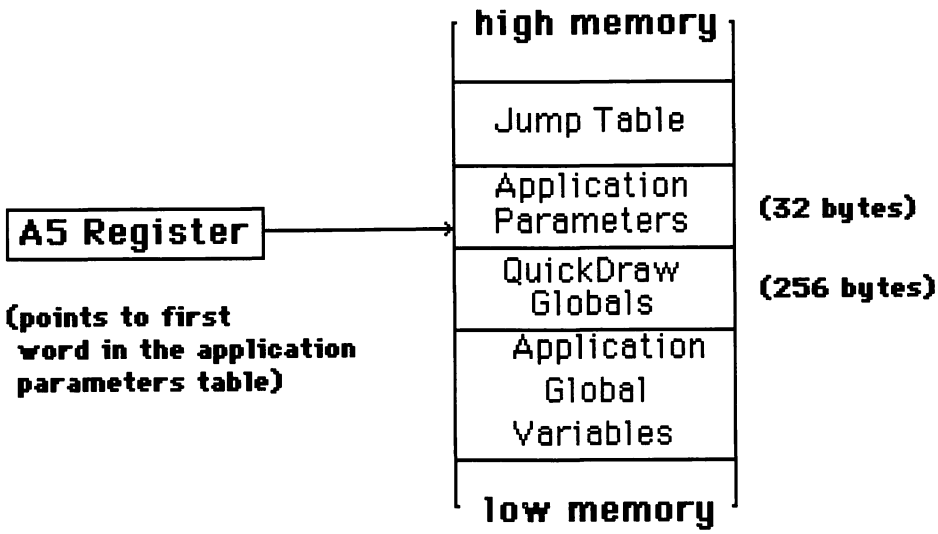


Figure 4-4. The Application Global Space.

an application can store them in the heap, if preferred. The usual space reserved for the QuickDraw global variable space by MDS is 256 bytes. (You can adjust the size of the space using a linker command called /GLOBALS, but you should rarely have to do this.)

Beneath the QuickDraw variables are the application's own variables. To access them (or the QuickDraw variables), you must use the A5 address register indirect with offset addressing mode, d16(A5). Fortunately, you do not have to know the absolute offsets to the variables in the space because they are calculated for you by the assembler; all you need do is specify the name of the variable in the operand. For example, if you have defined two long words called MyVariable as follows:

```
MyVariable  DS.L    2
```

and you want to store the contents of the D1 register in the first long word, you would use the instruction:

```
MOVE.L  D1,MyVariable(A5)
```

If you want to deal with the second long word, use:

```
MyVariable+4(A5)
```

as the destination operand.

A common error in assembly language programs is omitting the reference to the A5 register. It is required! You should also take care not to destroy the contents of the A5 register. If you do change it, you won't be able to access variables until you reload it with the address of the end of the variable space. This value is stored in a system global variable called CurrentA5.

The size of the application global space is not fixed. Rather, its size is controlled by the number of variables defined in your application. The advantage of this is that you can define as many variables as you want in your program, provided, of course, that you don't run out of RAM space.

Data Storage Within the Application Code

Data areas can be allocated within the application code itself using the DC and DCB assembler directives. These directives also place specific values in the spaces so allocated. Memory locations allocated with either of these two directives are called **constants** since their contents are not expected to change.

To store two constants, $BEAF (word) and $32 (long word), in a program, use:

```
MyWord  DC.W    $BEAF           ;Store BE AF
MyLWord DC.L    $32             ;Store 00 00 00 32
```

When constants are accessed by name, the MDS assembler forces you to use the program counter indirect with displacement addressing mode to ensure that your program will

be relocatable. If you want to read the constant MyLWord, you would use an instruction like:

```
MOVE.L  MyLWord(PC),D0
```

If you forget the (PC), the MDS assembler will supply it for you. Since the constants are located relative to the program counter, the program will be relocatable.

Constants are not supposed to be changed on the fly by a program, they're supposed to stay the same. Constants that are changed are really variables in disguise and should be defined as such using the DS assembler directive. Apple has even published warnings to programmers about writing to constants. The reason given is that it may cause your programs to be incompatible with future versions of the Macintosh, which may use hardware techniques to physically prevent you from writing to the code space. (Protecting the code space has merit; it's to prevent runaway programs from disturbing other programs in a multi-user environment.)

There is also a practical reason for not writing to a constant: It's simply awkward to do so from a programming point of view. This is because the program counter indirect addressing modes used to read constants cannot be used to write to them. Thus, an instruction like:

```
MOVE    #8,MyConstant(PC)       ;invalid instruction
```

is not permitted and will cause an error message during the assembly process. You can work around this limitation by loading the effective address of the constant into an address register (using LEA) and then using an address register indirect addressing mode:

```
LEA     MyConstant,A0
MOVE    #8,(A0)
```

but this is less efficient than defining MyConstant as a variable in the first place.

Chapter 5

Events and Input/Output Operations

There are two standard ways for a user to interact with a Macintosh program while it is running. The first is to enter commands from the keyboard, just as you would on any traditional personal computer. The second is to roll the mouse around the tabletop until its cursor appears above an "action" icon on the screen (this may be a rectangular push-button, a window's close box, or a square check box), and then click the mouse button to select the icon to perform the action associated with it. The Macintosh was the first personal computer to incorporate the mouse as a standard input device.

Variants of the basic click operation are **double-click** and **drag** operations. You double-click by quickly pressing and releasing the mouse button twice in succession. The maximum delay between the two clicks of a double-click can be set using the Control Panel desk accessory, and is stored in the system global variable DoubleTime. If the clicks are more widely separated, a program should consider them to be two separate clicks. You drag the mouse by moving it while the button is held down.

User-initiated activities such as a keystroke or a mouse click are just two of a group of 14 types of operations referred to as **events**. Each type of event on the Macintosh is summarized in Table 5-1. There is also another event, called a **null event**, that is reported only if no other event is pending.

Table 5-1. Macintosh Event Type Codes.

Symbolic Name for Event Type	*Value*	*Description*
NullEvt	0	No event occurred
MButDwnEvt	1	The mouse button was pressed
MButUpEvt	2	The mouse button was released
KeyDwnEvt	3	A character key was pressed
KeyUpEvt	4	A character key was released
AutoKeyEvt	5	A character key was auto-repeated
UpdatEvt	6	A window requires updating
DiskInsertEvt	7	A disk was inserted
ActivateEvt	8	A window was activated or deactivated
NetworkEvt	10	An AppleTalk network event occurred
IODrvrEvt	11	An I/O driver event occurred
App1Evt	12	An application-defined event
App2Evt	13	An application-defined event
App3Evt	14	An application-defined event
App4Evt	15	An application-defined event

An event usually represents a specific input/output (I/O) operation. It may, however, simply act as a reminder that a particular action, such as the redrawing of a window, must be performed. The classes of events a Macintosh application may have to respond to are as follows:

- Keyboard Events (key-down, key-up, auto-key)
- Mouse Events (button-down, button-up)
- Window Events (update, activate/deactivate)
- Disk-inserted Event
- AppleTalk network event
- I/O driver events

There are also four types of events you can simulate from within your application programs. They can relate to any occurrence you wish.

Several Macintosh operating system trap instructions are available to deal with events; they make up the part of the operating system called the **Event Manager**. In this chapter, I'll describe the most important of these instructions. I'll also

look at some other instructions relating to common input/output operations that aren't actually dealt with by the Event Manager: beeping the speaker and reading the time and date from the Macintosh's built-in clock/calendar. These instructions are summarized in Table 5-2.

Table 5-2. The Macintosh Event Manager and I/O Trap Instructions.

Trap Instructions

_Button — **Tests if the mouse button is down.**

```
CLR.B  -(SP)          ;BOOLEAN: space for result
_Button
MOVE.B (SP)+,D0       ;Result: true = button is down
                      ;        false = button is up
```

_Delay — **Does nothing for a fixed tick count.**

```
MOVE.L  #duration,A0  ;A0.L = Length of delay in ticks
_Delay                ;Result in D0.L = time on clock
                      ;                 after delay
```

_EventAvail — **Checks the event queue for the next event without removing the event from the queue.**

```
CLR.B   -(SP)         ;BOOLEAN: space for result
MOVE    #mask,-(SP)   ;INTEGER: the event mask
PEA     theEvent      ;VAR: an EventRecord
_EventAvail
MOVE.B  (SP)+,D0      ;Result: true = event occurred
                      ;        false = no event occurred
```

The size of an EventRecord is given by the constant EvtBlkSize. The structure of an event record is shown in Table 5-3.

Table 5-2. *continued*

Trap Instructions

_FlushEvents **Removes events from the event queue.**

```
MOVE.L #theMasks,D0     ;D0.L = event mask for events that
                        ;       will stop the search is in
                        ;       the upper word (stopMask).
                        ;       event mask for events that
                        ;       can be removed is in the
                        ;       the lower word (whichMask).
_FlushEvents
MOVE.W D0,stopEvent     ;D0.W = event type code that
                        ;       stopped the search
```

If stopMask is zero, the entire queue is examined.

_GetCursor **Loads a cursor record from a resource file.**

```
CLR.L  -(SP)            ;HANDLE: space for result
MOVE   #cursorID,-(SP)  ;INTEGER: cursor resource ID
_GetCursor
MOVE.L (SP)+,A0         ;Result: Handle to cursor record
```

_GetMouse **Gets the current mouse position.**

```
PEA     mouseLoc      ;VAR: a point. Local coordinates.
_GetMouse
```

_GetNextEvent **Checks the event queue for the next event and removes the event from the queue.**

```
CLR.B   -(SP)           ;BOOLEAN: space for result
MOVE    #mask,-(SP)     ;INTEGER: the event mask
PEA     theEvent        ;VAR: an EventRecord
_GetNextEvent
MOVE.B  (SP)+,D0        ;Result: true = event occurred
                        ;        false = no event occurred
```

Table 5-2. *continued*

Trap Instructions

_HideCursor — **Hides the current cursor.**

```
_HideCursor             ;No parameters
```

_InitCursor — **Sets the current cursor to the standard arrow and resets the cursor visibility level to zero.**

```
_InitCursor                 ;no parameters required
```

_IUDateString — **Gets a date string.**

```
MOVE.L  #seconds,-(SP)  ;LONGINT: seconds since Jan 1/1904
MOVE.B  #format,-(SP)   ;BYTE: 0 = short format
                        ;      1 = long format
                        ;      2 = abbreviated long format
PEA     theString       ;VAR: the returned date string
MOVE    #0,-(SP)        ;INTEGER: 0 = _IUDateString
_Pack6
```

_IUTimeString — **Gets a time string.**

```
MOVE.L  #seconds,-(SP)      ;LONGINT: seconds since Jan 1/1904
MOVE.B  #withSeconds,-(SP)  ;BOOLEAN: true  = include seconds
                            ;         false = no seconds
PEA     theString           ;VAR: the returned time string
MOVE    #2,-(SP)            ;INTEGER: 2 = _IUTimeString
_Pack6
```

_ObscureCursor — **Removes the cursor until the mouse is moved.**

```
_ObscureCursor              ;No parameters
```

_SetCursor — **Designates a new cursor as the current cursor.**

```
PEA   newCursor         ;VAR: a cursor record
_SetCursor
```

The size of a cursor record is given by the constant CursRec.

Table 5-2. *continued*

Trap Instructions	
_ShieldCursor	**Removes the cursor from the screen when it's within a certain rectangle.**

```
PEA    shieldRect           ;VAR: a rectangle
MOVE.L #globalOrigin,-(SP)  ;LONGINT: origin of coordinates
_ShieldCursor
```

_ShowCursor	**Displays a previously hidden cursor.**

```
_ShowCursor             ;No parameters
```

_StillDown	**Tests if the mouse button has been held down since the previous press, without removing any button-up event.**

```
CLR.B  -(SP)            ;BOOLEAN: space for result
_StillDown
MOVE.B (SP)+,D0         ;Result: true = button is still down
                        ;        false = button was released
```

_SysBeep	**Beeps the speaker.**

```
MOVE    #duration,-(SP) ;INTEGER: Length of beep in ticks
_SysBeep
```

_WaitMouseUp	**Tests if the mouse button has been held down since the previous press, and removes any button-up event.**

```
CLR.B  -(SP)            ;BOOLEAN: space for result
_WaitMouseUp
MOVE.B (SP)+,D0         ;Result: true = button is still down
                        ;        false = button was released
```

Table 5-2. *continued*

Trap Instructions		
System Variables		
Time	($20C)	The current time expressed in seconds since January 1, 1904. [long word]
Ticks	($16A)	The number of ticks since bootup. [long word]
SysEvtMask	($144)	The system event mask. [word]
DoubleTime	($2F0)	Maximum time (in ticks) between two clicks before they will be considered a double click. [long word]

The Event Manager

Generally speaking, when an event occurs on the Macintosh, a unique code representing the event is placed in an event queue maintained by the operating system. A typical application periodically checks this queue for the presence of an event and if it finds one, processes it. If the queue is empty, it keeps checking the queue until a recognizable event occurs. Certain types of events are not actually placed in the queue even though they are reported as if they were. These are the window update and activate/deactivate events. Only true I/O events are placed in the queue.

At the very beginning of an application you should ensure any stray events that are pending when the application is launched are removed (or **flushed**) from the queue. To do this, use the _FlushEvents instruction:

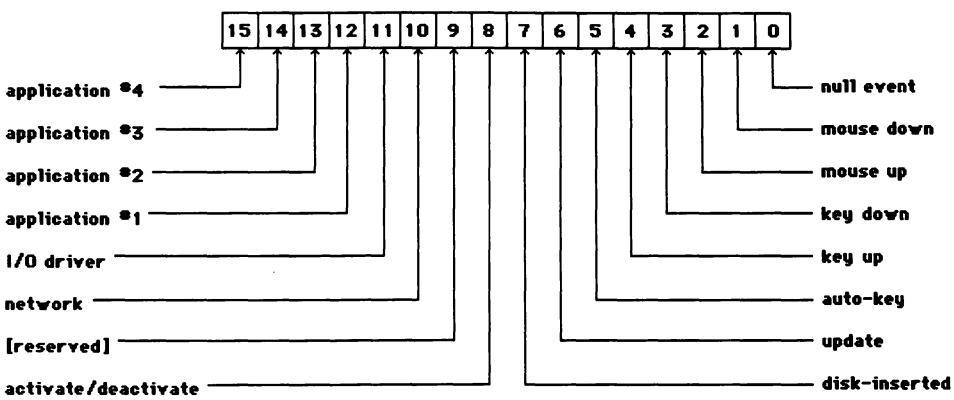
```
MOVE.L #$0000FFFF,D0    ;Flush all events
_FlushEvents
```

The low-order word of the long word stored in D0 (called whichMask) is an **event mask** that tells _FlushEvents which events to remove from the queue. As shown in Figure 5-1, each bit in an event mask corresponds to one of the 16 event

types that the Event Manager controls. To remove a particular type of event, simply set the appropriate bit to one. (Null events cannot be masked out, however.) In the example just given, the low-order word is $FFFF, which means "flush all types of events."

The high-order word in D0 (called stopMask) is also an event mask. It indicates how many events are to be removed from the queue. All events in the queue up to and including the first event of a type whose bit is set in the word are removed. The correspondence of bits to events is the same as for whichMask. If all events are to be flushed, the high-order word must be $0000.

When a Macintosh application first begins, *any* type of event that occurs will be posted in the event queue, even though your application may not be designed to respond to all events. If you want to restrict the types of events to be posted, store an appropriate system event mask in the system global variable SysEvtMask. Alternatively, your program can simply ignore any unsupported events that are fished out of the queue using the instructions described in the next section.



An event type is selected if its bit is 1; otherwise it is ignored.

Figure 5-1. The Format of an Event Mask.

Getting an Event

Once the event queue has been flushed and, optionally, a system event mask has been set up, your program can begin to get events from the queue and act on them. There are two instructions for doing this: _GetNextEvent and _EventAvail.

_GetNextEvent is the one you'll be using most often, so let's look at it first. Here's the type of subroutine you might call to get an event to deal with:

```
GetEvent CLR.B   -(SP)           ;Space for Boolean result
         MOVE    #$FFFF,-(SP)    ;Event mask (look for all)
         PEA     EventRecord     ;Address of event record
         _GetNextEvent
         TST.B   (SP)+           ;Pop and test Boolean result
         BEQ     GetEvent        ;Branch if no event
         RTS
EventRecord     DCB.B   EvtBlkSize,0  ;EvtBlkSize = 16
```

_GetNextEvent returns a Boolean result indicating whether a non-null event has been removed from the queue. If one has, the result is true (non-zero), otherwise it is false (zero). To make room for this result, you must clear space on the stack with a CLR.B -(SP) instruction. (Recall from Chapter 1 this actually decrements the stack pointer by two bytes, not one.)

The first parameter _GetNextEvent requires is an event mask, reflecting the types of events that may be retrieved from the queue. _GetNextEvent ignores any other types of events that may be in the queue. In the example, the event mask is $FFFF, meaning that all events are retrievable. The event mask is passed on the stack.

The second parameter passed on the stack is the address of a 16-byte data structure called an **event record**. The event record is where the results of the call to _GetNextEvent are stored. The size of the event record is given by the system constant EvtBlkSize so space for it can be reserved with a directive of the form DCB.B EvtBlkSize,0.

After calling _GetNextEvent, you can pop the Boolean result from the stack and test it with a TST.B (SP)+ instruction. If no event (other than a null event) is pending, the zero flag is set to one, and you can call _GetNextEvent once again by looping with a BEQ instruction. This simple loop involving _GetNextEvent is called an **event loop** and forms the backbone of most interactive programs.

Table 5-3. The Structure of an Event Record.

Description of Field	*Size (bytes)*	*Symbolic Offset name*
Event type code	2	evtNum
Event message	4	evtMessage
Time of event	4	evtTicks
Mouse coordinates	4	evtMouse
Event modifier (high)	1	evtMeta
Event modifier (low)	1	evtMBut

If a non-null event occurs, the event record is filled with information describing the event and the event code is removed from the queue. As shown in Table 5-3, the event record is made up of six fields, which you can access using the offset names indicated. For example, to read the event type code into D1, use the instruction:

```
MOVE  EventRecord+evtNum,D1  ;EventRecord defined with DC
```

The word stored in the evtNum field, sometimes called the **What** field, indicates the type of event that occurred. Each bit in evtNum has the same meaning as in an event mask. (See Figure 5-1.) The symbolic names for each of the 16 event type codes are shown in Table 5-1.

The long word stored in the evtTicks field (sometimes called **When**) is the time at which the event occurred, in units of ticks. (A tick is roughly one-sixtieth of a second.) The time

is measured from the time when the Macintosh was first turned on.

The evtMouse field (sometimes called **Where**) of the event record indicates the position of the mouse when the event occurred. The position is expressed in global coordinates where the coordinate origin is at the top-left corner of the screen. (See Chapter 6 for a description of coordinate systems.) The high-order word is the vertical position and the low-order word is the horizontal position.

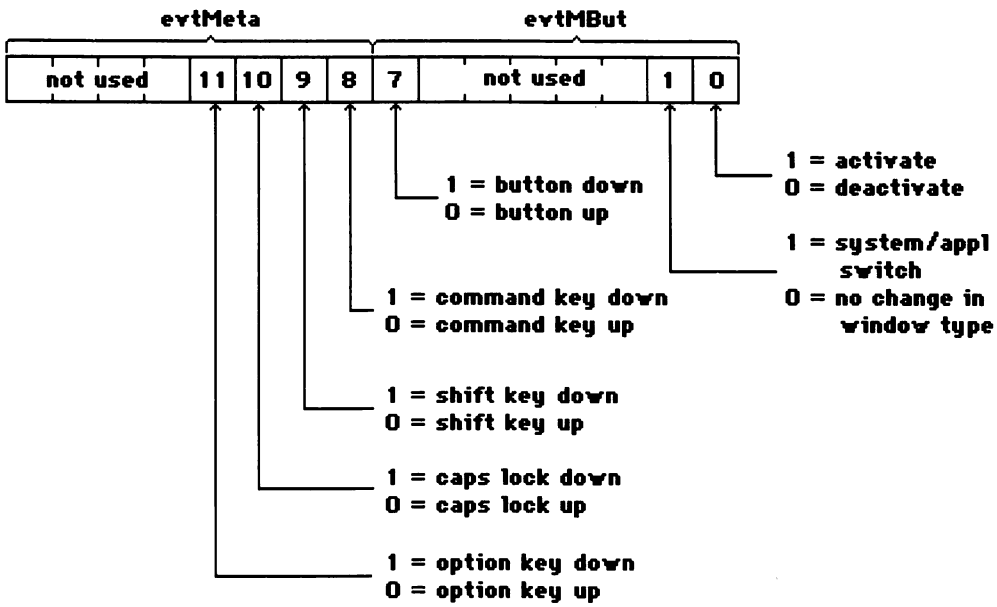
The evtMeta and evtMBut fields describe the status of the modifier keys and the mouse button when the event occurred. EvtMBut also contains bits used as flags to mark whether an activate or deactivate event occurred, and whether the newly activated window is of a different type than the previously active window. There are two types of windows: **application** and **system**. System windows are desk accessory windows. The meaning of each bit in the evtMeta and evtMBut bytes is shown in Figure 5-2.

The last field is evtMessage (a long word). I've left it to last because the information it holds depends on the type of event that is reported. The format of the evtMessage field for keyboard, window, and disk-inserted events is shown in Figure 5-3. EvtMessage fields for other events are either undefined (null and mouse events), for the private use of the operating system (network and I/O driver events), or defined by the application (application events).

The low-order word of the evtMessage field for a keyboard event reflects the key code and character code for the key (or combination of keys) that was pressed or released. I'll discuss the meaning of these codes later in this chapter.

For a window event, the evtMessage field contains a long word pointer to the window's data structure. I'll describe this data structure in Chapter 6.

For a disk-inserted event, the high-order word of the evtMessage field contains the result code generated by the operating system instruction that attempts to mount the disk. The low-order word contains the drive number (1 for



Symbolic names for the modifier bits:

Name	Value	Description
OptionKey	11	Option key (either one)
AlphaLock	10	Caps Lock key
ShiftKey	9	Shift key (either one)
CmdKey	8	Command key
BtnState	7	Mouse Button
ActiveFlag	0	Activate/Deactivate

Figure 5-2. The Format of the Modifiers Field of an Event Record.

internal, 2 for external). See *Inside Macintosh* for a description of the disk mounting trap instruction.

The second instruction you can use for inspecting the event queue is_EventAvail. This instruction works just like _GetNextEvent, except it does not remove the reported event from the queue. It is useful for checking whether a par-

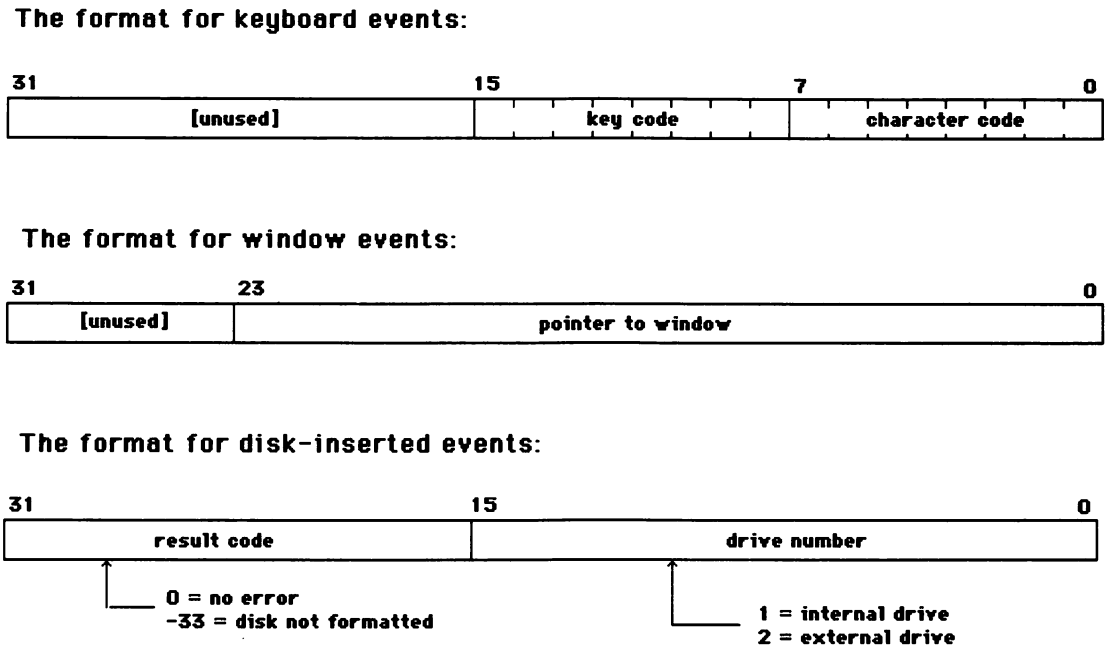


Figure 5-3. The Format of the evtMessage Field of an Event Record.

ticular event has happened without actually having to act on it right away.

Dealing With An Event

When a non-null event is returned by _GetNextEvent, it is up to your application to deal with it in an appropriate way. In this section, we'll explore some of the alternatives open to you for the common event types.

Every program dealing with events contains an event dispatcher subroutine that determines what event has occurred and calls the appropriate subroutine to handle it. The shell of a general-purpose event dispatcher is shown in Listing 5-1. You would call it whenever a call to _GetNextEvent indicates a non-null event has occurred.

Listing 5-1. The Asm Source File, Linker Control File, and RMaker Source File for the Dispatcher Program.

```
; Asm Source File
; Dispatcher.Asm
; This is an example of how to use an
; event dispatcher subroutine.

AppleID         EQU     1       ;Menu ID for Apple Menu
FileID          EQU     2       ;Menu ID for File Menu
WindID          EQU     128     ;Window ID

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QuickDraw globals
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; Create and draw a window on the screen:

        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file
        MOVE.L  (SP),WindowPtr(A5) ;Save ptr, but leave on stack
        _SetPort                ;Make window the active GrafPort
```

Listing 5-1. *continued*

```
; Create two standard menus:

        CLR.L   -(SP)           ;Space for handle
        MOVE    #AppleID,-(SP)  ;Menu ID number
        _GetRMenu               ;Get Menu from resource file
        MOVE.L  (SP)+,AppleH(A5);Save menu handle

        CLR.L   -(SP)           ;Space for handle
        MOVE    #FileID,-(SP)   ;Menu ID number
        _GetRMenu               ;Get menu from resource file
        MOVE.L  (SP)+,FileH(A5) ;Save menu handle

; Add menus to menu bar:

        MOVE.L  AppleH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        MOVE.L  FileH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        _DrawMenuBar            ;Display menu bar

MainLoop
        BSR     GetEvent
        BSR     HandleEvent
        BRA     MainLoop

GetEvent
        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1,-(SP)       ;Allow ALL events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result flag
        BEQ     GetEvent        ;Branch if no pending event
        RTS


* HandleEvent is the event dispatcher. It takes the event type
* code returned by _GetNextEvent and calls the subroutine that
* handles it. Access to the event handling subroutines is
* through a 16-entry jump table.
```

Listing 5-1. *continued*

```
HandleEvent

        MOVE    EventRecord+evtNum,D0
        ASL     #2,D0           ;Two shifts = times 4
        JMP     JumpTable(PC,D0);Jump to handler

JumpTable

        JMP     Ignore          ;Null event (never used)
        JMP     DoMouseDown     ;Button-down
        JMP     Ignore          ;Button-up
        JMP     DoKeyDown       ;Key-down
        JMP     Ignore          ;Key-up
        JMP     DoKeyDown       ;Auto-key
        JMP     DoUpdate        ;Update
        JMP     Ignore          ;Disk-inserted
        JMP     DoActivate      ;Activate
        JMP     Ignore
        JMP     Ignore
        JMP     Ignore
        JMP     Ignore
        JMP     Ignore
        JMP     Ignore
        JMP     Ignore

Ignore
        RTS

DoKeyDown
        RTS

DoUpdate
        RTS

DoActivate
        RTS

DoMouseDown

        CLR     -(SP)           ;Space for result
```

Listing 5-1. *continued*

```
        MOVE.L  EventRecord+evtMouse,-(SP)   ;Where
        PEA     WindowPtr(A5)
        _FindWindow             ;Where was button pressed?

        MOVE    (SP)+,D0        ;Get result
        CMP     #InMenuBar,D0   ;Pressed in menu bar?
        BEQ     QuitCheck       ;Yes, so check it out
        RTS                     ;Ignore everything else

; See if "QUIT" was selected from File menu:

QuitCheck

        MOVE.L  #0,-(SP)        ;result = menu/item selected
        PEA     EventRecord+evtMouse   ;Where
        _MenuSelect             ;Get menu selection
        MOVE    (SP)+,MenuNum(A5)   ;Save menu number
        MOVE    (SP)+,D0        ;Discard item number

        MOVE    #0,-(SP)
        _HiliteMenu             ;Remove highlight from menu title

        CMP     #FileID,MenuNum(A5)   ;In the FILE menu?
        BNE     GetEvent


* Must have selected QUIT command, so return to Finder by
* popping the subroutine return address before RTS. (We could
* also return just by executing a _ExitToShell instruction.)


        MOVE.L  (SP)+,D0        ;Pop the return address (long!)

        RTS                     ;Return to Finder

; Record for _GetNextEvent:

EventRecord     DCB.B   EvtBlkSize,0   ;Reserve space for record

; Here are the program globals. Use (A5) addressing.

AppleH          DS.L    1       ;Handle to Apple menu
FileH           DS.L    1       ;Handle to File menu
```

Listing 5-1. *continued*

```
WindowPtr       DS.L    1       ;Pointer to window

MenuNum         DS.W    1       ;Menu number selected
```

```
; Linker Control File
; Dispatcher.Link
;
; Link this file to create application
;  (without resources).
Dispatcher
$
```

```
* RMaker Source File
* Dispatcher.R
*
* Compile this after assembling and linking Dispatcher.Asm
*
* The next command appends the resources to the application:
!Dispatcher


Type MENU
,1                      ;;Resource ID
\14                     ;;Title is the Apple symbol (ASCII $14)
About this demo...      ;;About box

,2                      ;;Resource ID
File                    ;;Menu Title
  Quit                  ;;Only item is "Quit"

Type WIND
,128                    ;;Resource ID
Event Dispatcher Demo   ;;Title for Window
40 5 332 502            ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)
```

The HandleEvent subroutine in Listing 5-1 first loads D0 with the event type code from the evtNum field of the event record. It then multiplies the code by four (with two bit shifts to the left) to get the relative position within JumpTable of the jump to its event handler. This works because each JMP instruction in the table is four bytes long and the JMPs are in event type code order.

Finally, control passes to the handler with a jump instruction that uses the program counter indirect with index addressing mode. This technique is much more convenient (and elegant) than performing a series of CMP instructions to check for each event type separately.

Event dispatching is easy. It's writing the event handling subroutines referred to in the jump table that's difficult!

Keyboard Events

The three keyboard events are:

KeyDwnEvt (key-down)
KeyUpEvt (key-up)
AutoKeyEvt (auto-key)

The most important of these events is the **key-down event** that occurs when the user presses a **character key** on the keyboard. A character key is any key other than Caps Lock, Option, Shift, and Command. These keys are called **modifier keys**. Whether you deal with the key press may depend on whether the program is in a text insertion mode. If it is, the next step would be to display the entered character on the screen. I'll illustrate one method of displaying characters in the next chapter.

Most programs ignore **key-up events** because there is rarely a need to know when a key is released.

The **auto-key event** occurs when a character key begins to repeat after you've held it down for a short length of time. You can set the delay time with the Control Panel desk accessory. This event is usually treated in the same way as a standard key-down event.

Mouse Events

There are two mouse events dealt with by the Event Manager:

MButDwnEvt (mouse button down)
MButUpEvt (mouse button up)

As with key-up events, button-up events are usually ignored. A button-down event, however, indicates the user has clicked the mouse somewhere on the screen. When you detect a click, you should first determine what part of the screen is involved by using a trap instruction called _FindWindow. As you will see in the next chapter, _FindWindow returns a numeric code indicating whether the click occurred in the menu bar at the top of the screen, on the desktop, or in some part of an application or system window.

Clicks in the desktop are usually ignored, menu bar clicks are handled by the Menu Manager (see Chapter 7), and clicks in a window are handled by the Window Manager. (See Chapter 6.)

If the click is in a window, you should read the evtMessage field of the event record to get the pointer to the window involved. If it's not the pointer to the currently active window (use the _FrontWindow instruction to get its pointer if you haven't kept track of it), the appropriate step to take is to deactivate the currently active window and activate the new one by calling _SelectWindow. If the click is in the active window, the code returned by _FindWindow will indicate exactly what part of the window: the close box, the drag area, the grow box, the zoom box, or the content region. Suggestions for handling clicks in these areas are presented in Chapter 6.

Window Events

There are two types of window events: **update events** and **activate/deactivate events**. An update event, UpdatEvt, occurs whenever a portion of any window on the screen needs to be redrawn because it has just become exposed to view. Update events occur when you enlarge a window,

when you move aside an overlapping window, or a new window is created. A program must react to an update event by drawing the newly exposed portion of the screen.

An **activate event** occurs when a previously inactive window is to be made active. To handle it, bring the window to the front of the screen and redraw its scroll bars, if necessary.

A **deactivate event** usually occurs in conjunction with an activate event since no two windows can be active at the same time. To handle the event, mark the window as inactive by dimming its scroll bar and grow box area.

Activate and deactivate events both generate an ActivateEvt event type. To distinguish between activate and deactivate, you must check bit 0 of the evtMBut byte in the event record. If it is 1, you're dealing with an activate event; otherwise it's a deactivate event. Here are the instructions you would use to check the status of this bit:

```
MOVE.B  EventRecord+evtMBut,D0
BTST    #0,D0                   ;Is bit 0 set to 1?
BNE     MyActivate              ;Yes, so activate
```

In this example, MyActivate marks the start of the code that handles the activate event. The code after the BNE instruction would handle the deactivate event.

Disk-Inserted Events

A **disk-inserted event**, DiskInsertEvt, occurs when you place a disk into the internal or external disk drive. If you listen carefully, you'll hear the disk drive motor whir as soon as you do this. What is happening is that the lowest level of the Macintosh operating system detects the insertion and tries to mount the disk by reading some directory information into memory. It then posts the event in the event queue so you get a chance to deal with it further, if you wish. You may, for example, want to initialize a new disk as soon as it is inserted. I won't be describing the specifics of how to handle disk-inserted events in this book; refer to *Inside Macintosh* instead.

AppleTalk network events (NetworkEvt), I/O driver events (IODrvrEvt), and application-defined events (App1Evt, App2-Evt, App3Evt, App4Evt) are also not covered in this book.

Monitoring the Mouse Button

You can use three instructions to check the status of the mouse button without scanning the event queue with _GetNextEvent: _Button, _StillDown, and _WaitMouseUp. None of these require parameters, but all do return a Boolean result on the stack, so the calling sequence for each of them is of the form:

```
CLR.B  -(SP)          ;Space for result
_Button               ;(or _StillDown, _WaitMouseUp)
MOVE.B (SP)+,D0       ;Pop result from stack
```

The first instruction is _Button. It returns a true (non-zero) result if the mouse button is currently down; otherwise it returns false (zero).

_StillDown returns a true result if the mouse button is currently down and there is no pending button-up event in the event queue. If there is a pending button-up event, it is *not* removed from the queue. You will usually use _StillDown in situations where you want to identify when a drag operation has been completed.

The third instruction, _WaitMouseUp, works just like _StillDown except that it removes the pending button-up event from the event queue.

Keyboard Input

We saw earlier that whenever any of the three keyboard events occurs (key-down, key-up, or auto-key), the low-order word of the evtMessage field of _GetNextEvent's event record contains both the key code and the character code for the key involved. Each key on the keyboard gener-

ates an event when it is pressed, except the modifier keys: the Caps Lock key, the two Shift keys, the two Option keys (there is only one Option key on the Macintosh Plus keyboard), and the Command key. As you will see, a modifier key simply affects the character code generated when a character key is pressed.

To detect the pressing of a modifier key by itself, you have to use an instruction called _GetKeys. _GetKeys returns a bit map showing which keys on the keyboard are currently pressed. See *Inside Macintosh* for a description of this instruction.

Each physical key on the Macintosh keyboard and keypad is associated with a unique number, called a **key code**, between 0 and 255. There are two exceptions: The two Shift keys share the same key code, as do the two Option keys on the original Macintosh keyboard. Your applications will rarely have to deal with key codes.

The code you are usually more interested in is the **character code**. This is a number between 0 and 255 representing the alphanumeric symbol (a letter, number, or punctuation mark) associated with the keystroke. The character code is usually different if the same key is pressed while one or two modifier keys are held down.

For a given character font, the correspondence of character codes from 32 to 127 to printable symbols invariably follows the ASCII standard shown in Figure 5-4. It is the duty of the designer of the font to make the conventional symbol assignment, however.

The first 32 character codes, from 0 to 31, are called **control characters** since they have traditionally been used by computers to control various aspects of the output to a screen display, printer, or communications device. For example, the carriage return code ($0D) causes the print head of a printer to return to the left edge of the paper. The form feed code ($0C) causes the printer to skip to the top of the next page. A few of the control characters on the Macintosh actually correspond to special symbols: a cloverleaf, a check mark, a diamond, and an apple icon, for example. These symbols are not part of the ASCII standard.

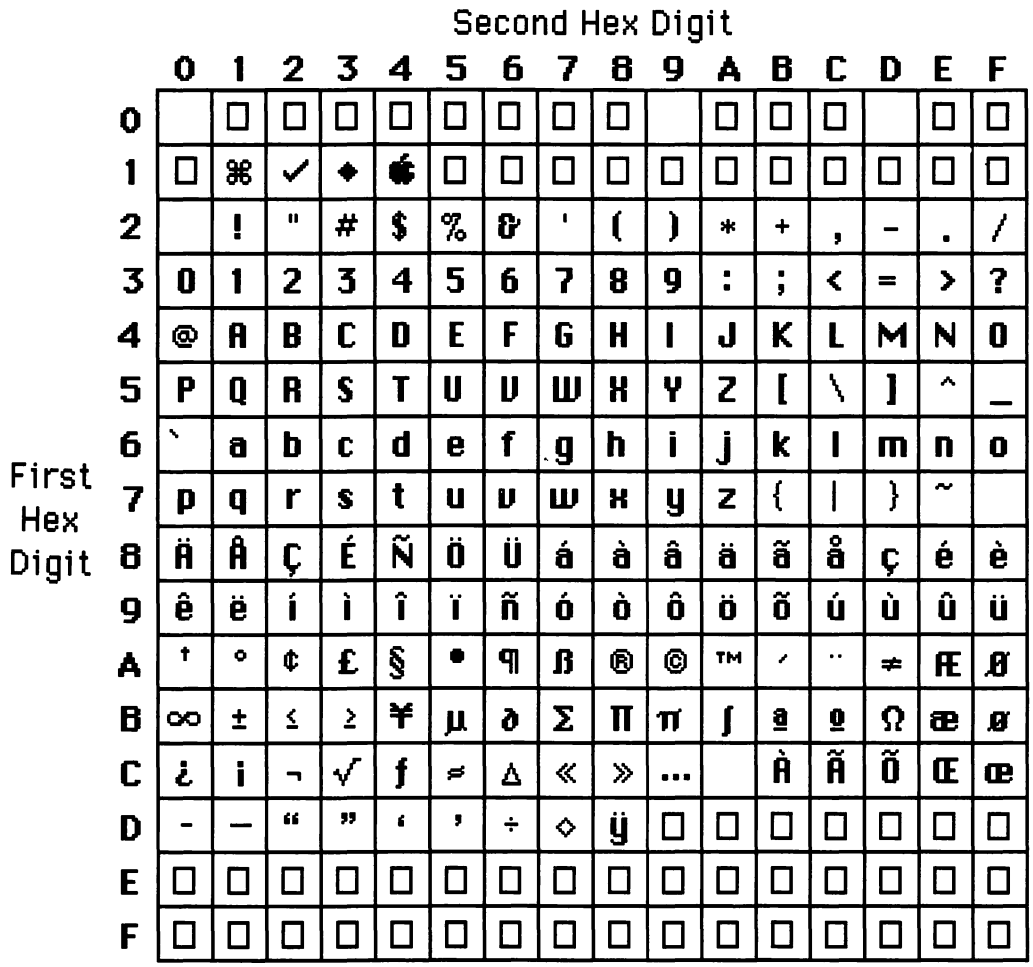


Figure 5-4. The ASCII Character Set. (Shown in the System Font.)

The symbol assignment of the highest 128 character codes, from 128 to 255, are not defined by the ASCII standard either. The symbols to which they correspond on the Macintosh are usually a mixed bag of foreign-language characters and special icons like Greek characters and copyright and trademark notices. Again, the assigned symbols depend on the particular font you're using.

To enter a certain character code from the keyboard, you must either press a key by itself or in combination with one or two modifier keys. The easiest way to determine what key combination corresponds to what character is to use the standard Key Caps desk accessory. If there is no symbol for a particular character code in the font, Key Caps displays a small rectangle over the key. The keys that generate control characters are the Return, Enter, Tab, and Backspace keys, and, if you are using the Macintosh Plus keyboard, the four arrow keys and the Clear key. Here are the codes these keys generate:

- The Return key generates code $0D.
- The Enter key generates code $03.
- The Tab key generates code $09.
- The Backspace key generates code $08.
- The Space Bar generates code $20.
- The Clear key generates code $1B.
- The Left Arrow key generates code $1C.
- The Right Arrow key generates code $1D.
- The Up Arrow key generates code $1E.
- The Down Arrow key generates code $1F.
- The = key on the keypad generates the same code as , on the main keyboard. Use its key code ($82) to distinguish it from the comma. The key code for comma is $B2.

The mapping of keystroke combinations to character codes is completely arbitrary and is handled by two INIT resources (having ID codes 1 and 2) in the System program. These resources contain assembly language subroutines that take a key code and a modifier key status byte as input and return the corresponding character code. By changing these resources, you can easily redefine the keyboard layout to whatever suits you—perhaps a Dvorak arrangement. Of course, you'll probably want to mark each key with its new symbol if you do this.

One of the common things you'll do in a program is check for the entry of particular character codes from the keyboard. Here is the type of subroutine you'd call after

detecting a key-down event to see if a certain character, say 'Y' or 'y', was pressed:

```
CheckKey  MOVE.L EventRecord+evtMessage,D0  ;Put key/char code in D0
          CMP.B  #'Y',D0                    ;Was 'Y' pressed?
          BEQ    YesPress                   ;Yes, so branch
          CMP.B  #'y',D0                    ;Was 'y' pressed?
          BEQ    YesPress                   ;Yes, so branch
          RTS
YesPress  NOP                               ;Insert handler here
          RTS
```

Notice that the evtMessage long word placed in D0 is compared with 'y' and 'Y' using the byte form (.B) of CMP. This is done to isolate the portion of the evtMessage field that contains the character code.

The Mouse Position and Cursors

Strangely enough, the most common input activity on the Macintosh, moving the mouse, does not generate a reported event. Instead, the lowest level of the operating system automatically monitors the position of the mouse and updates the position of its cursor (also called a pointer) on the screen.

If you want to determine the position of the mouse cursor, use the _GetMouse instruction. _GetMouse requires only one parameter on the stack, the address of a long word data area where the result (a coordinate point) will be stored:

```
          PEA     mouseLoc      ;Push addr of mouseLoc
          _GetMouse
mouseLoc  DC.W  0               ;Vertical position
          DC.W  0               ;Horizontal position
```

Notice that instead of defining mouseLoc with a DC.L directive, I've used two DC.W directives to emphasize the fact

that the coordinate is made up of a vertical and horizontal position.

The mouse position returned by _GetMouse is expressed in the local coordinate system of the currently active window, not the global coordinate system used by _GetNextEvent to store a point in the evtMouse field of an event record. You'll learn about coordinate systems in Chapter 6.

You can also control the appearance of the cursor, including whether it should be displayed or not. This is convenient because you'll probably want to display a different cursor in areas of the screen used for different purposes. For example, you might use a standard arrow cursor when the cursor is outside an active window, but an I-beam cursor when it's inside the window. The appearance of the I-beam serves as a reminder that text can be entered if the mouse is clicked.

Before we look at the instructions that affect the cursor, let's look at the data structure that defines a cursor. As shown in Table 5-4, a **cursor record** begins with two groups of 32 bytes each. Each group defines a 16x16 bit image that corresponds to a 16x16 square of screen pixels. The first word corresponds to the first row of the screen image, the second to the second row, and so on. In addition, the left-most bit in a word (bit 15) corresponds to the left-most position in the screen image. A one bit in a word means that the corresponding pixel is black. The symbolic name for the size of a cursor record is CursRec. The size of the cursor record is given by the constant CursRec.

Table 5-4. The Structure of a Cursor Record.

Description of Field	*Size (bytes)*	*Symbolic Offset name*
Data defining cursor	32	data
Data defining cursor mask	32	mask
Active cursor position	4	hotSpot

The first 32 bytes are the data that define the shape of the cursor on the screen. The second 32 bytes define a mask that combines with this data to form the image that actually appears on the screen. Only those pixels in the cursor's image corresponding to black pixels in the mask are copied to the screen. By defining a mask the same shape as the cursor's bit image, but completely black, and surrounding it with a fringe one pixel wide, you ensure that the cursor will be visible against a black background, because the fringe area will be white.

Following the two bit images in the cursor record is a **point** (two words defining a vertical and horizontal position) called the hotSpot. This is the position within the cursor image that appears on the screen at the current mouse position. The coordinates of the hotSpot are measured relative to the (0,0) position in the top left corner of the cursor's bit image. Just like the Macintosh screen, horizontal coordinates increase to the right and vertical coordinates increase to the bottom.

The standard cursor used on the Macintosh is the arrow. Four other cursor definitions are stored in the resource file that forms a part of the Macintosh operating system and is always available for use by an application. Their resource IDs are:

```
IBeamCursor  = 1 (an I-beam)
CrossCursor  = 2 (a "thin" plus sign)
PlusCursor   = 3 (a "thick" plus sign)
WatchCursor  = 4 (a wristwatch)
```

The resource type for a cursor resource is CURS. The data for the resource is simply one cursor record.

The Cursor Instructions

Now that you've seen how cursors are constructed, let's see how to use them. When you first begin a program you'll probably want to call _InitCursor. (It has no parameters.) This instruction makes the standard arrow cursor the current cursor and makes it visible. Thereafter, you can make

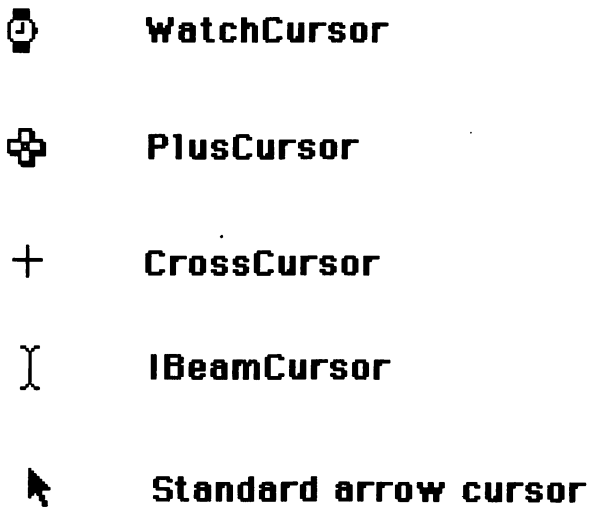


Figure 5-5. The Standard Macintosh Cursors.

any other cursor active by passing the address of the cursor record to _SetCursor as follows:

```
PEA  CursorRecord  ;Pointer to cursor record
_SetCursor
CursorRecord DCB.B  CursRec,0   ;CursRec = 68
```

where `CursorRecord` is a label for the start of the cursor record in the program constant area. (If you put it in the variable area, use CursorRecord(A5)). If you have allocated space for the cursor record on the heap, push the pointer to it instead.

You can load a cursor record from a resource file by passing the CURS resource ID to _GetCursor as follows:

```
        CLR.L  -(SP)              ;Room for handle
        MOVE   #IBeamCursor,-(SP) ;Resource ID
        _GetCursor
        MOVE.L (SP)+,IBeamH(A5)   ;Pop and save handle

IBeamH  DS.L   1                  ;Handle to cursor record
```

_GetCursor allocates space for the cursor record in the heap and returns a handle to it. If the handle is zero, the cursor record could not be found.

To make this cursor active, de-reference the handle returned by _GetCursor and pass the result (the address of the cursor record) to _SetCursor like this:

```
MOVE.L  IBeamH(A5),A0   ;Get handle in A0
MOVE.L  (A0),-(SP)      ;Put address in A0 (ptr) on stack
_SetCursor
```

The standard arrow cursor does not reside in a resource file, so the easiest way to make it active is to call _InitCursor. The cursor record for the arrow cursor is stored in the QuickDraw global variable area, at the offset given by Arrow. I'll discuss this global area in the next chapter.

Cursor Visibility

There are four cursor instructions that affect the visibility of the cursor on the screen. The first instruction, _HideCursor, removes the cursor from the screen by decrementing an internal counter, called the **cursor level**, by one. The cursor will only appear on the screen if the cursor level is zero, its initial value. To add one to the cursor level, use _ShowCursor. This instruction makes the cursor visible if _HideCursor has only been called once.

The _ObscureCursor instruction removes the cursor from the screen temporarily. It reappears the next time the mouse is moved. You might want to use _ObscureCursor to avoid having the mouse cursor interfere with a text entry cursor, for example.

The last of the four visibility instructions is _ShieldCursor and is used to remove the cursor from the screen if it falls inside a given rectangle on the screen. _ShieldCursor is the only instruction of the four requiring parameters: the address of the data structure containing the top, left, bottom, and right points of the rectangle, and a point that contains the origin of the coordi-

nate system for the rectangle expressed in global coordinates. Here's what the call to _ShieldCursor looks like:

```
            PEA     Rectangle
            MOVE.L  #$00230015,-(SP)  ;Point=(21,35)
            _ShieldCursor

Rectangle   DC.W    10,10,100,200     ;TLBR
```

Notice that the first half of the long word containing the point represents the vertical coordinate; the second half contains the horizontal coordinate. This is the reverse of the standard (h,v) order used by mathematicians.

The data structures for a rectangle and a point will be described in greater detail in the next chapter.

The Speaker

The Macintosh toolbox contains a small group of instructions making up the Sound Driver. These instructions can be used to generate simple harmonic tones or complex sound effects on the Macintosh.

The only speaker-related toolbox instruction I'm going to cover, however, is the _SysBeep instruction. If you want to make beautiful music on the Macintosh, refer to the "Sound Driver" chapter of *Inside Macintosh*.

The _SysBeep instruction, as you might guess, beeps the speaker for a fixed length of time. Here is how to use it:

```
MOVE    #34,-(SP)       ;Duration (in ticks)
_SysBeep                ;Beep the speaker
```

The word pushed on the stack before calling _SysBeep is the duration of the beep, in ticks. The beep sound gradually decays from loud to soft when you call _SysBeep.

You can control the volume of the sound using the Control Panel desk accessory. If the sound is turned off completely, the menu bar blinks once instead.

The System Clock

The Macintosh has a built-in, battery-operated clock that maintains the current date and time of day. With it you can calculate time increments as fine as one-sixtieth of a second.

There are several instructions in the toolbox that access the system clock. These are the ones that will be most useful to you: _Delay (delay a length of time), _IUTimeString (read the time), and _IUDateString (read the date). We'll also look at two global system variables that reflect the current time and date: Ticks and Time.

A **delay loop** is a portion of code used to kill time between two operations. Such loops are commonly used in animation programs to fix the film speed and in music generation programs to fix the frequency of the sound. The toolbox _Delay instruction can be called for these purposes:

```
            MOVE.L  Duration,A0     ;A0 = Length of delay
            _Delay

Duration    DC.L    453             ;Constant
```

Here, `Duration` is a long word constant that represents the length of the delay in ticks. On exit, the D0.L register contains the time on the system clock in ticks when the delay loop ends. You can also read this time from the system variable Ticks.

You may be tempted to generate delays by inserting dummy instruction loops in your program instead. If you do this, you can calculate the approximate delay by counting the number of cycles the 68000 needs to execute the instructions and multiplying the result by the cycle time (which is the reciprocal of 7.8 MHz, the clock frequency of the 68000 on the Macintosh). The Motorola M68000 *Programmer's Reference Manual* contains the cycle times for each 68000 instruction. You should avoid this method, however, because interrupts caused by the mouse (and other sources) will

make the delay seem longer than expected and future versions of the Macintosh may operate at a faster clock rate.

If you want to measure the time interval between the happening of two events, simply read the value stored at Ticks once when the first event occurs and again just after the second event. The elapsed time, in seconds, is simply the difference between the two values, divided by 60.

Reading the Time of Day and Date

Although the toolbox has several instructions you can use to set the time of day and the date, we're not going to look at them here because you'll rarely use them. When you want to change the time and date, it's much more convenient to use the Control Panel desk accessory.

What you'll usually want to do is read the current time and date in order to display it on the screen; to do this, use the _Pack6 instruction. _Pack6 is actually a multipurpose instruction that provides access to a package of related time and date instructions. An instruction in the package is selected by pushing a **routine selector** word on the stack before calling _Pack6.

To read the time, use _Pack6 with a routine selector of 2:

```
        MOVE.L  Time,-(SP)      ;Get seconds since Jan 1/1904
        MOVE.B  #-1,-(SP)       ;-1=seconds/0= no seconds
        PEA     TString(A5)     ;String returned here
        MOVE    #2,-(SP)        ;SELECTOR: 2 = Read time
        _Pack6

TString DS      6               ;time string
```

The first number pushed on the stack is the long word value stored in the global system variable, Time. This holds the number of seconds since midnight on January 1, 1904. A rather odd time base, to be sure, but, in any event, _Pack6 converts this tick count it into a string of the form:

```
HH:MM:SS XM  (X = A or P)
```

The seconds part of the time string (:SS) is actually returned only if you push a Boolean value of true (−1) after pushing the Time value. If you push false (0), seconds are ignored.

The other useful routine selector for _Pack6 is 0. Use it to return a date string in one of the following three forms:

9/21/86	**short date form**
Sunday, September 21, 1986	**unabbreviated long date form**
Sun, Sep 21, 1986	**abbreviated long date form**

Here's how to return any of these strings:

```
        MOVE.L  Time,-(SP)          ;Seconds count
        MOVE.B  #1,-(SP)            ;form code, 0=short,
                                    ; 1=long, 2=abbrev.
        PEA     DString(A5)         ;Date string variable
        MOVE    #0,-(SP)            ;Routine selector
        _Pack6

Dstring DS      29                  ;Enough room for longest
                                    ; string.
```

If you prefer, you can define macros for calls to _Pack6 to make it easier to remember what it is you're doing. Here are two macros for _IUTimeString (_Pack6, selector 2) and _IUDateString (_Pack6, selector 0):

```
MACRO _IUTimeString =
MOVE    #2,-(SP)
_Pack6
|

MACRO _IUDateString =
MOVE    #0,-(SP)
_Pack6
|
```

These macros are equivalent to two of the same name in the PackMacs.txt system equate file on the MDS disk.

If you include these definitions in your program source file, read the time and date by calling _IUTimeString and _IUDateString, instead of explicitly pushing a routine selector and calling _Pack6.

By the way, the "IU" in these names stands for **International Utilities**. These are utilities that are country-dependent—that is, the formats of the strings they return vary depending on national requirements. Two INTL resources (with IDs of 0 and 1) contain information describing how date and time strings are to be formatted. The Macintosh is shipped with the INTL resources appropriate to the country in which it is sold.

Chapter 6

Windows and Video Output

This chapter examines the most fundamental element of the Macintosh user interface: the window. This is where applications display their text and graphic output so it can be viewed by the user.

The Macintosh interface permits the handling of windows in a very flexible way: There can be several windows on the desktop at any time and each can be moved (or dragged) around the screen independently of the others. Unlike some operating systems that use the window metaphor, Macintosh windows can overlap one another. In fact, any window may totally obscure another.

Although several windows can coexist on the screen, only one is said to be **active** at any given time. By convention, the active window is always at the front of the screen and its drag region and scroll controls (more about these window parts later) are highlighted. To activate another window, all you have to do is click within its frame.

In the next section, you'll see how a window is represented in memory and what the various parts of a window are. You'll learn how to create windows, destroy them, move them around on the screen, and resize them. At the end of the chapter, some of the instructions used to draw text and graphics in a window will be analyzed.

Introduction to Windows

To the user, a window is just a rectangular box on the screen containing the output of a program. From a program-

mer's point of view, however, a window is much more than that. Its definition includes the window's position on the screen; the font; style; and size of the characters to be used when writing text in it; its title; whether it has a close box; and more. All this information is kept in a data structure called a window record.

Table 6-1. The Window Manager Trap Instructions.

_BeginUpdate — **Saves the window's visible region, then assigns the visible region to the update region.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to window record
_BeginUpdate

Call _BeginUpdate in response to an update event for a
window.
```

_CloseWindow — **Removes a window from the screen but does not free up the window record.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to window record
_CloseWindow
```

Use this instruction if you created the window by passing a nonzero wStorage parameter to _NewWindow; otherwise, use _DisposWindow.

_DisposWindow — **Removes a window from the screen and frees up all memory associated with the window record.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to window record
_DisposWindow
```

Use this instruction if you created the window by passing a zero wStorage parameter to _NewWindow; otherwise, use _CloseWindow.

Table 6-1. *continued*

_DragWindow — **Drags a window around the screen in response to the movement of the mouse and redraws it when the mouse button is released.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to window record
MOVE.L startPoint,-(SP)    ;LONGINT: point where mouse was
                                     pressed (global)
PEA    limitRect           ;POINTER: to rectangle limiting
                           ;         the scope of the drag
_DragWindow
```

The points for the limitRect rectangle are stored in global coordinates.

_DrawGrowIcon — **Draws the window's size box and the "elevator shafts" for the scroll bars.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to the window
_DrawGrowIcon
```

_EndUpdate — **Restores the window's visible region to the region saved when _BeginUpdate was called.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to window record
_EndUpdate
```

_Call _EndUpdate at the end of your update-handling code.

_FindWindow — **Returns a code indicating the part of a window in which a mouse click occurred.**

```
CLR    -(SP)               ;INTEGER: space for result
MOVE.L mousePoint,-(SP)    ;LONGINT: point on screen where
                           ;     mouse was pressed (global)
PEA    theWindow           ;VAR: pointer to window that was
                           ;     clicked is returned here
_FindWindow
MOVE   (SP)+,D0            ;Result: window part code
```

Table 6-1. *continued*

_FrontWindow — **Returns a pointer to the currently active window.**

```
CLR.L  -(SP)              ;POINTER: space for result
_FrontWindow
MOVE.L (SP)+,A0           ;Result: pointer to window
```

_GetNewWindow — **Loads a new window from a WIND resource file and displays it.**

```
CLR.L  -(SP)              ;POINTER: space for result
MOVE   #templateID,-(SP)  ;INTEGER: resource ID of WIND
MOVE.L wStorage,-(SP)     ;POINTER: to window record
MOVE.L behindWindow,-(SP) ;POINTER: to window in front
_GetNewWindow
MOVE.L (SP)+,A0           ;Result: pointer to window
```

_GetWTitle — **Returns the title of a window.**

```
MOVE.L theWindow,-(SP)    ;POINTER: to the window
PEA    newTitle           ;VAR: the title string
_SetWTitle
```

_GlobalToLocal — **Convert global coordinates to local coordinates.**

```
PEA    thePoint           ;VAR: a point (long word)
_GlobalToLocal
```

_InitGraf — **Initializes the QuickDraw drawing environment.**

```
MOVE.L globalVars,-(SP)   ;POINTER: to QD global variables
_InitGraf
```

_InvalRect — **Adds a rectangular region to the current window's update region.**

```
PEA    badRect            ;POINTER: to a rectangle (local)
_InvalRect
```

Table 6-1. *continued*

_InvalRgn — **Adds a region to the current window's update region.**

```
MOVE.L badRegion,-(SP)     ;POINTER: to a region
_InvalRgn
```

_LocalToGlobal — **Convert local coordinates to global coordinates.**

```
PEA    thePoint            ;VAR: a point (long word)
_LocalToGlobal
```

_NewWindow — **Creates and displays a new window.**

```
CLR.L  -(SP)               ;POINTER: space for result
MOVE.L wStorage,-(SP)      ;POINTER: to window record
PEA    windowRect          ;POINTER: to port rectangle
PEA    title               ;POINTER: to window title
MOVE.B #visible,-(SP)      ;BOOLEAN: true = visible
                           ;         false = invisible
MOVE   #windowType,-(SP)   ;INTEGER: window defn ID
MOVE.L behindWindow,-(SP)  ;POINTER: to window in front
MOVE.B #hasClose,-(SP)     ;BOOLEAN: true = close box
                           ;         false = no close box
MOVE.L #refCon,-(SP)       ;LONGINT: reference constant
_NewWindow
MOVE.L (SP)+,A0            ;Result: pointer to window
```

_SelectWindow — **Deactivates the previous window, activates a new window, redraws the new window in the front of the screen, and generates all necessary activate and update events.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to window to activate
_SelectWindow
```

_SetWTitle — **Sets the title of a window.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to the window
PEA    newTitle            ;POINTER: to the new title
_SetWTitle
```

Table 6-1. *continued*

_SizeWindow — **Draws a window with new dimensions.**

```
MOVE.L theWindow,-(SP)     ;POINTER: to the window
MOVE   #newWidth,-(SP)     ;INTEGER: new width
MOVE   #newHeight,-(SP)    ;INTEGER: new height
MOVE.B #update,-(SP)       ;BOOLEAN: true = updates OK
                           ;         false = no updates
```

The Boolean update parameter indicates whether newly exposed regions of the window are to be placed in the window's update region.

_SystemClick — **Passes a button-down event to a desk accessory for processing.**

```
PEA    theEvent            ;POINTER: to the event record
MOVE.L theWindow,-(SP)     ;POINTER: to window where event
                           ;         occurred
_SystemClick
```

_TrackBox — **Checks if the mouse button is released when the mouse cursor is still in the zoom box.**

```
CLR.B  -(SP)               ;BOOLEAN: space for result
MOVE.L theWindow,-(SP)     ;POINTER: to window involved
MOVE.L thePoint,-(SP)      ;LONGINT: mouse position (global)
MOVE   partCode,-(SP)      ;INTEGER: _FindWindow part code
_TrackBox
MOVE.B (SP)+,D0            ;Result: true = still in box
                           ;        false = not in box
```

_TrackGoAway — **Checks that the mouse button is released when the mouse cursor is still in the go-away box.**

```
CLR.B   -(SP)              ;BOOLEAN: space for result
MOVE.L  theWindow,-(SP)    ;POINTER: to window involved
MOVE.L  thePoint,-(SP)     ;LONGINT: mouse position (global)
_TrackGoAway
MOVE.B (SP)+,D0            ;Result: true = still in box
                           ;        false = not in box
```

Table 6-1. *continued*

_ValidRect	**Removes a rectangular region from the current window's update region.**

```
PEA     badRect             ;POINTER: to a rectangle (local)
_ValidRect
```

_ValidRgn	**Removes a region from the current window's update region.**

```
MOVE.L badRegion,-(SP)      ;POINTER: to a region
_ValidRgn
```

_ZoomWindow	**Zooms a window in or out.**

```
MOVE.L theWindow,-(SP)      ;POINTER: to window involved
MOVE   partCode,-(SP)       ;INTEGER: _FindWindow part code
MOVE.B #front,-(SP)         ;BOOLEAN: true = bring to front
                            ;         false = leave alone
```

Call this instruction with a partCode of #inZoomIn (zoom window to its pre-zoomed state) or #inZoomOut (zoom out the window).

The subroutines used to control windows make up the part of the Macintosh toolbox called the **Window Manager**. It is these subroutines that we'll be investigating for the next several pages. You should keep in mind, however, that the Window Manager ultimately relies on a group of fundamental screen drawing subroutines, collectively called **QuickDraw**, whenever it must display anything on the screen. It also uses the **Font Manager**, the part of the toolbox that deals with text characters. For these reasons, before any of the Window Manager commands can be used you must initialize QuickDraw and the Font Manager with the following instructions:

```
PEA     -4(A5)          ;Address of QuickDraw global area
_InitGraf               ;Initialize QuickDraw
_InitFonts              ;Initialize Font Manager
```

Notice that these are the first three instructions used in the standard program header described in Chapter 2.

QuickDraw Global Variables

MDS pre-allocates a 256-byte space for QuickDraw global variables just below the address pointed to by the A5 register, although only GrafSize (206) bytes are actually used. The address of the last variable in the space (a long word) is given by −4(A5) and that's the address passed to _InitGraf.

_InitGraf stores the address of the last variable in the QuickDraw variable space at the location pointed to by A5. This is the first entry in the system parameter table.

The end of the QuickDraw global area contains the following variables, shown in reverse order, from high to low memory:

ThePort	**(4 bytes)**	**a pointer to the active window**
White	**(8 bytes)**	**standard white pattern**
Black	**(8 bytes)**	**standard black pattern**
Gray	**(8 bytes)**	**standard gray pattern**
LtGray	**(8 bytes)**	**standard light gray pattern**
DkGray	**(8 bytes)**	**standard dark gray pattern**
Arrow	**(68 bytes)**	**standard arrow cursor record**
ScreenBits	**(14 bytes)**	**screen bitmap (see below)**
RandSeed	**(4 bytes)**	**seed for random number generator**

Additional QuickDraw variables below these are for the private use of QuickDraw. Below all the QuickDraw variables are the application's global variables.

The symbolic names given for the QuickDraw variables represent offsets from the highest addressed variable, ThePort. To access a QuickDraw variable, say RandSeed, use an instruction sequence like the following:

```
MOVE.L  (A5),A0          ;Get pointer to QD area in A0
MOVE.L  RandSeed(A0),D0 ;Access RandSeed
```

Alternately, you can access it directly with a MOVE.L

RandSeed−4(A5),D0 instruction, but only if you pass the effective address of −4(A5) to _InitGraf.

The Parts of a Window

Before you learn how to create windows, I'll summarize the terminology used to describe the various parts of a window. Figure 6-1 shows a typical Macintosh window whose constituent parts are labeled.

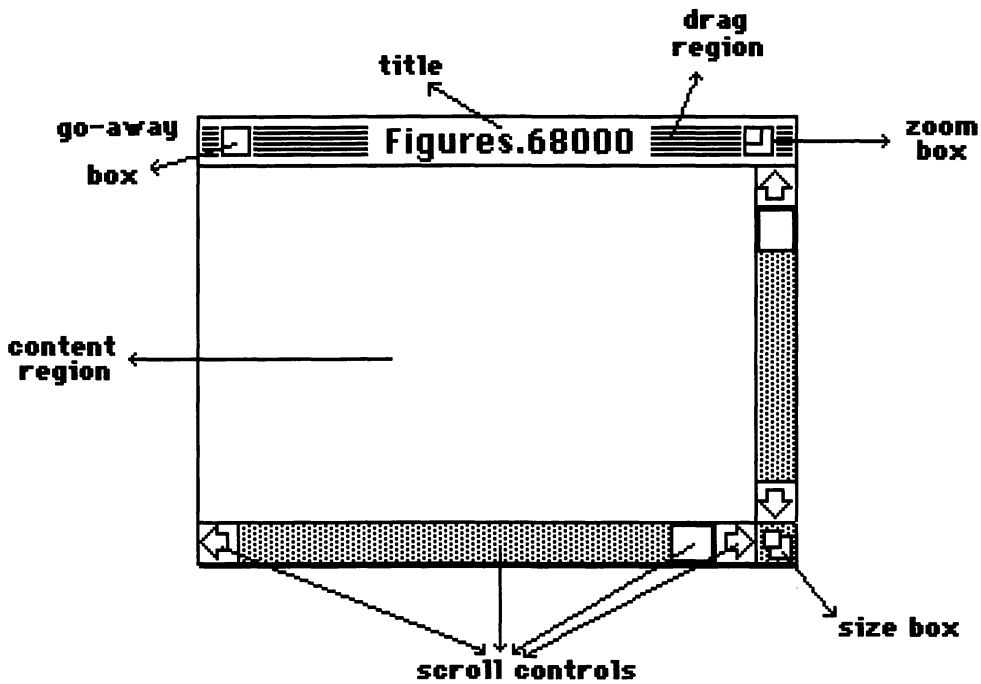


Figure 6-1. The Parts of a Macintosh Window.

The **go-away box** (also called the **close box**) is located in the top left-hand corner of the window. According to the Macintosh user-interface guidelines, if you click this box, the window is to close and disappear from the screen. Some windows, such as dialog and alert boxes, do not have a go-away box.

The **drag region** is the rectangular region on the top of a window containing the title of the window and, if the window is the active one, the "racing stripes" on either side of the title. The drag region does not include the go-away box. A window need not have a drag region.

If the mouse button is pressed while the cursor is in the drag region and the mouse is moved with the button still down, an outline of the window moves around the screen. When the button is released, the window is redrawn at its new position.

The **content region** is the area of the window within which you can draw and view text and graphics. It is bounded by the **window frame**, which includes the window's outline, the go-away box, and the drag region.

The **size box**, if present, is located in the lower right-hand corner of the window, usually within the content region. By positioning the mouse pointer in the size box and dragging the mouse, you can alter the size of the window.

A **zoom box** sometimes appears in the top right-hand corner of a window. The first time you click this box, the window expands to fill the entire screen, enabling you to quickly view as much information within the window as possible; on the next click in the zoom box, the window returns to its pre-zoomed size.

The **scroll controls** are found in horizontal and vertical "elevator shafts" that appear within the content region of a window. They include two arrows at either end of the shaft, and a movable control called a **thumb**. The scroll controls are used to move the portion of a text or graphics image within a window up and down or left and right. Movement can be line by line by clicking an arrow, or page by page by clicking in the space between an arrow and the thumb. You can move directly to any part of the document shown in the window by moving the thumb.

It is the responsibility of the programmer to adhere to the standard user-interface guidelines in response to mouse activity in the various parts of a window. For example, if the mouse is clicked in the go-away box, the window does not

automatically close; it is up to you to write your program in such a way that it does. We'll see how to do this later in this chapter.

Coordinate Systems

Now a word about the coordinate systems used by QuickDraw and the Window Manager. The first element in a window record is a QuickDraw data structure called a **GrafPort**, which contains information concerning the drawing environment for the window: this includes the pen characteristics for drawing operations, background patterns, and fill patterns. One important field in a GrafPort is called **PortBits**. It is a **bitmap** that describes the portion of a rectangular array of bits (a **bit image**) that drawing operations are to affect and imposes a coordinate system for the map. The structure of a bitmap record like PortBits is as follows:

BaseAddr	pointer
rowBytes	integer
boundsRect	rectangle

BaseAddr is a pointer to the memory location defining the upper left-hand corner of the bit image.

RowBytes is the width of the bit image in bytes; it must be an even number. Even though rowBytes must describe an integral number of words, the active portion of the bit image may be narrower. The **boundsRect** rectangle defines the portion of the bit image that is the active part of the bitmap; this rectangle must not extend beyond the boundaries of the bit image.

> The data structure for a **rectangle** is made up four integers representing the position of its top, left, bottom, and right boundaries, in that order. The symbolic names for the offsets to these points are (as you might expect) top, left, bottom, and right.

BoundsRect also defines a coordinate system where the bit in the top-left corner of the bit image has a coordinate equal to the top-left coordinate for boundsRect. This coordinate is not necessarily (0,0). In fact, when a window is created, the (0,0) position is assigned to the top-left corner of the content region of the window and boundsRect is adjusted to account for this. (This is done to make it easier for you to position items within a window.) The coordinate system defined by boundsRect is called a **local coordinate system** because it is used by drawing operations for the GrafPort with which it is associated.

> The notation (x,y) is the shorthand representation for the coordinates of a point on the screen. The first number, x, is the horizontal position and the second, y, is the vertical position.

The area within a GrafPort's bitmap that QuickDraw actually draws into is described by **PortRect**, another field in the GrafPort data structure. PortRect describes a rectangle that is usually wholly contained within the bitmap. It is defined using local coordinates. When a window is opened, PortRect is the rectangle enclosing the content region of the window.

A standard bitmap describing the Macintosh screen is stored at ScreenBits in the QuickDraw global variable area. If you passed an address of −4(A5) to _InitGraf, this address is given by ScreenBits−4(A5). For a 512K Macintosh, the BaseAddr value for this bitmap contains $7A700 (the start of the screen buffer), rowBytes contains 64, and the coordinates of boundsRect are 0,0, 342,512 (top, left, bottom, right). This means the bit image is the entire screen and that the entire screen forms part of the bitmap.

To compare coordinates in one GrafPort with those in another, you must first convert to a common coordinate system. The QuickDraw subroutines use a **global coordinate** sys-

tem where the top-left corner of the bitmap pointed to by BaseAddr is always considered to be at (0,0). As long as the BaseAddr pointers for the GrafPorts whose coordinates are being compared contains the same value, global coordinates map to memory locations in exactly the same way for each GrafPort; thus, comparisons are meaningful. For windows displayed on the Macintosh screen, this is indeed the case: BaseAddr always points to the starting address of the screen buffer.

In either coordinate system, the horizontal coordinates increase as you move to the right and the vertical coordinates increase as you move to the bottom.

You usually pass **global coordinates** to Window Manager subroutines. This is the same coordinate system used by _GetNextEvent for passing the position of the mouse when an event occurs. The instructions that draw text and graphics within a window use the **local coordinate** system, however.

Before a drawing instruction can use a global coordinate, the coordinate must first be converted to a local coordinate using the _GlobalToLocal trap instruction:

```
        PEA   Where(A5)         ;push addr of global coords
        _GlobalToLocal          ;convert global to local

Where   DS.L    1               ;Point: vertical, horizontal
```

_GlobalToLocal takes the global coordinate at Where, converts it to a local coordinate, and stores it at Where.

Notice that the Where variable is a data structure of type **point**. A point is simply a long word that contains the vertical (high-order word) and horizontal (low-order word) coordinates for a position on the screen. This order is the reverse of the order used when describing a point using the standard (x,y) notation. The MDS symbolic offsets to the vertical and horizontal components of a point are v and h. There is a related trap instruction, _LocalToGlobal, for performing the opposite conversion.

Creating Windows

Before defining windows on the Macintosh, you must call the _InitWindows instruction. This clears the desktop to its background pattern, erases the menu bar at the top of the screen, and initializes all window-related data structures. _InitWindows does not require any parameters and must only be called once at the beginning of a program.

There are eight pre-defined types of windows you can use on the Macintosh, each identified by a unique code called a **window definition ID**. These windows are shown in Figure 6-2. If you are using a Macintosh with the original 64K ROMs, you can only use the first six window types shown.

The use of the standard windows is dictated by the Macintosh user-interface guidelines. Windows with an ID = 0, 4, 8, or 12 are the most common and usually contain a document being acted on by the application. These window types are the same, except that two don't have a zoom box and two don't have a grow box.

Windows with an ID = 1, 2, and 3 are usually used as dialog and alert boxes. (See Chapter 8.) Windows with an ID = 16 are most commonly used by desk accessories, such as the calculator.

There are two basic ways to create a window. First, you can create it from scratch within the program. Second, you can use RMaker to create a WIND resource and store it in a resource file.

The _NewWindow instruction defines a window from scratch. Its general form is as follows:

```
CLR.L   -(SP)              ;Clear space for result
MOVE.L  #0,-(SP)           ;0=use heap for record
PEA     WindRect           ;Window dimensions
PEA     'Our Window'       ;Title for window
MOVE.B  #-1,-(SP)          ;-1 = visible (0=invisible)
MOVE    #0,-(SP)           ;0 = window definition ID
MOVE.L  #-1,-(SP)          ;-1 = this window in front
MOVE.B  #-1,-(SP)          ;-1 = draw a close box
```

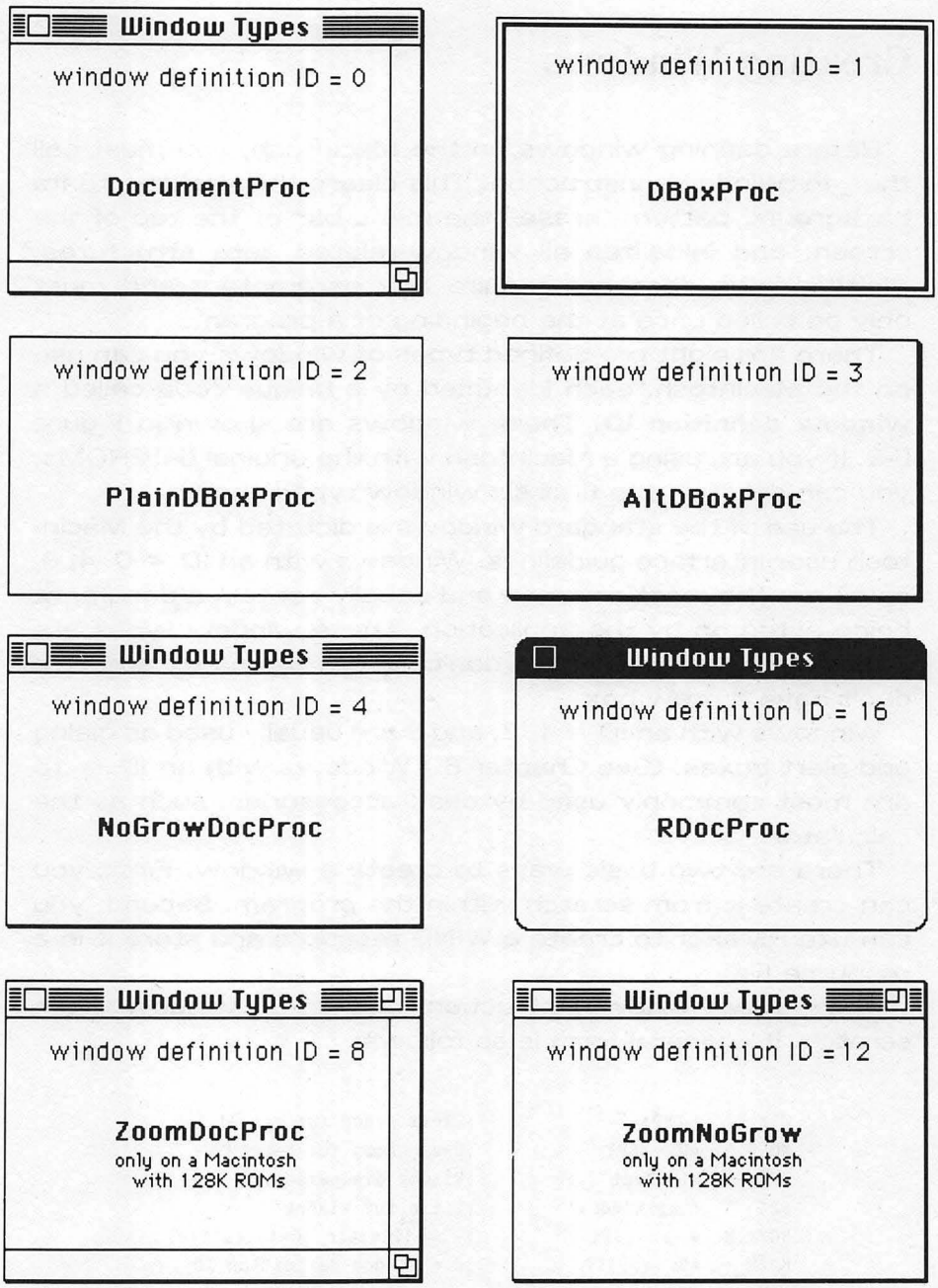


Figure 6-2. Standard Macintosh Window Types.

```
        CLR.L   -(SP)                   ;User-definable parameter
        _NewWindow
        MOVE.L  (SP)+,theWindow(A5) ;Pop long word pointer

WindRect  DC.W    50,50,200,300         ;Window rectangle (TLBR)

theWindow DS.L    1                     ;Space for long
```

_NewWindow is a function that returns on the stack a pointer to the window record. This means you must clear space for a long word on the stack before pushing the parameters _NewWindow requires. After calling _NewWindow, pop the pointer off the stack and store it in a variable so you can use it to access the window later on.

The first parameter passed to _NewWindow is a pointer to the area it uses to store the window record. You can reserve such an area (its size is given by the MDS constant WindowSize, 156 bytes) on the stack using _NewPtr. (See Chapter 4.) It's usually more convenient to ask _NewWindow to reserve this space automatically, however. To tell it to do this, push a zero pointer as in the above example.

The next parameter is a pointer to the coordinates of the window rectangle. The coordinates must be in top, left, bottom, right order.

The next five parameters relate to the appearance of the window. The first is its title, whether it is visible (true, −1) or invisible (false, 0). Next is the window definition ID code. The third refers to a pointer to the window in front of it (or −1 if the window is to be drawn in front) and the fourth concerns whether a close box is to be drawn (true, −1) or not drawn (false, 0).

Listing 6-1 is a program illustrating how to use _NewWindow. It creates and displays each of the eight basic window types whose definition IDs are kept in a table at WindID. The program also displays the window definition ID number in the window using some instructions (_Pack7, _MoveTo, and _DrawString) I haven't discussed yet. I'll be explaining these instructions later on in this chapter. To display each type of window in the program, keep clicking the mouse button.

After the last window type is displayed, you will return to the Finder.

Listing 6-1. The Source File and Linker Control File for the WindTypes Program.

```
* Asm Source File
* WindTypes.Asm
*
* This program displays the seven basic window types on the
* screen. Click the mouse button to move between the windows.

WindNum EQU     8               ;Number of window types

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QD globals area
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event


        LEA     WindIDs,A6      ;Load base address of ID table

        MOVE    #WindNum-1,D6   ;Set up loop count for DBF

; Draw a window on the screen. The window ID is
; contained in (A6).
DrawWind
```

Listing 6-1. *continued*

```
        CLR.L   -(SP)           ;Space for returned pointer
        MOVE.L  #0,-(SP)        ;0 = store window in stack
        PEA     Window          ;Window rectangle
        PEA     'Window Types'  ;Window Title
        MOVE.B  #-1,-(SP)       ;-1 = visible
        MOVE    (A6),-(SP)      ;First window type is at (A6)
        MOVE.L  #-1,-(SP)       ;-1 = front window
        MOVE.B  #-1,-(SP)       ;-1 = go away button
        MOVE.L  #0,-(SP)        ;refCon
        _NewWindow              ;Draw the window
        MOVE.L  (SP),WindPtr(A5)   ;Save pointer (don't pop)
        _SetPort                ;Make window current for drawing

        CMP     #0,(A6)         ;Is this a standard doc window?
        BEQ     @0              ;Yes, so branch
        CMP     #8,(A6)         ;Is this zoom with grow box?
        BNE     @1              ;No, so branch

@0      MOVE.L  WindPtr(A5),-(SP)
        _DrawGrowIcon           ;Draw the grow box

; Display the window definition ID number of the window:

@1      LEA     String(A5),A0   ;Address of string in A0
        CLR.L   D0              ;Make sure high word is zero
        MOVE    (A6)+,D0        ;Put ID in D0 and bump pointer
        MOVE    #0,-(SP)
        _Pack7                  ;_NumToString

        MOVE    #20,-(SP)       ;horizontal pos.
        MOVE    #20,-(SP)        ;vertical pos.
        _MoveTo                 ;Position the pen

        PEA     'window definition ID = '
        _DrawString             ;Display the string

        PEA     String(A5)
        _DrawString             ;Print the window type code

        JSR     GetButton       ;Wait for button press

        MOVE.L  WindPtr(A5),-(sp)   ;Erase window and remove it
```

Listing 6-1. *continued*

```
        _DisposWindow           ; from system

        DBRA    D6,DrawWind     ;Loop until D6=-1

        RTS                     ;Return to Finder

; Loop until the mouse button is pressed:

GetButton

        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1,-(SP)       ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result code
        BEQ     GetButton       ;Branch if no event

        MOVE    EventRecord+evtNum,D0   ;Get event type
        CMP     #mButDwnEvt,D0  ;Is it a button-down event?
        BNE     GetButton       ;No, so branch

        RTS

EventRecord     DCB.B   EvtBlkSize,0   ;Space for event record

Window          DC.W    50,25,200,225   ;window coordinates

WindIDs         DC.W    0,1,2,3     ;Valid IDs for windows
                DC.W    4,8,12,16


; Here are the program globals. Use (A5) addressing.


WindPtr         DS.L    1       ;Pointer to our window

String          DS.W    2       ;Space for number conversion

; Linker Control File
; WindTypes.Link
;
WindTypes
$
```

Another way to define a new window is to read its definition from a resource file using the _GetNewWindow instruction. Like _NewWindow, _GetNewWindow returns a pointer to the window data structure:

```
WindID    EQU     144                 ;Window resource ID

          CLR.L   -(SP)               ;Space for result
          MOVE    #WindID,-(SP)       ;Window resource ID
          MOVE.L  #0,-(SP)            ;window record on heap
          MOVE.L  #-1,-(SP)           ;Ptr to window in front
          _GetNewWindow
          MOVE.L  (SP)+,theWindow(A5) ;Save the result

theWindow DS.L    1
```

As you can see, you need only specify a resource ID number and a window pointer to load a window definition from a resource file. All the other parameters needed to describe the window record are contained within the resource file itself.

Of course, you can only use _GetNewWindow if you've previously stored the WIND resource in a resource file and that file is open. To create the resource, first use Edit to create a source code file for the window record for RMaker, the resource compiler. The form of the definition for a window is shown in Table 6-2. Next, place the name of the application file (preceded by !) at the beginning of the RMaker source file so that the window resource will be placed in the resource fork of the application file itself during the compilation process. This assumes the file has already been created by assembling and linking the main program file.

It is convenient to place the window resource in the application's resource fork because it means the resource will be automatically available to the application when it starts to run. You could also use RMaker to store the definition in a separate resource file, but that file would have to be explicitly opened using _OpenResFile.

Table 6-2. The RMaker Format of a WIND Resource Definition.

```
Type WIND
,128                    ;; Resource ID of window
A Pane in the Glass     ;; window title
20 20 350 400           ;; coordinates of window (TLBR)
I N                     ;; window status
4                       ;; window definition ID
0                       ;; reference value (user-definable)
```

The window definition ID can be 0, 1, 2, 3, 4, 8, 12, or 16. The window status can be Visible (V) or Invisible (I), NoGoaway (N) or Goaway (G).

Once a window has been created, you still can't draw anything in it because it is not the active drawing window. To activate a window to enable you to draw text and graphics in it, push the pointer to the window on the stack, and call the _SetPort instruction:

```
            MOVE.L  theWindow(A5),-(SP)
            _SetPort

theWindow   DS.L    1
```

This instruction presumes, of course, that the pointer returned by _NewWindow or _GetNewWindow was stored in a variable called theWindow.

Before you use _SetPort to designate a new active drawing window, you should save the pointer to the current drawing window using _GetPort. That way, you can easily return control to the original window with another _SetPort instruction.

To use _GetPort, pass the address of the location in which _GetPort is to return the pointer as follows:

```
            PEA     OldWindPtr      ;Return pointer here
            _GetPort

OldWindPtr  DC.L    0               ;This is a constant
```

Since OldWindPtr is a constant, the (A5) addressing mode is not used with PEA.

Destroying Windows

There are two instructions that will close a window, _CloseWindow and _DisposWindow. The one to use depends on how you initially created the window.

If you used _NewPtr to create space for the window record used by _NewWindow, use _CloseWindow to close the window. It takes a pointer to the window record as a parameter, removes the window from the screen, but does not free up the area used by the window record. To free up that space you must use the Memory Manager's _DisposPtr instruction. (See Chapter 4.)

If the Window Manager automatically allocated space for the window record on the heap (it always does if you use _GetNewWindow), use _DisposWindow to close the window. This instruction not only erases the window from the screen, it also deallocates the space reserved for the window record.

Both _CloseWindow and _DisposWindow cause update events if previously hidden parts of other windows are exposed when a window disappears. An activate event also occurs if the active window is closed and there are other windows on the screen; the window nearest the front of the screen is activated.

Reacting to Window-Related Events

In the previous chapter you learned the Event Manager can post three types of events that should be processed by the Window Manager:

- UpdatEvt (window update event)
- ActivateEvt (window activate or deactivate event)
- MButDwnEvt (mouse button down event)

These events are by no means processed automatically; it is up to your program to detect them and take appropriate action.

Update Events

An update event (UpDatEvt) occurs when a previously hidden portion of a window comes into view. This happens if another window is closed, moved, or resized, or if the subject window is activated and moved to the front of the screen or is enlarged by dragging on its size box. The Window Manager instructions take care of automatically adding newly exposed regions of a window in a data structure called the **update region** and generating the update event.

When you respond to an update event, you must re-draw those portions of the window contained in the update region. To do this, first call _BeginUpdate to ensure that subsequent drawing within the window will be restricted (or **clipped**) to the update region only. This is done by temporarily assigning the window's visible region—the part you can see on the screen—to the intersection of the existing visible region and the update region. This means you can redraw the entire window, but only the update region is affected.

Next, you have to redraw the screen. To do this, of course, you must know what the contents of the screen were just before the update event occurred. This requires careful planning on your part, and virtually dictates that you maintain some sort of data structure in memory describing the contents of the window at any given time, so you can re-create it when necessary.

When the screen has been redrawn, call _EndUpdate. This empties the update region and resets the visible region of the window to its original value.

Here's what the entire procedure looks like:

```
MOVE.L  theWindow(A5),-(SP)
_BeginUpdate
.
[re-write the screen here]
.
MOVE.L  theWindow(A5),-(SP)
_EndUpdate
```

You can force update events to occur under program control by adding regions or rectangular areas to the accumulated update region using _InvalRgn and _InvalRect. (See *Inside Macintosh* for a technical description of a region.) This is a handy way of forcing the redrawing of a portion of the screen. You use _InvalRgn and _InvalRect as follows:

```
            PEA     Rectangle          ;address of rectangle coords
            _InvalRect

Rectangle   DS.W    10,10,50,75        ;TLBR (local)
```

and

```
          MOVE.L  RgnHndl(A5),-(SP)    ;push handle to region
          _InvalRgn

RgnHndl DS.L    1
```

Notice that the rectangle coordinates used by _InvalRect are local coordinates.

There are corresponding instructions for *removing* regions and rectangular areas from the update region: _ValidRgn and _ValidRect. Use them if you want to prevent an update event from being posted in the event queue.

Activate Events

An activate event (ActivateEvt) occurs when either of two events occurs: A window is activated or a window is deacti-

vated. To determine which of these two related events has occurred, check the ActiveFlag bit (bit 0) of the evtMBut field of the EventRecord returned by _GetNextEvent:

```
MOVE.B  EventRecord+evtMBut,D0
BTST    #ActiveFlag,D0          ;Is activate bit on?
BEQ     DeActivateIt            ;No, so deactivate
```

If it's an activate event, the bit will be 1 and the BEQ branch will not be taken.

An activate event occurs when a new window is brought to the front of the screen. This happens when you call _SelectWindow when the mouse is clicked inside an inactive window or when the currently active window is closed. To handle an activate event, you should call _SetPort to make the activated window the current drawing window, and then redraw the size box (using _DrawGrowIcon) and the scroll bars, if necessary. You may also want to highlight or dim certain items in the menus at the top of the screen, depending on whether they are applicable to the newly activated window. (You'll see how to do this in the next chapter.)

For a deactivate event, you should dim the scroll bar and grow box by calling _DrawGrowIcon. Since the window isn't active, the Window Manager will not draw the highlighted grow box icon as it normally would. The deactivation subroutine may also involve highlighting or removing highlighting from menu items.

Button-Down Events

Your response to a button-down event (MButDwnEvt) depends on precisely where the mouse button was pressed: in a close box, the drag area, a content region, or another identifiable part of a window. To determine the location, use the _FindWindow instruction:

```
            CLR     -(SP)           ;Space for result
            MOVE.L  EventRecord+evtMouse,-(SP) ;coordinates (global)
            PEA     theWindow(A5)   ;Window pointer returned here
            _FindWindow
            MOVE    (SP)+,partCode(A5)

EventRecord     DCB.B   EvtBlkSize,0    ;Event record
theWindow       DS.L    1
partCode        DS.L    1
```

The word result returned by _FindWindow, stored at `partCode` in the above example, is a **part code** that reflects the region on the screen in which the mouse button was pressed. There are nine possible part codes:

0	**inDesk**	**(in the desktop)**
1	**inMenuBar**	**(in the menu bar)**
2	**inSysWindow**	**(in a desk accessory window)**
3	**inContent**	**(in a content region)**
4	**inDrag**	**(in a drag region)**
5	**inGrow**	**(in a size region)**
6	**inGoAway**	**(in a go-away box)**
7	**inZoomOut**	**(in a zoom box)**
8	**inZoomIn**	**(in a zoom box)**

The last two part codes, inZoomOut and inZoomIn, cannot be returned if the Macintosh is using the original 64K ROMs.

The inSysWindow part code is generated if the mouse is pressed in any part of a system (desk accessory) window. This means inContent, inDrag, inGrow, inGoAway, inZoomOut, and inZoomIn codes refer to regions in application windows only.

inDesk. If the part code is inDesk you will probably want to ignore the button press because no specific action is dictated by the user-interface guidelines.

inMenuBar. If the part code is inMenuBar, you should pass control to the Menu Manager so it can take care of pulling down menus and selecting menu items. You'll see how to do this in the next chapter.

inSysWindow. A part code of inSysWindow means there has been a click in the window for a desk accessory. You'll learn about desk accessories in detail in Chapter 9, but for now all you need to know is that you should pass control to the desk accessory using the _SystemClick instruction:

```
            PEA     EventRecord
            MOVE.L  theWindow(A5),-(SP)
            _SystemClick

EventRecord DCB.B   EvtBlkSize,0    ;Event record
theWindow   DS.L    1               ;[returned by _FindWindow]
```

EventRecord is the same record used by the _GetNextEvent instruction that reported the button-down event.

inContent, inDrag, inGrow, inGoAway, inZoomIn, inZoomOut. After you call _FindWindow and you've determined that the button was pressed in an application window, you should check whether the window is currently active or not. This can be done by comparing the window pointer returned by _FindWindow, which was stored at theWindow(A5) in the example above, with the pointer returned by the _FrontWindow function. If they aren't the same, simply call _SelectWindow to activate the window in which the click occurred. Here's how to do this:

```
        CLR.L   -(SP)               ;A pointer is returned
        _FrontWindow                ;Get pointer to active window
        MOVE.L  (SP)+,A6            ;Pop the window pointer
        CMP.L   theWindow(A5),A6    ;Are windows same?
        BEQ     Continue            ;Yes, so proceed normally
        MOVE.L  A6,-(SP)            ;Push new window pointer
        _SelectWindow               ; and select new window.
        RTS
Continue
```

The call to _SelectWindow automatically generates an activate event.

If the window is already active, you would proceed to Continue, which would be the part of the program that processes

inContent, inDrag, inGrow, inGoAway, inZoomIn, and inZoomOut part codes.

inContent. There is no standard procedure to follow when the button is clicked in the content region of a window; it will depend on the nature of your application. If, for example, the program is a word processor, you will probably want to place an I-beam cursor at the mouse position to indicate a new text insertion point. On the other hand, the click may be within an action box you've drawn on the screen, so you would perform the action associated with it.

inDrag. If the button is pressed in the drag region of a window, call _DragWindow. When you do this, an outline image of the window will be moved around the screen as you move the mouse with the button still down. If you move the mouse outside the limits of a bounding rectangle you pass to _DragWindow, the outline disappears and reappears only if the mouse is dragged back into range again.

When the button is released, the window is redrawn at its new position. If the mouse is released outside the bounding rectangle, however, the window stays at its original position. Here's what the calling sequence for _DragWindow looks like:

```
MOVE.L  theWindow(A5),-(SP) ;window pointer
MOVE.L  EventRecord+evtMouse,-(SP) ;coordinates (global)
PEA     boundRect           ;bounding rectangle (TLBR)
_DragWindow
```

where boundRect is a constant made up of two global coordinates: the top-left and bottom-right coordinates of the rectangle within which the mouse pointer must be kept during the drag operation.

inGrow. When you detect a button press in the grow box, call _GrowWindow. _GrowWindow displays an outline of the window that expands and contracts as the mouse is moved back and forth. To avoid shrinking windows to a miniscule size or expanding them to an enormous size, you can specify

minimum and maximum values for the final rectangle's height and width. Here's how to use _GrowWindow:

```
            CLR.L   -(SP)                       ;Space for long word
            MOVE.L  theWindow(A5),-(SP)         ;Push pointer to window
            MOVE.L  EventRecord+evtMouse,-(SP) ;coordinates (global)
            PEA     sizeRect                    ;Dimension limits
            _GrowWindow
            MOVE.L  (SP)+,D0                    ;Pop result into D0

SizeRect    DC.W    10,10,300,270               ;height (min), width
                                                ;(min), height (max),
                                                ;width (max)
```

Notice that since sizeRect refers to a group of constants, it is not followed by (A5).

When the mouse button is released, _GrowWindow ends and a long word is returned on the stack. The high-order word of this number represents the new height of the window, in pixels. The low-order word represents the new width. If the size did not actually change, a zero is returned.

If the window has, indeed, changed in size, you must call _SizeWindow to redraw it. If the height and width are in D0, here's what you would do:

```
MOVE.L  theWindow(A5),-(SP)
MOVE.L  D0,-(SP)              ;push width and height
MOVE.B  #-1,-(SP)             ;-1 = generate updates
_SizeWindow
```

The last item pushed on the stack is a Boolean quantity indicating whether any new portions of the window that come into view are to be automatically added to the window's update region (true) or not (false). You would normally set this Boolean item to true (−1) and then redraw the window contents when processing the update event returned by the next call to _GetNextEvent.

You also have to redraw the grow box on the window if the window is resized. This is done as follows:

```
MOVE.L  theWindow(A5),-(SP)     ;Pointer to window
_DrawGrowIcon
```

If the window contains any scroll controls, they will also have to be redrawn using the appropriate Control Manager instructions. The Control Manager is described in *Inside Macintosh*.

inGoAway. According to the user-interface guidelines, if the button is pressed in a go-away box, you are not to immediately call _CloseWindow or _DisposWindow to erase the window from the screen. Rather, you must call _TrackGoAway to check that the mouse cursor is still positioned in (or very near) the go-away box when the button is released. Only if it is are you to close the window. The purpose of calling _TrackGoAway is to prevent closing a window on the basis of an errant mouse click.

_TrackGoAway is a function that returns a Boolean result. You pass to it a pointer to the window and the coordinates of the point where the mouse was pressed, in global coordinates:

```
CLR.B   -(SP)                   ;Space for Boolean result
MOVE.L  theWindow(A5),-(SP)     ;Push window pointer
MOVE.L  EventRecord+evtMouse,-(SP) ;global coordinates
_TrackGoAway
TST.B   (SP)+                   ;Is the result 0 (false)?
BEQ     NoClose                 ;Yes, so branch
MOVE.L  theWindow(A5),-(SP)
_DisposWindow                   ;Close the window
```

Notice that the TST.B instruction is used to check if the Boolean result is true or false and pop the result from the stack at the same time. If the result is zero, or false, the BEQ instruction transfers control to NoClose. NoClose is the label for an instruction somewhere else in the program.

inZoomIn and inZoomOut. When either of these two part codes is returned, call _TrackBox to check that the mouse button is released when its cursor is still in the zoom box:

```
CLR.B   -(SP)                       ;Space for Boolean result
MOVE.L  theWindow(A5),-(SP)         ;Push window pointer
MOVE.L  EventRecord+evtMouse,-(SP) ;coordinates (global)
MOVE    partCode(A5),-(SP)          ;push _FindWindow result
_TrackBox
TST.B   (SP)+                       ;BNE succeeds if in box
```

Like _TrackGoAway, _TrackBox returns a Boolean result indicating whether the button was released in the box or not.

If the result is true, you should immediately call _ZoomWindow to handle the zoom activity. _ZoomWindow zooms the window out to full screen size (for inZoomOut) or zooms it back to the pre-zoomed size (for inZoomIn). Here is how to call _ZoomWindow:

```
MOVE.L theWindow(A5),-(SP)      ;Push window pointer
MOVE   partCode(A5),-(SP)       ;Push _FindWindow result
MOVE.B #-1,-(SP)                ;-1 = window in front
_ZoomWindow
```

Notice that the third parameter is a Boolean, indicating whether the window will be brought to the front (true) or left where it is (false).

The ROM subroutines called by the _TrackBox and _ZoomWindow instructions are not included in the Macintosh's original 64K ROM. Since these instructions are used only in response to inZoomOut or inZoomIn part codes, however, and these codes cannot be generated by a Macintosh using a 64K ROM, a program that includes them will still work on an older Macintosh.

A Window Application

The program in Listing 6-2 demonstrates how to react to button-down events when a window is on the screen. With it you can drag a window around the screen, resize it, and close it by clicking in the goaway box. It uses some Menu Manager instructions, such as _DisableItem and _EnableItem, not covered yet, but we'll be looking at them in the next chapter.

Listing 6-2. The Source File, Linker Control File, and RMaker File for the MainWind Program.

```
; Asm Source File
; MainWind.Asm

; This program shows how to manipulate a
; single window on the screen.

WindID          EQU     128     ;Window ID

MenuBarID       EQU     128     ;Menu bar ID
 AppleID        EQU     1       ;Menu ID for Apple menu
 FileID         EQU     2       ;Menu ID for File menu

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QuickDraw globals
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; Read menu bar from MBAR resource, then make it current
; using _SetMenuBar and draw it using _DrawMenuBar:

        CLR.L   -(SP)           ;Space for result
        MOVE    #MenuBarID,-(SP)   ;Push resource ID
        _GetNewMBar
        _SetMenuBar             ;Handle already on stack
        _DrawMenuBar            ;Display menu bar

        CLR.L   -(SP)
        MOVE    #FileID,-(SP)
```

Listing 6-2. *continued*

```
        _GetMHandle             ;Get handle to file menu
        MOVE.L  (SP)+,FileHndl(A5)   ;Save it for later use

        BSR     OpenWindow      ;Open up the window

MainLoop
        BSR     GetEvent
        BSR     HandleEvent
        BRA     MainLoop

GetEvent
        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #$FFFF,-(SP)    ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result flag
        BEQ     GetEvent        ;Branch if no pending event
        RTS

HandleEvent
        MOVE    EventRecord+evtNum,D0   ;Get event type code
        CMP     #MButDwnEvt,D0  ;Is it a button-down event?
        BEQ     @2              ;Yes, so branch

        CMP     #UpdatEvt,D0    ;Update event?
        BEQ     @1              ;Yes, so branch

        RTS                     ;Ignore everything else


* Handle an update event by redrawing the grow box.
* In a complete application, you would redraw the text
* and graphics in the window as well.

@1      MOVE.L  OurWindow(A5),-(SP)
        _BeginUpdate            ;Restrict to update region

        MOVE.L  OurWindow(A5),A0
        PEA     PortRect(A0)    ;The window rectangle
        _EraseRect              ;Erase the window

        MOVE.L  OurWindow(A5),-(SP)
        _DrawGrowIcon           ;Redraw the size box

; (redraw window contents here)
```

Listing 6-2. *continued*

```
        MOVE.L  OurWindow(A5),-(SP)
        _EndUpdate              ;Clear update region

        RTS

* Handle mouse clicks:

@2      CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)   ;Where info
        PEA     ClickWindow     ;VAR window involved
        _FindWindow             ;Where was button pressed?
        MOVE    (SP)+,D0        ;Pop the result

        CMP     #6,D0           ;Above 6?
        BHI     Ignore          ;Yes, so ignore
        ASL     #2,D0           ;Times 4 to step into table
        JMP     ClickTable(PC,D0)

Ignore  RTS

ClickTable
        JMP     Ignore          ;InDesk
        JMP     DoMenuBar       ;InMenuBar
        JMP     Ignore          ;InSysWindow
        JMP     Ignore          ;InContent
        JMP     DoDrag          ;InDrag
        JMP     DoGrow          ;InGrow
        JMP     DoGoAway        ;InGoAway

; Get menu selection:

DoMenuBar
        CLR.L   -(SP)           ;space for result
        PEA     EventRecord+evtMouse   ;Where
        _MenuSelect             ;Get menu selection
        MOVE    (SP)+,D6        ;Save menu number in D6
        MOVE    (SP)+,D7        ;Save item number in D7

        MOVE    #0,-(SP)
        _HiliteMenu             ;Remove highlight from menu title

        CMP     #FileID,D6      ;In the FILE menu?
        BNE     GetEvent        ;No, so branch
```

Listing 6-2. *continued*

```
        CMP     #2,D7           ;QUIT selected?
        BNE     @1              ;No, so branch

        _ExitToShell            ;Return to Finder

; Open the window:

@1      BSR     OpenWindow      ;Open window again
        RTS

; Drag the window:

DoDrag
        MOVE.L  OurWindow(A5),-(SP)
        MOVE.L  EventRecord+evtMouse,-(SP)   ;where
        PEA     boundsRect      ;bounding rectangle (constant)
        _DragWindow             ;Move window around screen
        RTS

DoGoAway
        CLR.B   -(SP)           ;Space for Boolean result
        MOVE.L  OurWindow(A5),-(SP)
        MOVE.L  EventRecord+evtMouse,-(SP)   ;where
        _TrackGoAway
        TST.B   (SP)+           ;Is the result true?
        BNE     @1              ;Yes, so branch and close
        RTS

@1      MOVE.L  OurWindow(A5),-(SP)
        _DisposWindow           ;Get rid of window

; Enable the "open window" item:

        MOVE.L  FileHndl(A5),-(SP)
        MOVE    #1,-(SP)
        _EnableItem
        RTS


* Track the mouse in the grow box until the button is released.
* Then redraw the window with its new size.
```

Listing 6-2. *continued*

```
DoGrow
        CLR.L   -(SP)           ;Space for result
        MOVE.L  OurWindow(A5),-(SP)
        MOVE.L  EventRecord+evtMouse,-(SP)   ;where
        PEA     sizeRect
        _GrowWindow
        MOVE.L  (SP)+,D6        ;Get new height, width

        BEQ     @1              ;Branch if size didn't change

; Resize and accumulate all of new window into update region.
; When the update event is handled, the window contents are
; erased and the grow box is redrawn.

        MOVE.L  OurWindow(A5),-(SP)
        MOVE.L  D6,-(SP)        ;New dimensions
        MOVE.B  #-1,-(SP)       ;-1 = create update events
        _SizeWindow             ;Redraw window with new size

        MOVE.L  OurWindow(A5),A0
        PEA     PortRect(A0)    ;New window rectangle
        _InvalRect              ;Force update of entire window

@1      RTS

; Create and draw a window on the screen with grow box:

OpenWindow
        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file

        MOVE.L  (SP),-(SP)      ;Replicate pointer on stack
        MOVE.L  (SP),OurWindow(A5)   ;Save pointer for later
        _DrawGrowIcon           ;Draw the grow box

; The next step ensures that our new window is the active
; drawing window. The pointer to the window is already on the
; stack.
```

Listing 6-2. *continued*

```
        _SetPort                ;Make window the active GrafPort

; Disable inapplicable menu item:

        MOVE.L  FileHndl(A5),-(SP)
        MOVE    #1,-(SP)
        _DisableItem            ;Disable "open window" item
        RTS


; The application constants:

EventRecord     DCB.B   EvtBlkSize,0    ;Space for event record

ClickWindow     DC.L    0               ;Pointer to window

boundsRect      DC.W    30,30,340,500   ;Drag rectangle

sizeRect        DC.W    30,200,327,490  ;h,w (min)  h,w (max)

; The application variables:

OurWindow       DS.L    1       ;Pointer to our window

FileHndl        DS.L    1       ;Handle to file menu
```

```
; Linker Control File
; MainWind.Link
;
; Link this file to create application
;  (without resources).
MainWind
$
```

```
* RMaker Source File
* MainWind.R
*
* Compile this after assembling and linking MainWind.Asm
*
* The next command appends the resources to the application:
!Book:MainWind
```

Listing 6-2. *continued*

```
Type MBAR = GNRL          ;;Menu bar resource
,128
.I
2                         ;;Two menus
1                         ;;ID of 1st menu
2                         ;;ID of 2nd menu

Type MENU
,1                        ;;Resource ID
14                        ;;Title is the Apple symbol (ASCII $14)
About this demo...        ;;About box

,2                        ;;Resource ID
File                      ;;Menu Title
  Open Window
  Quit

Type WIND
,128                      ;;Resource ID
Window Demo               ;;Title for Window
40 5 250 400              ;;Window coordinates (TLBR)
Visible GoAway            ;;Visible window/ goaway box
0                         ;;Window ID. 0 = document window
0                         ;;User-definable item (not used)
```

The only subroutine in this program that requires more explanation than found in the program's comments is DoGrow, the one that handles activity in the size box. If the window is to be resized (_GrowWindow returns a nonzero result), _SizeWindow is called to redraw the window with its new size. The entire content region of the window is then accumulated into the window's update region by calling _InvalRect. This causes the next update event to act on the entire window. In this program, update events erase the entire window.

If the entire window was not made invalid like this, and the window was enlarged, the screen clearing operation would not erase the old scroll control shafts and size box because _SizeWindow only places the newly exposed areas of the window into the update region.

The Window Title

The Window Manager has two instructions you can use to get the title of a window or set the title of a window: _GetWTitle and _SetWTitle. To get a title, use the following portion of code:

```
            MOVE.L  theWindow(A5),-(SP) ;Push window pointer
            PEA     TitleString(A5)     ;Push address of string
            _GetWTitle

TitleString DS  40                      ;String returned here
```

where TitleString is the address of a block of memory in the variable space where the string representing the window's title will be stored. The string is returned with a preceding length byte. The 40 bytes reserved with the DS 40 directive should be enough for the longest title string you're likely to use.

The calling sequence for setting a title is similar:

```
MOVE.L theWindow(A5),-(SP) ;Push window pointer
PEA    'New Title'         ;[this pushes an address]
_SetWTitle
```

Notice how the PEA instruction is used here. Although its operand is a string constant, the MDS assembler converts the operand to the address at which the assembler stores the string. This will be somewhere after the end of the program code space in the area reserved for constants. This form of PEA is equivalent to an instruction sequence of the form:

```
          PEA    MyString
          NOP

MyString DC     'New Title'
```

provided that the STRING_FORMAT directive is set to 3 so that the DC string is preceded by a length byte.

Displaying Text

Once you've created a window, you can easily display text in it using several QuickDraw instructions designed for that purpose. (See Table 6-3.) Unlike most computers, the text characters can be displayed in a variety of typefaces, styles, and sizes, thus you can craft the appearance of your output very carefully.

Table 6-3. Trap Instructions Used to Draw text.

_CharWidth — **Returns the width of a character in pixels.**

```
CLR     -(SP)               ;INTEGER: space for result
MOVE    #theChar,-(SP)-     ;CHAR: character to test
_CharWidth
MOVE    (SP)+,D0            ;Result: width of character
```

_DrawChar — **Draws a character at the current pen position.**

```
MOVE    #theChar,-(SP)      ;CHAR: character to draw
_DrawChar
```

_DrawString — **Draws a character string at the current pen position.**

```
PEA     theString           ;POINTER: to the string
_DrawString
```

_DrawText — **Draws a sequence of characters at the current pen position.**

```
PEA     theText             ;POINTER: to a sequence of
                            ;         characters
MOVE    #firstChar,-(SP)    ;INTEGER: Position of 1st
                            ;         character to draw
MOVE    #charCount,-(SP)    ;INTEGER: Number of characters
                            ;         to draw
_DrawText
```

Table 6-3. *continued*

_GetFontInfo — **Returns the characteristics for the current font.**

```
PEA     info                ;VAR: font information record
_GetFontInfo
```

The font information record is four words long. The offsets to its fields are given by ascent, descent, widMax, and leading (all integers).

_GetPen — **Returns the current pen position.**

```
PEA     penLoc              ;POINTER: to a point structure
_GetPen
```

_Move — **Moves the pen relative to its current position.**

```
MOVE    #horiz,-(SP)        ;INTEGER: horizontal movement
MOVE    #vert,-(SP)         ;INTEGER: vertical movement
_Move
```

_MoveTo — **Moves the pen to an absolute position.**

```
MOVE    #horiz,-(SP)        ;INTEGER: horizontal position
MOVE    #vert,-(SP)         ;INTEGER: vertical position
_MoveTo
```

_ScrollRect — **Scrolls the bits within a rectangle.**

```
PEA     theRect             ;POINTER: to scroll rectangle
MOVE    #hScroll,-(SP)      ;INTEGER: horizontal distance
MOVE    #vScroll,-(SP)      ;INTEGER: vertical distance
MOVE.L  updateRgn,-(SP)     ;HANDLE: to update region
_ScrollRect
```

To scroll down and to the left, use positive scrolling distances. To scroll up and to the right, use negative scrolling distances. The newly exposed area is cleared to the window's background color and is added to the update region.

Table 6-3. *continued*

_StringWidth — **Returns the width of a character string.**

```
CLR   -(SP)              ;INTEGER: space for result
PEA   theString          ;POINTER: to the string
_StringWidth
MOVE   (SP)+,D0          ;Result: width of string
```

_TextFace — **Sets the character style.**

```
MOVE   #typeStyle,-(SP)  ;INTEGER: style word
_TextFace
```

_TextFont — **Selects the current drawing font.**

```
MOVE   #fontNumber,-(SP) ;INTEGER: font number
_TextFont
```

_TextMode — **Sets the source transfer mode for character drawing.**

```
MOVE   #mode,-(SP)       ;INTEGER: text transfer mode
_TextMode
```

_TextSize — **Sets the point size for the current font.**

```
MOVE   #pointSize,-(SP)  ;INTEGER: type size in points
_TextSize
```

_TextWidth — **Returns the width of a sequence of characters.**

```
CLR   -(SP)              ;INTEGER: space for result
PEA   theText            ;POINTER: to the text
MOVE   #firstChar,-(SP)  ;INTEGER: Position of 1st
                         ;         character in text
MOVE   #charCount,-(SP)  ;INTEGER: Number of characters
                         ;         to measure
_TextWidth
MOVE   (SP)+,D0          ;Result: width of text
```

The word **font** is used to describe a group of characters having the same general typeface and size. On the Macintosh, font definitions are resources of type FONT and are

usually stored in the resource fork of the System file so they are available to any application. A standard System file contains a great many fonts, including the system font (Chicago 12), and the default application font (Geneva 12). Others can be removed (to save disk space) or added with an Apple utility program called Font/DA Mover.

The size of a font is measured in a unit called **points** reflecting the height of the matrix in which the characters are defined and drawn. A point is roughly one seventy-second of an inch, so each character in a 12 point font, for example, is roughly one sixth of an inch high.

Each character in a font is defined within an imaginary rectangle, called the **font rectangle**, which encloses the pixels used by the largest character in the font. (See Figure 6-3.) (Another rectangle, called the **character rectangle**, is the smallest rectangle enclosing the outline of the character.) Each of these pixels may be on or off.

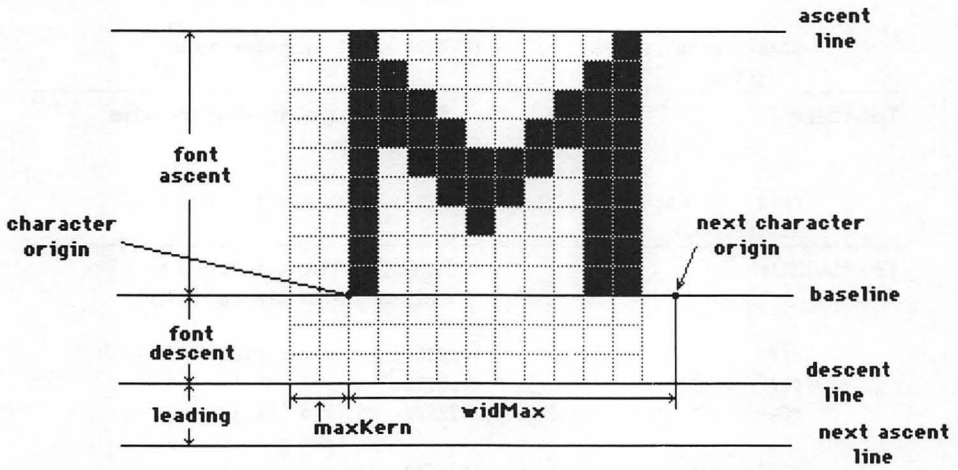


Figure 6-3. The Characteristics of a Macintosh Font.

A character is positioned relative to two landmarks within the font rectangle: the **character origin** and the **baseline**. This is done to ensure that the characters will line up smoothly on

the same line. The baseline is an imaginary line on which the character is written. It serves much the same purpose as a line on a page of notebook paper.

The **ascent** of a font is the number of pixels above the baseline and below the ascent line. The ascent line for a font is located just above the highest pixel of the tallest character in the font. Most characters occupy only the area between the ascent line and the baseline.

The **descent** of a font is the number of pixels below the baseline and above the descent line. The descent line is located just below the lowest pixel used by any character in the font. The pixels between these two lines hold the descenders of letters such as g, j, p, q, and y.

The **leading** of a font is the number of pixels between the descent line of one row of characters and the ascent line of the next row below. This means the number of pixels between two adjacent baselines is equal to ascent plus descent plus leading.

The **kern** is the number of pixels between the character origin and the left edge of the font rectangle. There is usually at least two columns of blank kern, so there will always be white space between adjacent characters, even if the previous character is the widest one in the font.

You should also be familiar with the quantity called **widMax**. This represents the maximum width of a character in the font, and is simply the number of pixels between two adjacent character origins. WidMax is an important attribute to know because you'll use it to quickly check whether you've got room to display a character on the current line in a window.

The _GetFontInfo instruction returns the characteristics of the currently active font in a **font information record**. The four items in a font information record have the symbolic offsets of ascent, descent, widMax, and leading. They are all integers.

_GetFontInfo takes a pointer to the record as a parameter, then fills that record with the values appropriate to the current font:

```
            PEA     FontInfo    ;Address of record
            _GetFontInfo

FontInfo    DCB.W   4,0         ;Four words in record
```

The individual elements within the FontInfo record can be accessed using the fixed offsets referred to above, and are defined in the MDS symbol definition files. Here's how to calculate the distance between rows for a given font:

```
MOVE    FontInfo+ascent,D0
ADD     FontInfo+descent,D0
ADD     FontInfo+leading,D0
```

This sequence merely adds together the ascent, descent, and leading fields in the FontInfo record and stores the result in D0.

Positioning the Pen

Now that you've seen how a character is defined, let's see how to draw one in a window. The first thing to do is set the current drawing location to a position within the window. For obvious reasons, this location is called the **pen position**.

When you first begin to draw text in the window, you will probably want to use the _MoveTo instruction to move to a particular horizontal and vertical position without drawing anything. For example, to move to location (50,75), use the following instructions:

```
MOVE    #50,-(SP)       ;Horizontal position
MOVE    #75,-(SP)       ;Vertical position
_MoveTo                 ;Move the pen
```

Notice that the coordinates passed to _MoveTo are *local* coordinates. Recall also that when a window is first created, the (0,0) local coordinate refers to the top left-hand corner of its content region and that the coordinates increase to the right (horizontal) and down (vertical).

Use the _Move instruction to move the pen to a position relative to its current position. For example, to move the pen position down 10 pixels and five pixels to the right, use the following instructions:

```
MOVE    #5,-(SP)        ;Horizontal distance to move
MOVE    #10,-(SP)       ;Vertical distance to move
_Move
```

The horizontal pen position is automatically advanced when you draw characters or strings of characters using the subroutines you'll see in the next section. This means you don't have to explicitly set it after every drawing operation.

If you ever want to know exactly where your pen is, use the _GetPen instruction:

```
        PEA     penLoc          ;addr. of record used by _GetPen
        _GetPen

penLoc  DC.L    0               ;this is a point: (h,v)
```

On return from _GetPen, the pen position is stored as a point at penLoc. Being a point record, you can access the coordinates separately by accessing the words at penLoc+v (vertical) and penLoc+h (horizontal).

Here's a subroutine you can call to simulate the effect of a carriage return/line feed operation. It moves the pen position to the left side of the next character row on the screen. To do this, it first calculates the new vertical position by reading the current pen location and then adding ascent plus descent plus leading to its vertical component:

```
        PEA     penLoc
        _GetPen                 ;Get current location
        PEA     FontInfo
        _GetFontInfo            ;Need ascent, leading
        MOVE    penLoc+v,D0     ;Get current vertical
        ADD     FontInfo+ascent,D0
        ADD     FontInfo+descent,D0
        ADD     FontInfo+leading,D0
```

```
            MOVE    #2,-(SP)        ;Horiz. pos. (left edge)
            MOVE    D0,-(SP)        ;Vert. pos (next line)
            _MoveTo
            RTS

penLoc      DC.L    0               ;this is a point: (h,v)
FontInfo    DCB.W   4,0             ;Four words in record
```

Notice that I've set the left edge of the line to position 2 rather than position 0. This was done so there would be room to display any pixels in the kern area of the character rectangle. The pen position always represents the character origin, therefore by setting the horizontal position to 2, you have room for two columns of kern pixels.

It's up to you to ensure that you've got room to display another row of characters in the window before calling the carriage return/line feed subroutine. You can do this by precalculating the new pen position, adding the font descent value to it, and comparing the result to the bottom coordinate of the window that is stored at offset PortRect+bottom from the start of the window record. If the new position is larger, you've run out of room.

Setting Text Characteristics

When a window is first created and then selected with _SetPort, a set of default text characteristics is initialized: the font to be used for drawing (Geneva), the style or typeface in which to draw the font (normal), and the size of the font (12-point). With the instructions described in this section, you can override these defaults.

To change the font used, pass a font number to the _TextFont instruction:

```
MOVE    #fontNumber,-(SP)
_TextFont
```

The symbolic names for the various font numbers are shown in Table 6-4. With the exception of font numbers 0 and

1, these numbers are related to the resource ID number as follows: The ID number is 128 times the font number plus the size of the font, in points. Thus, the resource ID for a 24-point London font (ID=6) is 792 (792 = 6*128 + 24).

Table 6-4. The Symbolic Names for the Font Numbers Passed to _TextFont.

Symbolic Name	*Font Number*
sysFont	0
applFont	1
newYork	2
geneva	3
monaco	4
venice	5
london	6
athens	7
sanFran	8
toronto	9
cairo	11
losAngeles	12
times	20
helvetica	21
courier	22
symbol	23
mobile	24

Font number zero (SysFont) refers to the system font used for such things as drawing the menu bar and window titles; this font is also called Chicago. Font number one (ApplFont) refers to the default application font, Geneva. This is the font used to draw text within windows if you haven't specifically selected another font.

You may also want to change the size of the font from time to time. To do this, use _TextSize by passing the point size on the stack as follows:

```
MOVE    #24,-(SP)       ;Select 24-point
_TextSize
```

If no font resource for that size is available, the resource for another size in the same font type will be scaled. The scaled characters will look best if the scaling factor is an even multiple. If a point size of zero is selected, the font resource having a size closest to the system font size (12) will be selected.

Another text feature you can select is the style of typeface of the characters to be displayed. The style attributes are not part of the font resource file. They are added to the "raw" characters by QuickDraw as the characters are drawn. To select the style of the text, pass a style word on the stack to _TextFace:

```
        MOVE    Style(A5),-(SP)
        _TextFace

Style   DS.W    1
```

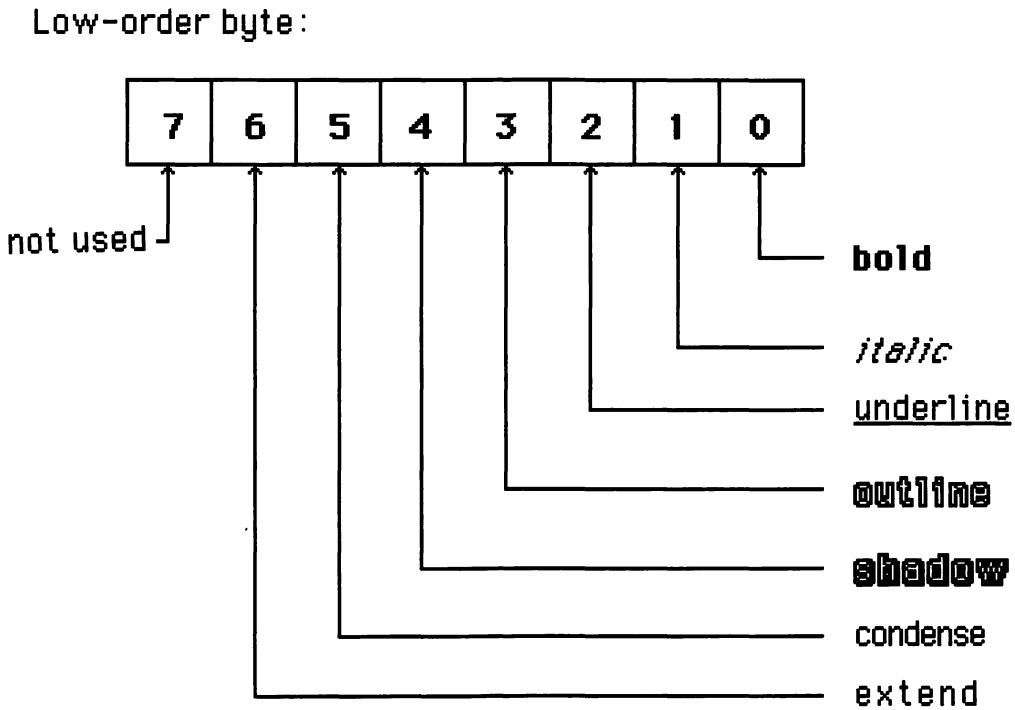
As shown in Figure 6-4, only the low-order seven bits in the style word are used, and each has a symbolic name associated with it that reflects the style attribute it controls. You can set any combination of these bits using the BSET instruction, to mix and match the basic style attributes:

```
CLR     Style(A5)               ;Normal style
BSET    #outlineBit,Style(A5)   ;Set outline bit
BSET    #ulineBit,Style(A5)     ;Set underline bit
```

The above instructions configure the Style variable for characters that are both outlined and underlined.

There is one other drawing characteristic you may want to set up before you draw characters in a window: the **source transfer mode**. This mode governs how the pixels within a character rectangle are logically combined with the corresponding pixels on the writing surface in the window to form the pixel actually placed on the screen. The eight different transfer modes are summarized in Figure 6-5.

The default transfer mode is srcOr, which means the two rectangles are superimposed to generate the result. This



Symbolic names for the bits in the style word:

Name	Bit #
BoldBit	0
ItalicBit	1
UlineBit	2
OutlineBit	3
ShadowBit	4
CondenseBit	5
ExtendBit	6

Figure 6-4. The Style Word Used with _TextFace.

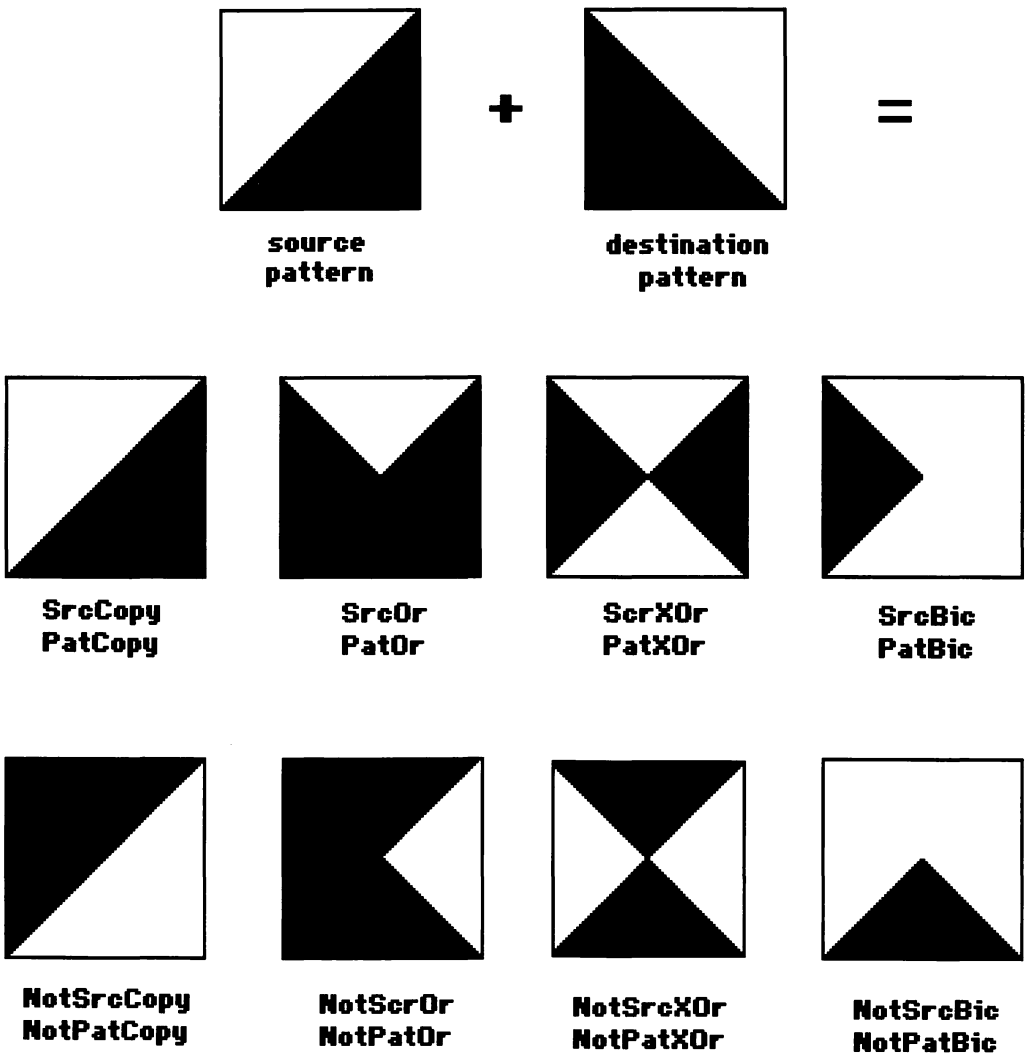


Figure 6-5. Source Transfer Modes and Pattern Transfer Modes for Text and Graphics Operations.

means any pixels in the destination rectangle that are below black pixels in the character rectangle will be forced to black; pixels below white pixels are not affected. For obvious reasons, srcOr is called an **overlay transfer mode**.

Contrast this with srcCopy where whatever is in the character rectangle replaces what's in the destination rectangle; srcXor, where screen pixels beneath black character pixels are inverted; and srcBic, where screen pixels beneath black character pixels are erased to white. The other four transfer modes are notsrcCopy, notsrcOr, notsrcXor, and notsrcBic. They all involve inverting the pixels in the destination rectangle before performing the combination calculation.

You'll probably never have to use a mode other than srcOr, unless you're overwriting a non-white background and want to ensure that the text is readable. For example, if the background is black, you can select the srcXor mode and the characters will appear in white; you can't use srcOr because you wouldn't see anything drawing black on black.

Here's how to change the source transfer mode to srcXor:

```
MOVE    #srcXor,-(SP)
_TextMode
```

Notice that "#" must precede the srcXor symbol because it is a constant, not a memory location.

Drawing Text

Well, we're finally ready to actually display something in a window! There are only three basic instructions for doing this: _DrawChar, _DrawString, and _DrawText.

Use _DrawChar if you simply want to display a single character on the screen. Here's how you would display the letter "a" at the current pen position:

```
MOVE    #'a',-(SP)    ;Push character on stack
_DrawChar
```

If you want to display a string of characters, it's much more convenient to use _DrawString rather than to make repeated calls to _DrawChar for each character:

```
            STRING_FORMAT 3         ;DC strings have length byte

            PEA     MyString        ;A string constant
            _DrawString             ;Draw it!

MyString    DC.B    'Hello world'   ;(MDS inserts length byte)
```

Notice that the string is preceded by a length byte that is automatically inserted by MDS because the STRING_FORMAT directive is 3.

Another way to print a string is to specify the string as the operand of the PEA instruction:

```
PEA     'Hello world'
_DrawString
```

In this case, MDS actually stores the string bytes at the end of the program code.

Use the third character drawing subroutine, _DrawText, to print any sequence of characters within a data structure. The sequence must not begin with a length byte because you pass the length explicitly:

```
        MOVE.L  textPtr(A5),-(SP)   ;Pointer to a block of text
        MOVE    #40,-(SP)           ;Start at byte #40
        MOVE    #22,-(SP)           ;Print 22 characters
        _DrawText

textPtr DS.L    1
```

Note that the first byte in a text string is byte zero, so byte 40 actually describes the forty-first byte.

This method of drawing text is most useful when you're accessing a group of fixed length messages.

Spacing Control

Use _CharWidth, _StringWidth, and _TextWidth to determine the width (in pixels) of the characters or text you would print with _DrawChar, _DrawString, and _DrawText, respectively. The arguments are passed to the Width instructions in the same way as the corresponding Draw instructions, but you must use a CLR -(SP) instruction to clear space for an integer result first. After the call, be sure to pop the width result. You'll use the width instructions in situations where you want to ensure that you won't write past the right edge of a window.

Example Programs Using Text Handling Instructions

Listing 6-3 shows a program that uses many of Quickdraw's text handling instructions. It uses _TextFont to set the font to SysFont (Chicago) and _TextSize to set the font size to 12-point. An underlined font style is then selected by setting the underline bit in a style word and passing it to _TextFace.

Listing 6-3. The Source File and Linker Control File for the Text Program.

```
* Asm Source File
* Text.Asm
*
* This program shows how to display text in a window.


   STRING_FORMAT   3       ;Precede DC strings with length

   INCLUDE ToolEqu.D       ;Toolbox equates
   INCLUDE QuickEqu.D      ;QuickDraw equates
   INCLUDE SysEqu.D        ;Operating system equates
   INCLUDE Traps.D         ;Trap instructions
```

Listing 6-3. *continued*

```
; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QD globals area
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; Draw a window on the screen:

        CLR.L   -(SP)           ;Space for returned pointer
        MOVE.L  #0,-(SP)        ;0 = store window in stack
        PEA     Window          ;Window rectangle
        PEA     'Text Demo'     ;Window Title
        MOVE.B  #-1,-(SP)       ;-1 = visible
        MOVE    #documentProc,-(SP)   ;Standard window type
        MOVE.L  #-1,-(SP)       ;-1 = front window
        MOVE.B  #0,-(SP)        ;0 = no go away button
        MOVE.L  #0,-(SP)        ;refCon
        _NewWindow              ;Draw the window
        MOVE.L  (SP),WindPtr(A5)   ;Save pointer to window
        _SetPort                ;Make window active port

; Here are some text drawing instructions:

        MOVE    #SysFont,-(SP)  ;Select system font (Chicago)
        _TextFont

        MOVE    #12,-(SP)       ;12 point text
        _TextSize

        CLR     D0              ;Clear style bits to 0
        BSET    #ulineBit,D0    ;Set underline style bit
        MOVE    D0,-(SP)
        _TextFace               ;Select the style (typeface)

        MOVE    #20,-(SP)       ;h
```

Listing 6-3. *continued*

```
        MOVE    #30,-(SP)       ;v
        _MoveTo                 ;Move to vertical position 30

        LEA     TheString,A6    ;Get EA of string to use
        BSR     Center          ;Draw string in center of window

        BSR     DoBeep

        RTS                     ;Return to Finder


* This subroutine centers a string on the current line.
* On entry, A6 points to the string.

Center
        MOVE.L  WindPtr(A5),A0
        MOVE    PortRect+right(A0),D6  ;Right edge of window
        MOVE    PortRect+left(A0),D5   ;Left edge of window
        SUB     D5,D6           ;Width of window in D6

        CLR     -(SP)           ;Space for result
        MOVE.L  A6,-(SP)        ;Push pointer to string
        _StringWidth            ;Get width of string
        MOVE    (SP)+,D0        ;Pop the result

        SUB     D0,D6           ;Calculate size of white space
        LSR     #1,D6           ;Divide by 2 to get left size
        ADD     D5,D6           ;Add left size to left edge

        PEA     penLoc
        _GetPen
        MOVE    penLoc+v,D4     ;Get vertical coordinate

        MOVE    D6,-(SP)        ;h d6
        MOVE    D4,-(SP)        ;v d4
        _MoveTo                 ;Move to proper position

        MOVE.L  A6,-(SP)        ;Pointer to string
        _DrawString             ;Draw the string

        RTS


* Beep, wait for a mouse click, then clear screen:
```

Listing 6-3. *continued*

```
DoBeep  MOVE    #30,-(SP)       ;1/2 second beep
        _SysBeep

GetMyEvent
        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1,-(SP)       ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result flag
        BEQ     GetMyEvent      ;Branch if null event

        MOVE    EventRecord+evtNum,D0   ;Get event type
        CMP     #mButDwnEvt,D0  ;Is it a button-down event?
        BNE     GetMyEvent      ;No, so loop

        MOVE.L  WindPtr(A5),A0
        PEA     PortRect(A0)    ;PortRect contains window coordinates
        _EraseRect              ;Erase the content region

        RTS

; Here are the program constants:

EventRecord     DCB.B   EvtBlkSize,0    ;Space for event record

Window          DC.W    50,50,300,450   ;window rectangle

penLoc          DC.L    0       ;Pen position

TheString       DC      'This string is centered and underlined'

; The program variables begin here:

WindPtr         DS.L    1       ;Pointer to window
```

```
; Linker Control File
; Text.Link
;
Text
$
```

The main subroutine in the program is called **Center**. It centers the display of a string on the current line. When you call it, A6 must point to a standard string (one that is preceded by a length byte).

In the example, the string is defined with a DC directive. Since the default MDS format for such a string is text with no length or trailing 0 byte, STRING_FORMAT is set equal to 3 at the beginning of the program. This directs the assembler to include the preceding length byte, as required by _DrawString.

Center determines where to start drawing the text string by first calculating the width of the window rectangle. It does this by subtracting the left edge, stored PortRect+left bytes into the window record from the right edge, stored at an offset of PortRect+right. It then uses _StringWidth to get the width of the string in pixels, and subtracts it from the window width. This yields the width of the unused part of the line. By dividing this number by two and adding it to the left position of the window, the program determines the horizontal position at which to begin drawing.

To get the vertical position, it first calls _GetPen to determine the current pen location. The vertical coordinate is located v bytes into the point record returned by _GetPen. The program then calls _MoveTo to position the pen and draws the string with _DrawString.

Another interesting program is shown in Listing 6-4. It demonstrates how to display characters in a window without drawing past its right edge or below its bottom. If there isn't enough room on the right side, the program draws the character on the left side of the next line. If you're already on the last line, the contents of the window are scrolled up one line, clearing a new bottom line in the process. Scrolling also occurs if you press RETURN while on the bottom line.

Listing 6-4. The Source File, Linker Control File, and RMaker File for the Scroll Program.

```
; Asm Source File
; Scroll.Asm
; This program shows how to display text in a window.
; The window is scrolled if RETURN is pressed on the bottom line.
```

Listing 6-4. *continued*

```
WindID          EQU     128     ;Window ID
 FileID         EQU     2       ;Menu ID for File menu

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QuickDraw globals
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor
        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; Create and draw a window on the screen:

        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file
        _SetPort                ;Make the window the current

; Create the menu bar and display it:

        CLR.L   -(SP)
        MOVE    #128,-(SP)      ;Menu bar ID
        _GetNewMBar             ;Load from resource file
        _SetMenuBar             ;Make it current
        _DrawMenuBar            ;Draw it

;Move the pen to the start of the first line:
```

Listing 6-4. *continued*

```
        PEA     InfoRecord
        _GetFontInfo            ;Get font characteristics

        MOVE    #2,-(SP)        ;Start at left edge
        MOVE    InfoRecord+ascent,-(SP) ;Leave room for ascent
        _MoveTo

GetEvent
        JSR     DoCursor        ;Display a cursor

GetEvent1
        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #$FFFF,-(SP)    ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result flag
        BEQ     GetEvent1       ;Branch if no pending event


        MOVE    EventRecord+evtNum,D0   ;Get event type code
        CMP     #KeyDwnEvt,D0   ;Key-down event?
        BEQ     DoKeyDown       ;Yes, so branch

        CMP     #AutoKeyEvt,D0  ;Key repeat?
        BEQ     DoKeyDown       ;Yes, so branch

        CMP     #mButDwnEvt,D0  ;Is it a button-down event?
        BNE     GetEvent1       ;No, so branch

; Handle mouse button down event:

        CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)   ;Where info
        PEA     ClickWindow     ;VAR window involved
        _FindWindow             ;Where was button pressed?
        CMPI    #InMenuBar,(SP)+   ;Pressed in menu bar?
        BNE     GetEvent1       ;No, so ignore

; See if QUIT was selected from File menu:

        CLR.L   -(SP)           ;space for result
        PEA     EventRecord+evtMouse   ;Where
```

Listing 6-4. *continued*

```
        _MenuSelect             ;Get menu selection
        MOVE    (SP)+,D6        ;Save menu number in D6
        MOVE    (SP)+,D0        ;Discard item number

        MOVE    #0,-(SP)
        _HiliteMenu             ;Remove highlight from title

        CMP     #FileID,D6      ;In the FILE menu?
        BNE     GetEvent1       ;No, so branch

        _ExitToShell            ;Return to Finder

; Handle the key-down event:

DoKeyDown
        MOVE    EventRecord+evtMessage+2,D6   ;Get ASCII code
        CMP.B   #32,D6          ;Is code >= 32?
        BHS     ShowChar        ;Yes, so branch

        CMP.B   #13,D6          ;Is it RETURN?
        BNE     @1              ;No, so ignore it

        JSR     CRLF            ;Advance to next line
        BRA     GetEvent

@1      MOVE    #10,-(SP)       ;Beep for control characters
        _SysBeep                ;Beep for 1/6 second
        BRA     GetEvent1

ShowChar
        PEA     penLoc
        _GetPen                 ;Get current pen position

        CLR     -(SP)
        MOVE    D6,-(SP)
        _CharWidth              ;Get width of character
        MOVE    (SP)+,D5
        ADD     penLoc+h,D5     ;D5 = new position if we draw

        PEA     ActiveWindow
        _GetPort                ;Get active drawing window
```

Listing 6-4. *continued*

```
        MOVE.L  ActiveWindow,A0
        CMP     PortRect+right(A0),D5   ;Past right edge?
        BLO     @1              ;It's lower, so branch

        JSR     CRLF            ;Advance to next line

@1      JSR     UndoCursor      ;Remove the cursor

        MOVE    D6,-(SP)
        _DrawChar               ;Draw the character
        BRA     GetEvent

* Move the pen to the left side of the next line,
* scrolling the screen, if necessary:

CRLF    JSR     UndoCursor      ;Remove cursor

        PEA     InfoRecord
        _GetFontInfo            ;Get font characteristics

        PEA     penLoc
        _GetPen                 ;Get pen position

        PEA     ActiveWindow
        _GetPort                ;Get active window

        MOVE    InfoRecord+ascent,D0
        ADD     InfoRecord+descent,D0
        ADD     InfoRecord+leading,D0   ;Get height of font
        MOVE    D0,Height(A5)

        MOVE.L  ActiveWindow,A4

        MOVE    penLoc+v,D0
        ADD     Height(A5),D0
        ADD     InfoRecord+leading,D0   ;Distance to next line
        CMP     PortRect+bottom(A4),D0  ;Room for next line?
        BHS     Scroll                  ;No, so scroll

        MOVE    penLoc+v,D0
        ADD     Height(A5),D0   ;new v
        MOVE    #2,-(SP)        ;h
        MOVE    D0,-(SP)        ;v
```

Listing 6-4. *continued*

```
        _MoveTo
        RTS

Scroll  CLR.L   -(SP)
        _NewRgn                 ;Get handle to new empty region
        MOVE.L  (SP)+,A6

        PEA     PortRect(A4)    ;Window rectangle
        MOVE    #0,-(SP)        ;Don't scroll horizontally
        MOVE    Height(A5),D0   ;# of pixels to scroll
        NEG     D0              ;Negative ==> scroll up
        MOVE    D0,-(SP)
        MOVE.L  A6,-(SP)        ;Handle to update region
        _ScrollRect             ;Scroll, clear bottom line

        MOVE    #2,-(SP)        ;h = left edge
        MOVE    penLoc+v,-(SP)  ;Keep the same v
        _MoveTo                 ;Move to left side

        MOVE.L  A6,-(SP)
        _DisposRgn              ;Destroy region (it's not used)

        RTS

; Display a black-box cursor:

DoCursor
        PEA     penLoc
        _GetPen                 ;Get pen position

        PEA     InfoRecord
        _GetFontInfo            ;Get font dimensions

        MOVE    penLoc+h,D0     ;Calculate left, right
        MOVE    D0,CursRect+left(A5)   ; coords of cursor box
        ADDQ    #8,D0           ;(8 pixels wide)
        MOVE    D0,CursRect+right(A5)

        MOVE    penLoc+v,D0     ;Calculate top, bottom
        MOVE    D0,CursRect+bottom(A5)   ; coords of cursor box
        SUB     InfoRecord+ascent,D0
        MOVE    D0,CursRect+top(A5)
```

Listing 6-4. *continued*

```
        PEA     CursRect(A5)
        PEA     AllBlack        ;Pointer to pattern
        _FillRect               ;Black box cursor
        RTS

; Erase the cursor rectangle:

UndoCursor
        PEA     CursRect(A5)
        _EraseRect              ;Remove the cursor
        RTS


; The application constants:

EventRecord     DCB.B   EvtBlkSize,0      ;Space for event record

ClickWindow     DC.L    0                 ;Pointer to window

penLoc          DC.L    0

ActiveWindow    DC.L    0

InfoRecord      DCB.W   4,0               ;Font information record

AllBlack        DC.B    $FF,$FF,$FF,$FF   ;A solid black pattern
                DC.B    $FF,$FF,$FF,$FF

; The application variables:

Height          DS      1       ;Height of font stored here

CursRect        DS.W    4       ;Cursor rectangle
```

```
; Linker Control File
; Scroll.Link
;
; Link this file to create an application
;  (without resources).

Scroll
$
```

Listing 6-4. *continued*

```
* RMaker Source File
* Scroll.R
*
* Compile this after assembling and linking Scroll.Asm
*
* The next command appends the resources to the application:
!Book:Scroll


Type MBAR = GNRL        ;;Menu bar resource
,128                    ;;Resource ID
.I                      ;;Decimal integers follow
2                       ;;Number of menus
1                       ;;ID of 1st menu
2                       ;;ID of 2nd menu

Type MENU
,1                      ;;Resource ID
\14                     ;;Title is the Apple symbol
  About this demo...    ;;About box
  (-                    ;;Dimmed line

,2                      ;;Resource ID
File                    ;;Menu Title
  Quit                  ;;Only item is Quit

Type WIND
,128                    ;;Resource ID
Text Entry Window       ;;Title for Window
40 5 332 502            ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)
```

The portion of the program that processes key-down events starts at DoKeyDown. It reacts to character codes corresponding to the RETURN key (code 13) or printable symbols only (codes 32 to 255); anything else causes the speaker to beep for one-sixth of a second.

The main subroutine in the program is called CRLF. It positions the pen on the left side of the next line in the window, scrolling the window contents if necessary. To determine if scrolling is necessary, it first adds the height of the font to the current vertical position, which is always on the character baseline, and to the leading. If the result is higher than or the same as the bottom coordinate of the window, a scrolling operation is needed and the BHS Scroll branch takes place (BHS is the same as BCC); otherwise, _MoveTo is used to move the pen to the left of the next line.

Scrolling is accomplished using _ScrollRect, a trap instruction called as follows:

```
PEA     theRect          ;Rectangle in which to scroll
MOVE    #hScroll,-(SP)   ;horizontal scroll distance
MOVE    #vScroll,-(SP)   ;vertical scroll distance
MOVE.L  updateRgn,-(SP) ;Handle to update region
_ScrollRect
```

The scrolling distances are in pixels. Positive values are used to scroll down (hScroll) and to the left (vScroll); negative values are used to scroll up (hScroll) and to the right (vScroll). The area exposed as a result of the scroll is cleared to the window's background color (usually white) and is accumulated in the region whose handle is given by updateRgn. The bits that disappear during the scroll are not saved. If necessary, you can force an update event to occur by passing this handle to _InvalRgn. An update handler would redraw the newly exposed portion of the data structure being displayed in the window. Since this program doesn't need to fill the bottom line with anything, it does not do this.

This program uses _NewRgn to get a handle to a new, empty region _ScrollRect can use. The calling sequence for _NewRgn is:

```
CLR.L  -(SP)             ;Space for result
_NewRgn
MOVE.L (SP)+,A0          ;Pop handle to region
```

Since we don't need the region filled by _ScrollRect, we destroy it by calling _DisposRgn before leaving the CRLF subroutine. If this wasn't done, the region would continue to grow as CRLF is called again and again.

The CRLF subroutine is called if RETURN is pressed, or if a character is pressed near the right edge of the window. The portion of the program beginning at ShowChar shows how to check that the horizontal pen position will not extend past the right edge of the window should a character be drawn. If there's room, _DrawChar is called right away; otherwise it is called after calling CRLF.

The program also has a subroutine called DoCursor that displays a solid black cursor on the screen. The cursor is a rectangle with a width of eight pixels and a height the same as the ascent for the font. The bottom left position of the rectangle coincides with the current pen position. The rectangle is filled with black using the _FillRect instruction described in the next section of this chapter.

The UndoCursor subroutine removes the cursor and is called before drawing a character or before moving the cursor with the CRLF subroutine. The cursor is erased using the _EraseRect instruction.

Handy Utilities

One of the more common things you will want to display on the screen is the numeric result of a calculation. To do this, you first have to convert the number, which is usually in binary form, to a string of ASCII characters representing the decimal representation of the number. Once you convert, it's an easy matter to display the string using the _DrawString instruction.

The Macintosh operating system has a standard instruction for converting binary numbers into an ASCII string. It is part of a **package** of standard subroutines accessed using the _Pack7 instruction:

```
        LEA     String(A5),A0   ;Put address of string in A0
        MOVE.L  #myNumber,D0    ;Put number in D0.L
        MOVE    #0,-(SP)        ;SELECTOR: 0 = number to string
        _Pack7

String  DS.W    16              ;Space for string
```

The word placed on the stack just before calling _Pack7 is called the routine selector because it determines which of the subroutines contained in the package is to take control. In this case, the selector 0 selects the number to string conversion.

If you prefer you can replace the two-instruction sequence:

```
MOVE    #0,-(SP)
_Pack7
```

with the easier-to-remember instruction, _NumToString. This instruction is really an MDS macro defined in the standard symbol file called PackMacs.txt. Use the INCLUDE directive to merge this file with your program source file.

There are only two valid _Pack7 routine selectors; the other is 1, which selects the opposite conversion, from string to binary number. The macro for this selection is _StringToNum. Before calling it, transfer the address of the string into the A0 register. The result is returned in D0.L.

Displaying Graphics

The Macintosh is probably more famous for its ability to display graphics than anything else. QuickDraw contains many, many instructions that can be used to draw points, rectangles, ovals, rounded-corner rectangles, arcs, regions, and pictures on the screen. In this section we'll look at a few of these instructions.

Table 6-5. Trap Instructions Used to Draw Graphics.

_ClosePgon — Closes the open polygon record.

```
_ClosePgon              ;(no parameters)
```

_EraseArc — Erases an arc.

```
PEA     inRect          ;POINTER: to arc's rectangle
MOVE    #startAngle,-(SP)   ;INTEGER: starting angle
MOVE    #arcAngle,-(SP)     ;INTEGER: extent of the arc
_EraseArc
```

_EraseOval — Erases an oval.

```
PEA     inRect          ;POINTER: to oval's rectangle
_EraseOval
```

_ErasePoly — Erases a polygon.

```
MOVE.L thePolygon,-(SP)     ;HANDLE: to the polygon
_ErasePoly
```

_EraseRect — Erases a rectangle.

```
PEA     theRect         ;POINTER: to the rectangle
_EraseRect
```

_EraseRoundRect — Erases a round-corner rectangle.

```
PEA     theRect             ;POINTER: to the rectangle
MOVE    #cornerWidth,-(SP)  ;INTEGER: width of corner oval
MOVE    #cornerHeight,-(SP) ;INTEGER: height of corner oval
_EraseRoundRect
```

_FillArc — Fills an arc with a pattern.

```
PEA     inRect          ;POINTER: to arc's rectangle
MOVE    #startAngle,-(SP)   ;INTEGER: starting angle
MOVE    #arcAngle,-(SP)     ;INTEGER: extent of the arc
PEA     fillPat         ;POINTER: to the fill pattern
_FillArc
```

Table 6-5. *continued*

_FillOval — **Fills an oval with a pattern.**

```
PEA    inRect              ;POINTER: to oval's rectangle
PEA    fillPat             ;POINTER: to the fill pattern
_FillOval
```

_FillPoly — **Fills a polygon with a pattern.**

```
MOVE.L thePolygon,-(SP)    ;HANDLE: to the polygon
PEA    fillPat             ;POINTER: to the fill pattern
_FillPoly
```

_FillRect — **Fills a rectangle with a pattern.**

```
PEA    theRect             ;POINTER: to the rectangle
PEA    fillPat             ;POINTER: to the fill pattern
_FillRect
```

_FillRoundRect — **Fills a round-corner rectangle with a pattern.**

```
PEA    theRect             ;POINTER: to the rectangle
MOVE   #cornerWidth,-(SP)  ;INTEGER: width of corner oval
MOVE   #cornerHeight,-(SP) ;INTEGER: height of corner oval
PEA    fillPat             ;POINTER: to the fill pattern
_FillRoundRect
```

_FrameArc — **Frames an arc.**

```
PEA    inRect              ;POINTER: to arc's rectangle
MOVE   #startAngle,-(SP)   ;INTEGER: starting angle
MOVE   #arcAngle,-(SP)     ;INTEGER: extent of the arc
_FrameArc
```

_FrameOval — **Frames an oval.**

```
PEA    inRect              ;POINTER: to oval's rectangle
_FrameOval
```

_FramePoly — **Frames a polygon.**

```
MOVE.L thePolygon,-(SP)    ;HANDLE: to the polygon
_FramePoly
```

Table 6-5. *continued*

_FrameRect — **Frames a rectangle.**

```
PEA     theRect             ;POINTER: to the rectangle
_FrameRect
```

_FrameRoundRect — **Frames a round-corner rectangle.**

```
PEA     theRect             ;POINTER: to the rectangle
MOVE    #cornerWidth,-(SP)  ;INTEGER: width of corner oval
MOVE    #cornerHeight,-(SP) ;INTEGER: height of corner oval
_FrameRoundRect
```

_GetPenState — **Returns the pen characteristics in a pen state record.**

```
PEA     curState            ;POINTER: to pen state record
_GetPenState
```

The pen state record is PSRec (18) bytes long.

_InverRect — **Inverts a rectangle.**

```
PEA     theRect             ;POINTER: to the rectangle
_InverRect
```

_InverRoundRect — **Inverts a round-corner rectangle.**

```
PEA     theRect              ;POINTER: to the rectangle
_InverRoundRect
```

_InvertArc — **Inverts an arc.**

```
PEA     inRect              ;POINTER: to arc's rectangle
MOVE    #startAngle,-(SP)   ;INTEGER: starting angle
MOVE    #arcAngle,-(SP)     ;INTEGER: extent of the arc
_InvertArc
```

_InvertOval — **Inverts an oval.**

```
PEA     inRect              ;POINTER: to oval's rectangle
_InvertOval
```

Table 6-5. *continued*

_InvertPoly — **Inverts a polygon.**

```
MOVE.L thePolygon,-(SP)    ;HANDLE: to the polygon
_InvertPoly
```

_KillPoly — **Destroys a polygon record and frees up the space it uses.**

```
MOVE.L thePolygon,-(SP)    ;HANDLE: to the polygon record
_KillPoly
```

_Line — **Draws a line to a position relative to the current pen position.**

```
MOVE   #horiz,-(SP)        ;INTEGER: horizontal movement
MOVE   #vert,-(SP)         ;INTEGER: vertical movement
_Line
```

_LineTo — **Draws a line to an absolute pen position.**

```
MOVE   #horiz,-(SP)        ;INTEGER: horizontal position
MOVE   #vert,-(SP)         ;INTEGER: vertical position
_LineTo
```

_NumToString — **Converts a binary number to an ASCII string.**

```
LEA    theString,A0        ;A0 = pointer to string
MOVE.L #theNumber,D0       ;D0.L = number to convert
MOVE   #0,-(SP)            ;INTEGER: 0 = _NumToString
_Pack7
```

_OpenPoly — **Creates an empty polygon record and opens it.**

```
CLR.L  -(SP)               ;HANDLE: space for result
_OpenPoly
MOVE.L (SP)+,A0            ;Result: handle to polygon
                           ;        record
```

Table 6-5. *continued*

_PaintArc	**Fills an arc with the pen pattern.**

`PEA inRect ;POINTER: to arc's rectangle`
`MOVE #startAngle,-(SP) ;INTEGER: starting angle`
`MOVE #arcAngle,-(SP) ;INTEGER: extent of the arc`
`_PaintArc`	

_PaintOval	**Fills an oval with the pen pattern.**
`PEA inRect ;POINTER: to oval's rectangle` `_PaintOval`	
_PaintPoly	**Fills a polygon with the pen pattern.**
`MOVE.L thePolygon,-(SP) ;HANDLE: to the polygon` `_PaintPoly`	
_PaintRect	**Fills a rectangle with the pen pattern.**
`PEA theRect ;POINTER: to the rectangle` `_PaintRect`	
_PaintRoundRect	**Fills a round-corner rectangle with the pen pattern.**
`PEA theRect ;POINTER: to the rectangle` `_PaintRoundRect`	
_PenMode	**Sets the new pen transfer mode.**
`MOVE #newMode,-(SP) ;INTEGER: new pen transfer mode` `_PenMode`	
_PenNormal	**Assigns default values to the pen size, pattern, and transfer mode.**
`_PenNormal ;(no parameters)`	

Table 6-5. *continued*

_PenPat — **Sets the new pen pattern.**

```
PEA     newPat              ;POINTER: to the new pattern
_PenPat
```

_PenSize — **Sets the new width and height of the graphics pen.**

```
MOVE    #newWidth,-(SP)     ;INTEGER: new pen width
MOVE    #newHeight,-(SP)
```

_StringToNum — **Converts an ASCII digit string to a binary number.**

```
LEA     theString,A0        ;A0 = pointer to string
MOVE    #1,-(SP)            ;INTEGER: 1 = _StringToNum
_Pack7                      ;Result is in D0.L
```

Setting Pen Characteristics

Just as when drawing text, you must first set up the pen characteristics before starting to draw graphic images in a window, if you're not satisified with the default values. The three instructions for doing this are _PenSize, _PenMode, and _PenPat. (See Figure 6-6.)

Use _PenSize to set the dimensions of the rectangular nib of the pen. The nib of the pen hangs down and to the right of the pen's position. The top-left corner of the pen is aligned with the pen coordinates. The standard dimensions of the pen nib are (1,1), which means that it consists of just one pixel. Here's how you would set the pen size to 10 pixels wide by 15 pixels high:

```
MOVE    #10,-(SP)       ;width
MOVE    #15,-(SP)       ;height
_PenSize
```

The _PenMode instruction sets the pattern transfer mode for graphic drawing operations. These modes are analogous

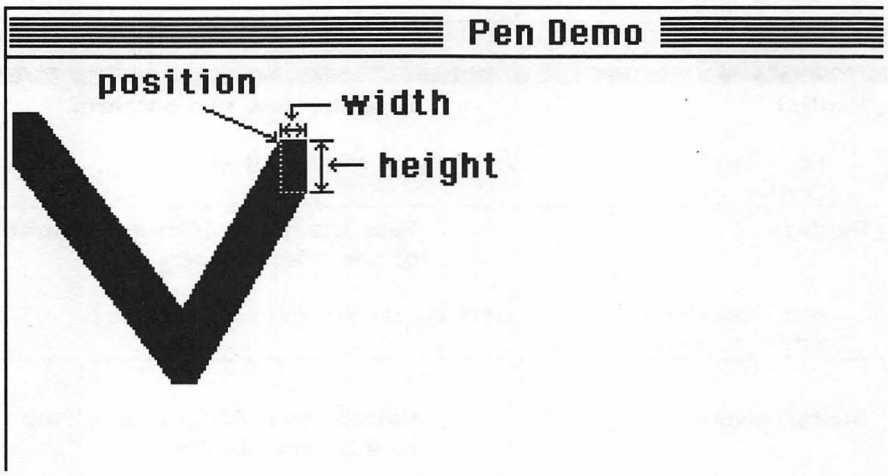


Figure 6-6. The Macintosh Drawing Pen.

to the source transfer modes described above that are used in connection with text drawing operations. The default transfer mode is patCopy, the overlay mode.

The _PenPat instruction is used to set the pattern of the "ink" that comes out of the pen as you draw. A pattern is simply an 8x8 rectangular bit image defined by a sequence of eight bytes in memory. As shown in Figure 6-7, each byte defines the image in one row of the pattern; as usual, a one bit corresponds to a black pixel, a zero bit to a white pixel.

To set the pen pattern, you must pass the address of the pattern's data structure on the stack:

```
        PEA     MyPattern
        _PenPat

MyPattern DC.B   $81,$42,$24,$18,$18,$24,$42,$81
```

The pattern defined is the X shown in Figure 6-7. The default pen pattern is a solid black image.

If you've been fiddling with the pen characteristics and want to restore the standard default conditions, use _PenNormal. This sets the pen size to (1,1), the pen pattern to

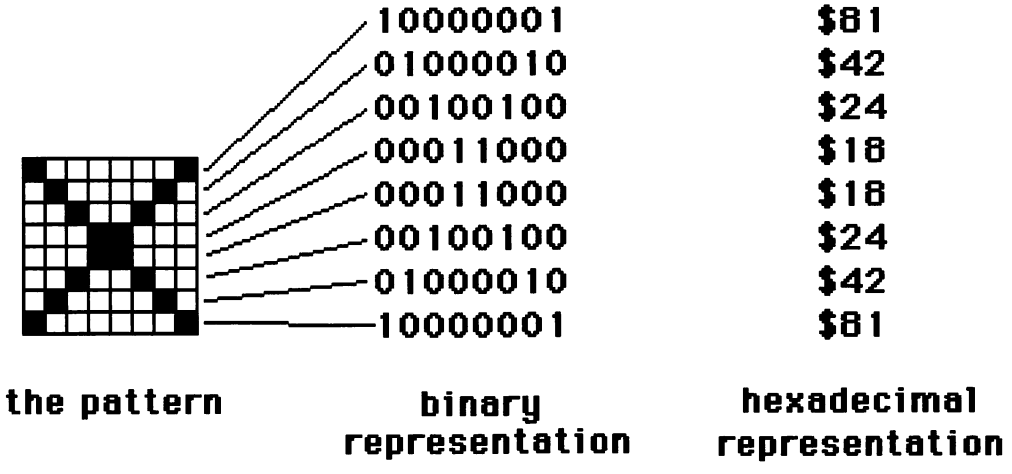


Figure 6-7. Defining a Pattern.

black, and a transfer mode of patCopy. _PenNormal requires no parameters.

If you don't know the current pen characteristics, use _GetPenState to return them in a pen state record:

```
            PEA   PenState          ;Address of record
            _GetPenState

PenState    DC.L    0               ;Point: location of pen
            DC.L    0               ;Width and height of pen
            DC.W    0               ;Pen transfer mode
            DCB.B   8,0             ;Pen pattern
```

The standard names for the offsets from PenState to the four elements in the pen state record are psLoc (0), psSize (4), psMode (8), and psPat (10).

Drawing Lines

To position the pen for a graphics drawing operation, use the _Move and _MoveTo instructions, just as you do when positioning it for text drawing operations. The pen does not

draw anything on the screen when these instructions are used.

To draw lines in the window, use the _Line and _LineTo instructions. _Line draws a line to a point that is expressed in terms of relative coordinates. The destination position is calculated by adding the current pen coordinates to the distances passed on the stack as follows:

```
MOVE    #13,-(SP)       ;horiz=horiz+13
MOVE    #22,-(SP)       ;vert=vert+22
_Line
```

If you wish to move to an absolute position in the window instead, you can use _LineTo like this:

```
MOVE    #50,-(SP)       ;horizontal position
MOVE    #60,-(SP)       ;vertical position
_LineTo                 ;Move to (50,60)
```

Both _Line and _LineTo automatically assign the current pen location to the destination location.

Drawing Shapes

QuickDraw has many instructions you can use to draw any of several classes of shapes on the screen. In this section I'll describe those used with **rectangles**, **round-corner rectangles**, **ovals**, **arcs**, and **polygons**.

There are five fundamental shape-drawing operations supported by the QuickDraw instructions:

Framing: drawing an outline of the shape.
Painting: filling a shape with the current pen pattern using the current pen transfer mode.
Erasing: filling a shape with the window's current background pattern (usually white). The transfer mode used is always patCopy.
Inverting: changing white pixels within the shape to black and vice versa.
Filling: filling a shape with a given pattern. The transfer mode used is always patCopy.

Let's see how to perform these operations on the most common QuickDraw shapes.

Rectangles

As you saw earlier in this chapter, a rectangle is a data structure made up of two points that define its top-left and bottom-right coordinates, in that order. This ordering is often abbreviated as "TLBR". For example, suppose the top-left and bottom-right coordinates of a rectangle are (10,20) and (200,300). The data structure for the rectangle would be:

```
Rectangle   DC.W    20,10,300,200   ;TLBR
```

The standard drawing instructions for rectangles are _FrameRect, _PaintRect, _EraseRect, _InverRect, and _FillRect. The first four of these are called by first pushing the address of the rectangle's data structure on the stack. _FillRect also requires you to push the address of the data structure containing the 8-byte pattern definition.

Ovals

The data structure for an **oval** is actually the same as a rectangle because the shape of the oval is dictated by the dimensions of a bounding rectangle in which it is inscribed. The standard drawing instructions are _FrameOval, _PaintOval, _EraseOval, _InvertOval, and _FillOval. They are called in the very same way as the corresponding instructions for rectangles.

Round-Corner Rectangles

A **round-corner rectangle** is a rectangle whose corners are rounded in the shape of small ovals that are invisibly inscribed in its corners. The curvature of the corner is dictated by the height and width of the oval's axes. The standard drawing

instructions are _FrameRoundRect, _PaintRoundRect, _EraseRoundRect, _InverRoundRect, and _FillRoundRect. Before you call them, you must first push the address of the rectangle's data structure and the width and height of the axes of the corner oval. For _FillRoundRect you must also push the address of the data structure for the fill pattern.

Arcs

An **arc** is simply a portion of the outline of an oval. (See Figure 6-8.) It is defined in terms of the bounding rectangle for the oval; the angle at which the arc begins, measured clockwise from the positive vertical axis of the oval; and the angular extent of the arc.

Note that all angles are expressed in degrees (from 0 to 359), not radians. In addition, since the angles to the corners of the rectangle are always deemed to be 45, 135, 225, and 315, respectively, the angles used by the arc instructions are not true circular degrees unless the rectangle is a square.

The standard drawing instructions are _FrameArc, _PaintArc, _EraseArc, _InverArc, and _FillArc. The parameters you must pass to them on the stack are, in order: the address of the rectangle's data structure, the starting angle, and the angular extent. If you use _FillArc, you also have to push the address of the data structure that defines the fill pattern.

The area affected by the paint, erase, invert, and fill operations is actually the wedge bounded by the arc and the two radial lines extending from the center of the oval to the starting and end points of the arc.

Polygons

A **polygon** is a relatively complex data structure made up of a bounding rectangle and a series of points representing the vertices of the shape. To create a polygon, you first call _OpenPoly as follows:

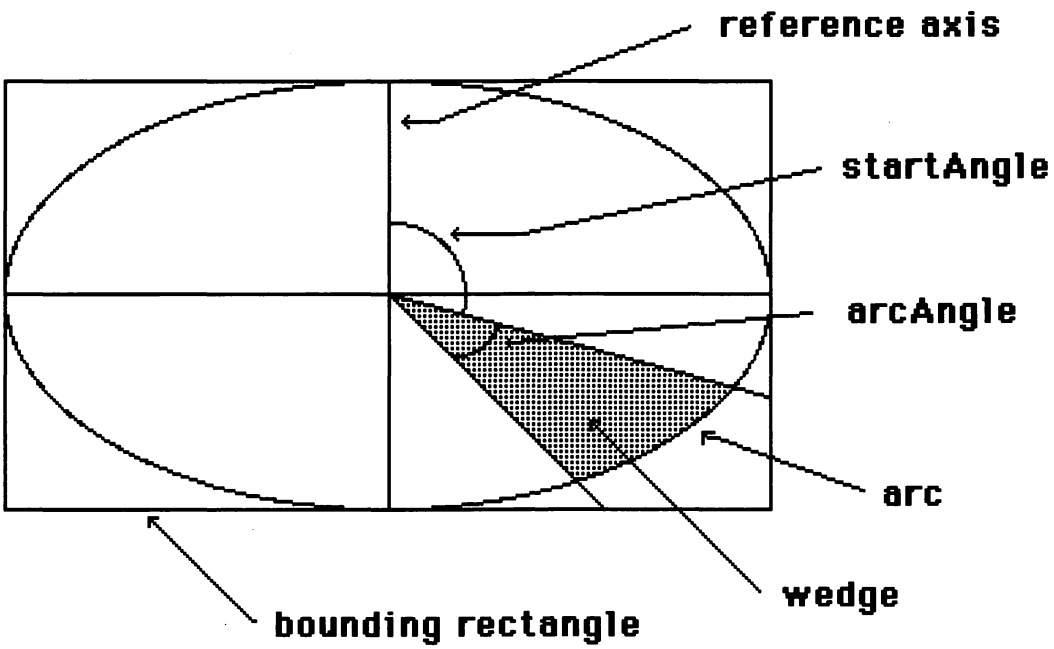


Figure 6-8. QuickDraw Arcs and Wedges.

```
            CLR.L   -(SP)              ;Space for handle
            _OpenPoly
            MOVE.L  (SP)+,PolyHndl(A5) ;Save handle

PolyHndl    DS.L    1
```

As you can see, _OpenPoly returns a handle to a polygon data record. You will need this handle to perform the standard drawing operations with polygons.

Once the polygon record has been opened, you define your polygon by making a series of calls to _Line and _LineTo, just as you if you were drawing the polygon on the screen. The lines don't actually appear on the screen because _OpenPoly hides the pen. When you're done, call _ClosePgon (no parameters required).

The standard drawing instructions for polygons are _FramePoly, _PaintPoly, _ErasePoly, _InvertPoly, and _FillPoly. Before calling any of them, push the handle to the polygon on the stack. For _FillPoly you also have to push the address of the data structure defining the fill pattern.

Once you're through with a polygon, you can deallocate the memory space it occupies by pushing its handle on the stack and calling _KillPoly.

Chapter 7

Menus

Another prominent characteristic of the Macintosh user interface is the **menu bar**, which invariably appears across the top of the screen when an application is running. A typical menu bar contains the titles of one or more **menus**, and each menu is made up of one or more **items**. These items can be selected with the mouse or, sometimes, from the keyboard.

The items within a menu remain hidden beneath its title until you **pull down** the menu by positioning the mouse cursor above the menu's title in the menu bar and pressing the mouse button. You can then select any particular item by dragging the mouse down (or back up) until the item is highlighted, then releasing the mouse button. Before you release the mouse button, you can move to an adjacent menu by moving the mouse far enough sideways while it's in the menu bar area.

In this chapter you'll learn how to access the Macintosh Menu Manager instructions from assembly language. The Menu Manager is made up of all the instructions you'll need to create menu bars and menus, to manage the items contained within each menu, and to easily implement the standard pull-down menu selection technique. The instructions are summarized in Table 7-1.

Table 7-1 The Menu Manager Trap Instructions.

_AddResMenu	**Adds the names of resources to the end of a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
MOVE.L #rsrcType,-(SP)    ;LONGINT: resource type code
_AddResMenu
```

_AppendMenu	**Adds one or more items to the end of a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
PEA    itemString         ;POINTER: to item name string
_AppendMenu
```

_CheckItem	**Checks or unchecks a menu item.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
MOVE.B #checked,-(SP)     ;BOOLEAN: true = check
                          ;         false = uncheck
     _CheckItem
```

_ClearMenuBar	**Removes all menus from the active menu bar.**

```
_ClearMenuBar             ;no parameters
```

_CountMItems	**Returns the number of items in a menu.**

```
CLR    -(SP)              ;INTEGER: space for result
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
_CountMItems
MOVE   (SP)+,D0           ;Result: number of items in menu
```

_DeleteMenu	**Removes a menu from the menu bar.**

```
MOVE   #menuID,-(SP)      ;INTEGER: ID of menu to delete
_DeleteMenu
```

Table 7-1. *continued*

_DelMenuItem — **Deletes a menu item.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
MOVE   #afterItem,-(SP)   ;INTEGER: Item # to delete
_DelMenuItem
```

This instruction is only available if you are using a Macintosh with a 128K ROM.

_DisableItem — **Disables an item within a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
_DisableItem
```

_DisposMenu — **Frees up the space used by a menu record.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
_DisposMenu
```

Use _DisposMenu only if the menu was created with _NewMenu. The equivalent instruction for menus created with _GetRMenu is _ReleaseResource.

_DrawMenuBar — **Draws the current menu bar.**

```
_DrawMenuBar              ;no parameters
```

_EnableItem — **Enables an item within a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
_EnableItem
```

_GetItem — **Returns the name of a menu item.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
PEA    itemString         ;VAR: item's name returned here
_GetItem
```

Table 7-1. *continued*

_GetItmStyle — **Gets the character style for an item in a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
PEA    theStyle           ;VAR: new character style (word)
_GetItmStyle
```

_GetMenuBar — **Makes a copy of the current menu bar.**

```
CLR.L  -(SP)              ;HANDLE: space for result
_GetMenuBar
MOVE.L (SP)+,A0           ;Result: handle to the copy
```

_GetMHandle — **Returns the handle to a menu in the menu bar.**

```
CLR.L  -(SP)              ;HANDLE: space for result
MOVE   #menuID,-(SP)      ;INTEGER: menu ID
_GetMHandle
MOVE.L (SP)+,A0           ;Result: handle to menu
```

_GetNewMBar — **Loads a menu bar definition into memory from a MBAR resource.**

```
CLR.L  -(SP)              ;HANDLE: space for result
MOVE   #menuBarID,-(SP)   ;INTEGER: resource ID of menu
                          ;         bar
_GetNewMBar
MOVE.L (SP)+,A0           ;Result: handle to menu bar
```

_GetRMenu — **Loads a predefined menu from a MENU resource file.**

```
CLR.L  -(SP)              ;HANDLE: space for result
MOVE   #menuID,-(SP)      ;INTEGER: resource ID for menu
_GetRMenu
MOVE.L (SP)+,A0           ;Result: handle to menu record
```

Table 7-1. *continued*

_HiliteMenu — **Highlights or removes highlights from a menu title in the menu bar.**

```
MOVE   #menuID,-(SP)      ;INTEGER: ID of menu
_HiliteMenu
```

_InitMenus — **Initializes the Menu Manager.**

```
_InitMenus                ;no parameters
```

_InsertMenu — **Inserts a menu after a given menu in the menu bar.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
MOVE   #beforeID,-(SP)    ;INTEGER: menu number to insert
                          ;         after (0=add to end)
_InsertMenu
```

_InsMenuItem — **Inserts a menu item after a given item.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
PEA    itemString         ;POINTER: to item string
MOVE   #afterItem,-(SP)   ;INTEGER: Item # to insert after
_InsMenuItem
```

This instruction is only available if you are using a Macintosh with a 128K ROM.

_InsertResMenu — **Inserts the names of resources after a given item in a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to menu record
MOVE.L #rsrcType,-(SP)    ;LONGINT: resource type code
MOVE   #afterItem,-(SP)   ;INTEGER: item number to insert
                          ;         after (0=beginning)
_InsertResMenu
```

Table 7-1. *continued*

_MenuKey — **Returns the menu ID and item number corresponding to a particular command key.**

```
CLR.L  -(SP)              ;LONGINT: space for result
MOVE   #ch,-(SP)          ;CHAR: character typed with
                          ;      command key
_MenuKey
MOVE.L (SP)+,D0           ;Result: Menu ID (high word)
                          ;        Item number (low word)
```

If the command key does not correspond to a menu item, the result is zero.

_MenuSelect — **Tracks mouse down activity in the menu bar by pulling down menus and highlighting items, when necessary. Returns the menu ID and item number selected.**

```
CLR.L  -(SP)              ;LONGINT: space for result
MOVE.L #startPoint,-(SP)  ;LONGINT: Point where mouse
                          ;         was pressed (global)

_MenuSelect
MOVE.L (SP)+,D0           ;Result: Menu ID (high word)
                          ;        Item number (low word)
```

If no item was selected, the result is zero.

_NewMenu — **Creates a new, empty menu record.**

```
CLR.L  -(SP)              ;HANDLE: space for result
MOVE   #menuID,-(SP)      ;INTEGER: ID code for menu
PEA    menuTitle          ;POINTER: to menu title string
_NewMenu
MOVE.L (SP)+,A0           ;Result: handle to menu record
```

Table 7-1. *continued*

_SetItem — **Renames a menu item.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
PEA    itemString         ;POINTER: to new name for item
_SetItem
```

_SetItmStyle — **Sets the character style for an item in a menu.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
MOVE   #theStyle,-(SP)    ;INTEGER: new character style
_SetItmStyle
```

_SetItmIcon — **Displays an icon to the left of an item name.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
MOVE   #iconNum,-(SP)     ;BYTE: icon number (1..255)
_SetItmIcon
```

_SetItmMark — **Marks a menu item with a given character.**

```
MOVE.L theMenu,-(SP)      ;HANDLE: to the menu
MOVE   #theItem,-(SP)     ;INTEGER: item number in menu
MOVE   #markChar,-(SP)    ;CHAR: marking character
_SetItmMark
```

If markChar is zero, the item is unmarked.

_SetMenuBar — **Makes a menu bar the current one.**

```
MOVE.L menuBar,-(SP)      ;HANDLE: to menu bar
_SetMenuBar
```

Initializing the Menu Manager

Before using Menu Manager instructions, you must call the _InitMenus instruction once at the beginning of your program. This initializes the Menu Manager's internal data structures so it can properly deal with subsequent Menu Manager instructions.

Some of the instructions within the Menu Manager portion of the ROM toolbox also use the QuickDraw drawing instructions, the Font Manager, and the Window Manager. This means you must also call _InitGraf, _InitFonts, and _InitWindows (in that order) before calling _InitMenus. These calls are part of the standard opening sequence to every program described in Chapter 2.

Creating a Menu

One of the first things you'll use the Menu Manager for is to create a menu bar. For the purposes of illustration, let's generate the menu bar shown in Figure 7-1. This is the minimum menu structure most applications support so they will work properly with desk accessories. (See Chapter 9 for further discussion.)

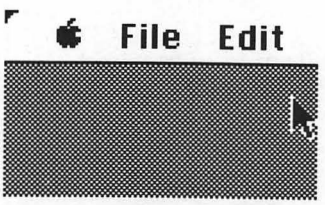


Figure 7-1. The Standard Apple-File-Edit Menu Bar.

Building the Menus

As with windows, you can create a menu in one of two ways: from scratch or by loading it directly from a resource file. The resource type for a menu is MENU.

If you're creating the menu from scratch, you first have to build a data structure, called a **menu record**, to hold the menu items, then fill it in by adding the menu items to it. To build the record, call _NewMenu as follows:

```
            CLR.L   -(SP)              ;Clear space for handle
            MOVE    #1,-(SP)           ;Menu ID (here it's 1)
            PEA     M1Name             ;Title (Apple symbol)
            _NewMenu
            MOVE.L  (SP)+,MenuH1(A5)   ;Save the handle

M1Name      DC.B    1,20               ;Length+"Apple"
MenuH1      DS.L    1
```

As you can see, _NewMenu returns a handle to the data structure for the new menu, so we first clear space for it on the stack. Then we push two parameters, a menu identification code, and the address of a string (preceded, as usual, by a length byte) that will be the title of the menu. After _NewMenu finishes, the returned handle is popped from the stack and stored in the MenuH1 variable.

Note that the menu ID code can be any positive integer not already used by another of your menus. Negative integers are reserved for menus defined by desk accessories, and a menu ID code of zero is not permitted.

Now that the menu record exists, we can add the names of menu items to the menu. For this we use _AppendMenu. To illustrate this, here's how to add the first item in the standard Apple menu, "About Demo Program...":

```
MOVE.L  MenuH1(A5),-(SP)        ;Push handle
PEA     'About Demo Program...' ;Name of item
_AppendMenu
```

_AppendMenu adds the specified item to the end of the list of menu items. A menu can hold up to 31 items.

Notice the three periods following the name of the menu item in the above example. By convention, this means if the item is selected, a dialog box will appear on the screen requesting user input. (See Chapter 8.) Selecting an About... item, for example, typically brings up an alert box containing a copyright notice, a description of the active program, and an OK button that must be clicked before you can return to the application.

The string you pass to _AppendMenu can actually contain the names of more than one menu item. Item names are separated from each other by a semicolon. For example, if you want to quickly create a standard Edit menu, use these instructions:

```
CLR.L   -(SP)              ;Space for handle
MOVE    #3,-(SP)           ;Menu #3
PEA     'Edit'             ;Name of menu
_NewMenu                   ;Create MenuInfo record
PEA     'Undo/Z;(-;Cut/X;Copy/C;Paste/V;Clear'
_AppendMenu
```

Since _NewMenu returns a handle to the menu on the stack, you don't have to explicitly push a copy of it before calling _AppendMenu. You may want to save the handle in case you need it later, however, so put a MOVE.L (SP),MenuH3(A5) instruction after _NewMenu and define a variable called MenuH3 with DS.L directive.

Notice the second item name in the string, "(-". The leading left parenthesis is a **modifier character** that disables (dims) the name of the item following it. Disabled items cannot be selected from the menu unless they are first enabled with the _EnableItem instruction. The single hyphen following the parenthesis is actually expanded into a line of hyphens across the width of the menu. A dimmed line of hyphens is convenient for physically separating groups of related items in a menu.

There are four other modifier characters you can use to affect the appearance of a menu item. They let you display the keyboard equivalent for an item (/), set the style of text used to display the item (<), display a special symbol (usually a check mark) to the left of the item (!), and display an icon to the left of the item (^). These modifier characters are summarized in Table 7-2. With the exception of the (and ; modifier characters, each modifier is followed by a single character, called the **argument**, containing the value of the modifier.

Table 7-2. Menu item modifier characters.

Modifier character	*Meaning*
/	**Keyboard equivalent**
<	**Character style**
!	**Mark item**
^	**Icon item**
(	**Disable the item**
;	**Multiple item separator**

Except for (and ; , each modifier character must be followed by a single argument character. The argument character following the / modifier can be any printable symbol enterable from the keyboard. The character style can be <B (bold), <I (italic), <U (underline), <O (outline), or <S (shadow). The argument character following the ! modifier can be any printable symbol. The icon item can be ^1, ^2, and so on. The ASCII code of the argument character, plus 208, is the resource ID of the icon.

Here are two examples of item names containing modifier characters:

'Cut/X' A command-X will appear to the right of the item name to indicate that the menu item can be selected by pressing the X key while holding down the command key.

'Read<B' The item name, Read, will be boldfaced.

You can also concatenate modifier sequences to combine two or more features. For example, use Read<B<U/R to display the item "Read" in a boldfaced, underlined style, with a command-R keyboard equivalent. You'll see other examples of how to use modifier characters later in this chapter.

After you've added all the items to the Edit menu, you can proceed to define other menus, remembering to assign each of them unique menu identification numbers.

There are no subroutines in the Macintosh 64K ROM that permit you to rearrange items once the menu has been created, so be sure to add items in the proper order. There are two new instructions in the 128K ROM that correct this deficiency, however. The first, _InsMenuItem, lets you insert an item after any specified item.

```
MOVE.L MenuHl(A5),-(SP)        ;Handle to menu
PEA    'New Choice'            ;New item name
MOVE   #2,-(SP)                ;Item # to add after
_InsMenuItem
```

The second, _DelMenuItem, can be used to delete a menu item:

```
MOVE.L MenuHl(A5),-(SP)        ;Handle to menu
MOVE   #3,-(SP)                ;Delete item #3
_DelMenuItem
```

Remember that these two instruction are available on a Macintosh equipped with a 128K ROM only. You can check whether this ROM is installed by examining the value stored at ROM85 ($28E). This value is $7FFF for a 128K ROM or $FFFF otherwise.

MENU Resource Files

An easier way to construct a menu is to first define it in an RMaker source file and compile it with RMaker to put it into a resource file. Menu definitions have resource type codes of

MENU. The RMaker format of the source file for such a resource is shown in Table 7-3.

Table 7-3. The RMaker Format of a MENU Resource.

```
Type MENU
,131                    ;;Resource ID for MENU
MyMenu                  ;;Menu title
First Entry             ;;First menu item
Second                  ;;Second menu item
Last Item               ;;Third menu item
```

The MENU resource definition must be followed by a blank line. For example, the RMaker statements required to define a standard Edit menu resource are as follows:

```
Type MENU
,130                    ;;Resource ID
File                    ;;Menu title
New                     ;;1st menu item
Open...                 ;;2nd menu item
Close
Save
Save As...
Page Setup...
Print...
Quit                    ;;last menu item
```

A blank line must follow the line containing the last menu item. This tells RMaker the list of menu items is complete. If you specify an attribute byte on the second line, don't make the resource purgeable or you will cause a system error.

When you add these lines to the application's RMaker source file and compile it, MENU resource #130 will be available for use by the application.

When you have MENU resources at your disposal, you don't use _NewMenu to create a menu. Instead, use _GetRMenu like this:

```
FileMID     EQU     130                 ;ID code for File menu

            CLR.L   -(SP)               ;Space for handle
            MOVE    #FileMID,-(SP)      ;Push menu resource ID
            _GetRMenu
            MOVE.L  (SP)+,MenuH2(A5)    ;Pop handle from stack

MenuH2      DS.L    1
```

Since there may be several MENU resources in a resource file, you must pass the menu resource ID code on the stack for _GetRMenu to know which one to access. As with _NewMenu, a handle to the data structure for the menu is returned on the stack.

We've now used two different techniques to create the standard Apple, File, and Edit menus. In practice, you would probably create them all using the same technique.

Destroying Menus

When a particular menu is no longer needed by your application, you should formally destroy it to free up the memory space it occupies. Before you do this, make sure the menu is first removed from the menu bar using the _DeleteMenu instruction and then redraw the menu bar using _DrawMenuBar. You'll see how to use these two instructions later on in this chapter.

There are two ways to deallocate the memory space used by a menu, depending on how it was created. For menus created with _NewMenu, use _DisposMenu. Since menus created with _GetRMenu are resources handled by the Resource Manager, you must use _ReleaseResource to dispose of them.

Both _DisposMenu and _ReleaseResource require one parameter on the stack, the handle to the menu data structure:

```
MOVE.L  MenuH2(A5),-(SP)        ;Handle for 2nd menu
_DisposMenu                     ;or _ReleaseResource
```

After a menu is destroyed, it's gone for good. If you want it back you'll have to read it in from the resource file using _GetRMenu or re-create it with _NewMenu and _AppendMenu.

Adding Items From Resource Files

It is often convenient to take the names of menu items from the names of resources in a resource file. For example, if you have a Font menu, it would be nice to add to it the names of all the available font resources (type FONT) in one simple step. It would also be useful for adding the names of all available desk accessories to the standard Apple menu. A **desk accessory** is a resource of type DRVR.

As you've probably guessed by now, the Menu Manager has instructions to do exactly this. _AddResMenu adds the names of all resources of a particular type to the bottom of a given menu, and _InsertResMenu inserts them after any given item. The resources added can be in any open resource file.

The parameters for _AddResMenu are the handle to the menu and the four-character (one long word) resource type:

```
        MOVE.L MenuHndl(A5),-(SP)       ;Handle to menu
        MOVE.L #'FONT',-(SP)            ;Resource type code
        _AddResMenu

MenuHndl        DS.L    1               ;Handle stored here
```

Notice the method for passing the name of the resource type on the stack. Each character in the name occupies exactly eight bits in the long word pushed on the stack.

The menu handle needed by _AddResMenu is the one returned by _NewMenu or _GetRMenu when the menu was first created or loaded from the MENU resource file. If you didn't save the handle, or the menus were loaded with an MBAR resource, use _GetMHandle to get an existing menu's handle. When _AddResMenu finishes, the name of every font in the system and application resource files (and any other

open resource files) will appear in the menu whose handle is stored in MenuHndl.

The _InsertResMenu instruction requires one additional parameter, the number of the item after which the new items are inserted:

```
MOVE.L MenuHndl(A5),-(SP)       ;Handle to menu
MOVE.L #'FONT',-(SP)            ;Resource type code
MOVE   #3,-(SP)                 ;Insert after item #3
_InsertResMenu
```

If you indicate you want to insert after item #0, the names are inserted before the first item in the menu.

You should know that _AddResMenu and _InsertResMenu can only be used to add the names of those resources having names associated with them. As you saw in the Chapter 2 discussion of RMaker, resource names are optional and not every resource has a name. Font and desk accessory resources are *always* associated with names, however.

A resource is also ignored if its name begins with a period. The Macintosh device drivers for I/O devices, like the disk and the printer, begin with periods; the device driver for the disk is .Sony, for example.

The versions of _AddResMenu and _InsertResMenu in the Macintosh 128K ROM alphabetize the added resources before putting them into the menu. The 64K ROM versions add them in the order they are located.

Determining the Number of Items in a Menu

To determine the number of items in a given menu, use _CountMItems by passing the handle to the menu on the stack:

```
CLR     -(SP)               ;Space for result
MOVE.L  MenuHndl(A5),-(SP)  ;Push handle
_CountMItems
MOVE    (SP)+,D0            ;Pop result into D0
```

The number returned can be from 0 to 31.

The only time you really have to use _CountMItems is after using _AddResMenu to add an indeterminate number of resource names to a menu. If you don't use _AddResMenu, your application should be able to keep track of the number of items.

Building a Menu Bar

Once you've created all your menus and added the necessary items to them, it's time to create the menu bar so you can display it at the top of the screen. The first step is to call _ClearMenuBar (no parameters) to clear the menu bar data structure and erase the menu bar area of the screen. (If you're just starting your program, you don't actually have to do this because _ClearMenuBar is called by _InitMenus.)

The next step is to insert the various menus into the menu bar using the _InsertMenu instruction, as follows:

```
                MOVE.L  MenuHndl(A5),-(SP)  ;handle to menu
                MOVE    #0,-(SP)            ;beforeID
                _InsertMenu

MenuHndl        DS.L    1
```

The beforeID number, zero in this case, indicates the ID code of the menu in the menu bar before which the new menu is to be inserted. If the number is zero, the menu is placed to the right of the rightmost menu.

It is also possible to remove menus from the menu bar. For this, use the _DeleteMenu instruction:

```
MenuId    EQU     3               ;Equate for menu ID

          MOVE    #MenuID,-(SP)   ;menu ID (EQU constant)
          _DeleteMenu
```

The only parameter _DeleteMenu requires is the menu ID code. In this example, menuID is a symbol representing this code and was defined using the EQU assembler directive. Be

sure to precede the symbolic name with a # in the MOVE instruction. If you don't, the instruction uses the number stored at location $000003 instead of the number three itself.

The most convenient way to build a menu bar is to read its definition from a resource file using _GetNewMBar. A menu bar resource has a resource type of MBAR and has the following structure:

Number of menus (one word)
Resource ID of first menu (one word)
.
.
.
Resource ID of last menu (one word)

Since RMaker does not directly support this type, you'll have to simulate it by equating it to the GNRL RMaker resource type. Here's what the source code looks like for a menu bar that has three menus having ID numbers of 1, 7, and 8:

```
TYPE MBAR = GNRL        ;;Create a MBAR resource
 ,128                   ;;Resource ID
.I                      ;;Using decimal integers
3                       ;;Three menus
1                       ;;Menu ID = 1
7                       ;;Menu ID = 7
8                       ;;Menu ID = 8
```

To use _GetNewMBar, first clear a space for the handle returned on the stack, then push the resource ID of the MBAR resource to be used:

```
IDcode      EQU     128

            CLR.L   -(SP)               ;Space for handle
            MOVE    #IDcode,-(SP)       ;resource ID (immediate)
            _GetNewMBar
            MOVE.L  (SP)+,MBarHndl(A5)  ;Pop handle

MBarHndl    DS.L    1                   ;Handle to menu bar
```

_GetNewMBar not only creates the menu bar record, it also automatically appends the menus specified in the resource.

A menu bar created with _GetNewMBar is not automatically made the currently active menu bar. To do this, you must call _SetMenuBar:

```
MOVE.L  MBarHndl(A5),-(SP) ;Handle to menu bar
_SetMenuBar
```

Before calling _SetMenuBar, you will probably want to read the handle stored at the MenuList system variable and store it in one of your program's own variables. MenuList contains the handle to the current menu bar and you'll need it if you want to make the original bar the active menu bar later on.

If you need to access one of the menus associated with an MBAR resource, perhaps to append further items to it or change an item name, you can use _GetMHandle to get its handle:

```
MenuID      EQU     7

            CLR.L   -(SP)               ;Space for handle
            MOVE    #MenuID,-(SP)       ;Push menu ID number
            _GetMHandle
            MOVE.L  (SP)+,A0            ;Pop handle into A0
```

Once you have the handle, you can use _AppendMenu, _AddResMenu, or any other instruction requiring a menu handle.

Displaying the Menu Bar

Well, you've now done everything except actually display the menu bar on the screen. For that, simply use _DrawMenuBar (no parameters required).

Modifying the Menu Bar

You can define as many menu bars as you want in a program, but only one can be displayed at a time. As you just saw, use _SetMenuBar to select any particular menu bar.

If you want to make a few changes to the existing menu bar and later restore it to its original state, it's best to make a copy of the menu bar using _GetMenuBar.

```
CLR.L   -(SP)              ;Space for handle
_GetMenuBar                ;Make copy of menu bar
MOVE.L  (SP)+,OldMBar(A5) ;Save new handle

OldMBar DS.L    1          ;Handle to old menu bar
```

After you do this, make the changes to the current menu bar, then display it using _DrawMenuBar.

To redisplay the original menu bar, use the following instructions:

```
MOVE.L  OldMBar(A5),-(SP)
_SetMenuBar                ;Activate the old menu bar
_DrawMenuBar               ;Display it!
```

OldMBar is the handle to the copy we made of the menu bar before the original was altered.

Menu Title Display

The _HiliteMenu instruction is used to highlight a menu title in the menu bar (white letters on a black background). To use _HiliteMenu, pass the ID number of the menu on the stack:

```
MOVE    #3,-(SP)        ;Menu number 3
_HiliteMenu
```

You probably won't have to highlight a menu title like this very often because it's done automatically by the _MenuSelect instruction you call when an item is selected from a menu. You will use _HiliteMenu more often to remove highlighting from a menu title *after* you've selected an item. To do this, push a zero on the stack before calling _HiliteMenu. A menu ID number of zero means "remove highlighting from all menu titles."

Menu Item Display

The toolbox has several commands you can use to affect the display of items in a pull-down menu. You'll learn about them in this section.

Changing the Name of an Item

To change the name of an item in a menu use _SetItem:

```
            STRING_FORMAT 3              ;Need length for DC

            MOVE.L  MenuHndl(A5),-(SP)   ;Handle to menu
            MOVE    #2,-(SP)             ;Item number in menu
            PEA     ItemName             ;Address of new string
           _SetItem

ItemName    DC.B    'New Name'           ;(Preceded by length)
MenuHndl    DS.L    1                    ;Handle to menu
```

The first item in a menu is item number one, not zero. Notice that I set STRING_FORMAT to 3 in this example to force MDS to put a length byte in front of the ItemName string, as required by _SetItem. There is also an instruction called _GetItem you can use to read the current name of a particular item. It is called just like _SetItem and the name is returned in ItemName.

Disabling and Enabling Item Names

At some points in a program, certain menu items may become meaningless because they have no meaning in that environment. You saw an example of this in the window program in Listing 6-2 of Chapter 6—when the window is open, the "Open Window" menu item is superfluous.

Rather than let the user select items that will be ignored by the application, you should disable them using the _DisableItem instruction. The operating system dims disabled items in pull-down menus and will not permit them to be selected.

To use _DisableItem, pass the handle to the menu and the menu item number on the stack like this:

```
MOVE.L  MenuHndl(A5),-(SP)      ;Handle to menu
MOVE    #3,-(SP)                ;Disable item #3
_DisableItem
```

If you want to disable a menu item when the menu is first installed, place a left parenthesis, (, in front of its name when the item is added to the menu with _AppendMenu. This same technique works when you define the names of menu items using RMaker source statements.

To reactivate a menu item, use _EnableItem. It requires the same two parameters as _DisableItem.

If you pass an item number of zero to _DisableItem or _EnableItem, the menu *title* is disabled or enabled, respectively. If you do this, you must call _DrawMenuBar to show the change.

Changing the Style of Item Names

It is also possible to change the style of the characters the Macintosh uses to draw the name of a menu item. The style can be bold, italic, underline, outline, shadow, condense, extend, or any combination of these seven basic type styles.

To set the style of an item, first push the handle to the

menu and the menu item number on the stack. Next, push a style word and call _SetItmStyle. As shown in Figure 7-2, the low-order seven bits in the style word control the basic style features: The feature associated with a bit is enabled when the bit is one. The symbolic names for these bits are also shown in Figure 7-2.

If you want to display a bold, underlined menu item, use the following instructions:

```
MOVE.L  MenuHndl(A5),-(SP)   ;Handle to menu
MOVE    #3,-(SP)             ;Menu item 3
CLR     D0                   ;No style!
BSET    #BoldBit,D0          ;Set bold bit (0)
BSET    #UlineBit,D0         ;Set underline bit (2)
MOVE    D0,-(SP)             ;Push style word
_SetItmStyle
```

Set the style word to zero if you don't want to use any special character style attributes.

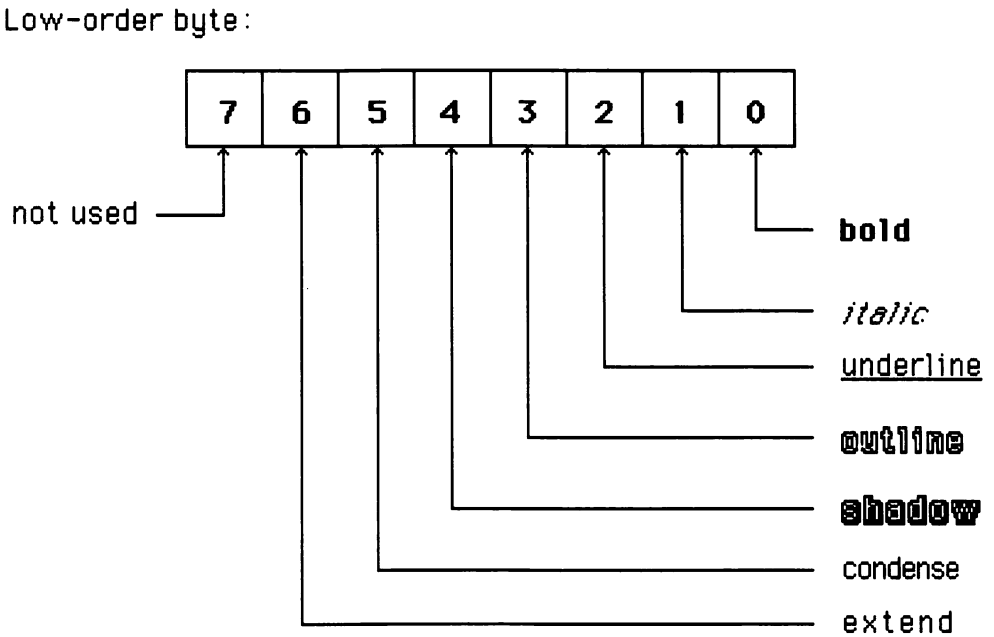
Some elements of the style of a menu item can also be set by including the < modifier character in the item's name when the menu is formed. As you saw earlier in this chapter, appending <B to the name selects bold, <I selects italic, <U selects underline, <O selects outline, and <S selects shadow. There are no modifiers for selecting the condense and extend styles, so you must select these styles using _SetItmStyle.

If you want to determine what the current style is, use _GetItmStyle. Instead of pushing a style word, push the address of the variable in which the style word is to be returned:

```
        MOVE.L  MenuHndl(A5),-(SP)   ;Handle to menu
        MOVE    #3,-(SP)             ;Third menu item
        PEA     Style(A5)            ;Address of variable
        _GetItmStyle

Style   DS.W    1               ;Style word returned here
```

Use the BTST instruction to check individual style bits. For example, if you execute these instructions:



Symbolic names for the bits in the style word:

Name	Bit #
BoldBit	0
ItalicBit	1
UlineBit	2
OutlineBit	3
ShadowBit	4
CondenseBit	5
ExtendBit	6

Figure 7-2. The Style Word Used with _SetItmStyle and _GetItmStyle.

```
BTST #ItalicBit,Style(A5) ;Check bit 1 (italic bit)
BNE  BoldOn               ;Branch if bit is 1
```

the BNE branch will be taken if italic bit is active.

Checking and Marking Item Names

Use _CheckItem to place a mark character to left of the name of any menu item. The standard mark character is a check mark symbol (ASCII code 18). You can also use _CheckItem to remove any mark character.

Marks are usually used to identify which of several related menu items is active. When you pull down the MacWrite Fonts menu, for example, a check mark appears to the left of the name of the current font and font size. All other fonts and font sizes are unmarked.

To tell _CheckItem whether to mark or remove a mark from an item, push a true or false Boolean parameter on the stack just before calling _CheckItem.

Here's how to place a mark character to the left of the third item in a menu whose handle is stored at MenuHndl(A5):

```
MOVE.L  MenuHndl(A5),-(SP)      ;Handle to menu
MOVE    #3,-(SP)                ;Item #3
MOVE.B  #-1,-(SP)               ;Boolean, -1 = mark
_CheckItem
```

Change the Boolean value to false if you want to remove the check mark (or other marking character) to the left of the item name.

If you don't want to use the standard check mark to mark a menu entry, use _SetItmMark instead.

```
MOVE.L  MenuHndl(A5),-(SP)      ;Handle to menu
MOVE    #3,-(SP)                ;Item #3
MOVE    #'x',-(SP)              ;marking character
_SetItmMark
```

Notice that the ASCII code for the marking character is placed on the stack just before calling _SetItmMark. If this code is zero, any marking character present is removed.

When you first add items to a menu using _AppendMenu or within an RMaker source file, you can use the ! modifier character to place a mark in front of an item when the menu is first installed. This saves you the extra step of using _CheckItem or _SetItmMark after creating the menu.

Associating Icons with Item Names

The most dramatic way to liven up your menus is to place an icon to the left of an item name using _SetItmIcon. Before you see how to do this, a word about icons.

An **icon** is a 32 by 32 screen image represented in memory by a sequence of 32 long words. The bits in each long word define the pixels in one row of the icon, starting with the leftmost position (bit 31) and ending with the rightmost position (bit 0). If the bit is one, the corresponding pixel is black; zero bits correspond to white pixels. The long words are arranged in top-to-bottom row order.

Suppose we want to determine the numeric representation for an X icon. The first step is to draw the icon on a 32 by 32 grid of squares and then convert each row to the corresponding long word.

Since most operations involving icons require that the icons be referred to by a resource ID number, the next step is to store the icon definition in a resource file. The resource file type for a single icon definition is ICON.

RMaker does not support the ICON resource type directly, so you have to equate it to RMaker's GNRL type to create it. Here's what the source code looks like:

```
TYPE ICON = GNRL
 ,323                    ;;Resource ID
.H                       ;;Hexadecimal numbers follow
[insert the hex numbers
 in the example above]
```

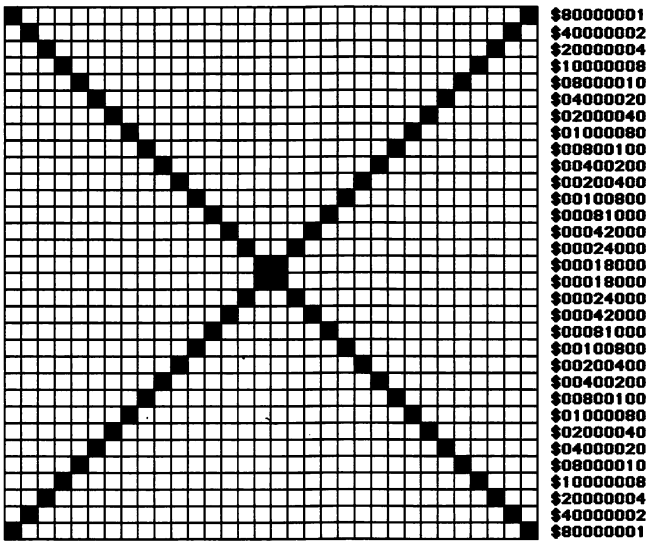


Figure 7-3. Defining An Icon.

If you compile the resource definition and append it to the application (use the !filename RMaker output file command for this), the icon resource is opened when you launch the application.

Now that you've mastered icons, let's get back to _SetItmIcon. This instruction takes three parameters: a handle to the menu, the number of the item within the menu to be associated with the icon, and the icon number. The **icon number** is the resource ID of the icon minus 256. It is not the resource ID itself.

Icons that can be used in menus must have resource IDs from 257 to 511. This means the icon number passed to _SetItmIcon will be an integer from 1 to 255.

If the icon was stored in a resource file under number 323, here's how you would assign it to the second item in a menu:

```
MOVE.L  MenuHndl(A5),-(SP)      ;Handle to menu
MOVE    #2,-(SP)                ;Item number
MOVE    #67,-(SP)               ;Icon number (ID=323)
_SetItmIcon
```

Other ways to assign icons to menu items are to add the item to the menu using _AppendMenu or add it right in the definition of the MENU resource in the RMaker source statements. To do this, follow the item name with a ^ (a modifier character) and the character whose ASCII code is 208 less than the resource ID for the icon. What this means is that '1' (ASCII value 49) refers to resource ID 257 (the first menu icon), '2' to resource ID 258, and so on. If you do this, you don't have to bother with _SetItmIcon unless you want to assign a new icon to a menu item. For example, use the string 'Delete^3' to place the icon with a resource ID of 259 to the left of the Delete menu item.

If the menu item is checked, the icon appears to the right of the check mark but before the text of the menu item.

Selecting Items From a Menu

Once you've defined a menu bar and the menus it is to contain, you're ready to write the main body of your application program. The program must contain the code needed to check whether the user wants to select a menu item and the code to be executed when any particular item is selected.

In a typical program, the main event loop keeps polling the event queue using _GetNextEvent until an event occurs. If it's a button-down event, you would call _FindWindow to determine where the button was pressed.

```
        CLR.W   -(SP)          ;Space for result (part code)
        MOVE.L  EventRecord+evtMouse,-(SP)  ;Mouse location
        PEA     theWindow(A5) ;Window pointer returned here
        _FindWindow
        MOVE.W  (SP)+,D0       ;Move part code into D0
        CMP     #inMenuBar,D0 ;Were we in the menu bar?
        BEQ     DoMenu         ;Yes, so branch

EventRecord     DCB.B   EvtBlkSize,0  ;_GetNextEvent's record

theWindow       DS.L    1     ;Pointer to window
```

The position parameter passed to _FindWindow is fetched from the evtMouse field of the event record used by the call to _GetNextEvent.

If the result returned by _FindWindow is anything but inMenuBar, you can pass it on to the part of the program that processes other window-related events such as dragging, resizing, and closing. If the result is inMenuBar, the button was pressed in the menu bar region of the screen and you must pass control to _MenuSelect to handle the standard pull-down menu chores. _MenuSelect tracks the mouse until the mouse button is released and returns codes to indicate what menu and what menu item was selected (if any). By using it, you avoid writing the complex program needed to implement the standard pull-down menu interface.

To use _MenuSelect, push space for a long word result on the stack, then push the global coordinates of the point where the mouse was pressed, like this:

```
CLR.L   -(SP)           ;Space for long word result
MOVE.L  EventRecord+evtMouse,-(SP)  ;Global coordinates
_MenuSelect
MOVE.L  (SP)+,D0        ;Grab the result
BEQ     Ignore          ;Do nothing if 0 result
```

The high-order 16 bits of the long word result is the ID number of the menu selected. The low-order 16 bits is the item number selected. If the result is zero, no menu item was chosen. Figure 7-4 shows a diagram of the format of the result.

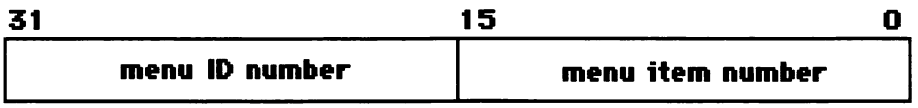


Figure 7-4. The Format of the Result Returned by _MenuSelect and _MenuKey

When _MenuSelect returns, it highlights the title of the selected menu. Before you continue, it's good practice to

remove the highlight from the name by passing a zero to the _HiliteMenu instruction:

```
MOVE        #0,-(SP)                ;Remove highlight from all menus
_HiliteMenu
```

Accessing Menu Items from the Keyboard

You can sometimes select menu items by pressing a character key while holding down the command key. This works if you assign a command key equivalent to the menu item when the menu is created. This is done by following the menu item name with a / and the command character. For example, to assign command-V to "Paste", specify a menu item with the name Paste/V when you create the menu. The command key equivalent appears to the right of the item name when the menu is pulled down with the mouse.

When you retrieve a key-down event from the event queue, check the evtMeta field of the event record to see whether the command key was down when the event occurred. If it wasn't, it doesn't correspond to a menu item, and you can continue with the part of the program that deals with other types of key presses.

Here's how to check for a key press and test whether the command key was down:

```
GetEvent  CLR.B   -(SP)              ;Space for Boolean result
          MOVE    #-1,-(SP)         ;Allow all events
          PEA     EventRecord       ;Record for _GetNextEvent
          _GetNextEvent
          TST.B   (SP)+             ;Pop and test the result flag
          BEQ     GetEvent          ;Branch if nothing

          MOVE    EventRecord+evtNum,D0   ;Get event type
          CMP     #KeyDwnEvt,D0     ;Key-down?
          BNE     NotAKey           ;No, so branch

          MOVE    EventRecord+evtMeta,D0   ;Get modifiers word
          BTST.W  #CmdKey,D0        ;Is bit 8 (CmdKey) on?
```

```
        BEQ     NotACommand         ;No, so branch
        NOP                         ;Begin command key handling
EventRecord     DCB.B   EvtBlkSize,0  ;Allocate event record
```

Notice that I used the CmdKey symbol to represent the bit number (8) of the flag holding the state of the command key. It is relative to the word, not byte, beginning at evtMeta. If the command key is down, the bit is 1, and the BTST instruction clears the zero flag in the status register. This means control does not pass to the target of the BEQ instruction, which is the part of the program that handles standard key presses.

If the command key was down, the next step is to call _MenuKey to determine if the character pressed with it corresponds to a menu item.

```
CLR.L   -(SP)               ;Space for long word result
MOVE    EventRecord+evtMessage+2,-(SP) ;The character code
_MenuKey
MOVE.L  (SP)+,D0            ;Remove result code
```

Recall from Chapter 5 that the low-order byte of the word beginning at offset evtMessage+2 from the start of the event record is the character code for the key pressed.

The format of the result code returned by _MenuKey is the same as for _MenuSelect. That is, the low-order word contains the number of the menu item selected, and the high-order word contains the ID of the menu in which the item appears. If the result is zero, the command key entered does not correspond to any menu item and can be ignored.

Example Program Using Menu Manager Instructions

The program in Listing 7-1 illustrates how to use many of the important Menu Manager instructions. It creates a menu

bar containing three menus: Apple, File, and Font. The Apple menu contains an About... item, but nothing happens if you select it; in a complete application you would display an alert box. The File menu contains a Quit item you can select to leave the application and return to the Finder.

Listing 7-1. The Source File, Linker Control File, and RMaker File for the Menus Program.

```
; Asm Source File
; Menus.Asm
;
; This is an example of how to install a Font menu

MenuBarID       EQU     128     ;Menu Bar resource ID

WindID          EQU     128     ;Window ID
 AppleID        EQU     1       ;Menu ID for Apple menu
 FileID         EQU     2       ;Menu ID for File menu
 FontID         EQU     3       ;Menu ID for Font menu

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QuickDraw globals
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; Create and draw a window on the screen:

        CLR.L   -(SP)           ;Space for returned pointer
```

Listing 7-1. *continued*

```
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file
        MOVE.L  (SP),OurWindow(A5)   ;Save window ptr for later
        _SetPort                ;Make window current GrafPort

; Get the menu bar with two menus:

        CLR.L   -(SP)           ;Space for result
        MOVE    #MenuBarID,-(SP)   ;MBAR resource ID
        _GetNewMBar             ;Read in menu bar
        _SetMenuBar             ;handle already on stack

; Create the Font menu and the add the items to it:

        CLR.L   -(SP)           ;Space for result
        MOVE    #FontID,-(SP)   ;Menu ID number
        PEA     'Font'          ;Name of menu
        _NewMenu                ;Create the menu
        MOVE.L  (SP),FontH(A5)  ;Save menu handle + on stack

        MOVE.L  #'FONT',-(SP)   ;Resource type
        _AddResMenu             ;Add named font resources

        MOVE.L  FontH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        _DrawMenuBar            ;Display menu bar

MainLoop

        BSR     GetEvent
        BSR     HandleEvent
        BRA     MainLoop

GetEvent

        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1,-(SP)       ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
```

Listing 7-1. *continued*

```
        TST.B   (SP)+           ;Pop and test the result
        BEQ     GetEvent        ;Branch if no pending event
        RTS

* HandleEvent is the event dispatcher. It takes the event type
* code returned by _GetNextEvent and calls the subroutine that
* handles it. Access to the event-handling subroutines is
* through a 16 entry jump table.

HandleEvent

        MOVE    EventRecord+evtNum,D0   ;Get event code
        CMP     #8,D0           ;Event 9_15?
        BHI     Ignore          ;Yes, so branch
        ASL     #2,D0           ;Two shifts = times 4
        JMP     JumpTable(PC,D0)   ;Jump to handler

Ignore  RTS

JumpTable

        JMP     Ignore          ;Null event (never used)
        JMP     DoMouseDown     ;Button-down
        JMP     Ignore          ;Button-up
        JMP     DoKeyDown       ;Key-down
        JMP     Ignore          ;Key-up
        JMP     DoKeyDown       ;Auto-key
        JMP     DoUpdate        ;Update
        JMP     Ignore          ;Disk-inserted
        JMP     DoActivate      ;Activate

DoKeyDown
DoUpdate
DoActivate

        RTS

DoMouseDown

        CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)   ;Where
        PEA     ClickWindow     ;VAR window involved
        _FindWindow             ;Where was button pressed?
```

Listing 7-1. *continued*

```
        MOVE    (SP)+,D0        ;Get result
        CMP     #InMenuBar,D0   ;Pressed in menu bar?
        BEQ     DoMenu          ;Yes, so check it out
        RTS                     ;Ignore everything else

; Handle clicks in menu bar:

DoMenu

        CLR.L   -(SP)           ;Space for result
        PEA     EventRecord+evtMouse   ;Where
        _MenuSelect             ;Get menu selection
        MOVE    (SP),theMenu(A5)   ;Save menu number (high word)
        MOVE    2(SP),theItem(A5)   ;Save item number (low word)

        MOVE    #0,-(SP)
        _HiliteMenu             ;Remove highlight from menu

        TST.L   (SP)+           ;Test and pop _MenuSelect result
        BNE     @1              ;Branch if item selected
        BRA     GetEvent

@1      CMP     #FileID,theMenu(A5)   ;In the File menu?
        BNE     @2

; must have selected QUIT command, so return to Finder

        _ExitToShell            ;Return to Finder

@2      CMP     #FontID,theMenu(A5)   ;Font menu?
        BNE     MenuExit        ;No, so branch

; Font menu being used. First remove checks from all items, then check
; the one that was selected.

        CLR     -(SP)           ;Space for result
        MOVE.L  FontH(A5),-(SP)   ;Handle to menu
        _CountMItems            ;Get # of items in menu
        MOVE    (SP)+,D6        ;Pop count into D6

; Use a DBRA loop to remove checks from everything. The count is in D6
; because D6 is not destroyed by toolbox calls.
```

Listing 7-1. *continued*

```
        SUBQ    #1,D6           ;Count-1 for DBcc loops
UnMark  MOVE.L  FontH(A5),-(SP)
        MOVE    D6,-(SP)        ;Item number (minus 1)
        ADDQ    #1,(SP)         ;Add 1 for true item number
        MOVE.B  #0,-(SP)        ;0 (false) = remove check from the item
        _CheckItem              ;Do it!
        DBRA    D6,UnMark       ;Keep looping until D6 = -1

        MOVE.L  FontH(A5),-(SP)
        MOVE    theItem(A5),-(SP)   ;Item number
        MOVE.B  #-1,-(SP)       ;-1 (true) = checked
        _CheckItem              ;Check the selected item

; Get name of font resource from the menu item:

        MOVE.L  FontH(A5),-(SP)   ;Handle to font menu
        MOVE    theItem(A5),-(SP) ;Menu item number
        PEA     fontName        ;VAR name of font
        _GetItem                ;Get the name

; Get the font number corresponding to the font name. For this
; use _GetFMenu:

        PEA     fontName        ;Pointer to font name
        PEA     fontNumber      ;VAR font number
        _GetFNum                ;Read number into fontNumber

; Set the new typeface:

        MOVE    fontNumber,-(SP);Push font number
        _TextFont               ;Select new typeface

        MOVE    #0,-(SP)        ;0 = closest to system size
        _TextSize               ;Pick an available font size

; Position the drawing pen, clear the screen, and draw our
; test pattern:

        MOVE    #30,-(SP)       ;h
        MOVE    #20,-(SP)       ;v
        _MoveTo

        MOVE.L  OurWindow(A5),A0
```

Listing 7-1. *continued*

```
        PEA     PortRect(A0)    ;PortRect holds window rectangle
        _EraseRect              ;Erase the window

        PEA     'The quick brown fox jumped over the lazy dog.'
        _DrawString

        RTS

; We get to here if Apple menu was selected:

MenuExit
        RTS                     ;Ignore Apple menu


; Record for _GetNextEvent:

EventRecord     DCB.B   EvtBlkSize,0   ;Reserve space for record

ClickWindow     DC.L    0       ;Window where mouse was clicked

fontNumber      DC      0       ;Font number
fontName        DCB.B   16,0    ;Name of font

; Here are the program globals. Use (A5) addressing.

FontH           DS.L    1       ;Handle to Font menu

OurWindow       DS.L    1       ;Pointer to window we defined

theMenu         DS      1       ;Menu # selected
theItem         DS      1       ;Item # selected
```

```
; Linker Control File
; Menus.Link
;
; Link this file to create application
;  (without resources).

Menus
$
```

Listing 7-1. *continued*

```
* RMaker Source File
* Menus.R
*
* Compile this after assembling and linking Menus.Asm
*
* The next command appends the resources to the application:
!Book:Menus


Type MBAR = GNRL
,128
.I                      ;;Integers follow
2                       ;;Number of menus
1                       ;;ID of 1st menu
2                       ;;ID of 2nd menu

Type MENU
,1                      ;;Resource ID
\14                     ;;Title is the Apple symbol
  About this demo...    ;;About box

,2                      ;;Resource ID
File                    ;;Menu Title
  Quit                  ;;Only item is Quit

Type WIND
,128                    ;;Resource ID
Font Menu Example       ;;Title for Window
40 5 332 502            ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)
```

The last menu, Font, is the most interesting one. (See Figure 7-5.) It contains the names of all the active fonts in the Macintosh System program. When you select a name from the Font menu, the application prints a "quick brown fox" test string in the window, using the selected font. A check mark appears to the left of the name of the last font item selected.

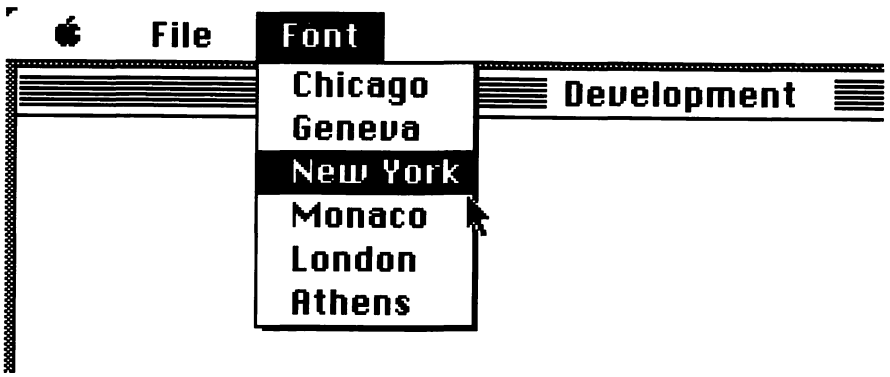


Figure 7-5. A Font Menu.

To create the application, assemble the Menus.Asm file, link using the Menus.Link linker control file, then compile the Menus.R file with RMaker. (Remember to change the disk volume prefix in the !Book:Menus statements, if necessary.) The application is stored in a file called Menus you can double-click to launch. Let's take a closer look at the assembly language source code to see how this application has been put together.

As with most of the applications in this book, it starts by initializing the various toolbox managers it uses. It then loads a new window definition from the application's resource fork using _GetNewWindow, and makes it the active window for drawing operations using _SetPort. Next, it creates a menu bar with an Apple and File menu by loading in MBAR resource #128 using _GetNewMBar. This menu bar is then made the current one using _SetMenuBar.

The Font menu cannot be included in the MBAR resource because there is no way to tell in advance what fonts will be stored in the System file. That's because it is quite common for users to add new fonts to it or remove fonts from it to save disk space, using Apple's Font/DA Mover program.

To create the Font menu, the program uses _NewMenu. The names of all the active fonts are added to its item list using _AddResMenu. (The resource type passed to

_AddResMenu is FONT.) To complete the menu bar definition, the program uses _InsertMenu to add the Font menu to the right side of the menu bar. The program then displays the menu bar using _DrawMenuBar.

The next step is to wait for a button-down event in the menu bar and process it. The program does this by entering an event loop starting with the label MainLoop. When a button-down event occurs, control passes to DoMouseDown where the program uses _FindWindow to determine whether the event occurred in the menu bar area. If it didn't, control returns to the main event loop.

Clicks in the menu bar are handled by the code beginning at DoMenu. The program calls _MenuSelect to determine what menu item, if any, was selected. Notice how _MenuSelect's long word result is handled by the program: the first word on the stack, at (SP), which is the high-order word of the menu number/item number result, is transferred to the theMenu variable and the second word, at 2(SP), is transferred to theItem. Post-increment addressing is not used because the program needs to keep the result on the stack so it can easily check whether it is zero using TST.L (SP)+. If it is non-zero, an item was selected, then the zero flag is clear and the BNE @1 branch will take place.

Tests are then made to see what menu was pulled down. If it was the File menu, the only item is Quit, so the program calls _ExitToShell to return to the Finder. If it was the Font menu, the program has some housekeeping to do before selecting the new font and drawing the test string. In particular, it must erase the check mark for the previously selected font name and place a check mark to the left of the new font name.

To do this, the program first determines how many items are in the Font menu using _CountMItems, then puts the result in the D6 register. It next uses a DBRA loop to remove check marks from each item in the menu. The condition tested by this instruction is always true, so looping always continues until the counter reaches -1. Since DBcc stops looping when its counter reaches -1, the number in D6 (the

counter) is reduced by one with SUBQ before entering the loop. Since the _CheckItem instruction within the loop requires the item number to be on the stack, D6 is first pushed (the item number minus one), then the value on the stack is incremented with an ADDQ #1,(SP) instruction. A zero byte (a Boolean false) is pushed on the stack just before calling _CheckItem to direct _CheckItem to remove a check mark from the item.

After all items have the check marks removed, _CheckItem is called again to check the item selected. This time a -1 byte (Boolean true) is pushed to signify the item is to be checked.

The program must now set the new typeface for character drawing operations. To do this, it calls _GetItem to get the name of the new font, and then _GetFNum to convert this name to a font number that can be used with _TextFont. (You haven't come across _GetFNum before. It expects two long words on the stack; a pointer to the font name string; and the location at which the integer result, the font number, is to be stored.)

A call to _TextFont sets the proper typeface used by the subsequent _DrawString instruction. Before the test string is actually displayed, the drawing window is cleared using _EraseRect; the window rectangle it needs is located PortRect bytes from the start of the window record.

This application does not let you change the point size of the text. Instead, it calls _TextSize with a zero parameter to select a size for the active font that is closest to the point size of the system font. If it didn't do this, and the active font is not defined in the default size used within windows, the operating system scales a differently sized set of characters of the same font to the 12 point size. This results in distortion if the size of the scaled font is not an exact multiple of an existing font size.

Chapter 8

Dialogs and Alerts

The Macintosh user-interface guidelines describe two special types of windows, **dialog boxes** and **alert boxes**, which are used to request input from, or convey messages to, a user. You can create and control them with a group of Macintosh toolbox trap instructions that make up the Dialog Manager. (See Table 8-1.) You'll learn how to use the Dialog Manager in this chapter.

Table 8-1. The Dialog Manager Trap Instructions.

_Alert	**Draws an alert box on the screen. The box is defined in an ALRT resource.**

```
CLR     -(SP)              ;INTEGER: space for result
MOVE    #alertID,-(SP)     ;INTEGER: resource ID of ALRT
MOVE.L filter,-(SP)        ;POINTER: to filter procedure
_Alert
```

_CautionAlert	**Draws an alert box on the screen with the Caution icon in the top-left corner. The box is defined in an ALRT resource.**

```
CLR     -(SP)              ;INTEGER: space for result
MOVE    #alertID,-(SP)     ;INTEGER: resource ID of ALRT
MOVE.L filter,-(SP)        ;POINTER: to filter procedure
_CautionAlert
```

Table 8-1. *continued*

_CloseDialog	**Frees up the space used by a dialog record and removes the dialog box from the screen.**

```
MOVE.L theDialog,-(SP)     ;POINTER: to dialog
_CloseDialog
```

Use _CloseDialog only if you specified a nonzero value for dStorage when you created the dialog. If the value was zero, use _DisposDialog instead.

_DialogSelect	**Handles an event in a modeless dialog box and indicates whether the event related to an enabled dialog item.**

```
CLR.B  -(SP)              ;BOOLEAN: space for result
PEA    theEvent           ;POINTER: to the event record
PEA    theDialog          ;VAR: pointer to dialog affected
PEA    itemNumber         ;VAR: item number selected
_DialogSelect
MOVE.B (SP)+,D0           ;Result: true = enabled item
                          ;               selected
                          ;        false = no enabled item
                          ;                selected
```

_DisposDialog	**Frees up the space used by a dialog record and the records it refers to, and removes the dialog box from the screen.**

```
MOVE.L theDialog,-(SP)     ;POINTER: to dialog record
_DisposDialog
```

_DrawDialog	**Draws the contents of a dialog box on the screen.**

```
MOVE.L theDialog,-(SP)     ;POINTER: to dialog record
_DrawDialog
```

Table 8-1. *continued*

_GetCtlValue — **Returns the current value of a control item.**

```
CLR    -(SP)              ;INTEGER: space for result
MOVE.L theControl,-(SP)   ;HANDLE: to control item
_GetCtlValue
MOVE   (SP)+,D0           ;Result: value of item
```

_GetDItem — **Gets the properties of an item in a dialog box.**

```
MOVE.L theDialog,-(SP)    ;POINTER: to dialog record
MOVE   #itemNumber,-(SP)  ;INTEGER: item number
PEA    itemType           ;VAR: handle to item type code
PEA    itemHandle         ;VAR: handle to item
PEA    dispRect           ;VAR: display rectangle
_GetDItem
```

_GetIText — **Gets the text of a text item.**

```
MOVE.L itemHandle,-(SP)   ;HANDLE: to text item
PEA    theText            ;VAR: the text string
_GetIText
```

The maximum size of the text string is 241 characters. Use _GetDItem to get the handle to the text item.

_GetNewDialog — **Loads a predefined dialog from a DLOG resource file.**

```
CLR.L  -(SP)              ;POINTER: space for result
MOVE   #templateID,-(SP)  ;INTEGER: resource ID of DITL
MOVE.L dStorage,-(SP)     ;POINTER: to dialog record
MOVE.L behindWindow,-(SP) ;POINTER: to window in front
                          ;         of the dialog
_GetNewDialog
MOVE.L (SP)+,A0           ;Result: handle to dialog record
```

Table 8-1. *continued*

_GetResource — **Loads a resource into memory. Use it to load a DITL resource.**

```
CLR.L  -(SP)              ;HANDLE: space for result
MOVE.L #rsrcType,-(SP)    ;LONGINT: resource type code
MOVE   #rsrcID,-(SP)      ;INTEGER: resource ID
_GetResource
MOVE.L (SP)+,A0           ;Result: handle to resource
```

_HiliteControl — **Highlights a control item.**

```
MOVE.L theControl,-(SP)   ;HANDLE: to control item
MOVE   #hiliteState,-(SP) ;INTEGER: highlighting code
_HiliteControl
```

_InitDialogs — **Initializes the Dialog Manager.**

```
MOVE.L restartProc,-(SP)  ;POINTER: to restart procedure
_InitDialogs
```

To use the standard restart procedure, push a zero pointer.

_IsDialogEvent — **Indicates whether a given event relates to a particular modeless dialog window.**

```
CLR.B -(SP)               ;BOOLEAN: space for result
PEA   theEvent            ;POINTER: to the event record
_IsDialogEvent
MOVE.B (SP)+,D0           ;Result: true = dialog-related
                          ;        false = not related
```

_ModalDialog — **Handles user activity in a modal dialog box and returns the number of the item selected.**

```
MOVE.L filter,-(SP)       ;POINTER: to filter procedure
PEA    itemNumber         ;VAR: item number (integer)
_ModalDialog
```

Table 8-1. *continued*

_NewDialog — **Creates a new, empty dialog record.**

```
CLR.L  -(SP)              ;POINTER: space for result
MOVE.L dStorage,-(SP)     ;POINTER: to dialog record
PEA    windowRect         ;POINTER: to dialog box rectangle
PEA    title              ;POINTER: to title for dialog box
MOVE.B #visible,-(SP)     ;BOOLEAN: true = visible
                          ;         false = invisible
MOVE   #windowType,-(SP)  ;INTEGER: window type code
MOVE.L behindWindow,-(SP) ;POINTER: to window in front
                          ;         of the dialog
MOVE.B #goAwayFlag,-(SP)  ;BOOLEAN: true = has close box
                          ;         false = no close box
MOVE.L #refCon,-(SP)      ;LONGINT: reference constant
MOVE.L itemList,-(SP)     ;HANDLE: to item list
_NewDialog
MOVE.L (SP)+,A0           ;Result: handle to dialog record
```

_NoteAlert — **Draws an alert box on the screen with the Note icon in the top-left corner. The box is defined in an ALRT resource.**

```
CLR    -(SP)              ;INTEGER: space for result
MOVE   #alertID,-(SP)     ;INTEGER: resource ID of ALRT
MOVE.L filter,-(SP)       ;POINTER: to filter procedure
_NoteAlert
```

_ParamText — **Sets the values for the four dialog text placeholders ^0, ^1, ^2, and ^3.**

```
PEA    subText0           ;POINTER: to ^0 string
PEA    subText1           ;POINTER: to ^1 string
PEA    subText2           ;POINTER: to ^2 string
PEA    subText3           ;POINTER: to ^3 string
_ParamText
```

If a pointer is zero, the current value of the placeholder string is not affected.

Table 8-1. *continued*

_SelIText — **Selects a range of text in a variable text box.**

```
MOVE.L theDialog,-(SP)     ;POINTER: to dialog record
MOVE   #itemNumber,-(SP)   ;INTEGER: item number
MOVE   #selStart,-(SP)     ;INTEGER: starting position
MOVE   #selEnd,-(SP)       ;INTEGER: ending position
_SelIText
```

_SetCtlValue — **Sets the value of a control item.**

```
MOVE.L theControl,-(SP)    ;HANDLE: to control item
MOVE   #newValue,-(SP)     ;INTEGER: new value for item
_SetCtlValue
```

The handle to the control item is the handle returned by _GetDItem.

_SetDItem — **Sets the properties of an item in a dialog box.**

```
MOVE.L theDialog,-(SP)     ;POINTER: to dialog record
MOVE   #itemNumber,-(SP)   ;INTEGER: item number
MOVE   #itemType,-(SP)     ;INTEGER: new item type code
MOVE.L itemHandle,-(SP)    ;HANDLE: to new item handle
PEA    dispRect            ;POINTER: to new display rectangle
_SetDItem
```

_SetIText — **Sets the text of a text item.**

```
MOVE.L itemHandle,-(SP)    ;HANDLE: to text item
PEA    theText             ;POINTER: to new text string
_SetIText
```

Use _GetDItem to get the handle to the text item.

Table 8-1. *continued*

_StopAlert	**Draws an alert box on the screen with the Stop icon in the top-left corner. The box is defined in an ALRT resource.**

```
CLR     -(SP)              ;INTEGER: space for result
MOVE    #alertID,-(SP)     ;INTEGER: resource ID of ALRT
MOVE.L filter,-(SP)        ;POINTER: to filter procedure
_StopAlert
```

System global variables:

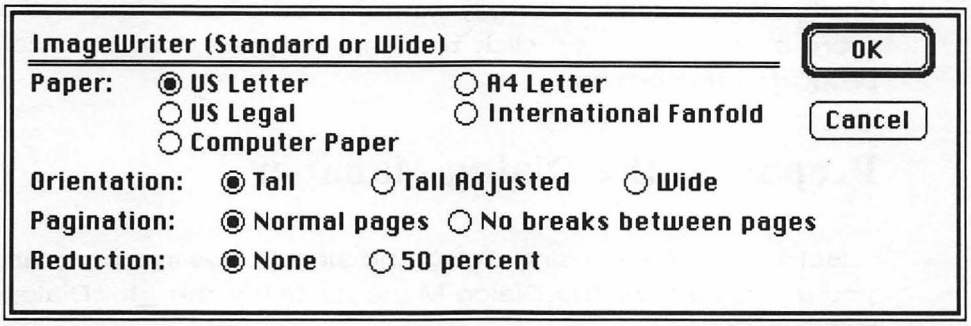
ACount	**($A9A)**	**Stage of last alert minus 1 [word].**
ANumber	**($A98)**	**The resource ID of the last alert used [word].**

The two standard types of dialog boxes are shown in Figure 8-1. These boxes are conventionally used to request certain types of input from the user. They can be composed of several user-alterable **items** (data input fields) containing such things as lines of text that can be edited, check boxes, and buttons. They can also contain static items that cannot be modified, such as fixed text, icons, and pictures. Toolbox instructions let you easily determine which items have been selected and what the current settings of the items are.

An **alert box**, as its name suggests, normally warns a user of the consequences of a proposed action that might result in the destruction or loss of data. In a typical application, an alert box contains "OK" and "Cancel" buttons that you can click to either verify the action or abort it. For instance, if you try to drag an application icon to the trash can with the Finder, you'll see the alert box shown in Figure 8-2.

Alerts are often used to display status information as well. Most About... items in the standard Apple menu use alert boxes to display authorship and copyright information, for example.

(a) A modal dialog box



(b) A modeless dialog box

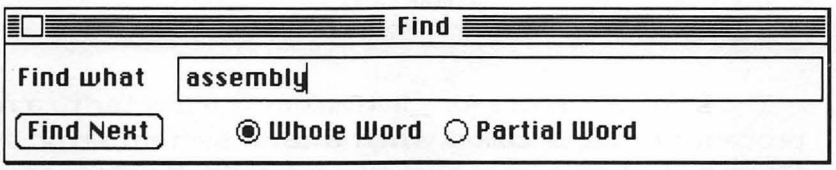


Figure 8-1. Macintosh Dialog Boxes.

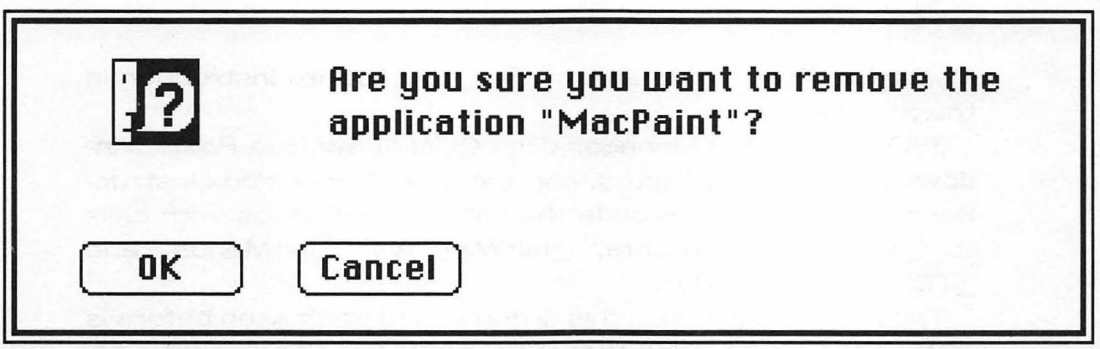


Figure 8-2. A Macintosh Alert Box.

The main difference between alerts and dialogs is that alerts don't contain any alterable items such as text strings or check boxes. They simply contain static items and one or more buttons you can click to dismiss the alert so the main program can continue.

Preparing the Dialog Manager

Before you begin using dialog and alert boxes in a program, you must initialize the Dialog Manager using the _InitDialogs instruction:

```
MOVE.L  #0,-(SP)        ;Pointer to restart procedure
_InitDialogs
```

The sole parameter for _InitDialogs is a pointer to a restart procedure that is called when a fatal system error occurs. Using a pointer of zero, as in this example, indicates you want to use the standard system procedure, which displays a bomb alert box and forces you to reboot the system. A more elegant restart procedure would be one that simply executes an _ExitToShell instruction to take you to the Finder. To set it up, use the following two instructions:

```
PEA     MyRestartProc
_InitDialogs
```

MyRestartProc is the label of the _ExitToShell instruction in the program.

Since the Dialog Manager uses QuickDraw, the Font, Window, and Menu Managers, and the Text Edit toolbox instructions, you have to precede the call to _InitDialogs with calls to _InitGraf, _InitFonts, _InitWindows, _InitMenus, and _TEInit, in that order.

The only instruction in this group you haven't seen before is _TEInit, the instruction that initializes the toolbox's text edit-

ing manager, Text Edit. These instructions are used to implement the standard "cut and paste" editing operations described in the Macintosh user-interface guidelines. The MDS Edit program is an example of a program that uses Text Edit for all its editing operations.

Creating Dialog Boxes

There are two general classes of dialog boxes you can implement on the Macintosh: **modal** and **modeless**. A **modal dialog box** is one that, once displayed, internally handles all keyboard and mouse events until the user dismisses the box by clicking a button. Mouse clicks outside an item in the dialog box are ignored, so you can't pull down a menu, select another window, or use a desk accessory until the modal dialog box is dismissed. In fact, that is how the modal dialog box gets its name: When you're using it, you're confined to a special operating mode until a button is clicked.

A **modeless dialog box**, on the other hand, is just like any other window on the screen. It has a goaway box and a title bar, but no grow box. The user is free to switch between the modeless dialog box and any other window on the screen without restriction. You remove a modeless dialog box from the screen just like any other window: by clicking its goaway box or selecting Close from a File menu.

Modal and modeless dialog boxes are defined and created using the same general programming techniques. The difference in their behavior arises because different instructions are used to interact with them while they are on the screen.

Just as with a window or a menu, there are two ways to create a dialog box, depending on whether a template for it has been saved in a resource file of type DLOG.

If the dialog definition is not in a DLOG resource file, use _NewDialog to create it. _NewDialog is rather complex in that it requires you to pass on the stack nine parameters

describing the properties of the dialog. It returns a pointer to a record describing the dialog.

> For the purposes of QuickDraw's drawing instructions, a dialog pointer is equivalent to a window pointer. This is because a dialog record is a superset of a window record.

Here is the calling sequence for _NewDialog:

```
        CLR.L   -(SP)               ;Space for result
        MOVE.L  #0,-(SP)            ;0 = use heap for record
        PEA     DialogRect          ;Dialog rectangle
        PEA     ''                  ;Title (null)
        MOVE.B  #-1,-(SP)           ;-1 = visible
        MOVE    #DBoxProc,-(SP)     ;window definition ID
        MOVE.L  #-1,-(SP)           ;-1 = front window
        MOVE.B  #0,-(SP)            ;0 = no goaway box
        MOVE.L  #0,-(SP)            ;refCon (usually 0)
        MOVE.L  ItemHndl(A5),-(SP) ;Handle to item list
        _NewDialog
        MOVE.L  (SP)+,A0            ;Move pointer into A0

DialogRect  DC.W  50,50,200,200   ;Coordinates of dialog rectangle
ItemHndl    DS.L  1               ;Handle to item list
```

The first parameter pushed on the stack (after making space for the result) is a pointer to the area where the dialog record is to be kept. You can get such a pointer by reserving a space DWindLen (170) bytes long using _NewPtr. If you use a pointer of 0, as in the example, _NewDialog allocates its own space for a dialog record on the heap.

The next parameter is a pointer to the coordinates of the rectangle in which the dialog box is displayed. The coordinates are expressed in global coordinates.

The next two parameters are a pointer to the title of the dialog box (if applicable) and a Boolean value indicating whether the box is to be visible (true) or invisible (false).

The window definition ID can be any of the values used when creating standard windows. (See Chapter 6.) By con-

vention, however, you should only use the DBoxProc, PlainDBoxProc, and AltDBoxProc ID codes for modal dialog boxes. For modeless dialog boxes, use NoGrowDocProc (a window with a close box but no size box).

The next long word parameter is the window pointer of the window behind which the dialog box is to be drawn. If the pointer is zero, the dialog box goes behind all other windows. If it is minus one, it goes at the front and becomes the active window. You will usually pass a minus one pointer.

The goaway box Boolean parameter will normally be false for modal dialog boxes and true for modeless dialog boxes. Modal dialog boxes don't use goaway boxes or titles.

The refCon value is for the private use of your application. It can be set to any value (a long word) you like. Its meaning is completely up to you.

The last parameter pushed on the stack before calling _NewDialog is a handle to a DITL resource containing a list of the items the dialog uses. You'll see how to define such a resource in just a moment. For now, let's see how to load a DITL resource into memory and get its handle:

```
CLR.L   -(SP)                   ;Space for result
MOVE.L  #'DITL',-(SP)           ;Resource type
MOVE    #128,-(SP)              ;Resource ID
_GetResource
MOVE.L (SP)+,ItemHndl(A5)       ;Pop the handle
```

_GetResource is a general purpose instruction for reading any type of resource into memory, not just DITL resources. It returns a handle to the resource record, just what we need for _NewDialog.

By far the most convenient way to create a dialog is to read it in from a DLOG resource file with _GetNewDialog. This instruction requires only three parameters: the ID of the DLOG resource, a pointer to the dialog record area, and the pointer to the window behind which the dialog window is to appear (or minus one if the dialog is to be the front window and active).

```
        CLR.L   -(SP)                   ;Space for result
        MOVE    #128,-(SP)              ;Resource ID of DLOG
        MOVE.L  #0,-(SP)                ;0 = storage on heap
        MOVE.L  #-1,-(SP)               ;Put dialog in front
        _GetNewDialog
        MOVE.L  (SP)+,DlogHndl(A5)      ;Move ptr into variable

DlogHndl  DS.L  1                       ;Handle to dialog record
```

The calling sequence for _GetNewDialog is much simpler than that for _NewDialog because most of the information needed to form the dialog record is contained in the DLOG resource.

As usual, you use RMaker to create a DLOG resource. Here is the source format for a typical DLOG resource:

```
TYPE DLOG
  128               ;;resource ID of this DLOG
A Dialog Box        ;;title for the dialog box
50 50 200 400       ;;coords of box (TLBR)
Visible NoGoAway    ;;window attributes
1                   ;;window definition ID
0                   ;;refCon value (usually 0)
133                 ;;resource ID of DITL (item list)
```

The title for a dialog box is displayed only if you are using a modeless dialog box. The coordinates are global coordinates relative to the top left-hand corner of the screen.

The attributes of a dialog window can be Visible (display the dialog box) or Invisible (don't display it), and GoAway (use a close box) or NoGoAway (no close box). You can use any other words beginning with V, I, G, N, respectively, if you wish. You will usually want to define a Visible dialog box so you don't have to display it with _ShowWindow after loading it into memory. Use the GoAway attribute for modeless dialog boxes and the NoGoAway attribute for modal dialog boxes.

The window definition ID is the same as the one you would pass to _NewDialog. For modeless dialogs, use 4 (NoGrow-

DocProc). For modal dialogs, use either 1 (DBoxProc), 2 (PlainDBoxProc), or 3 (AltDBoxProc).

The refCon parameter has the same meaning as the one passed to _NewDialog.

The DLOG resource is linked to a DITL resource that describes the various items used by the dialog. You'll learn about DITL resources in the next section.

Items and Item Lists

As you have just seen, both _NewDialog and _GetNewDialog use an important resource of type DITL (Dialog Item List). Such a resource is made up of a list of items that are to appear in a dialog or alert box. Associated with each item are the coordinates of the rectangle in which it is to be displayed, in the local coordinate system of the dialog or alert box window.

The common types of items that can be defined in a DITL resource are as follows:

- Static text—a line of text that cannot be modified
- Variable text box—a line of text that can be modified
- A control item—button, check box, radio button
- An icon
- A QuickDraw picture

It is also possible to use controls defined in a CNTL resource file or to create application-defined items, but I won't be considering them here.

The RMaker source format for a DITL resource looks like this:

```
TYPE DITL
 ,133                ;;resource ID of this DITL
5                    ;;number of items in list

StaticText Disabled  ;;static text (disabled)
10 121 26 177        ;;Item rectangle (TLBR)
My text              ;;this text never changes
```

```
Edit                ;;variable text box
45 31 62 195        ;;TLBR
Change this         ;;text to be edited

Button              ;;button item
109 33 140 94       ;;TLBR
Cancel              ;;Name for the button

Check               ;;check box item
78 28 95 107        ;;TLBR
Sound on            ;;check box name

Radio               ;;radio button item
77 171 95 259       ;;TLBR
High Rate           ;;radio button name
```

The item types in a DITL resource are identified by specific words:

- StaticText, StatText, or Stat for static text
- EditText or Edit for variable text boxes
- Button or BtnItem for standard buttons
- CheckBox, ChkItem, or Check for check boxes
- RadioButton, RadioItem, or Radio for radio buttons

The other standard item types are supported by MDS 2.0 only. Their identification words are:

- Icon or IconItem for icons
- Pic or PicItem for QuickDraw pictures
- User or UserItem for application-defined items
- ResCItem or ResCtl for controls defined in a CNTL resource file

These last few item types are not used very often.

The coordinates given for the items are in the local coordinates of the window in which the items will appear, and are in standard top, left, bottom, right order.

If an item name in the dialog template is followed by the word Disabled, as with the static text item above, clicks in its rectangle are ignored. Items whose names are followed by the word Enabled are active items. If neither word is used, the item is considered to be enabled.

Item Types

This section will describe each of the different types of items you can use within a dialog box. The symbolic names for these items are shown in Table 8-2.

Table 8-2. Item Type Codes for Dialog and Alert Boxes.

Symbolic Name	*Code*	*Description*
UserItem	0	User-defined item
CtrlItem	4	Control item
BtnCtrl	4+0	Button control
ChkCtrl	4+1	Check box control
RadCtrl	4+2	Radio button control
ResCtrl	4+3	Control in CNTL resource
StatText	8	Static text
EditText	16	Variable text box
IconItem	32	Icon
PicItem	64	QuickDraw picture
ItemDisable	128	Disabled item

Add the constant ItemDisable to the code for an item to disable that item.

Static Text

A **static text item** (StatText) is a string of characters that appears in the dialog box, but cannot be edited. Such an item could be used to hold a command or an explanatory message, or to pose a question, for example.

If a static text item is wider than the width of the rectangle in which it is to be displayed, the end of the text wraps to the next line in the rectangle, but you won't see it if the rectangle is not deep enough. Entire words wrap together, so words are not broken up over two lines.

It is often convenient to be able to change the precise wording of a static text item after it is initially defined, by inserting names or phrases that can't be predicted in

advance. The easiest way to do this is to use the ^0, ^1, ^2, and ^3 text place holders when you first define the static text item in the DITL resource.

You can assign a text string to each of these place holders to ensure when the dialog box is drawn that the strings are substituted for the place holders. The instruction to use for this is _ParamText:

```
        STRING_FORMAT 3         ;Need length+string for DS

        PEA  String0(A5)        ;String for ^0
        PEA  String1(A5)        ;String for ^1
        MOVE.L #0,-(SP)         ;(Don't change ^2 string)
        PEA  String3(A5)        ;String for ^3
        _ParamText

String0  DS.B  'placeholder 0'
String1  DS.B  'placeholder 1'
String3  DS.B  '4th placeholder'
```

If a particular string is not to be changed, push a long word zero on the stack instead of a pointer to the string. This was done for the ^2 place holder in the example.

Suppose you use a dialog box to ask for verification of a file deletion operation. Instead of using a general static text item like "Are you sure you want to delete the file?", you can define an item such as:

```
"Are you sure you want to delete ^0?"
```

and then use _ParamText to subsitute for ^0 the actual name of the file selected. This must be done before displaying the dialog, of course.

Variable Text Box

A **variable text box** (EditText) is a rectangle within which a line or lines of text is displayed. The text can be up to 241 characters long and can be edited using the standard Macin-

tosh text editing techniques. This means you can click the mouse somewhere in the text box to select an insertion point for subsequent keystrokes. (The insertion point is marked by a blinking vertical bar.) You can select a range of text for deletion by dragging the mouse across the text, then pressing any key to replace it. Further, you can extend a previous selection range by holding down the SHIFT key while you select another range. Selected text appears as white characters on a black background.

As you will see later on, the toolbox contains instructions you can use to pre-select a range of text or set a text insertion point. There are also instructions to change the text displayed in the box and to determine what the current text is.

If there is more than one variable text box item in a dialog, you can use the TAB key to move from one to the next. If you're in the last text box when you press TAB, you will go to the first one.

Control Items

Control items (CtrlItem) represent the most common item types used in dialog boxes. The major types are:

- Buttons (CtrlItem + BtnCtrl)
- Check boxes (CtrlItem + ChkCtrl)
- Radio buttons (CtrlItem + RadCtrl)

There is also a control item that can be set by the application (UserItem) and one defined by a control template in a CNTL resource file (ResCtrl). I will not be considering these types of control items here.

BUTTONS. Buttons are rounded-corner rectangles that can be clicked to dismiss a dialog or alert box (remove it from the screen) and cause a particular action to occur. The name associated with a button appears within the body of the button.

Most modal dialog boxes and alert boxes contain at least one button so they can be dismissed in accordance with the user interface guidelines.

There is a special visual form for the **default button** in a modal dialog or alert. This button can be selected by pressing the RETURN or ENTER key on the keyboard. The default button is easily identifiable because it is enclosed by a dark, black border. If you are using more than one button in your dialog, the one most likely to be selected should be made the default so it can be easily selected with a keyboard command.

For modal dialog boxes, the first item in the item list is always the default item, so make sure it is a button. Such a button is typically labeled as the OK button. For alert boxes, either the first or second item in the item list can be designated as the default when you create the alert's item list.

The Dialog Manager instructions that display dialog boxes do not automatically draw the dark border around the default button. You must take care of this yourself using techniques discussed later in this chapter. The alert box drawing instructions do take care of highlighting the default button, however.

CHECK BOXES. Check boxes are associated with parameters that can be in one of two states: on and off, selected and not selected, high and low, and so on. A check box appears as a small square in the dialog box. If it is on (1), it has an X drawn in it. The name associated with the check box appears to its right.

A check box is independent of all other check boxes and other items. This means that its setting should not affect the setting of any other item in the dialog box.

RADIO BUTTONS. Radio buttons invariably appear in groups of two or more buttons, with each button representing a different value that can be associated with one particular parameter. They appear as small circles in a dialog box and the one that is on has a smaller black circle inscribed in it. The radio button's name appears to its right on the screen. Radio buttons derive their name from the fact that when a radio button is selected by clicking it, all other buttons in the group are turned off, just like when you select a station on a standard car radio.

Icons

An icon (IconItem) is another type of item you can associate with a dialog box. In the DITL item list resource an icon is referred to by its resource ID number. The resource type for an icon is, naturally enough, ICON.

An icon is scaled to fit the rectangle associated with it in the DITL resource. For best results, select a rectangle that is 32 pixels wide and 32 pixels high. This is the exact size of an icon, so there will be no distortion due to scaling.

An icon item is similar to a static text item in that it cannot be modified by the user.

Pictures

The last standard item type is a QuickDraw picture (PicItem). Like an icon item, a picture is referred to by its resource ID number. The resource type is PICT. Also like icons, pictures are scaled to fit the display rectangles associated with them.

Disabling Items

Any item in a dialog can be defined as disabled by adding the ItemDisable constant to its item type code or by placing the word Disabled after its name in the DITL resource definition. It's a good idea to disable static items like icons, pictures, and text so mouse clicks in their rectangles will be ignored.

Changing Item Attributes

There are several toolbox instructions you can use to read and change the attributes of the items in a dialog box item list: _SetDItem, _GetDItem, _SetIText, _GetIText, _SelIText, _GetCtlValue, and _SetCtlValue.

Use _GetDItem to determine the item type, the handle to the data defining the item's behavior, and the coordinates of the display rectangle for the item. The calling sequence is as follows:

```
        MOVE.L  DialogPtr(A5),-(SP)     ;Push dialog pointer
        MOVE    #1,-(SP)                ;Item number
        PEA     itemType                ;VAR Item type
        PEA     itemHandle              ;VAR Item handle
        PEA     itemRect                ;VAR Item rectangle
        _GetDItem

itemType   DC.W 0                       ;Item type code
itemHandle DC.L 0                       ;Item handle
itemRect   DCB.W 4,0                    ;Item rectangle

DialogPtr  DS.L 1                       ;Dialog pointer
```

The item handle refers to data or code related to the item type. For control items, for example, the handle refers to a control record that defines the control in question. For a variable text box item, the handle refers to the string of characters currently displayed in the box.

To change the type, handle, or rectangle of a dialog item, you can use _SetDItem.

```
MOVE.L  DialogPtr(A5),-(SP)     ;Push dialog pointer
MOVE    #1,-(SP)                ;Item number
MOVE    itemType(A5),-(SP)      ;New Item type
MOVE.L  itemHandle(A5),-(SP)    ;New Item handle
PEA     dispRect(A5)            ;New Display rectangle
_SetDItem
```

_SetDItem is very useful for disabling certain dialog items so that the Dialog Manager will ignore clicks in them. To do this, add ItemDisable (decimal 128) to the standard item type code for the item.

Do not confuse disabling an item with making it inactive. An inactive item (always a control item) is dimmed in the dialog

box, but a disabled item is not. To make a control item inactive, use _HiliteControl:

```
MOVE.L  itemHandle(A5),-(SP)  ;Handle to control item
MOVE    #255,-(SP)            ;255 = inactive
_HiliteControl
```

You should make inactive and disable any control items in a dialog box inappropriate to the particular operating environment you're in.

To return a control item to its normal state, pass a highlighting code of zero to _HiliteControl:

```
MOVE.L  itemHandle(A5),-(SP)  ;Handle to control item
MOVE    #0,-(SP)              ;0 = highlight
_HiliteControl
```

You should not use _SetDItem to change the text of a static text or variable text item, however. For this, use _SetIText as follows:

```
        STRING_FORMAT 3         ;Need length+string for DC

        MOVE.L  itemHandle(A5),-(SP)    ;Handle to text item
        PEA     MyText                  ;The new text string
        _SetIText

MyText  DC.B    'Use this string'
```

You can use _GetDItem to get the proper value for itemHandle before calling _SetIText.

It's often quite convenient to determine the text associated with a particular variable text box to enable the program to take the user's input and deal with it. For this, use _GetIText.

```
        MOVE.L  itemHandle(A5),-(SP)    ;Handle to text item
        PEA     CurrentText(A5)         ;Address of string var
        _GetIText

CurrentText  DS.B  242                  ;Space for text
```

Since text items can be up to 241 characters long, you have to reserve 242 bytes for CurrentText. The extra byte is used for the leading length byte.

One other thing you can do with text in a variable text box is to "pre-select" all or a portion of it. Selected text is highlighted in white letters on a black background and is deleted and replaced when you type a character from the keyboard other than RETURN or a modifier key. If the text box contains a default string, you will probably want to select the entire string. That way, the default will disappear when the user types a character to change the entry; assuming, of course, that if the user changes even one character, he's likely to be entering a whole different response. This is usually a valid presumption.

To select text, use _SelIText:

```
MOVE.L  itemHandle(A5),-(SP)    ;Handle to text item
MOVE    #1,-(SP)                ;Item number
MOVE    #0,-(SP)                ;Starting char. position
MOVE    #241,-(SP)              ;Ending char. position
_SelIText
```

The starting and ending position parameters passed to _SelIText run from zero up to the maximum size of the text string. A zero value refers to a position to the left of the first character, one refers to the left of the second character, and so on. If the starting and ending positions are equal, a blinking vertical bar appears at that position. Typed characters are inserted at this point.

The settings of control items such as check boxes and radio buttons can be read and changed using _GetCtlValue and _SetCtlValue. Since check boxes and radio buttons are either on or off, their settings are either zero or one.

To use _GetCtlValue or _SetCtlValue you must first obtain the handle to the check box or radio button. To do this, use _GetDItem. _GetCtlValue returns the current value of the control item so you can tell if it is on or off.

```
CLR     -(SP)                ;Space for result
MOVE.L  itemHandle(A5),-(SP) ;Handle to control item
_GetCtlHandle
MOVE    (SP)+,D0             ;On if D0=1
```

You will use the _SetCtlHandle instruction to take care of selecting and disabling check boxes and radio buttons. For example, if you click the mouse in a check box, you will want to turn it on (put an X in it) using the following instructions:

```
MOVE.L  itemHandle(A5),-(SP) ;Handle to control item
MOVE    #1,-(SP)             ;New value (1=on)
_SetCtlValue
```

Remember to follow the Macintosh user interface conventions when enabling radio buttons: Only one button in a group can be selected at any given time. This means if a user clicks the third button in a group of four, your program will have to turn off the first, second, and fourth buttons, and turn on the third button. To do this, you have to make four calls to _SetCtlValue.

Using Dialog Boxes

The proper way to handle a dialog box once it's on the screen depends on whether it is a modal or modeless dialog box. Let's begin by considering modal boxes and then move on to explore modeless boxes.

Modal Dialog Boxes

As soon as you display a modal dialog box on the screen with _NewDialog or _GetNewDialog, you must call _ModalDialog to monitor events within the box and get a result indicating what item was selected. You can then deal with the result as you see fit before calling _ModalDialog once again to get more input or to dismiss the dialog box. To dis-

miss it, call _HideWindow to erase it from the screen. (See Chapter 6 for a description of this instruction.) Or use _DisposDialog or _CloseDialog to erase it and free up the heap space it uses (more on these two instructions later in this section).

A program subroutine that implements a simple two-button dialog box is shown in Listing 8-1. It uses _GetNewDialog to create a dialog box defined in a DLOG resource file.

Listing 8-1. The Source File, Linker Control File, and RMaker File for the Modal Program.

```
; Asm Source File
; Modal.Asm
;
; This is an example of how to use modal dialog boxes.

AppleID         EQU     1       ;Menu ID for Apple Menu
FileID          EQU     2       ;Menu ID for File Menu
WindID          EQU     128     ;Window ID
ModalID         EQU     128     ;Modal Dialog ID

        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions
; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QuickDraw globals
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event


; Create and draw a window on the screen:
```

Listing 8-1. *continued*

```
        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file
        _SetPort                ;Make window the active GrafPort

; Create two standard menus:

        CLR.L   -(SP)           ;Space for handle
        MOVE    #AppleID,-(SP)  ;Menu ID number
        _GetRMenu               ;Get Menu from resource file
        MOVE.L  (SP)+,AppleH(A5)   ;Save menu handle
        CLR.L   -(SP)           ;Space for handle
        MOVE    #FileID,-(SP)   ;Menu ID number
        _GetRMenu               ;Get menu from resource file
        MOVE.L  (SP)+,FileH(A5) ;Save menu handle

; Add menus to menu bar:

        MOVE.L  AppleH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        MOVE.L  FileH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        _DrawMenuBar            ;Display menu bar

        BSR     DoDialog

MainLoop

        BSR     GetEvent
        BSR     HandleEvent
        BRA     MainLoop

GetEvent

        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1,-(SP)       ;Allow all events
```

Listing 8-1. *continued*

```
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (sp)+           ;Pop and test the result code
        BEQ     GetEvent        ;Branch if null event
        RTS


* Draw a dialog box on the screen and handle it:

DoDialog
        CLR.L   -(SP)           ;Space for result
        MOVE    #ModalID,-(SP)  ;Resource ID of template
        MOVE.L  #0,-(SP)        ;0 = storage on heap
        MOVE.L  #-1,-(SP)       ;-1 = window at front
        _GetNewDialog           ;Create the dialog
        MOVE.L  (SP)+,DialogPtr(A5)   ;Save dialog pointer

        BSR     DoDefault       ;Highlight default button

DialogLoop

        MOVE.L  #0,-(SP)        ;No filter procedure
        PEA     itemNumber(A5)  ;Item number returned here
        _ModalDialog            ;Get user input

        CMP     #1,itemNumber(A5)   ;"You Bet!" button?
        BEQ     DialogErase     ;Yes, so remove dialog

        MOVE    #60,-(SP)       ;One-second (60 tick) beep
        _SysBeep
        BRA     DialogLoop      ;And try again

DialogErase

        MOVE.L  DialogPtr(A5),-(SP)
        _DisposDialog           ;Get rid of dialog box
        RTS


* DoDefault draws a three-pixel wide border around the
* default button in a dialog box. (The button must be the
* first item.) The border is separated from the button
```

Listing 8-1. *continued*

```
* rectangle by a one-pixel gap.

DoDefault

        PEA     OldPort         ;VAR result
        _GetPort                ;Get current drawing window

        MOVE.L  DialogPtr(A5),-(SP)
        _SetPort                ;Make dialog active for drawing

        MOVE.L  DialogPtr(A5),-(SP)   ;Dialog pointer
        MOVE    #1,-(SP)        ;Item #1
        PEA     itemType        ;VAR item type
        PEA     itemHndl        ;VAR item handle
        PEA     itemRect        ;VAR item rectangle
        _GetDItem               ;Get item info
        PEA     itemRect        ;VAR item rectangle
        MOVE    #-4,-(SP)       ;Expand left/right 4 pixels
        MOVE    #-4,-(SP)       ;Expand top/bottom 4 pixels
        _InsetRect              ;Calculate new rectangle

        MOVE    #3,-(SP)        ;Pen width
        MOVE    #3,-(SP)        ;Pen height
        _PenSize                ;Set new pen size

        PEA     itemRect        ;VAR item rectangle
        MOVE    #16,-(SP)       ;Width of corner oval
        MOVE    #16,-(SP)       ;Height of corner oval
        _FrameRoundRect         ;Draw dark border

        MOVE.L  OldPort,-(SP)
        _SetPort                ;Restore original drawing window

        RTS


* HandleEvent is the event dispatcher. It takes the event
* type code returned by _GetNextEvent and calls the subroutine
* that handles it. Access to the event handling subroutines is
* through a 16 entry jump table.

HandleEvent
```

Listing 8-1. *continued*

```
        MOVE    EventRecord+evtNum,D0
        CMP     #8,D0           ;Event 9-15?
        BHI     Ignore          ;Yes, so ignore
        ASL     #2,D0           ;Two shifts = times 4
        JMP     JumpTable(PC,D0)   ;Jump to handler

Ignore  RTS

JumpTable

        JMP     Ignore          ;Null event (never used)
        JMP     DoMouseDown     ;Button-down
        JMP     Ignore          ;Button-up
        JMP     DoKeyDown       ;Key-down
        JMP     Ignore          ;Key-up
        JMP     DoKeyDown       ;Auto-key
        JMP     DoUpdate        ;Update
        JMP     Ignore          ;Disk-inserted
        JMP     DoActivate      ;Activate

DoKeyDown
        RTS
DoUpdate
        RTS
DoActivate
        RTS
DoMouseDown

        CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)   ;Where
        PEA     WindowPtr(A5)
        _FindWindow             ;Where was button pressed?

        MOVE    (SP)+,D0        ;Get result
        CMP     #InMenuBar,D0   ;Pressed in menu bar?
        BEQ     QuitCheck       ;Yes, so check it out
        RTS                     ;Ignore everything else

; See if "QUIT" was selected from File menu:

QuitCheck
```

Listing 8-1. *continued*

```
        MOVE.L  #0,-(SP)        ;result = menu/item selected
        PEA     EventRecord+evtMouse   ;Where
        _MenuSelect             ;Get menu selection
        MOVE    (SP)+,MenuNum(A5)   ;Save menu number
        MOVE    (SP)+,D0        ;Discard item number

        MOVE    #0,-(SP)
        _HiliteMenu             ;Remove highlight from menu title

        CMP     #FileID,MenuNum(A5)   ;In the FILE menu?
        BNE     GetEvent


* Must have selected QUIT command, so return to Finder by
* popping the subroutine return address before RTS. (We
* could also return just by executing an _ExitToShell
* instruction.)

        MOVE.L  (SP)+,D0        ;Pop the return address (long!)

        RTS                     ;Return to Finder

; Record for _GetNextEvent:

EventRecord     DCB.B   EvtBlkSize,0   ;Reserve space for record

oldPort         DC.L    0       ;Window ptr for _GetPort
itemType        DC      0       ;Item type for _GetDItem
itemHndl        DC.L    0       ;Item handle for _GetDItem
itemRect        DCB.W   4,0     ;Dialog rectangle for _GetDItem

; Here are the program globals. Use (A5) addressing.

AppleH          DS.L    1       ;Handle to Apple menu
FileH           DS.L    1       ;Handle to File menu

WindowPtr       DS.L    1       ;Pointer to window

MenuNum         DS.W    1       ;Menu number selected

DialogPtr       DS.L    1       ;Pointer to dialog record

itemNumber      DS.W    1       ;Item number for modal dialog
```

Listing 8-1. *continued*

```
; Linker Control File
; Modal.Link
;
; Link this file to create application
;  (without resources).

Modal
$
```

```
* RMaker Source File
* Modal.R
*
* Compile this after assembling and linking Modal.Asm
*
* The next command appends the resources to the application:
!Book:Modal

Type MENU
,1                      ;;Resource ID
14                      ;;Title is the Apple symbol (ASCII $14)
About this demo...      ;;About box

,2                      ;;Resource ID
File                    ;;Menu Title
  Quit                  ;;Only item is "Quit"

Type WIND
,128                    ;;Resource ID
Modal Dialog Demo       ;;Title for Window
40 5 332 502            ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)

Type DLOG               ;;Modal Dialog
,128                    ;;Resource ID
                        ;;No title
100 100 200 350         ;;TLBR
V N                     ;;Visible, No Goaway
1                       ;;Standard dialog box type
```

Listing 8-1. *continued*

```
0                       ;;User-definable (not used)
128                     ;;Resource ID of DITL resource

Type DITL               ;;Item list for DLOG (128)
,128                    ;;Resource ID
3                       ;;Number of items

Button                  ;;Button (item #1 - default)
60 20 90 90
OK

Button                  ;;Button (item #2)
60 135 90 205
No Way!

Static Disabled         ;;Static text item (item -3 - disabled)
20 30 40 400
Do you want to continue?
```

To use _ModalDialog, first push the pointer to a filter procedure, then the address of the variable in which an item number is to be returned:

```
            MOVE.L  #0,-(SP)        ;No filter procedure
            PEA     itemNumber(A5)  ;Item number variable
            _ModalDialog

ItemNumber DS.W     1               ;Item number returned here
```

A **filter procedure** is a subroutine that _ModalDialog calls after it detects an event but before it responds to it. This lets you modify the effect of events any way you want. If you're not using a custom filter procedure (the usual case), just push a long word zero on the stack. This invokes the standard filter function, converting the press of the RETURN or ENTER key to a mouse click in the first item in the dialog box. In typical applications, the first item will be a button. See

Inside Macintosh for the technical specifications of a dialog filter procedure.

When _ModalDialog takes over, it handles any update events related to the dialog (caused when a control item like a button or a check box changes value), and monitors all events until an active item is selected. It beeps if the mouse is clicked outside the dialog window and ignores all clicks inside the window if they're not also inside the display rectangle of an enabled item. Button-down events in control items like buttons and check boxes are monitored until the button is released; if the mouse is not in the control at the end of the click, the click is ignored.

When _ModalDialog finishes, the itemNumber variable contains the number of the item selected. With one exception, disabled items cannot be selected, so this number corresponds to an active item. The exception occurs when you use the standard filter procedure and RETURN or ENTER is pressed: in this situation, _ModalDialog always returns a one even if the first item is disabled. The first item should always be an enabled button, however.

In the subroutine in Listing 8-1, the dialog is erased and disposed of with a call to _DisposDialog if the first button is clicked (the "You Bet!" button). If the other button is clicked, the speaker beeps for one second and the subroutine calls _ModalDialog again. _ModalDialog does not report clicks in the text item because it is marked as disabled in the resource file.

Listing 8-1 also shows how to highlight the default button (the first item). The DoDefault subroutine uses _GetDitem to determine the bounding rectangle for the button item, extends this rectangle by four pixels in all directions using _InsetRect with negative parameters, then draws a dark border around the new rectangle with a three-pixel wide pen.

The action a program takes when _ModalDialog returns a result depends on the type of item selected. If it was a check box, the box should be checked if it was previously disabled or vice versa. If it was a radio button, the radio button should

be selected and all other radio buttons in the group should be disabled.

When a button is selected, you will normally erase the dialog window from the screen. See the following section, **Removing Dialog Boxes From the Screen**, for instructions on how to do this.

Key-down events are processed by _ModalDialog only if there is a variable text box in the dialog. If the text box is enabled, its item number is returned after every key press. If it's not, no item number is returned, but you can still edit the string in the text box. It's probably best to disable text boxes so you don't have to keep looping back to _ModalDialog after every key press. Instead, after a button is pressed to dismiss the dialog, you can use _GetIText to determine the final value of the text string.

The TAB key is used to move the text insertion cursor from one variable text box to the next. If you're in the last text box when you press TAB, you will proceed to the first text box.

Button-down events in a variable text box are automatically handled by _ModalDialog in a manner consistent with the Macintosh user interface guidelines for text selection. That is, if the mouse is clicked, a blinking cursor appears and subsequent keystrokes are inserted at that point. If the mouse is dragged, a highlighted selection area appears that can be deleted by pressing the key for the next character to be inserted or the DELETE key. Later on in this chapter you'll see an example of how to manipulate items in a dialog box.

Modeless Dialog Boxes

A modeless dialog box is a bit more difficult to handle than a modal dialog box. When it is on the screen, the user is not restricted from performing other operations like moving to another window or selecting a desk accessory, before dismissing it to remove it from the screen. In this respect it's simply like any other standard window. Unlike a modal dialog or an alert, it does not retain control until you select an active

item. You simply feed it events one at a time and it returns a Boolean result that tells you whether the event related to the dialog box or not.

The Macintosh toolbox does make it somewhat easier to deal with a modeless dialog box than a window, however. Whenever your program calls _GetNextEvent and detects an event has occurred, it should call _IsDialogEvent to determine whether the event relates to the modeless dialog box.

```
        CLR.B   -(SP)                   ;Space for Boolean result
        PEA     EventRecord             ;_GetNextEvent record
        _IsDialogEvent
        MOVE.B  (SP)+,D0                ;Pop true/false result

EventRecord     DCB.B   EvtBlkSize,0    ;_GetNextEvent's record
```

If the result is false, the event was not dialog-related and can be processed as usual. If the result is true, you must immediately call _DialogSelect to handle the event.

```
        CLR.B   -(SP)           ;Space for Boolean result
        PEA     EventRecord     ;_GetNextEvent record (constant)
        PEA     DialogPtr(A5)   ;Dialog pointer (variable)
        PEA     itemNumber      ;Item number (constant)
        _DialogSelect
        MOVE.B  (SP)+,D0        ;Pop the result

itemNumber      DC.W    0       ;Item number returned
DialogPtr       DS.L    1       ;Pointer to dialog record
```

_DialogSelect takes the event, processes it, and returns a Boolean result indicating whether it related to an enabled dialog item. If the result is false, it didn't, and you don't have to do anything further. _DialogSelect always returns a false value if you pass it a window update, deactivate, or activate event; these events are processed internally.

If the result is true, the itemNumber and DialogPtr variables will contain the number of the item selected and a pointer to the active dialog record. You can then deal with the result in the same way you deal with a result returned by

a call to _ModalDialog for a modal dialog box. _DialogSelect always returns a true result in situations where _ModalDialog would have reported an item-related event to you.

Drawing Within Dialog Boxes

You can also display text and graphics within a dialog box using the same instructions used with ordinary windows. (See Chapter 6.) Before drawing, however, make the dialog box the active drawing window using the _SetPort instruction.

If you erase items within a dialog box, you can redraw them by pushing the pointer to the dialog window on the stack and calling _DrawDialog. You must do this because erasing does not cause an update event that will be dealt with during the next call to _ModalDialog.

Removing Dialog Boxes From the Screen

When a dialog box is dismissed, you can use _HideWindow to make it invisible with these instructions:

```
MOVE.L  DlogPtr(A5),-(SP)       ;Pointer to dialog
_HideWindow
```

If you want to make it visible again later, use _ShowWindow, by passing it the same dialog pointer on the stack.

When you're through with a dialog box for good, erase it from the screen, remove it from the list of active windows maintained by the Macintosh operating system, and free up the memory it occupies.

The method of erasing depends on how you created the dialog in the first place. If you told the toolbox to automatically allocate storage for the dialog record on the heap, use _DisposDialog. This frees up all storage associated with the dialog, including the space used by the item list template rec-

ord. (Recall if you pass a pointer of zero to the dialog record that _NewDialog or _GetNewDialog requires, the toolbox automatically allocates storage space on the heap for the dialog record.)

If you passed your own pointer to the space for the dialog record, use _CloseDialog instead. This will free up the space associated with the various fields in the dialog record, except the item list record. The dialog record itself is not affected. To free up the spaces used by the item list and dialog records, use _DisposPtr. (See Chapter 4.)

Both _CloseDialog and _DisposDialog require only one parameter to be passed on the stack: The pointer to the dialog window be destroyed.

A Dialog Box Program

The program in Listing 8-2 shows how to create the dialog box shown in Figure 8-3. This box is made up of nine items: one button (the default item), an variable text box, three radio buttons, a check box, and three static text items. The static text and variable text items are disabled, so _ModalDialog will not return a result when they are clicked, or if text is entered. When the button is clicked to dismiss the dialog, the final value of the variable text string is displayed in the active window.

Listing 8-2. The Source File, Linker Control File, and RMaker File for the Items Program.

```
; Asm Source File
; Items.Asm
;
; This is an example of how to manipulate items in dialog boxes.

AppleID         EQU     1       ;Menu ID for Apple Menu
FileID          EQU     2       ;Menu ID for File Menu
WindID          EQU     128     ;Window ID
DialogID        EQU     129     ;Dialog ID
```

Listing 8-2. *continued*

```
        INCLUDE ToolEqu.D       ;Toolbox equates
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QD globals area
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event


; The resources are in a separate resource file so that we
;  don't have to re-Asm and re-Link every time we fiddle
;  with a resource:


        CLR     -(SP)
        PEA     'Items.Rsrc'
        _OpenResFile
        MOVE    (SP)+,D0

; Create and draw a window on the screen:

        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file


; The next step is very important. It ensures that our new
; window is the active port, so we can draw in it.
```

Listing 8-2. *continued*

```
        _SetPort                ;Make window current GrafPort

; Create two standard menus:

        CLR.L   -(SP)           ;Space for handle
        MOVE    #AppleID,-(SP)  ;Menu ID number
        _GetRMenu               ;Get Menu from resource file
        MOVE.L  (SP)+,AppleH(A5);Save menu handle
        CLR.L   -(SP)           ;Space for handle
        MOVE    #FileID,-(SP)   ;Menu ID number
        _GetRMenu               ;Get menu from resource file
        MOVE.L  (SP)+,FileH(A5) ;Save menu handle

; Add menus to menu bar:

        MOVE.L  AppleH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        MOVE.L  FileH(A5),-(SP)
        MOVE    #0,-(SP)        ;(0 = add to end)
        _InsertMenu             ;Add to menu bar

        _DrawMenuBar            ;Display menu bar

        CLR.L   -(SP)           ;Space for result
        MOVE    #DialogID,-(SP) ;Resource ID of template
        MOVE.L  #0,-(SP)        ;0 = storage on heap
        MOVE.L  #-1,-(SP)       ;-1 = window at front
        _GetNewDialog           ;Create the dialog
        MOVE.L  (SP)+,DialogPtr(A5)   ;Save dialog pointer

        BSR     DoDefault       ;Highlight default button


* Set the states of all the control items. Their values are all
* zero at the beginning, so we only have to change those that
* are non-zero.

        MOVE    #4,D5           ;Item 4 (1200 baud)
        MOVE    #1,D6           ;Value = 1 (on)
        BSR     SetIStatus      ;Highlight radio button
```

Listing 8-2. *continued*

```
        MOVE    #6,D5           ;Item 6 (check box)
        MOVE    #1,D6           ;On
        BSR     SetIStatus      ;Put X in box


* Select the entire variable text string. The selected text is
* marked by the following _ModalDialog instruction.

        MOVE.L  DialogPtr(A5),-(SP)
        MOVE    #2,-(SP)        ;It's item #2
        MOVE    #0,-(SP)        ;Start at beginning!
        MOVE    #241,-(SP)      ;End at end! (241 is max size)
        _SelIText               ;Do the selection

DialogLoop
        MOVE.L  #0,-(SP)        ;No filter procedure
        PEA     itemNumber(A5)  ;Item number returned here
        _ModalDialog            ;Get user input

        MOVE    itemNumber(A5),D5   ;Put item number in D5

        CMP     #1,D5           ;OK button?
        BEQ     DialogExit      ;Yes, so we're all done

        CMP     #6,D5           ;Check box
        BNE     @1              ;No, so branch


* Toggle the state of the check box by changing it to 0
* if it is 1 or to 1 if it is 0 using EOR.

        BSR     GetItemInfo

        CLR     -(SP)           ;Space for result
        MOVE.L  itemHndl,-(SP)
        _GetCtlValue            ;Read value for check box
        MOVE    (SP)+,D6        ;Get result (0 or 1)

        EORI    #1,D6           ;Flip bit 0 (contains value)
        BSR     SetIStatus      ;Remove the X
        BRA     DialogLoop      ;Back to dialog!
```

Listing 8-2. *continued*

```
* The only other possibilities are the three radio buttons
* (3, 4, 5). First turn them all off, then turn on the one
* that was selected:

@1      MOVE    D5,D7           ;Save item # in D7 for now

        MOVEQ   #3,D5           ;Start with item 3
        MOVEQ   #0,D6           ;0 = off

@2      BSR     SetIStatus
        ADDQ    #1,D5           ;Move to next item #
        CMP     #6,D5           ;Past the end?
        BNE     @2              ;No, so branch

        MOVE    D7,D5           ;Get item # back
        MOVEQ   #1,D6           ;1 = on
        BSR     SetIStatus
        BRA     DialogLoop      ;Back to dialog

DialogExit


* Read the value of the variable text box item:

        MOVEQ   #2,D5           ;Variable text item
        BSR     GetItemInfo     ;Get item attributes

        MOVE.L  itemHndl,-(SP)  ;Handle to text item
        PEA     theText         ;VAR the text string
        _GetIText               ;Read string into theText

        MOVE.L  DialogPtr(A5),-(SP)
        _DisposDialog           ;Get rid of dialog box


* Display the text string

        MOVE    #25,-(SP)
        MOVE    #25,-(SP)
        _MoveTo                 ;Position the pen

        PEA     'The text entered was...
```

Listing 8-2. *continued*

```
        _DrawString

        PEA     theText
        _DrawString

        JMP     MainLoop        ;All done.


* Here are the subroutines used in this example:
* GetItemInfo gets the properties of the dialog item whose
* number is in D5. The handle to the dialog must be stored at
* DialogPtr. The results are returned in the constants itemType,
* itemHndl, and itemRect.

GetItemInfo

        MOVE.L  DialogPtr(A5),-(SP)
        MOVE    D5,-(SP)        ;Item number
        PEA     itemType        ;VAR item type
        PEA     itemHndl        ;VAR item handle
        PEA     itemRect        ;VAR item rectangle
        _GetDItem               ;Get attributes of item
        RTS


* SetIStatus sets the value of a control item. On entry, D5.W
* contains the item number and D6.W contains its value. The
* pointer to the dialog must be in the DialogPtr variable.
*
* Buttons, check boxes, and radio buttons can be set to one
* of two values: off (0) or on (1).
*
* Note that _SetCtlValue causes an update event for the dialog
* window handled by _ModalDialog.

SetIStatus

        BSR     GetItemInfo     ;Get item attributes

        MOVE.L  itemHndl,-(SP)  ;Push handle to item
        MOVE    D6,-(SP)        ;0=off, 1=on
        _SetCtlValue            ;Set new value
```

Listing 8-2. *continued*

```
        RTS


* DoDefault draws a three-pixel wide border around the default
* button in a dialog box. (The button must be the first item).
* The border is separated from the button rectangle by a
* one-pixel gap.


DoDefault

        PEA     OldPort         ;VAR result
        _GetPort                ;Get current drawing window

        MOVE.L  DialogPtr(A5),-(SP)
        _SetPort                ;Make dialog active for drawing

        MOVE.L  DialogPtr(A5),-(SP)   ;Dialog pointer
        MOVE    #1,-(SP)        ;Item #1
        PEA     itemType        ;VAR item type
        PEA     itemHndl        ;VAR item handle
        PEA     itemRect        ;VAR item rectangle
        _GetDItem               ;Get item info

        PEA     itemRect        ;VAR item rectangle
        MOVE    #-4,-(SP)       ;Expand left/right 4 pixels
        MOVE    #-4,-(SP)       ;Expand top/bottom 4 pixels
        _InsetRect              ;Calculate new rectangle

        MOVE    #3,-(SP)        ;Pen width
        MOVE    #3,-(SP)        ;Pen height
        _PenSize                ;Set new pen size

        PEA     itemRect        ;VAR item rectangle
        MOVE    #16,-(SP)       ;Width of corner oval
        MOVE    #16,-(SP)       ;Height of corner oval
        _FrameRoundRect         ;Draw dark border

        MOVE.L  OldPort,-(SP)
        _SetPort                ;Restore original drawing window

        RTS
```

Listing 8-2. *continued*

```
* Constants and variables:

oldPort         DC.L    0       ;Window ptr for _GetPort
itemType        DC      0       ;Item type for _GetDItem
itemHndl        DC.L    0       ;Item handle for _GetDItem
itemRect        DCB.W   4,0     ;Dialog rectangle for _GetDItem

theText         DCB.B   242,0   ;Variable text item

DialogPtr       DS.L    1       ;Dialog pointer
itemNumber      DS.W    1       ;item number selected


* The common code begins here:

MainLoop

        BSR     GetEvent
        BSR     HandleEvent
        BRA     MainLoop

GetEvent

        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1, -(SP)      ;Allow all events
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        MOVE    (SP)+, D0       ;Pop the result code
        BEQ     GetEvent        ;Branch if null event
        RTS

* HandleEvent is the event dispatcher. It takes the event type
* code returned by _GetNextEvent and calls the subroutine that
* handles it. Access to the event handling subroutines is
* through a 16 entry jump table.

HandleEvent

        MOVE    EventRecord+evtNum,D0   ;Get event code
        CMP     #8,D0           ;Event 9-15?
        BHI     Ignore          ;Yes, so branch
        ASL     #2,D0           ;Two shifts = times 4
```

Listing 8-2. *continued*

```
        JMP     JumpTable(PC,D0)    ;Jump to handler

Ignore  RTS

JumpTable

        JMP     Ignore          ;Null event (never used)
        JMP     DoMouseDown     ;Button-down
        JMP     Ignore          ;Button-up
        JMP     DoKeyDown       ;Key-down
        JMP     Ignore          ;Key-up
        JMP     DoKeyDown       ;Auto-key
        JMP     DoUpdate        ;Update
        JMP     Ignore          ;Disk-inserted
        JMP     DoActivate      ;Activate

DoKeyDown
        RTS
DoUpdate
        RTS
DoActivate
        RTS
DoMouseDown

        CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)   ;Where
        PEA     WindowPtr(A5)
        _FindWindow             ;Where was button pressed?

        MOVE    (SP)+,D0        ;Get result
        CMP     #InMenuBar,D0   ;Pressed in menu bar?
        BEQ     QuitCheck       ;Yes, so check it out
        RTS                     ;Ignore everything else

; See if QUIT was selected from File menu:

QuitCheck

        MOVE.L  #0,-(SP)        ;Result = menu/item selected
        PEA     EventRecord+evtMouse   ;Where
```

Listing 8-2. *continued*

```
        _MenuSelect             ;Get menu selection
        MOVE    (SP)+,MenuNum(A5)   ;Save menu number
        MOVE    (SP)+,D0        ;Discard item number

        MOVE    #0,-(SP)
        _HiliteMenu             ;Remove highlight from title

        CMP     #FileID,MenuNum(A5)   ;In the FILE menu?
        BNE     GetEvent


; Must have selected QUIT command, so return to Finder by
; popping the subroutine return address before RTS. (You
; could also return just by executing an _ExitToShell
; instruction.)

        MOVE.L  (SP)+,D0        ;Pop the return address (long!)

        RTS                     ;Return to Finder

; Record for _GetNextEvent:

EventRecord     DCB.B   EvtBlkSize,0   ;Reserve space for record

; Here are some globals. Use (A5) addressing.

MenuNum         DS.W    1       ;Menu number selected
AppleH          DS.L    1       ;Handle to Apple menu
FileH           DS.L    1       ;Handle to File menu
WindowPtr       DS.L    1       ;Pointer to window
```

```
; Linker Control File
; Items.Link
;
; Link this file to create application
;  (without resources).

Items
$
```

Listing 8-2. *continued*

```
* RMaker Source File
* Items.R
*
* Compile this after assembling and linking Items.Asm
*
* The next command creates a separate resource file:
Items.Rsrc

Type MENU
,1                      ;;Resource ID
\14                     ;;Title is the Apple symbol (ASCII $14)
About this demo...      ;;About box

,2                      ;;Resource ID
File                    ;;Menu Title
Quit                    ;;Only item is Quit

Type WIND
,128                    ;;Resource ID
Dialog Items Demo       ;;Title for Window
40 5 332 502            ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)

Type DLOG               ;;Modal Dialog
,129                    ;;Resource ID
                        ;;No title
75 81 225 431           ;;TLBR
V N                     ;;Visible, No Goaway
1                       ;;Standard dialog box type
0                       ;;User-definable (not used)
129                     ;;Resource ID of DITL resource

Type DITL               ;;Item list for DLOG (129)
,129                    ;;Resource ID
9                       ;;Number of items

Button                  ;;Button (item #1 - default)
120 310 140 340
OK
```

Listing 8-2. *continued*

```
EditableText Disabled   ;;Variable text box (disabled)
40 130 56 280
687-7144                ;;The text

Radio
70 110 86 160
300                     ;;300 baud message

Radio
70 185 86 235
1200                    ;;1200 baud message

Radio
70 260 86 310
2400                    ;;2400 baud message

Checkbox
100 110 116 215
Capture Text

StaticText Disabled
10 70 26 320
\14 COMMUNICATIONS PARAMETERS \14

StaticText Disabled
40 10 56 110
Phone Number

StaticText Disabled
70 10 86 95
Baud Rate
```

The dialog box is created and displayed in the usual way, using _GetNewDialog. The DoDefault subroutine presented in the previous programming example is then called to highlight the default button.

Before calling _ModalDialog to monitor user input, the application sets the states of all the control items and selects the entire variable text string. Since this is done after displaying the dialog box on the screen, the box appears to flicker

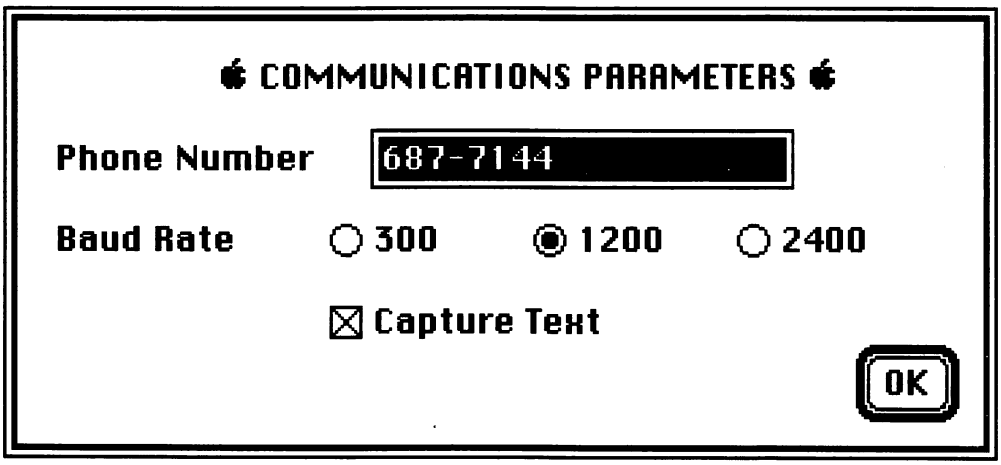


Figure 8-3. The Dialog Box Created by the Program in Listing 8-2.

slightly as the active control items are highlighted and the text is selected during the first call to _ModalDialog. You can eliminate the flicker by defining the dialog box as Invisible in the resource file; _GetNewDialog will create it but not display it. After adjusting the controls and text to the appropriate values, you can use _ShowWindow to display the dialog box window in its complete starting form. Remember from Chapter 6, _ShowWindow requires a pointer to a window record that is equivalent to the dialog pointer returned by _GetNewDialog as a parameter.

All the control items are initially off, as they always are right after a dialog box is loaded from a resource file. To indicate that the "1200 baud" radio button is the default, it is turned on by calling the SetIStatus subroutine.

SetIStatus expects the item number in D5 and its value in D6. It first calls the application's GetItemInfo subroutine to get the properties of the item, then it sets the value using _SetCtlValue. (GetItemInfo uses _GetDItem to determine the item type, handle, and bounding rectangle.) This call causes an update event for the dialog window that is handled during the text call to _ModalDialog. It is handled by drawing a small black circle inside the active radio button. Once the

proper radio button has been turned on, the check box is turned on using the very same technique.

The last preliminary step is to highlight the text in the variable text box. This is done to ensure that the whole entry will be deleted if the user starts entering a new phrase. To ensure that the whole string is selected, selection endpoints of 0 and 241 are pushed on the stack; the maximum length of this type of item is 241 characters.

The application then calls _ModalDialog to request input from the user. If the dialog button is clicked, control passes to DialogExit where the final value of the text string is loaded into theText. Notice how this is done: GetItemInfo is called to get the handle to the text item, which is then passed to _GetIText to create a standard text string preceded by a length byte. DialogExit next disposes of the dialog box with _DisposDialog before displaying the text on the screen with _DrawString.

If the user selects the check box item, its value is toggled by calling _GetCtlValue to get its current value, flipping bit zero, where the value is kept, with an EORI #1,D6 instruction, then calling the SetIStatus subroutine to record the change.

If one of the three radio buttons is clicked, they are first all turned off and then the selected one is turned on. This ensures that only one item in the related group of radio buttons is on at any given time, as required.

Notice that the technique used to create this application is slightly different than usual. If you examine the source listing for the RMaker file, you will notice that the resources are stored in a separate file called Items.Rsrc—they are not appended to the application file. (The application opens this file with a _OpenResFile instruction.) This was done to help speed up the development process. If changes are made to the resources (to reposition the items in the alert box or change their rectangles, for example), all you need to do to incorporate the changes is run RMaker once again; there is no need to assemble and link again because the application file is not affected. Of course, once you're satisfied the

resources are in final form, you can append them to the application file to avoid having to open the resource file explicitly or running the risk of forgetting to copy the resource file when you make a copy of the application.

Creating Alert Templates

The standard toolbox instructions for creating and displaying alert boxes insist that a template describing the form of the box be stored in a resource file. The resource type for an alert template is ALRT.

As usual, you can use RMaker to create ALRT resources. The form of the source statements is as follows:

```
TYPE ALRT
 ,128                ;;resource ID of this ALRT
35 35 300 300        ;;alert rectangle (TLBR)
128                  ;;resource ID of DITL (item list)
DDDD                 ;;stages word (must be hexadecimal)
```

Just like a DLOG resource, an ALRT resource refers to a DITL item list resource containing descriptions of the items to be displayed in the alert box. Your alert boxes should only use static items like text, icons, pictures, or simple buttons. Control items such as radio buttons and check boxes should not be used as they are not meaningful in an alert box environment.

One important parameter in an ALRT resource is the **stages word**. It defines the behavior of the alert in each of four different stages. When you use the alert box for the first time, it enters the first stage. As you keep calling it up, it progresses through the second, third, and fourth stages, in that order. Thereafter, the alert box always behaves as if it was in the fourth stage.

The stage number minus one of the last alert box is always stored in the global variable ACount. The resource ID of the alert is stored in ANumber. Both these numbers are words. If

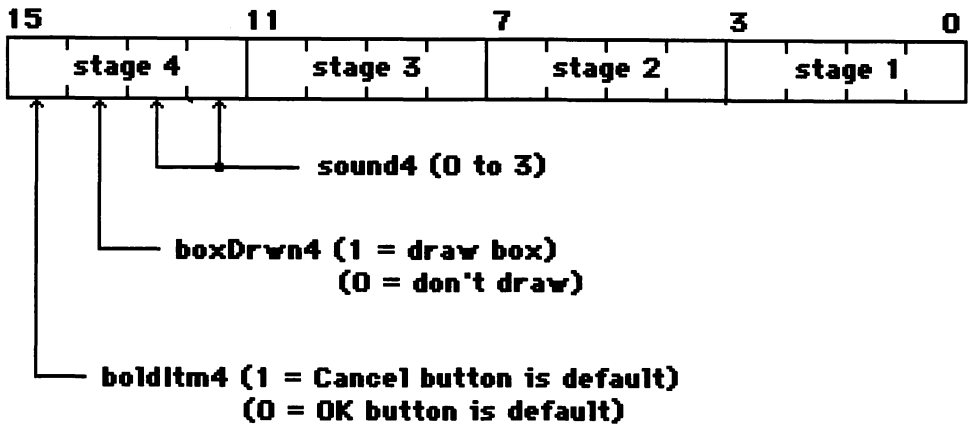


Figure 8-4. The Stages Word in an ALRT Resource.

you want to change the stage of an alert box, place the stage number, minus one, in ACount. Store minus one there if you want the first stage to be used the next time the alert box is called up.

Each of the four four-bit groups in the stages word define the characteristics of one stage. The high-order four bits control the fourth stage and the low-order four bits control the first stage. The characteristics associated with each stage are: what the default button is to be, what sound is to be emitted, and whether the alert box is to be drawn.

For a given stage, the first two bits (0 and 1) contain a sound number from zero to three. In most cases, this represents the number of times the speaker is to beep when an alert is called up at that stage level. It is possible to invoke a custom sound procedure that interprets these numbers differently, however. See *Inside Macintosh* for details of how to do this.

The next bit (bit 2), boxDrwn, indicates whether the alert box is to be drawn on the screen. You will usually set this bit to one (display the box), but it can be set to zero if you don't want the alert to be displayed at that stage level.

The last bit (bit 3), boldItm, controls which of two buttons is to be the default button. The default button is the one selected when RETURN or ENTER is pressed from the key-

board. If the bit is set to zero, the first button (usually an OK button) is the default; if it is one, the second button (usually a Cancel button) is the default.

In most applications you will probably want all stages to be equivalent, so all four fields will be the same. For example, if you want to beep the speaker once, display the alert box, and make the Cancel button the default button, use a stages word of $DDDD. For each stage this sets the sound number to 01 (one beep), the boxDrwn bit to 1 (draw the box), and the boldItm bit to 1 (the default is Cancel).

Using Alert Boxes

Alert boxes are very easy to use because all screen and event activities are handled by a single instruction that creates the alert record, draws the alert box and its items, interprets events until an active item is selected, erases the alert from the screen, and then disposes of any memory used by the alert record. The instruction returns the item number selected. All you have to do is monitor this result and take whatever action is appropriate. Compare this with dialog boxes where you have to use different instructions to create and dispose of the dialog.

There are four standard toolbox instructions you can use to display an alert box on the screen: _Alert, _NoteAlert, _CautionAlert, and _StopAlert. They all use the same calling sequence:

```
CLR      -(SP)          ;Space for result
MOVE     #133,-(SP)     ;ALRT Resource ID
MOVE.L   #0,-(SP)       ;Filter procedure pointer
_Alert                  ;or _NoteAlert, _CautionAlert
                        ; or _StopAlert
MOVE     (SP)+,D0       ;Get item number selected
```

The pointer to a filter procedure is similar to the one described earlier for dialog boxes. Pushing a zero value tells

the toolbox to use the standard filter procedure. It converts a RETURN or ENTER keypress into the click of the default button in the alert box.

The only difference between the four alert box instructions is the icon they display in the top left-hand corner of the box. _Alert displays no icon at all. The icons displayed by the other instructions are shown in Figure 8-5.

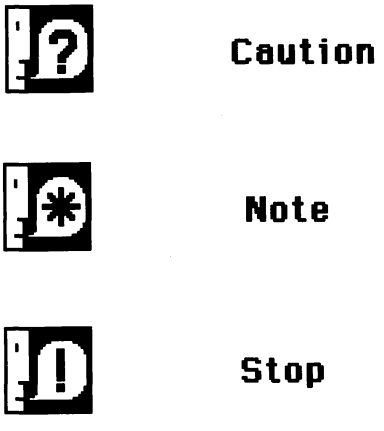


Figure 8-5. The Standard Alert Box Icons.

The coordinates of the top left-hand corner of an icon used by _NoteAlert, _CautionAlert, and _StopAlert are (10,20). Since an icon is 32-pixels square, don't define item rectangles that overlap the square whose corner points are (10,20) and (42,52).

Once _Alert, or its three relatives, take control, mouse clicks are handled just like _ModalDialog handles them. That is, mouse clicks outside the alert box produce error beeps, and clicks in disabled items are ignored.

Chapter 9

Supporting Desk Accessories

Everyone who uses the Macintosh soon comes to appreciate the **desk accessories** (DAs). Desk accessories, as you know, are small utility programs that are always there when you need them. While you're in the middle of a brainstorming session with an application, you can put it on hold and quickly call up a DA by selecting its name from the standard Apple menu. When you're through with the DA, you can return to the application and continue to use it as if you had never left it.

Some familiar desk accessories are the Scrapbook, Alarm Clock, Note Pad, Calculator, Key Caps, Control Panel, and Puzzle, some of which are shown in Figure 9-1. There are dozens of others you can purchase from independent publishers and add to the System program or your own applications.

It is important to realize that applications do not automatically support desk accessories. It is up to you, when writing an application, to include the instructions needed to activate them when appropriate events occur, and to switch between them and your application. In this chapter you'll see what these instructions are and how to use them. These instructions make up the Macintosh Desk Manager and are summarized in Table 9-1.

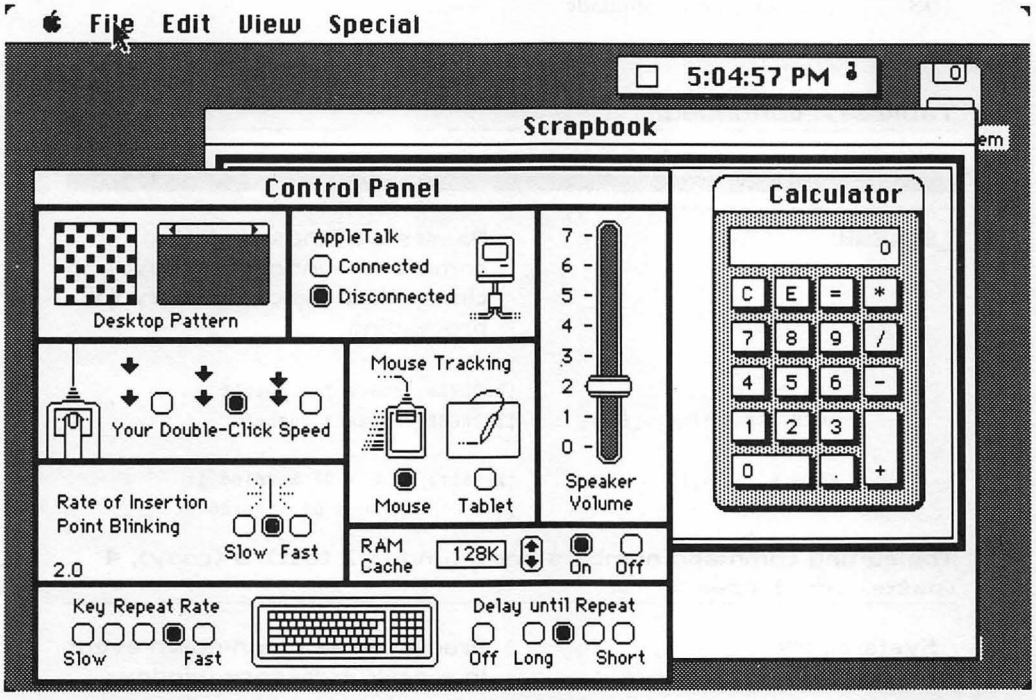


Figure 9-1. Some Macintosh Desk Accessories.

Table 9-1. Desk Manager Trap Instructions.

_CloseDeskAcc	**Closes a desk accessory and removes its window from the screen.**

```
MOVE    #refNum,-(SP)       ;INTEGER: reference number
_CloseDeskAcc
```

_OpenDeskAcc	**Opens a desk accessory and passes control to it.**

```
CLR     -(SP)               ;INTEGER: space for result
PEA     accName             ;STRING: name of DA to open
_OpenDeskAcc
MOVE    (SP)+,D0            ;Result is a reference number
```

Table 9-1. *continued*

_SysEdit — **Passes a standard editing command—undo, cut, copy, and clear—to a desk accessory for processing.**

```
CLR.B  -(SP)              ;BOOLEAN: space for result
MOVE   #editCmd,-(SP)     ;INTEGER: command number
_SysEdit
MOVE.B (SP)+,D0           ;Result: true = DA handled it
                          ;        false = DA ignored it
```

The editing command numbers are 0 (undo), 2 (cut), 3 (copy), 4 (paste), and 5 (clear).

_SystemClick — **Processes a button-down event in a desk accessory window.**

```
PEA    EventRecord        ;POINTER: to the event record
MOVE.L theWindow,-(SP)    ;POINTER: to window record
                          ;returned by _FindWindow
_SystemClick
```

_SystemTask — **Allows a desk accessory to perform a periodic function.**

```
_SystemTask               ;no parameters
```

Adding Desk Accessories to a Menu

A user cannot select a desk accessory unless its name appears in a menu at the top of the screen. By convention, the names of all the desk accessories available to an application are to be placed in a menu whose title appears on the left-hand side of the menu bar. Again, by convention, the title for this menu is the Apple symbol (ASCII code 20).

All desk accessories are stored in resource files and have resource types of DRVR. You can use _AddResMenu to add

their names to an Apple menu created with _GetRMenu or _NewMenu. (See Chapter 7.) All you need to do is execute these three instructions once you've created the menu:

```
MOVE.L MenuH1(A5),-(SP)         ;Handle to "Apple" menu
MOVE.L #'DRVR',-(SP)            ;Resource type code
_AddResMenu
```

MenuH1, the handle to the Apple menu, is the variable in which the handle returned by _NewMenu or _GetRMenu is stored when you first create the menu. For convenience, you should only use _AddResMenu after you've added application-specific items to the Apple menu using _AppendMenu. The two standard items you will normally add are an About... item and a dimmed dashed line item. The dashed line acts as a physical separator between your own special items and the general desk accessory items.

Putting it all together, here is a subroutine to create an Apple menu containing an About... item, a dashed line, and the names of every desk accessory in the System file, your program file, or any other open resource file:

```
        CLR.L   -(SP)               ;Clear space for handle
        MOVE    #1,-(SP)            ;Menu ID = 1
        PEA     M1Name              ;Title (Apple symbol)
        _NewMenu                    ;Create the Apple menu
        MOVE.L  (SP),-(SP)          ;Make copy of handle
        PEA  'About This Demo...;(-' ;Names of first two items
        _AppendMenu                 ;Add them to the menu

        MOVE.L #'DRVR',-(SP)        ;Resource type code
        _AddResMenu                 ;Add accessory names
        RTS

M1Name      DC.B    1,20            ;Length + Apple symbol
```

Notice that I pushed on the stack a second copy of the handle returned by _NewMenu. The first is eventually popped by the _AppendMenu instruction and the second by _AddResMenu.

You can further simplify this procedure by creating a MENU resource file that already includes the About... item and the dimmed dashed line. In this case, the subroutine you would use looks like this:

```
CLR.L   -(SP)              ;Space for handle
MOVE    #128,-(SP)         ;Resource ID of MENU
_GetRMenu                  ;Get the menu
MOVE.L  (SP),MenuHl(A5)    ;Save menu handle
MOVE.L #'DRVR',-(SP)       ;Resource type code
_AddResMenu                ;Add accessory names
RTS
```

You can omit the fourth instruction if you won't be needing the menu handle later on in your program.

Opening Desk Accessories

While your program is running and the Apple menu is enabled, a user can open a desk accessory by pulling down the Apple menu and selecting the DA by name. For this to be possible, however, your program must react in the usual way to button-down events in the menu bar.

After calling _FindWindow and determining that a button-down event has occurred, call _MenuSelect to determine the menu ID and the number of the item selected. If the menu ID isn't that of the Apple menu, you can process the event in the usual way. If the Apple menu was involved, however, check to see if the item number is that of a desk accessory. If you've followed the suggestions in the previous section and created a menu beginning with two custom items, menu IDs of three or higher refer to desk accessories.

To pass control to a desk accessory, push a pointer to its name and call _OpenDeskAcc. To determine what its name is, use the _GetItem instruction to convert the item number to an item name:

```
        MOVE.L  MenuHndl(A5),-(SP)      ;Handle to Apple menu
        MOVE    #4,-(SP)                ;Item number (assume 4)
        PEA     DAName                  ;Location for name
        _GetItem                        ;Get the name

        CLR     -(SP)                   ;Clear space for result
        PEA     DAName                  ;Pointer to name
        _OpenDeskAcc
        MOVE    (SP)+,DARef(A5)         ;Save DA reference #

DAName      DCB.B 16,0                  ;Name of DA (length+15)
MenuHndl    DS.L  1                     ;Handle to Apple menu
DARef       DS    1                     ;DA reference #
```

Since the name of a DA cannot exceed 15 characters, 16 bytes are reserved for DAName (the extra byte is for a preceding length byte).

You will not usually need to use the reference number returned by _OpenDeskAcc. It must be pushed on the stack before calling _CloseDeskAcc to close a desk accessory and remove it from the screen, but this operation is normally handled for you by the desk accessory itself when you click its close box. You may, however, want to close a desk accessory from your application if the Close item is selected from the standard File menu while a desk accessory window is active.

When you pass control to the desk accessory with _OpenDeskAcc, the DA takes over and performs its duties until you tell it you want to return to the main application. Exactly how you return depends on the accessory. Sometimes you must click a close box to dismiss the DA entirely; other times you can click your application's window to activate it and deactivate the accessory window. If you use the latter method, the window for the DA still appears on the screen, and if you click it the DA becomes active again.

Desk Accessories and Mouse Clicks

As you saw in Chapters 4 and 6, when an event loop in your program detects a button-down event, you usually call

_FindWindow to determine what part of the screen was clicked. If the number returned by _FindWindow is InSysWindow, a desk accessory window (also called a **system window**) was clicked. This type of window is created when you open a desk accessory and remains on the screen until you click the close box of the DA's window.

It is very easy to handle a click in a desk accessory window. Simply make a call to _SystemClick as follows:

```
PEA     EventRecord             ;record for _GetNextEvent
MOVE.L  theWindow(A5),-(SP)     ;Pointer to window
_SystemClick
```

`EventRecord` is the record filled in by the call to `_GetNextEvent` that returned the button-down event. The variable `theWindow` contains the pointer to the window in which the mouse was clicked and is returned by _FindWindow.

_SystemClick passes the click to the desk accessory so it can deal with it as follows:

- If the desk accessory window is not active, it is activated and a deactivate event is posted for the currently active window.
- If the desk accessory is already active and the click is in a close box, _SystemClick calls _TrackGoAway to see if the accessory window should be closed.
- If the desk accessory is already active and the click is in the title bar, _SystemClick calls _DragWindow so the window can be moved.
- Clicks in the content region of an accessory window are handled in a manner dictated by the desk accessory.

Upon return from _SystemClick, return to your event loop to get the next event to be processed.

Desk Accessories and Editing

Many desk accessories use the standard text-editing commands described in the Macintosh user-interface guidelines: undo, cut, copy, paste, and clear. A little support from the

application is needed before they will work properly, however. In particular, your application must always include a standard Edit menu in the menu bar. It must be the third menu in the bar, and the ordering of the items must be as follows:

Undo
(A disabled item)
Cut
Copy
Paste
Clear

Just before you open a desk accessory by calling _OpenDeskAcc or reactivate it with _SystemClick, you should ensure that all these editing commands are enabled using _EnableItem so they will be available when the accessory gains control. When the application takes over once again, you can disable any items that have no meaning to your application.

When a standard editing command is selected from the Edit menu, call _SysEdit to give a desk accessory a chance to claim it.

```
CLR.B   -(SP)           ;Space for result
MOVE    #2,-(SP)        ;Item number minus 1
_SysEdit
TST.B   (SP)+           ;Test and pop the result
BEQ     YouEdit         ;Branch if not claimed
BRA     EventLoop       ;Go get next event
```

Notice that the item number passed to _SysEdit is one less than the item number returned by _MenuSelect. The _SysEdit item numbers for the standard editing commands are as follows:

Undo	0
Cut	2
Copy	3
Paste	4
Clear	5

Do *not* try to pass any other numbers to _SysEdit.

If the editing command is not claimed, the result is false and the BEQ branch will succeed, then control passes to YouEdit so that your application can deal with it. If the desk accessory did handle the editing command, you have nothing to do, so you can go get another event.

You don't have to pass to _SysEdit editing commands entered using keyboard equivalents. When a desk accessory is active it automatically detects and processes keyboard editing commands itself.

Periodic Functions of Desk Accessories

Some desk accessories are designed to periodically perform certain activities. Examples of such time-dependent activities are the blinking of the Note Pad's cursor, displaying the current time by the Alarm Clock, and updating the key cap display by the Key Caps accessory.

Before a desk accessory can perform these periodic functions, however, your program must call the _SystemTask instruction at least once every timer tick. If you don't, the accessory will be totally inactive while your application is in control. For this reason, you should place the _SystemTask instruction in any event loops used by your program. It's also a good idea to call it periodically during any lengthy processing operations. _SystemTask requires no parameters.

Initializing Toolbox Managers

Even though your application may not use certain toolbox Managers, such as the Dialog Manager or TextEdit, for example, it should initialize them in case they're needed by a desk accessory. You will encounter no difficulties if you use the standard initialization header referred to in Chapter 2.

Some desk accessories may also need to insert a new menu in the menu bar. For this reason, you should always leave space for the addition of one menu to your application's menu bar.

An Application Program Supporting Desk Accessories

The program in Listing 9-1 illustrates how to write an application that supports desk accessories. It illustrates how to write a program that must work even when there is more than one window on the screen. The program creates a simple Apple-File-Edit menu bar, adds all available desk accessory items to the Apple menu, and then lets you switch between an application window and any open desk accessories in the usual way. To keep you posted on what's going on, it also displays an appropriate message in the application window: "I'm not active" (when the window is deactivated), "I'm active" (when it's activated), "Window needs updating" (when an update event occurs), or "Keyboard not supported" (when a key-down event occurs).

Listing 9-1. The Source File, Linker Control File, and RMaker File for the Accessory Program.

```
* Asm Source File
* Accessory.Asm
*
* This program shows how to develop an application
* that works with desk accessories.

MenuBarID       EQU     128     ;Menu Bar resource ID
  AppleID       EQU     1       ;Menu ID for Apple menu
  FileID        EQU     2       ;Menu ID for File menu
  EditID        EQU     3       ;Menu ID for Edit menu

WindID          EQU     128     ;Window resource ID

        INCLUDE ToolEqu.D       ;Toolbox equates
```

Listing 9-1. *continued*

```
        INCLUDE QuickEqu.D      ;QuickDraw equates
        INCLUDE SysEqu.D        ;Operating system equates
        INCLUDE Traps.D         ;Trap instructions

; Initialize the various Managers:

        PEA     -4(A5)          ;Start of QD globals area
        _InitGraf               ;Initialize QuickDraw
        _InitFonts              ;Font Manager
        _InitWindows            ;Window Manager
        _InitMenus              ;Menu Manager
        _TEInit                 ;TextEdit
        MOVE.L  #0,-(SP)        ;(no restart procedure)
        _InitDialogs            ;Dialog Manager
        _InitCursor             ;We want arrow cursor

        MOVE.L  #$0000FFFF,D0
        _FlushEvents            ;Get rid of every event

; Create and draw a window on the screen:

        CLR.L   -(SP)           ;Space for returned pointer
        MOVE    #WindID,-(SP)   ;Resource ID
        MOVE.L  #0,-(SP)        ;Store on heap
        MOVE.L  #-1,-(SP)       ;-1 = front window
        _GetNewWindow           ;Get window from resource file
        MOVE.L  (SP),OurWindow(A5)  ;Save window pointer
        _SetPort                ;Window ptr already on stack

; Read Apple, File, Edit menu bar from MBAR resource, then
; make it current using _SetMenuBar:

        CLR.L   -(SP)           ;Space for result
        MOVE    #MenuBarID,-(SP);Push resource ID
        _GetNewMBar
        _SetMenuBar             ;Handle already on stack


* Add desk accessory names to Apple menu:

        CLR.L   -(SP)           ;Space for result
        MOVE    #AppleID,-(SP)  ;Menu ID for Apple menu
```

Listing 9-1. *continued*

```
        _GetMHandle             ;Return menu handle on stack
        MOVE.L  (SP),AppleH(A5) ;Save it for later
        MOVE.L  #'DRVR',-(SP)   ;DAs are DRVR resources
        _AddResMenu             ;Put them in Apple menu

        _DrawMenuBar            ;Display menu bar

MainLoop

        BSR     GetEvent
        BSR     HandleEvent
        BRA     MainLoop
GetEvent
        _SystemTask             ;Let DAs do periodic functions
        CLR.B   -(SP)           ;Leave space for Boolean result
        MOVE    #-1,-(SP)       ;Allow all events (-1 = $FFFF)
        PEA     EventRecord     ;Results are returned here
        _GetNextEvent           ;Check for an event
        TST.B   (SP)+           ;Pop and test the result code
        BEQ     GetEvent        ;Branch if no pending event
        RTS


* HandleEvent is the event dispatcher. It takes the event type
* code returned by _GetNextEvent and calls the subroutine
* that handles it.  Access to the event handling subroutines is
* through a jump table arranged in event type code order.

HandleEvent

        MOVE    EventRecord+evtNum,D0   ;Get event type code
        CMP     #8,D0           ;Events 9_15?
        BHI     Ignore          ;Yes, so branch and ignore
        ASL     #2,D0           ;Times 4 to index into table
        JMP     JumpTable(PC,D0)  ;Jump to handler

Ignore  RTS

JumpTable

        JMP     Ignore          ;Null event (never used)
        JMP     DoMouseDown     ;Button-down
```

Listing 9-1. *continued*

```
        JMP     Ignore          ;Button-up
        JMP     DoKeyDown       ;Key-down
        JMP     Ignore          ;Key-up
        JMP     DoKeyDown       ;Auto-key
        JMP     DoUpdate        ;Update
        JMP     Ignore          ;Disk-inserted
        JMP     DoActivate      ;Activate

DoKeyDown

        MOVE    EventRecord+evtMeta,D0  ;Get modifiers word
        BTST    #CmdKey,D0      ;Is command key bit on?
        BNE     CommandTest     ;Yes, so branch

        BSR     ClearWindow

        MOVE    #30,-(SP)
        MOVE    #30,-(SP)
        _MoveTo
        PEA     'Keyboard not supported'
        _DrawString

        MOVE.L  oldPort,-(SP)
        _SetPort

        RTS                     ;Ignore other keystrokes

; Check for COMMAND key (might be a key equivalent for menu)

CommandTest
        CLR.L   -(SP)           ;Space for result
        MOVE    EventRecord+evtMessage+2,-(SP)  ;Push character
        _MenuKey                ;Get menu information
        JMP     Menu1


; In a typical program, you would handle update events by
; redrawing what was previously erased by calling _BeginUpdate,
; redrawing, then calling _EndUpdate. Here, I don't keep track
; of what's on the screen so I just clear the screen and display
; a message. _BeginUpdate and _EndUpdate are first called
; back-to-back to prevent the same update from begin reported
; again.
```

Listing 9-1. *continued*

```
DoUpdate
        MOVE.L  OurWindow(A5),D0   ;Move into D0 for CMP
        CMP.L   EventRecord+evtMessage,D0   ;Our window?
        BNE     @1              ;No, so branch

; These two instructions empty the update region:

        MOVE.L  OurWindow(A5),-(SP)
        _BeginUpdate
        MOVE.L  OurWindow(A5),-(SP)
        _EndUpdate

        BSR     ClearWindow

        MOVE    #30,-(SP)
        MOVE    #30,-(SP)
        _MoveTo
        PEA     'Window needs updating'
        _DrawString

        MOVE.L  oldPort,-(SP)
        _SetPort

@1      RTS

DoActivate
        MOVE.L  OurWindow(A5),D0   ;Move into D0 for CMP
        CMP.L   EventRecord+evtMessage,D0   ;Our window?
        BNE     @1              ;No, so branch

        MOVE    EventRecord+evtMeta,D0  ;Get modifiers word
        BTST    #ActiveFlag,D0   ;Is activate bit set?
        BEQ     DeActivate      ;No, so branch

        BSR     ClearWindow

        MOVE    #30,-(SP)
        MOVE    #30,-(SP)
        _MoveTo
        PEA     'I'm active!'
        _DrawString
```

Listing 9-1. *continued*

```
        MOVE.L  oldPort,-(SP)
        _SetPort

@1      RTS

Deactivate

        MOVE.L  OurWindow(A5),-(SP)
        _SetPort                ;Select our port for drawing

        BSR     ClearWindow

        MOVE    #30,-(SP)
        MOVE    #30,-(SP)
        _MoveTo
        PEA     'I'm not active!'
        _DrawString

        MOVE.L  oldPort,-(SP)
        _SetPort

        RTS

DoMouseDown

        CLR     -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)      ;Where
        PEA     WindowPtr       ;VAR window selected
        _FindWindow             ;Where was button pressed?
        MOVE    (SP)+,D0        ;Get result
        CMP     #6,D0           ;Result above 6?
        BHI     @1              ;Yes, so branch

        ASL     #2,D0           ;Times 4 to step into table
        JMP     ClickTable(PC,D0)

@1      RTS                     ;Ignore everything else

; Jump table to the seven click-handling subroutines:

ClickTable
        JMP     DeskTop         ;In the desktop
```

Listing 9-1. *continued*

```
        JMP     Menu            ;In the menu bar
        JMP     System          ;In DA window
        JMP     Content         ;In Content region
        JMP     Drag            ;In Drag region
        JMP     Grow            ;In Grow box
        JMP     GoAway          ;In Close box

GoAway
        RTS

Grow
        RTS

Drag
        RTS

; The click was in the content region of a window (and must be
; ours). Make sure it's selected using _SelectWindow to generate
; an update event. (Normally you would only do this if the
; window was not already selected.)

Content
        MOVE.L  WindowPtr,-(SP)
        _SelectWindow           ;Select the window
        RTS

System
        PEA     EventRecord
        MOVE.L  WindowPtr,-(SP)
        _SystemClick            ;Handle click in DA window
        RTS

DeskTop
        RTS                     ;Ignore clicks in desktop


* Handle clicks in the menu bar:

Menu
        CLR.L   -(SP)           ;Space for result
        MOVE.L  EventRecord+evtMouse,-(SP)   ;where?
        _MenuSelect
```

Listing 9-1. *continued*

```
Menu1   MOVE.L  (SP)+,D0        ;Pop the result
        TST.L   D0              ;Is D0=0 (no selection)?
        BNE     @1              ;No, so branch
        RTS

; SWAP D0 swaps the high word of D0 with the low word. This
; means the low word contains the menu number and the high
; word contains the item number.

@1      SWAP    D0
        CMP.W   #FileID,D0      ;Is it the File menu?
        BEQ     DoFileMenu      ;Yes, so branch

        CMP.W   #AppleID,D0     ;Is it the Apple menu?
        BEQ     DoAppleMenu     ;Yes, so branch

; You must be in the Edit menu. Pass the standard editing
; commands to the desk accessory.

        CLR.B   -(SP)           ;Space for result
        SWAP    D0              ;Item # in low word
        SUBQ    #1,D0           ;Reduce by 1 for _SysEdit
        MOVE    D0,-(SP)        ;Push item #
        _SysEdit
        MOVE.B  (SP)+,D0        ;Pop Boolean result
        BNE     @2              ;Branch if accessory handled it
        BSR     FixTitle
        RTS                     ;We don't support editing!

@2      BSR     FixTitle

        RTS

; Handle the Apple menu by passing control to a DA if the item
; number is greater than 2.

DoAppleMenu
        SWAP    D0              ;Get item number in D0.W
        CMP.W   #2,D0           ;Is it a DA?
        BHI     @1              ;Yes, so branch
        BRA     FixTitle        ;No, so branch
```

Listing 9-1. *continued*

```
@1      MOVE.L  AppleH(A5),-(SP)
        MOVE    D0,-(SP)        ;Push menu item number
        PEA     DAName          ;VAR name of DA
        _GetItem                ;Get name of accessory

        CLR     -(SP)           ;Space for result
        PEA     DAName          ;Name of accessory
        _OpenDeskAcc            ;Pass control to DA
        MOVE    (SP)+,D0        ;Pop the result

FixTitle
        MOVE    #0,-(SP)
        _HiLiteMenu             ;Return menu title to normal
        RTS

; Handle the File menu (only a Quit item):

DoFileMenu

        SWAP    D0              ;Get item # in low word
        CMP.W   #1,D0           ;Is it 1st item (Quit)?
        BEQ     @1              ;Yes, so branch
        RTS                     ;(should never get here)

@1      BSR     FixTitle        ;Remove highlight from title


; Return to Finder the easy way using _ExitToShell. You don't
; have to pop any pending subroutine return addresses if you do
; this.

        _ExitToShell


; ClearWindow erases our window. The dimensions of the window
; are located at position 16 from the start of the window record
; (a variable called portRect).

ClearWindow
        PEA     oldPort
        _GetPort                ;Get current drawing port
        MOVE.L  OurWindow(A5),-(SP)
```

Listing 9-1. *continued*

```
        _SetPort                ;Make our window current
        MOVE.L  OurWindow(A5),A0    ;Ready for indirect access
        PEA     PortRect(A0)    ;Address of port rectangle
        _EraseRect
        RTS

; Application constants:

EventRecord     DCB.B   EvtBlkSize,0   ;Space for event record

WindowPtr       DC.L    0       ;Pointer to window

DAName          DCB.B   16,0    ;Space for DA name string

oldPort         DC.L    0       ;Currently active drawing port

; Here are the program globals. Use (A5) addressing.

OurWindow       DS.L    1       ;Pointer to our window

AppleH          DS.L    1       ;Handle to Apple menu
```

```
; Linker Control File
; Accessory.Link
;
; Link this file to create application
;  (without resources).
Accessory
$
```

```
* RMaker Source File
* Accessory.R
*
* Compile this after assembling and linking Accessory.Asm
*
* The next command appends the resources to the application:
!Book:Accessory


Type MBAR = GNRL        ;;Menu bar resource
,128                    ;;Resource ID
.I                      ;;Decimal integers follow
```

Listing 9-1. *continued*

```
3                       ;;Number of menus
1                       ;;ID of 1st menu
2                       ;;ID of 2nd menu
3                       ;;ID of 3rd menu

Type MENU
,1                      ;;Resource ID
\14                     ;;Title is the Apple symbol (ASCII $14)
  About this demo...    ;;About box
  (-

,2                      ;;Resource ID
File                    ;;Menu Title
  Quit                  ;;Only item is Quit

,3                      ;;Resource ID
Edit                    ;;Menu Title
  Undo                  ;;Standard Edit menu
  (-
  Cut/X
  Copy/C
  Paste/V
  Clear
  (-

Type WIND
,128                    ;;Resource ID
DA Demo                 ;;Title for Window
100 5 332 502           ;;Window coordinates (TLBR)
Visible NoGoAway        ;;Visible window/ no goaway box
4                       ;;Window ID. 4 = title, no grow box
0                       ;;User-definable item (not used)
```

Let's take a closer look at the program to see how it supports desk accessories. First of all, a menu bar with two menus is loaded from the application file's resource fork using _GetNewMBar. To add the names of the desk accessories (DRVR resources) to the Apple menu using _AddResMenu, we first need to know the handle to the Apple menu—this is obtained using the _GetMHandle instruction.

To allow any DA to perform a periodic function associated with it, the program includes the _SystemTask instruction in the GetEvent subroutine that forms part of the main event loop. If you remove this instruction, the Alarm Clock won't keep ticking, the cursor in the Note Pad won't blink, and so on.

When a key-down event occurs, control passes to DoKeyDown, which clears the application window and uses _DrawString to display the "Keyboard not supported" message. Notice that the window clearing subroutine, ClearWindow, first uses _GetPort to save the pointer to the active drawing window (it may be a desk accessory window), then uses _SetPort to select the application window. When _DrawString finishes, the application calls _SetPort again to restore the original drawing window. It is important not to permanently switch to the application window because such an action may take the DA by surprise.

Update events occur when a desk accessory is moved aside to expose a new portion of the application window; they are handled by the code beginning at DoUpdate. In a complete application, you would redraw the portion of the window previously overlaid by the DA window. This means you have to keep track of what is in the window at all times. This application simply erases the window and displays a "Window needs updating" message. Notice that before it does this it calls _BeginUpdate and _EndUpdate to empty the window's update region. If you don't do this, the operating system will post the same update event again and again.

Activate and deactivate events are handled in the usual way, beginning at DoActivate. The only concern is remembering to use _SetPort to select the application window for drawing.

When handling button-down events you must be more concerned with the possibility that desk accessories are present. First of all, a call to _FindWindow may indicate the mouse was clicked in a system window: If it was, control passes to the System subroutine that calls _SystemClick to let the DA handle the click as it sees fit.

Clicks in the content region of the application window are handled by the Content subroutine. It selects the application window using _SelectWindow to bring it to the front of the screen, causing an update event to be posted in the event queue. It also removes the highlighting from the active DA window, if necessary. In a complete application you would probably call _SelectWindow only if the window is not already active. You can see if it's active by comparing the pointer returned by _GetPort with the OurWindow variable.

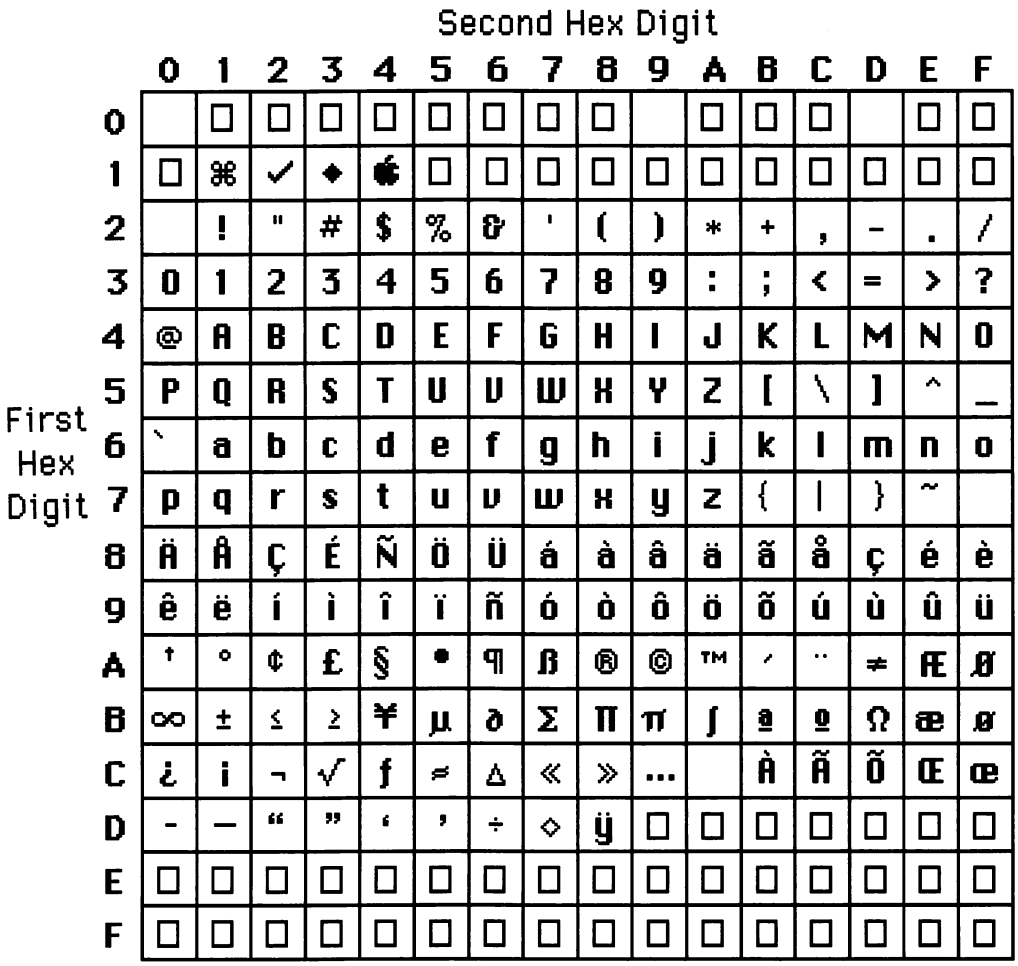
If there is a button-down event in the menu bar and an item is selected from the Apple menu, control passes to DoAppleMenu. This subroutine checks to see if a DA was selected by comparing the item number returned by _MenuSelect with two (the first two items are an About... item and a dimmed line). If the item number is greater than two, it's time to open the desk accessory with _OpenDeskAcc.

Recall, however, that _OpenDeskAcc requires the name of the DA as a parameter, not a menu item number. To get the name, the application uses _GetItem to read the item name into DAName. DAName is 16 bytes long to accommodate the maximum name size of 15 bytes and a leading length byte.

If an item in the Edit menu is selected, its item number minus one is passed to _SysEdit, giving a DA a chance to perform standard editing operations. If _SysEdit returns a Boolean false result, the application should deal with the edit command itself; here, it just ignores it.

Appendix A

The ASCII Character Set



Note: The characters shown are those defined in the system font resource. The characters used in other fonts may be different.

Appendix B

Finding, Fixing, and Avoiding Programming Errors

The errors listed below are the ones you're most likely to commit while developing 68000 assembly language programs on the Macintosh. To avoid them, follow the suggestions given.

Improperly managing the stack. This is probably the most common cause of programming errors on the Macintosh and invariably leads to the unwelcome appearance of the fatal bomb alert. Before calling a Macintosh trap instruction, make absolutely certain its parameters are pushed on the stack in the proper order, and that they are the proper size. For functions, also remember to push space for the result and to pop the result from the stack on return from the trap instruction.

Referring to unavailable resources. Make sure any resource you use in an application is either in the System file, has been appended to the application using the MDS !filename RMaker command or the MDS /RESOURCES Link command, or is in a resource file that has been specifically opened by the application with _OpenResFile. If you try to use a resource that is not active, your program will crash unless it does proper error checking.

Redefining symbols and labels. Once a symbol has been defined with the EQU directive, you cannot redefine it without causing a "Multiply defined symbol" or "Illegal line" MDS error. You will most often cause these errors by inadvertently attempting to define a symbol in your main source code file that is already defined in an included MDS equate text (.txt) file or a packed symbol (.D) file, respectively. If a subse-

quently included .D file attempts to redefine an existing symbol, no error occurs, but the new definition is ignored. You can't redefine instruction labels, either. If you do, you will cause a "Multiply defined label" MDS error.

Using the d16(PC) addressing mode when trying to store a value in the constant allocated with the DC directive. The d16(PC) addressing mode cannot be used as a destination operand, so use the following general instruction sequence to change the value of a constant.

```
LEA    MyConstant,A0  ;MyConstant is same as MyConstant(PC)
MOVE   #number,(A0)   ;Store "number" in MyConstant
```

If a program must often write to constants, it's best to create variables instead (using the DS directive) so that you can write to them directly with a MOVE #number,label(A5) instruction.

Forgetting to append the (A5) mode designator to the name of a variable allocated with the DS directive. If you define a variable, say MyVariable, access it with an operand of the form "MyVariable(A5)", not "MyVariable" by itself. If you don't, MDS uses the MyVariable(PC) addressing mode, which generates a completely different effective address, causing your program to behave unpredictably.

Improperly using a Bcc (branch conditionally) instruction after a CMP instruction. Remember that a CMP instruction compares the destination operand with the source operand to determine how to set the condition code flags, not the other way around. For example, a BLS instruction that follows it will cause a branch to the target address if the destination is lower or the same (LS) as the source, not if the source is lower or the same as the destination.

Improperly using the CMP instruction when the destination operand does not use a register direct addressing mode. If the destination operand for a CMP instruction is an operand other than Dn or An, the source operand must use the immediate addressing mode. CMP offset(A3),D3 is valid, but CMP offset(A3),myVariable(A5) is not. The only exception is the CMPM (Ay)+,(Ax)+ instruction.

Not preceding a string with a length byte. Most of the string-handling toolbox subroutines insist the strings you pass to them by address be preceded by a length byte. If you define a string with:

```
DC.B    'This is a string'
```

the string is not preceded by a length byte unless the STRING_FORMAT directive is set to 3 first. If you explicitly include the length byte with:

```
DC.B    16,'This is a string'
```

make sure that STRING_FORMAT is set to 1 to prevent duplication of the length byte.

Specifying an incorrect loop count when using the DBcc instruction. The data register used as the counter in a DBcc loop must hold the loop count *minus one* because looping ends when the counter becomes minus one, not zero.

Specifying an explicit destination operand for the PEA instruction. The "push" in a PEA instruction is implicit, so you must not use an instruction of the form "PEA theString,-(SP)". The correct syntax is "PEA theString".

Not specifying a disk prefix for files used by RMaker when using MDS 1.0. Remember that the original version of RMaker expects to find the file it is appending to (!filename), or a file it is including (INCLUDE filename), on the same disk as RMaker itself, even if the RMaker source file is on another disk. In addition, if it creates a new file, it saves it on the RMaker disk. To override this default behavior, use a file name that includes a specific disk prefix.

Using the wrong operand size to access a field in a data record. The Macintosh operating system uses a variety of data structures, or records, made up of fields of byte, word, or long word size. If you access a field, check that your operand size is the same size as the field; if it's not, you won't access the field properly. For example, the evtMessage field of an event record is a long word; for a key-down event

the last byte in the field is the character code. To read its value, use an instruction like MOVE.L EventRecord+evtMessage,D0 and then isolate the low-order byte with an AND.L #$FF,D0 instruction. Don't use a MOVE.B EventRecord+evtMessage,D0 instruction because this loads only the first byte of the evtMessage field into the last eight bits of the D0 register. For a key-down event, this number is meaningless.

Forgetting the # when specifying an immediate operand. If you forget to place a # in front of an immediate operand, the assembler thinks you're referring to an address and assembles the instruction using the absolute addressing mode. This means the number acted on is the one stored at the address given by the operand, not an immediate quantity. This error most often occurs when you use symbolic names to refer to immediate numbers. For example, the symbol IBeamCursor refers to a constant, not an address, so use an operand of the form #IBeamCursor in an instruction.

Reversing the order of instruction operands. Remember that 68000 assemblers insist that the source operand be specified before the destination operand. This error is common for those who also program in 8086 assembly language on the IBM PC, where the opposite operand order is required.

Confusing the use of BHS/BGE, BHI/BGT, BLS/BLE, and BLO/BLT. These Bcc instructions are most often used after comparing the relative sizes of two numbers with a CMP instruction. If the numbers being compared are unsigned numbers, use the BHS (BCC), BHI, BLS, and BLO (BCS) instructions only. If they are signed numbers, use their counterparts BGE, BGT, BLE, and BLT instead.

Appendix C
The MacsBug Debugger

The MDS master disk contains two **debuggers** you can use to help track down the source of programming errors that are not flagged during the assemble/link process. These are errors caused by faulty program logic or improper trap instruction calls rather than obvious syntax errors.

The particular debugger described here is a single-Macintosh debugger called xMacsBug. (The other, MacDB, requires two Macintoshes linked together by a cable.) To install it, change its name to MacsBug, put it in the System Folder of the disk (the one containing System and Finder), then boot from the disk. When the system starts up, the message "Macsbug installed" appears, verifying that the debugger is available for use.

Macsbug is most useful for executing a program one instruction at a time (a procedure called stepping) and for displaying the contents of registers after each step. These features make it fairly easy to determine whether your program is operating as expected or, if it's not, where it begins to lose control, and why.

Invoking MacsBug

You can invoke MacsBug in two ways. The most common way is to press the rear programmer's switch on the left side of the Macintosh (it's marked INTERRUPT) while the program you want to inspect is running. The program itself can exit to MacsBug using the _Debugger trap instruction.

When MacsBug takes over, it displays the contents of all the registers followed by a › prompt symbol. From here you can enter any of a number of commands that MacsBug supports. We're only going to look at a few of the commands here, but they are the most important ones.

> You may find that the disk drive may not stop whirring after MacsBug takes over. To stop the internal drive, enter the command DM DFF1FF right away; use DM DFF3FF for the external drive.

Locating the Program

The first thing you will want to do is locate the interrupted program in the application heap. Use the HD (Heap Dump) command for this. HD analyzes the heap and displays the position and size of each **free** (unused) block, pointer block, and handle block it contains. Here is what the first part of a typical heap dump looks like:

```
00000B00

00000B34 P 00000108           *
00000C3C P 00000074           *
00000CB0 H 0000001A 0 00000C18
00000CCA H 000001E2 E 00000C1C * 20 0001 CODE
```

The first line contains the address of the base of the application heap. The next four lines identify various blocks in the heap. The first three items in each of these lines hold the following information in the following order:

- the starting position of the block
- the block type code (P for pointer, H for handle, F for free)
- the size of the block (including Memory Manager overhead bytes)

For handle blocks only, these three items are followed by a four-bit hexadecimal digit describing the properties of the block. The attribute bits are locked (bit 3), purgeable (bit 2), and resource (bit 1). Bit 0 is not used and is always zero. The long word that follows the digit is the address of the master pointer to the block. (See Chapter 4.)

Notice that an asterisk is used to identify any blocks that are immovable, either because they are nonrelocatable or they are relocatable but locked. Recall from Chapter 4 that pointer blocks are always nonrelocatable.

For blocks corresponding to resource files you will also see the resource's reference number, ID number, and type code on the right side of the line. A program is actually a CODE resource with a resource ID of 1, so in the example the block containing the program begins at $0000CCDA.

The program actually begins $0C bytes from the start of the CODE block. The first $0C bytes are used by the Memory Manager to keep track of the block and its properties.

Disassembling the Program

To verify that the CODE block does, indeed, hold the program, disassemble the program using the IL (Instruction List) command. To do this, enter a command of the form "IL addr" where addr represents the address at which the disassembly is to begin. IL displays 16 program lines in assembly language form, but without symbolic labels. In our example, enter the command "IL CCD6" to begin disassembling the program. ($CCD6 is the address of the CODE block plus $0C. The preceding $ sign is optional.)

Standard names for 68000 instructions are shown in a disassembled listing. ROM trap instructions are identified by the phrases TOOLBOX $Axxx (user-interface toolbox instructions) or OSTRAP $Axxx (operating system instructions); their symbolic names are displayed in the comment field of a line.

Subsequent presses of the RETURN key will disassemble the next 16 lines in the program. You can also use IL to disassemble any portion of the Macintosh ROM. You may want to do this to discover how the toolbox and operating system subroutines really work.

Displaying and Setting Memory Locations

To display the contents of memory locations, use the DM (Display Memory) command. The form of this command is:

```
DM [ address [ number ] ]
```

where the brackets are used to indicate that the parameters they enclose are optional. Don't type the brackets when you enter the command. DM displays the contents of **number** bytes of memory beginning at **address**, in hexadecimal and ASCII form. If you don't

specify a number, 16 bytes are displayed. For example, if you enter the command DM 2E0, you will see the following line:

```
000002E0    0646 696E 6465 7220  2020 2020 2020 2020    .FINDER
```

The starting address is on the far left and is followed by the 16 bytes (arranged as eight words) stored in memory beginning at this location. On the far right is the ASCII character for each byte in the line. If the byte is less than $20, a period is displayed instead. The data stored at $2E0 is a string (preceded by a length byte) containing the name of the application launched when you execute an _ExitToShell instruction.

Note that if you now press RETURN by itself, the next 16 bytes in memory are displayed. This makes it easy to examine a range of memory with a minimum of typing.

To store data in consecutive memory locations, use the SM (Store Memory) command:

```
SM addr value1 [ value2 ... valueN ]
```

This command stores the specified values into memory beginning at the location given by addr. It is useful for changing program constants before running the program again.

Displaying and Setting Registers

To see what's in any of the 68000 registers after a program has been interrupted, simply type the name of the register (D1 to D7, A1 to A7, PC, SR) and press RETURN. To display the contents of all the registers at once, enter TD (Total Display).

To change the contents of any register, follow the name of the register with the value to be placed in the register. Do this to initialize registers before calling a subroutine (with the G, GT, T, or S commands) that expects parameters to be passed in registers.

Stepping and Tracing

To verify that a program is functioning properly, it is often convenient to execute it one step at a time. After each step you can check the values in the 68000 registers to see if everything is proceeding as expected.

Use the T (Trace) command to execute the instruction pointed to by the program counter. (Use the "PC value" command to change the value in the program counter.) After execution, the next instruction and the contents of all the registers are displayed. For the purposes of the T command, a trap call is considered to be a single instruction.

A similar command is S (Step), but it also steps through the ROM subroutine called by a trap instruction. If you wish, you can follow the S command with a number indicating the number of instructions to step through before returning to the MacsBug command line.

Sometimes you don't want to step through a particular subroutine or code segment line by line; you just want to execute it quickly and examine the final results. To do this, use the GT (Go Till) command. The form of this instruction is:

```
GT addr
```

where `addr` is an address in the program. When the program counter reaches the specified address, execution halts and control returns to MacsBug.

Use the G (Go) command to begin executing a program at a certain address. The starting address is either the value in the program counter or, if you specify an address after the G command, at that address. Control returns to MacsBug if the program (or subroutine) at the starting address ends with an RTS instruction. If it ends with an _ExitToShell instruction, you will return to the Finder instead; press the interrupt switch to return to MacsBug if this happens.

Leaving MacsBug

You can leave MacsBug in one of three ways:

RB—Reboot the system
ES—Exit to the shell program (usually the Finder)
EA—Exit and relaunch the interrupted application

The last command, EA, is not available on versions of MacsBug prior to version 5.0.

Appendix D
Utility Programs

There are many utility programs available to assist 68000 assembly language programmers in program development. Some of the most valuable are listed here.

ResEdit. Apple Computer, Inc., 20525 Mariani Ave., Cupertino, CA 95014, Tel: (408) 996-1010. This is Apple's official resource editor. With it you can view and change the definition of any resource file on a disk. This program comes with MDS (version 2.0 only), Apple's periodic Software Supplements, and is available from many commercial information utilities for no charge.

REdit. Apple Computer, Inc. Another of Apple's resource editors, developed by programmers at Apple France. It is included with Apple's periodic Software Supplements and is available from many commercial information utilities as well.

Fedit. Mac Master Software, #122 - 939 E. El Camino Real, Sunnyvale, CA 94087. A popular utility for snooping through files at the disk block level. It is also useful for changing file attributes in a disk directory, such as the bundle bit, and for recovering from disk crashes.

Purge Icons. Author unknown. This program is available from many bulletin boards and commercial information utilities. It removes any unused ICON, BNDL, and FREF resources from a disk's DeskTop file—these are associated with applications that have been deleted from the disk. You will want to use it when you're designing a custom icon for an application so that the Finder will not keep using the original icon. (See Chapter 2.) It's also handy for reducing the size of the DeskTop file.

Dialog Creator. Apple Computer, Inc. Another "visual resource editor," this one for defining dialog and alert boxes. It is included with Apple's periodic Software Supplement.

TMON. Icom Simulations, Inc., 626 Wheeling Rd., Wheeling, IL 60090, Tel: (312) 520-4440. This advanced, single-Macintosh debugger is more powerful and more useful than the MacsBug debugger that comes with MDS.

MacNosy. Jasik Designs, 343 Trenton Way, Menlo Park, CA 94025, (415) 322-1386. The self-proclaimed "disassembler for the rest of us." With it you can easily inspect the subroutines in the Macintosh ROM or in any program on disk in assembly language format. This is a great way to learn about professional programming techniques.

ConCode. Pixel Pathways, P.O. Box 4065, Mt. Penn, PA 19606. An invaluable desk accessory for assembly language programmers. When you select a 68000 instruction to work with, all the valid addressing modes for the instruction are shown. You can also enter numeric values for the instruction's two operands, then execute the instruction to determine the result. The settings of the condition code flags after execution are also shown.

The Macintosh Reference System. TOM Programs, Suite 34-T, 1500 Massachusetts Ave. NW, Washington, DC 20005, (202) 223-6813. A Microsoft File database containing summaries of all the trap instructions documented in *Inside Macintosh*. The same database is also available in a deck of 750 color-coded 3-by-5 cards.

MacExpress. ALsoft, Inc., P.O. Box 927, Spring, TX 77383-0927, (713) 353-4090. An Application Manager that directs and controls an application's user interface. By using it, you can concentrate on writing the guts of your application and leave the implementation of the Macintosh user interface to MacExpress.

Consulair Professional Development Tools: Utilities. Consulair Corp., 140 Campo Drive, Portola Valley, CA 94025, (415) 851-3272. A package made up of four useful utilities: SuperMake, for automatically rebuilding all changed parts of an application; Grep, for searching multiple files for a given string pattern; Diff, for displaying the differences between two text files; and Maximum Performance Analyzer, a desk accessory that monitors a program to determine the time it takes to execute its routines.

Bibliography

The following materials will either assist in understanding the Macintosh programming environment or in developing applications in 68000 assembly language, or both.

Books

Apple Computer, Inc., *Inside Macintosh, Volumes I, II, III, and IV*, Reading, MA: Addison-Wesley Publishing Company, 1985 (volumes I to III) and 1986 (volume IV). The definitive work on the Macintosh programming environment, written by Apple's developer support team. Every serious programmer must have a copy of all four volumes.

Apple Computer, Inc., *Macintosh 68000 Development System User's Manual*, Cupertino, CA: Apple Computer, Inc., 1984. This is the manual that comes with MDS.

Chernicoff, Stephen, *Macintosh Revealed, Volume 1: Unlocking the Toolbox*, Hasbrouck Heights, NJ: Hayden Book Company, 1985. An excellent introduction to the Macintosh programming environment.

Chernicoff, Stephen, *Macintosh Revealed, Volume 2: Programming with the Toolbox*, Hasbrouck Heights, NJ: Hayden Book Company, 1985. This book contains plenty of examples on how to program with the ROM toolbox. Although the examples are in Pascal, tables at the end of each chapter describe the assembly language equivalents of the Pascal functions and procedures discussed.

Mathews, Keith, *Assembly Language Primer for the Macintosh*, New York, NY: New American Library, 1985. This is an introduction to 68000 assembly language for novice programmers. Unfortunately, its lack of depth and numerous typographical errors make it difficult to follow.

Motorola Inc., *M68000 8/16/32-Bit Microprocessor Programmer's Reference Manual*, 5th ed., Prentice-Hall, Inc., 1986. A description of the 68000 written by the chip's designer.

Rosenzweig, Edwin and Harrison, Harland, *Programming the 68000: Macintosh Assembly Language*, Hasbrouck Heights, NJ: Hayden

Book Company, 1986. A good introduction to 68000 assembly language programming.

Williams, Steve, *Programming the Macintosh in Assembly Language*, Berkeley, CA: Sybex Inc., 1986. This book contains detailed descriptions of a series of general-purpose macros you can use to simplify the development of assembly language programs. Its repeated references to the CP/M environment are quite perplexing, however.

Periodicals

MacDeveloper: The Electronic Magazine for Macintosh Developers, Harry Chesley, 1850 Union St. #360, San Francisco, CA 94123. This newsletter is published about six times per year and is distributed via electronic bulletin board systems and national information services like Delphi and CompuServe.

Macintosh Technical Notes, Technical Notes, Apple Computer Mailing Facility, 467 Saratoga Avenue, Suite 621, San Jose, CA 95129. These are the "official" technical notes from Apple Computer, Inc., which are released at infrequent intervals during the year.

MacInTouch, Ford-LePage, Inc., P.O. Box 786, Framingham, MA 01701, (617) 527-5808. A magazine valuable for its useful reviews of many products, including those of interest to developers.

MacMag, 3743 Notre-Dame W., Montreal, Quebec, Canada H4C 1P8. This magazine often contains introductory articles about programming in assembly language.

MacTutor: The Macintosh Programming Journal, MacTutor, P.O. Box 400, Placentia, CA 92670, (714) 630-3730. A monthly magazine containing much useful material for the 68000 programmer. No other Macintosh magazine can match it for technical content.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

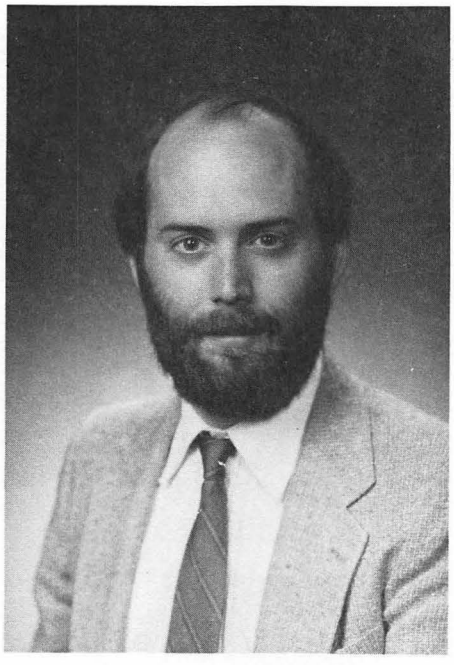
V

W

X, Y, Z

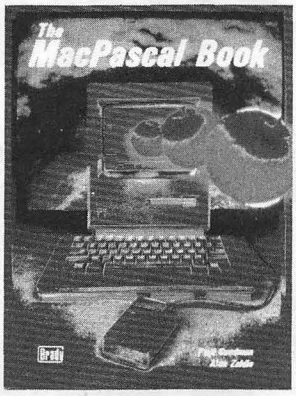
About the Author

Gary B. Little is a Macintosh and Apple II programmer who resides in Vancouver, British Columbia. He is also a practicing lawyer. Gary is a founding member of the Apples British Columbia Computer Society and of the prestigious, but not-too-serious SAGE (Serious Apple Group, Eh!). He is also an active member of several business organizations that promote and assist software developers. He has written numerous articles for several microcomputer periodicals and is the author of four recent microcomputer books published by Brady. He is also the author of the Apple II communications program, "Point-to-Point," published by Pinpoint Publishing.



BRADY'S got the Mac knack

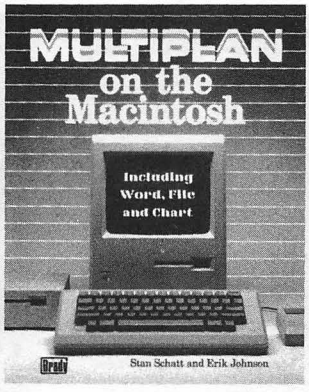
1. Whether you own a Macintosh or want to buy one, this book tells you everything there is to know about it as well as what it can do. You'll discover the Macintosh's best kept secrets by tapping all of its capabilities. $15.95

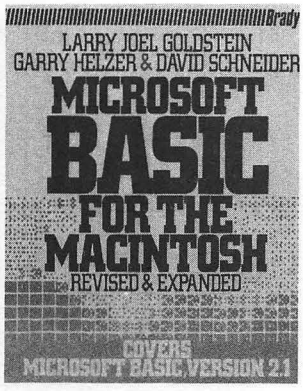


2. This book is for the novice who wants to learn how to program and the experienced Pascal programmer who wants to use Macintosh Pascal. Contains detailed descriptions of file usage on I/O. $19.95

3. This book gives artists and designers a basic understanding of the tools used in MacPaint to create drawings and generate ideas for design projects. Teaches one step-at-a-time in logical understandable increments. $15.95

4. Your step-by-step guide for using the Macintosh to run your office and handle all spreadsheet, graphing, file management and word processing needs. Includes introduction to Macintosh terminology, techniques and programs. $18.95





5. The latest on Microsoft BASIC, Version 2.1. Includes a tutorial with start-to-finish instructions for using and programming with commands statements and functions explained in narrative form; many example programs; BASIC commands listed alphabetically, and more. $23.95

Now at your book or computer store.
Or order toll-free today: **800-624-0023** In New Jersey: 800-624-0024

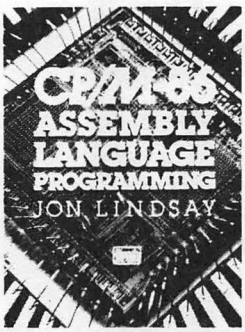
BRADY COMMUNICATIONS COMPANY, INC.
c/o Prentice Hall
P.O. Box 512, W. Nyack, NY 10994

Circle the numbers of the titles you want below.
(Payment must be enclosed; or, use your charge card.) Add $1.50 for postage and handling.
Enclosed is check for $_____ or charge to
☐ MasterCard ☐ VISA.

Acc't # _____ Exp. date _____
Signature _____
Name _____
Address _____
City _____ State _____ Zip _____
(New Jersey residents, please add applicable sales tax.)
Dept. 3

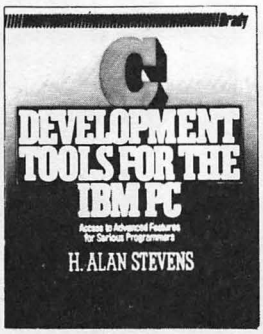
1 (0-89303-649-8) 2 (0-89303-644-7) 3 (0-89303-648-X)
4 (0-89303-678-1) 5 (0-89303-662-5)

BRADY Speaks Your Language

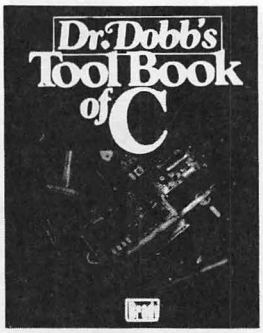


1. The perfect intro to CP/M-86 even if you've never programmed in assembly language before. Includes sections on sequential file handling and memory manipulation, and more. $21.95

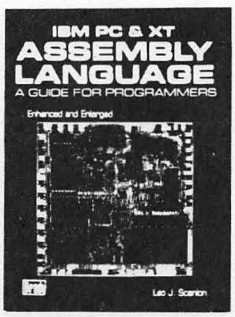
2. Beyond the basics, this guide explores new structured concepts to analyze and solve problems in C with tools of modularity, input and output functions, arrays and structures, and bit manipulation. $21.95



3. The special focus is on screen management and data retrieval. You'll find a library of reusable C software tools to help create interactive systems, including file management, screen forms, etc. $21.95

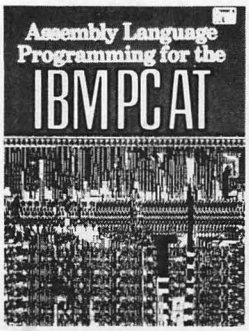


4. The best C articles from the highly respected *Dr. Dobb's Journal* dealing exclusively with C language and programming techniques. $24.95

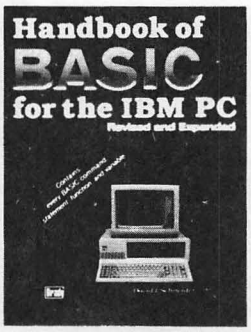


5. Our best-selling assembler book has been made even better! It now includes 30 assembler Macros and version 2.0 of the IBM Assembler. $21.95

6. For everyone who thought APL was too unorthodox to bother learning, this guide will unlock the secrets to its logic and show you how to write the fastest, most compact code. $22.95



7. The author of our best-selling assembler books now demonstrates his detailed and accurate style on the 80286 chip. $21.95



8. *PC* magazine calls it: "A truly remarkable book...A treasure trove of useful programming information on the IBM PC." $22.95

Now at your book or computer store.
Or order toll-free today: **800-624-0023**
In New Jersey: 800-624-0024

BRADY COMMUNICATIONS COMPANY, INC.
c/o Prentice Hall
P.O. Box 512, W. Nyack, NY 10994

Circle the numbers of the titles you want below. (Payment must be enclosed; or, use your charge card.) Add $1.50 for postage and handling.
Enclosed is check for $_____ or charge to
☐ MasterCard ☐ VISA

Acc't # _____ Exp. date _____
Signature _____
Name _____
Address _____
City _____ State _____ Zip _____
(New Jersey residents, please add applicable sales tax.)
Dept. 3

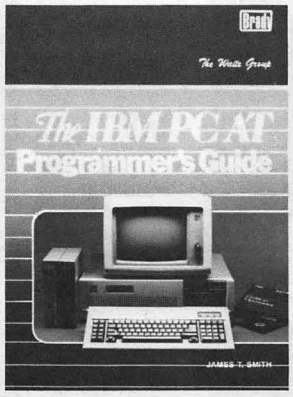
1 (0-89303-390-1) 2 (0-89303-473-8) 3 (0-89303-612-9) 4 (0-89303-599-8)
5 (0-89303-575-0) 6 (0-89303-567-X) 7 (0-89303-484-3) 8 (0-89303-510-6)

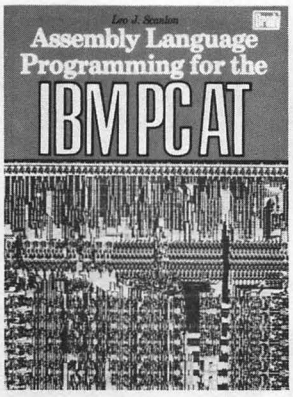
Brady

If you use an IBM PC-AT

BRADY has your number: 80286

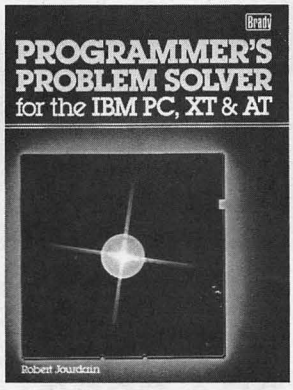
1. This guide presents the nuts and bolts of programming the 80286 to get the most out of the AT's power. It includes discussions of the entire intel CPU family for perspective and focuses on IBM BIOS to enable programmers to get the most out of its extended services. Examples include both assembly language and Pascal code to illustrate software interrupts for DOS services, extended memory access, and much, much more. $21.95

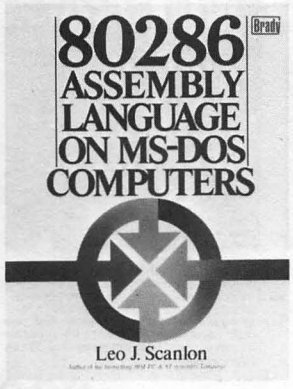




3. Written for AT "compatibles," Scanlon's plain English, tutorial style covers a crash course in computer numbering, the fundamentals of assembly language, assemblers, and the 80286's instruction set. The guide includes programs for doing high-precision arithmetic, sorting, and code conversions along with procedures for using Microsoft's Macro Assembler, EDLIN, SYMDEB debugger, and LINK. $21.95

2. Perfect for both beginners and experienced programmers, you'll find everything from the basics of computer numbering through to step-by-step instructions for using the IBM Macro Assembler. Clearly written, it presents logical groupings of the entire 80286 instruction set for quicker, easier learning along with complete coverage of BIOS and a library of over 30 macros for faster programming. It also covers graphics and sound control. $21.95 (Disk available)





4. Here's the ultimate reference source that includes over 150 solutions to common hardware-control tasks through high-level or assembly programming or system functions. It shows how to code for directory access, keyboard macros, and advanced video and sound control. Complete discussions of graphics on the EGA, port and modem control, printer manipulation, and file operations answer just about every question that arises in programming interfaces. $22.95

Now at your book or computer store.
Or order toll-free today: **800-624-0023** In New Jersey: 800-624-0024

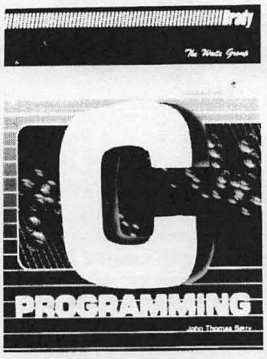
BRADY COMMUNICATIONS COMPANY, INC.
c/o Prentice Hall
P.O. Box 512, W. Nyack, NY 10994

Circle the numbers of the titles you want below. (Payment must be enclosed; or, use your charge card.) Add $1.50 for postage and handling.
Enclosed is check for $_____ or charge to
☐ MasterCard ☐ VISA.

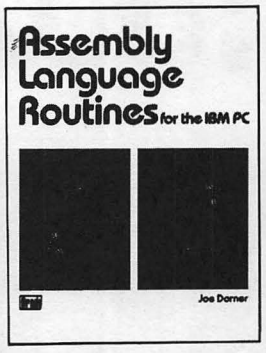
Acc't # _____ Exp. date _____
Signature _____
Name _____
Address _____
City _____ State _____ Zip _____
(New Jersey residents, please add applicable sales tax.)
Dept. 3

1 (0-89303-580-7) 2 (0-89303-484-3) 3 (0-89303-618-8) 4 (0-89303-787-7)

BRADY Knows Programming

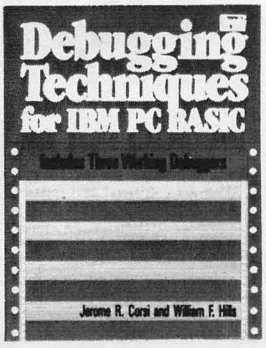


1. Beyond the basics, this guide explores new structured concepts to analyze and solve problems in C with tools of modularity, input and output functions, arrays and structures, and bit manipulation. $21.95

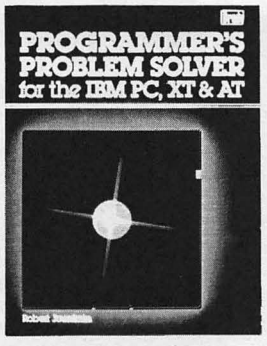


2. You learn by example with this guide through hundreds of subroutine listings. Discover how to enhance high level language programs (esp. BASIC) to quickly manipulate and sort large data files, generate graphics, integrate arrays, and use interrupts to access DOS. $17.95

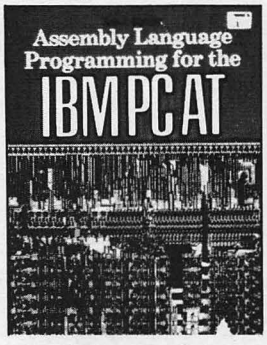
3. Learn the techniques used for creating assembly language utilities. Includes 10 of the most popular utilities such as DBUG SCAN, CLOCK, UNDELETE, ONE KEY, PCALC calculator and notepad and five others. $21.95 (Disk available)



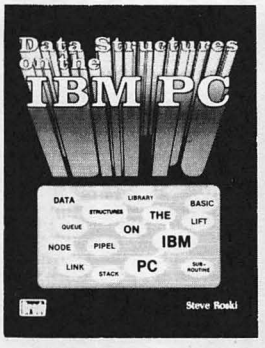
4. Includes code listings for three working debuggers including single-stepping, cross referencing, and mapping utilities. $19.95 (Disk available)



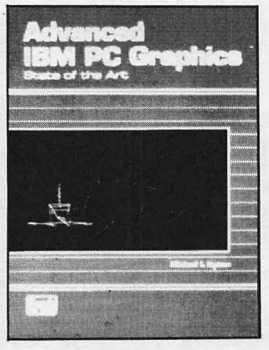
5. A definitive reference text for advanced programmers. You'll find over 150 discussions of common hardware-control tasks (in BASIC, Pascal, or C) as well as assembler overlays, drivers, and real-time operators. $22.95



6. Perfect for both beginners and experienced programmers, you'll find everything from the basics of computer numbering through to step-by-step instructions for using the IBM Macro Assembler. With complete coverage of BIOS and a library of over 30 macros for faster programming. $21.95 (Disk available)



7. Here's a compendium of many of the most useful, but often neglected, advanced programming concepts. A tutorial in format that uses BASIC for examples, it covers techniques such as: linked data structures; recursion; pipelining; and dynamic storage allocation. Includes listings for 25 subroutines. $21.95 (Disk available)



8. The title might say "advanced" but you'll find a guide that begins with the fundamentals of BASIC graphics and takes you through truly sophisticated 3-D assembly routines. Includes block graphics, creating a graphics editor, directly programming IBM's color graphics adapter, and much more. $21.95

Now at your book or computer store.
Or order toll-free today: **800-624-0023** In New Jersey: 800-624-0024

BRADY COMMUNICATIONS COMPANY, INC.
c/o Prentice Hall
P.O. Box 512, W. Nyack, NY 10994

Circle the numbers of the titles you want below. (Payment must be enclosed; or, use your charge card.) Add $1.50 for postage and handling.
Enclosed is check for $_____ or charge to
☐ MasterCard ☐ VISA

Acc't # _____ Exp. date _____
Signature _____
Name _____
Address _____
City _____ State _____ Zip _____
(New Jersey residents, please add applicable sales tax:)
Dept. 3

1 (0-89303-473-8) 2 (0-89303-409-6) 3 (0-89303-584-X) 4 (0-89303-587-4)
5 (0-89303-787-7) 6 (0-89303-484-3) 7 (0-89303-481-9) 8 (0-89303-476-2)

Brady

MAC ASSEMBLY LANGUAGE

Unlock the power of assembly language programming on the Macintosh with this master key—a thorough guide that shows you how to create the fastest and most efficient Mac programs possible, explains every nook and cranny of the 68000 microprocessor, and gives you plenty of example programs to learn from. The assembler used is version 2.0 of Apple's Macintosh 68000 Development System (MDS). This is the ideal book for programmers who want to quickly learn the fundamentals of assembly language programming on the Macintosh.

This book will lead you step-by-step through each stage in the development of an assembly language program. You will learn how to use such MDS programming tools as the editor, assembler, linker, and resource compiler. You'll also learn how to create programs that run in the unique Macintosh environment using the Window Manager, the Menu Manager, the Dialog Manager, and many other toolbox subroutines. An entire chapter is devoted to showing you how to write applications that work with desk accessories.

Cover design by Ben Santora

A Brady Book • Published by Prentice Hall Press • New York

0 21898 54143 7

ISBN 0-13-541434-2