MACINTOSH USER EDUCATION

Programming Macintosh Applications in Assembly Language /INTRO/ASSEM

See Also: Macintosh Memory Management: An Overview
The Memory Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Operating System Utilities: A Programmer's Guide
Putting Together a Macintosh Application

Modification History: First Draft S. Chernicoff 2/27/84

ABSTRACT

This manual tells you what you need to know to write all or part of your Macintosh application program in assembly language. The emphasis here is on general principles and methods; details on specific OS and Toolbox routines are given elsewhere.

TABLE OF CONTENTS

3 About This Manual
3 Definition Files
4 Memory Organization
8 The Dispatch Table
10 The Trap Mechanism
10 Format of Trap Words
12 Trap Macros
12 Calling Conventions
12 Register-Based Calls
14 Stack-Based Calls
17 Register-Saving Conventions
18 Pascal Interface to the OS and Toolbox
19 Mixing Pascal and Assembly Language
23 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.
Distribution of this draft in limited quantities does not constitute
publication.

ABOUT THIS MANUAL

This manual tells you what you need to know to write all or part of your Macintosh application program in assembly language. The emphasis here is on general principles and methods; details on specific OS and Toolbox routines are given elsewhere.

The manual assumes you already know how to write assembly language for the Motorola MC68000 (or just "68000" for short), the microprocessor used in the Macintosh. It also assumes you're familiar with Lisa Pascal and its associated software development tools, particularly the Assembler, the Pascal Compiler, and the Linker. *** (Currently, all software for the Macintosh must be developed on a Lisa computer and written on a Macintosh-formatted disk for execution on the Macintosh. Eventually development tools will be available on the Macintosh itself.) ***

The manual begins by discussing the various files of definitions pertaining to the OS and Toolbox, and what they contain. Then it describes the Macintosh's memory layout and organization. This is followed by a description of the dispatch table and the trap mechanism, which allow your program to use the OS and Toolbox while remaining independent of specific addresses in the Macintosh ROM. Next is a discussion of the calling conventions for using the OS and Toolbox from assembly language and for mixing Pascal and assembly language in your own programs. Finally, there's a glossary of terms used in this manual.

DEFINITION FILES

The primary aids to assembly-language programmers are a set of definition files that define various symbolic names for use in assembly-language programs. By naming the definition files in an .INCLUDE directive, you make the definitions available to your program.

The most important of the definition files are the equates files, which use .EQU directives to define values for symbolic names. There are separate _system_, _QuickDraw_, and _Toolbox equates files_ for definitions related to the Operating System, QuickDraw, and the User Interface Toolbox. There are also a number of specialized equates files, such as the memory equates file, which contains definitions pertaining to memory allocation. These specialized files are discussed in the individual manuals that apply to them (for instance, the memory equates file is covered in the Memory Manager manual).

The equates files define a variety of symbolic names for various purposes, such as:

- Useful numeric quantities. For example, the constant maxMenu stands for the maximum number of menus in a menu bar.

- Fixed memory addresses. For example, sysCom is the starting address of the system communication area.

- Addresses of system variables. For example, ticks is the address of a long-word integer variable containing the elapsed time in ticks (sixtieths of a second) since the system was last started up. Often the global variable in turn contains an address: for example, sysEvtBuf is the address of a pointer to the system event buffer (not the address of the buffer itself!).

- Masks. For example, tagMask is a mask for extracting the tag field from the header of a memory block.

- Bit numbers. For example, lock is the bit number of the lock bit in the first byte of a master pointer, defined for use with the bit manipulation instructions BTST (Bit Test), BSET (Bit Set), BCLR (Bit Clear), and BCHG (Bit Change).

- Codes. For example, inMenuBar is the code returned by the Window Manager function FindWindow when the user presses the mouse button inside the menu bar.

- Offsets into data structures. For example, wVisible is the offset of a window's "visible" flag relative to the beginning of the window record.

It's a good idea always to use the symbolic names defined in an equates file in place of the corresponding numerical values (even if you know them), since some of these values may be subject to change. One thing to watch out for is that the names of the offsets for a data structure don't always match the field names in the corresponding Pascal definition. In the OS and Toolbox documentation, the definitions are normally shown in their Pascal form; the corresponding offset constants for assembly-language use are listed in the summary at the end of each manual.

In addition to the equates files, there's also a _system errors file_, which defines symbolic names for all error codes returned by Operating System routines. Finally, there are the _system_, _QuickDraw_, and _Toolbox macro files_, which define the macros used to call OS and Toolbox routines from assembly language.

MEMORY ORGANIZATION

In its current configuration, the Macintosh has 128K bytes of volatile read/write memory (RAM) and 64K bytes of permanent read-only memory (ROM). The ROM contains the built-in code of the Operating System and User Interface Toolbox, which is available for use by any application program. In the 68000's 16-megabyte address space, RAM occupies addresses $0-$1FFFF and ROM is at addresses $400000-$40FFFF.

In addition, the various built-in input/output devices are "memory-mapped", meaning that they appear to the processor as addressable memory locations with special properties. The 6522 VIA (Versatile Interface Adapter) occupies addresses in the range $E00000-$EFFFFF, the 8530 SCC (Serial Communications Controller) $900000-$9FFFFF and $B00000-$BFFFFF, and the IWM ("Integrated Woz Machine") disk interface $D00000-$DFFFFF. You won't ordinarily need to know any details about these memory-mapped devices, since you'll deal with them exclusively through the Operating System.

(warning)

All specific memory addresses given in this section refer to the first-release, 128K Macintosh. The Lisa 2 Macintosh emulator uses a different memory layout, as will future versions of Macintosh with different memory capacities. For compatibility, always refer to these RAM addresses by their symbolic names (given in a table below) rather than their numeric values. For calls to OS and Toolbox routines located in ROM, use the 68000's unimplemented instruction trap, as described below under "The Trap Mechanism". This ensures compatibility by making all ROM references indirectly, through a dispatch table kept in RAM.

The organization of RAM is shown in Figure 1. The first $100 bytes (addresses $0-$FF) are reserved by the 68000 hardware for use as exception vectors. The next $300 bytes ($100-$3FF), referred to as the "system communication area", contain global variables used by various parts of the Macintosh system software. The next $400 bytes ($400-$7FF) contain the dispatch table for OS and Toolbox routines, discussed below under "The Dispatch Table". This is followed by $300 bytes ($800-$AFF) of additional system globals.

At (almost) the very end of memory are the main sound buffer ($1FD00-$1FFE3), used by the Sound Driver to control the sounds emitted by the built-in speaker, and the main screen buffer ($1A700-$1FC7F), which holds the bit image to be displayed on the Macintosh screen. If an interactive debugger such as MacsBug is installed, it immediately precedes the screen buffer. Then comes an area reserved for the application's parameters and global variables, which normally also includes a block of global variables belonging to QuickDraw. When the Segment Loader starts up an application, it adjusts the size of this area according to the application's needs and sets register A5 to point to the boundary between the application's parameters and globals. (This subject is covered in more detail in the Segment Loader manual.)

(note)

For special applications, there are an alternate screen buffer ($12700-$17C7F) and an alternate sound buffer ($1A100-$1A3E3). If you use either or both of these, the application parameters (or the debugger, if there is one) end at $126FF or $1A0FF instead of the normal $1A6FF, and the space available for dynamic allocation (see below) is reduced accordingly.

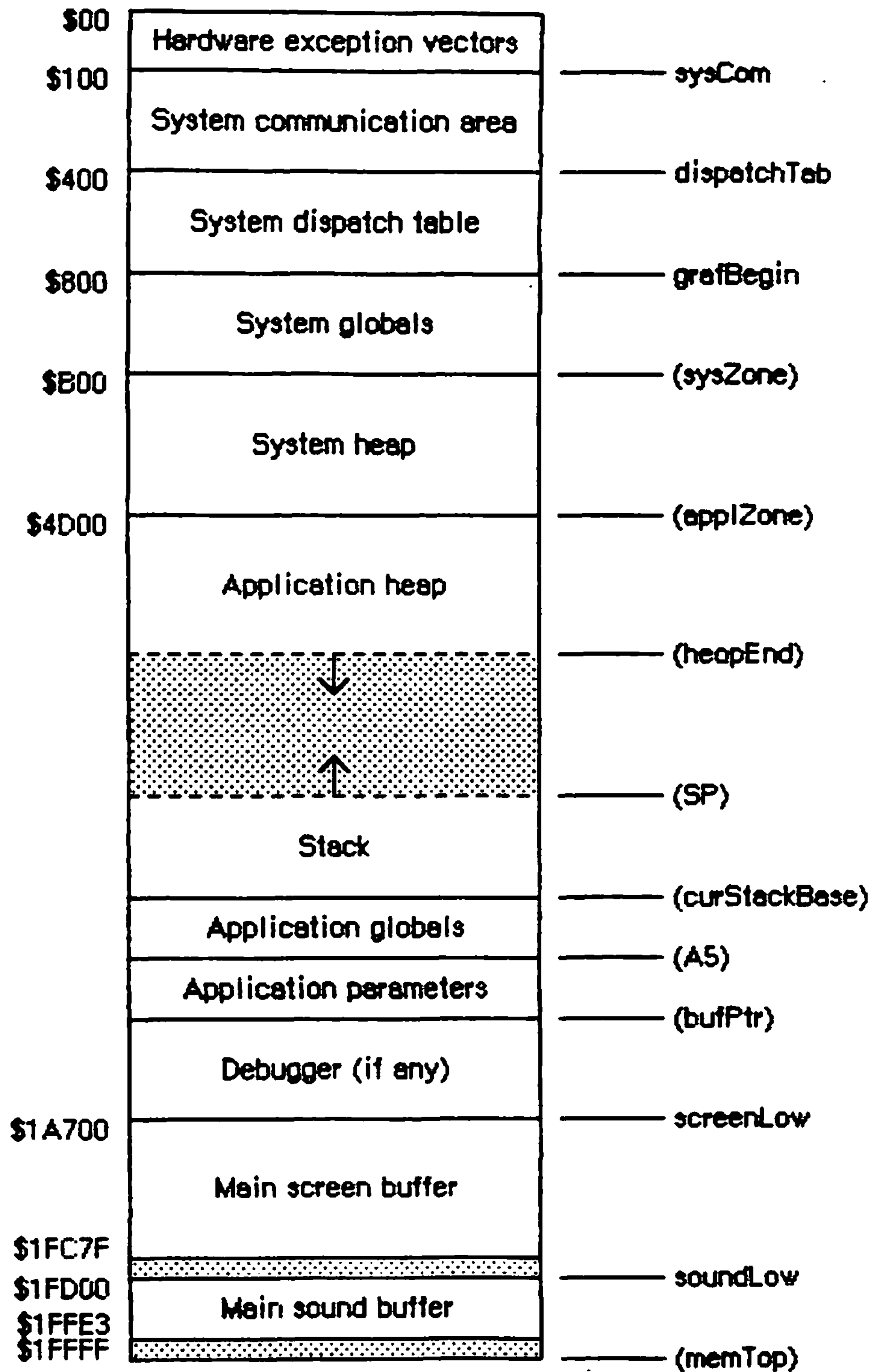


Figure 1. RAM Organization

All remaining space, between the end of the system globals ($B00) and the beginning of the application globals, is available for dynamic allocation by the running program. This space is shared between the stack and the heap, with the heap growing forward from the beginning of the space and the stack growing backward from the end. (The stack and the heap are discussed in general terms in the document "Macintosh Memory Management: An Overview" *** which will be the chapter preceding this one in the eventual "Inside Macintosh" manual *** and in greater detail in the Memory Manager manual.)

Immediately following the system globals is the system heap, which is initialized to a fixed size (currently 16.5K, or $4200 bytes) when the system is started up. The system heap is intended for the system's own private use; your application program should use the application heap for all of its heap allocation. (In particular, the code of the application itself resides in the application heap.) The application heap is initialized at the start of each new application program (currently to 6K, or $1800 bytes), and can then expand as required to accommodate the application's needs. The stack grows and shrinks from the other end of the space.

(warning)

> Although the 68000 hardware provides for separate user and supervisor stacks, each with its own stack pointer, the Macintosh maintains only one stack. All application programs run in supervisor mode and share the same stack with the system; the user stack pointer isn't used.

The boundaries between the various areas of RAM are marked by global constants and variables defined in the equates files. In the following table (as well as in Figure 1), names not shown in parentheses are constants that are equated directly to the designated address; those in parentheses are variables containing long-word pointers that in turn point to the address. Names identified as marking the end of an area actually refer to the address **following** the last byte in that area.

Name	Meaning
sysCom	Start of system communication area
dispatchTab	Start of system dispatch table
grafBegin	Start of additional system globals
(sysZone)	Start of system heap
(applZone)	Start of application heap
(heapEnd)	End of application heap
(curStackBase)	Base (end) of stack; start of application globals
(bufPtr)	End of application parameters
screenLow	Start of main screen buffer
(scrnBase)	Start of current screen buffer
soundLow	Start of main sound buffer
(soundBase)	Start of current sound buffer
(memTop)	End of RAM
romStart	Start of ROM

THE DISPATCH TABLE

The bulk of the Operating System and Toolbox resides in read-only memory (ROM). However, to allow flexibility for future development, application code must be kept free of any specific ROM addresses. So all references to OS and Toolbox routines are made indirectly, through a dispatch table in RAM containing the addresses of the routines. As long as the location of the dispatch table is known, the routines themselves can be moved to different locations in ROM without disturbing the operation of programs that depend on them.

Information about the locations of the various OS and Toolbox routines is encoded in compressed form in the ROM itself. When the system is started up, this encoded information is expanded to form the dispatch table. Because the dispatch table resides in RAM (locations $400-$7FF), individual entries can be "patched" to point to addresses other than the original ROM address. This allows changes to be made in the ROM code by loading corrected versions of individual routines into RAM at system startup and patching the dispatch table to point to them. It also allows an application program to replace specific OS and Toolbox routines with its own "custom" versions. A pair of utility routines for manipulating the dispatch table, GetTrapAddress and SetTrapAddress, are described in the Operating System Utilities manual.

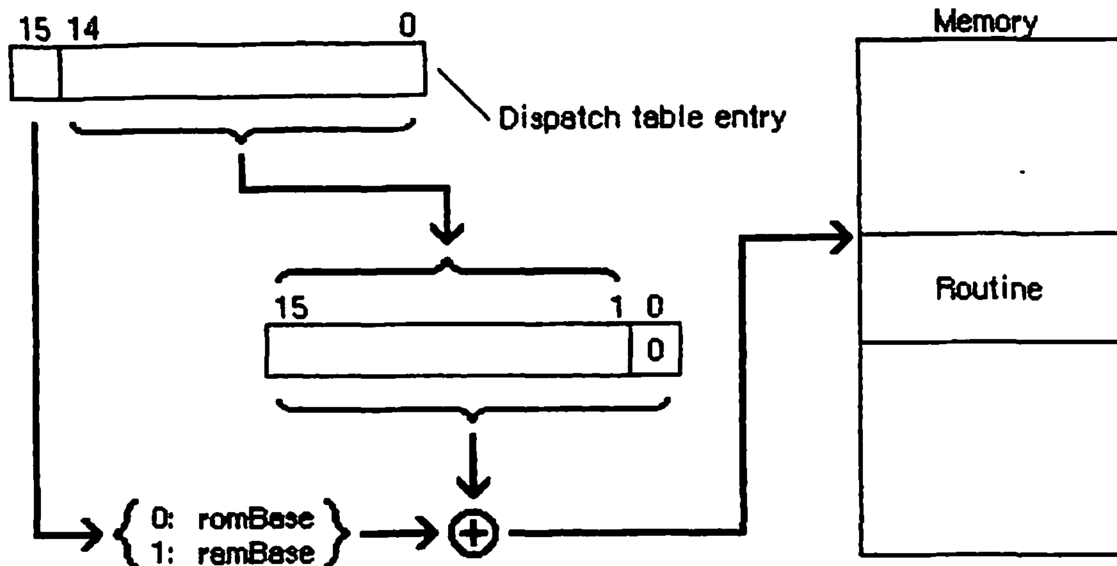


Figure 2. Dispatch Table Entry

For compactness, entries in the dispatch table are encoded into one word each, instead of a full long-word address (see Figure 2). Since the dispatch table is 1024 ($400) bytes long, it has room for 512 word-length entries. The high-order bit of each entry tells whether the routine resides in ROM (0) or RAM (1). The remaining 15 bits give the offset of the routine relative to a base address. For routines in ROM, this base address is the beginning of the ROM, address $400000; for routines in RAM, it's the beginning of the system heap, currently at address $B00.

(note)

The two base addresses are kept in a pair of global variables named romBase and ramBase.

The offset in a dispatch table entry is expressed in words instead of bytes, taking advantage of the fact that instructions must always fall on word boundaries (even byte addresses). To find the absolute address of the routine, the system checks the high-order bit of the dispatch table entry to find out which base address to use, doubles the offset to convert it from words to bytes, and adds the result to the designated base address.

Using 15-bit word offsets, the dispatch table can address locations within a range of 32K words, or 64K bytes, from the base address. Starting from romBase, this range is big enough to cover the entire ROM; but only half of the 128K RAM lies within range of ramBase. Since all RAM-based code resides in the heap, ramBase is set to the beginning of the system heap to maximize the amount of useful space within range.

Locations below the start of the heap ($BØØ) are used to hold global system data (including the dispatch table itself), and can never contain executable code; but if the heap is big enough, it's possible for some of the application's code to lie beyond the upper end of the dispatch table's range ($1ØAFF). Any such code is inaccessible through the dispatch table.

(note)

> This problem will become particularly acute on the Lisa 2 and on future versions of Macintosh with more than 128K of RAM. To make sure they lie within range of ramBase, patches to OS and Toolbox routines are typically placed in the system heap rather than the application heap.

THE TRAP MECHANISM

Calls to the OS and Toolbox via the dispatch table are implemented by means of the 68ØØØ processor's "1Ø1Ø emulator" trap. To issue such a call in assembly language, you use one of the trap macros defined in the system, QuickDraw, and Toolbox macro files. When you assemble your program, the macro generates a trap word in the machine-language code. A trap word always begins with the hexadecimal digit $A (binary 1Ø1Ø); the rest of the word identifies the routine you're calling, along with some additional information pertaining to the call.

Instruction words beginning with $A don't correspond to any valid machine-language instruction, and are known as unimplemented instructions. They're used to augment the processor's native instruction set with additional operations that are "emulated" in software instead of being executed directly by the hardware. On the Macintosh, the additional operations are the OS and Toolbox routines. Attempting to execute an unimplemented instruction causes a trap to the Trap Dispatcher, which examines the bit pattern of the trap word to determine what operation it stands for, looks up the address of the corresponding routine in the dispatch table, and jumps to the routine.

Format of Trap Words

As noted above, a trap word always begins with the digit $A in bits 12-15, the mark of an unimplemented instruction. Bit 11 tells whether the call is to the Operating System (Ø) or the Toolbox (1). The format of the rest of the word depends on whether it's an OS or a Toolbox call.

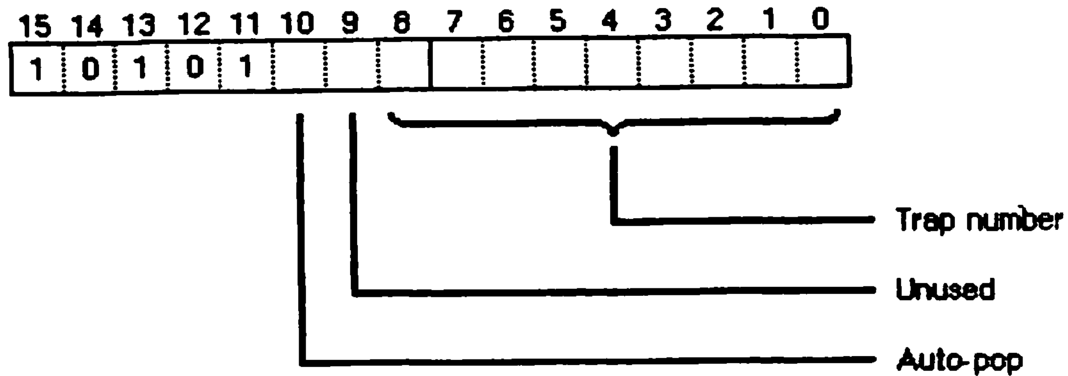


Figure 3. Trap Word Format for Toolbox Calls

Figure 3 shows the trap word format for Toolbox calls. Bits Ø-8 form a 9-bit trap number identifying the particular Toolbox routine being called. Bit 9 is unused; bit 1Ø is called the "auto-pop" bit and is discussed below under "Pascal Interface to the OS and Toolbox".

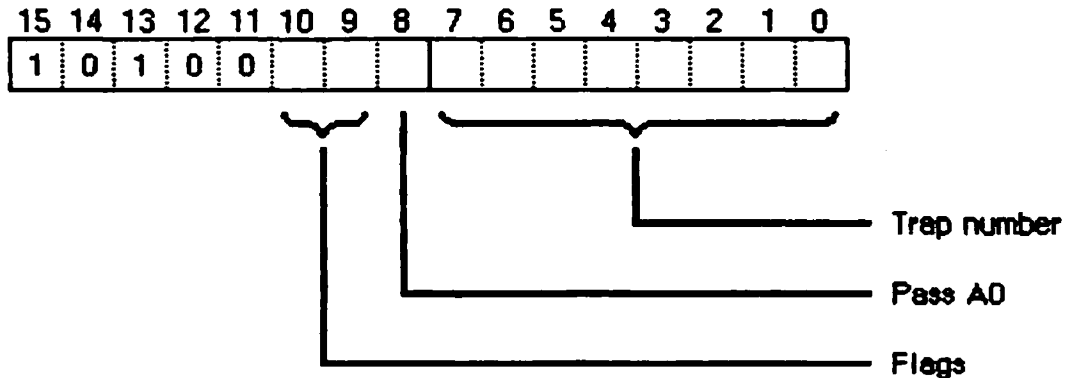


Figure 4. Trap Word Format for OS Calls

For Operating System calls, only the low-order 8 bits (bits Ø-7) are used for the trap number (see Figure 4). Thus of the 512 entries in the dispatch table, only the first 256 can be used for OS traps. Bit 8 of an OS trap has to do with register usage and is discussed below under "Register-Saving Conventions". Bits 9 and 1Ø have specialized meanings depending on which OS routine you're calling, and are covered where relevant in other manuals.

Trap Macros

The names of all trap macros begin with the underscore character (_), followed by the name of the corresponding routine. As a rule, the macro name is the same as the name used to call the routine from Pascal, as given in the OS and Toolbox documentation. For example, to call the Window Manager routine NewWindow, you would use an instruction with the macro name _NewWindow in the op code field. There are a few exceptional cases, however, in which the spelling of the macro name differs from the name of the routine itself; these exceptions are noted in the documentation for the individual routines.

Trap macros for Toolbox calls take no arguments; those for OS calls may have as many as three optional arguments. The first argument, if present, is used to load a register with a parameter value for the routine you're calling, and is discussed below under "Register-Based Calls". The remaining arguments control the settings of the various flag bits in the trap word. The form of these arguments varies with the meanings of the flag bits, and is described in the manuals on the relevant parts of the Operating System.

CALLING CONVENTIONS

The calling conventions for Operating System and Toolbox routines fall into two categories: register-based and stack-based. As the terms imply, register-based routines receive their parameters and return their results in the processor's registers; stack-based routines communicate via the stack, following the same conventions used by the Pascal Compiler for routines written in Pascal. Before calling any OS or Toolbox routine, you have to set up the parameters in the way the routine expects.

(note)

> As a general rule, Operating System routines are register-based and Toolbox routines stack-based, but there are exceptions on both sides. Throughout this documentation, register-based calling conventions are given for all routines that have them; if none is shown, then the routine is stack-based.

Register-Based Calls

By convention, register-based routines normally use register A0 for passing addresses (such as pointers to data objects) and D0 for other data values (such as integers). Depending on the routine, these registers may be used to pass parameters to the routine, result values back to the calling program, or both. For routines that take more than two parameters (one address and one data value), the parameters are normally collected in a parameter block in memory and a pointer to the parameter block is passed in A0. However, not all routines obey these

conventions; for example, some expect parameters in other registers, such as A1. See the documentation on each individual routine for details.

Whatever the conventions may be for a particular routine, it's up to you to set up the parameters in the appropriate registers before calling the routine. For instance, the Memory Manager utility procedure BlockMove, which copies a block of consecutive bytes from one place to another in memory, expects to find the address of the first source byte in register A0, the address of the first destination location in A1, and the number of bytes to be copied in D0. So to move 20 bytes beginning at address srcAddr to locations beginning at destAddr, you might write something like

```
LEA       srcAddr,A0       ;source address in A0
LEA       destAddr,A1      ;destination address in A1
MOVEQ     #20,D0           ;byte count in D0
_BlockMove                 ;trap to routine
```

Because many register-based routines expect to find an address of some sort in register A0, the trap macros allow you to specify the contents of that register as an argument to the macro instead of explicitly setting up the register yourself. The first argument you supply to the macro, if any, represents an address to be passed in A0. The macro will load the register with an LEA (Load Effective Address) instruction before trapping to the routine. So, for instance, to perform a Read operation on a file, you could set up the parameter block for the operation and then use the instruction

```
_Read     paramBlock       ;trap to routine with
                           ; pointer to parameter
                           ; block in A0
```

This feature is purely a convenience, and is optional: if you don't supply any arguments to a trap macro, or if the first argument is null, the LEA to A0 will be omitted from the macro expansion. Notice that A0 is loaded with the actual address denoted by the argument, not the contents of that address.

(note)

You can use any of the 68000's addressing modes to specify this address, with one exception: you can't use the two-register indexing mode ("address register indirect with index and displacement"). An instruction such as

```
_Read     offset(A3,D5)
```

won't work properly, because the comma separating the two registers will be taken as a delimiter marking the end of the macro argument.

Many register-based routines return a 16-bit result code in the low-order half of register DØ to report successful completion or failure due to some error condition. A negative result code always signals an error of some kind; a code of Ø denotes successful completion. (Some routines also use DØ to return an actual data result. In these cases, any nonnegative value in the low-order half of the register represents a true result and implies successful completion of the routine.) The system errors file defines symbolic names for all result codes reported by the various OS routines.

Just before returning from a register-based call, the Trap Dispatcher tests the low-order half of DØ with a TST.W instruction to set the processor's condition codes. You can then check for an error by branching directly on the condition codes, without any explicit test of your own: for example,

```
_PurgeMem                ;trap to routine
BMI       Error          ;branch on error

. . .                    ;no error--actual result
                         ; in low half of DØ
```

(warning)
Not all register-based routines return a result code. Some leave the contents of DØ unchanged; others use the full 32 bits of the register to return a long-word result. See the documentation of individual routines for details.

Stack-Based Calls

To call a stack-based routine from assembly language, you have to set up the parameters on the stack in the same way the compiled object code would if your program were written in Pascal. The number and types of parameters expected on the stack depend on the routine being called. The number of bytes each parameter occupies depends on its type:

Parameter type	Number of bytes	Contents
BOOLEAN	1 byte	Low-order bit = Ø (FALSE) or 1 (TRUE)
CHAR	1 byte	ASCII character code
INTEGER	2 bytes	Twos-complement integer
LongInt	4 bytes	Twos-complement integer
REAL	4 bytes	Sign bit, 8-bit biased exponent, 23-bit mantissa
String	4 bytes	Pointer to string; first byte pointed to gives length of string in characters
Record, array	1-4 bytes	Contents of structure if <= 4 bytes; otherwise pointer to structure
Pointer	4 bytes	Address of value
Handle	4 bytes	Address of master pointer
VAR parameter	4 bytes	Address of variable, regardless of type

If the routine you're calling is a function, the first step is to reserve space on the stack for the function result. Then, for both functions and procedures, push the parameters onto the stack in the order they occur in the routine's Pascal definition. Finally, call the routine by executing the corresponding trap macro. The trap pushes the return address onto the stack, along with an extra word of processor status information. The Trap Dispatcher removes this extra status word, leaving the stack in the state shown in Figure 5 on entry to the routine. The routine itself is responsible for removing its own parameters from the stack before returning. If it's a function, it leaves its result on top of the stack; if it's a procedure, it restores the stack to the same state it was in before the call.

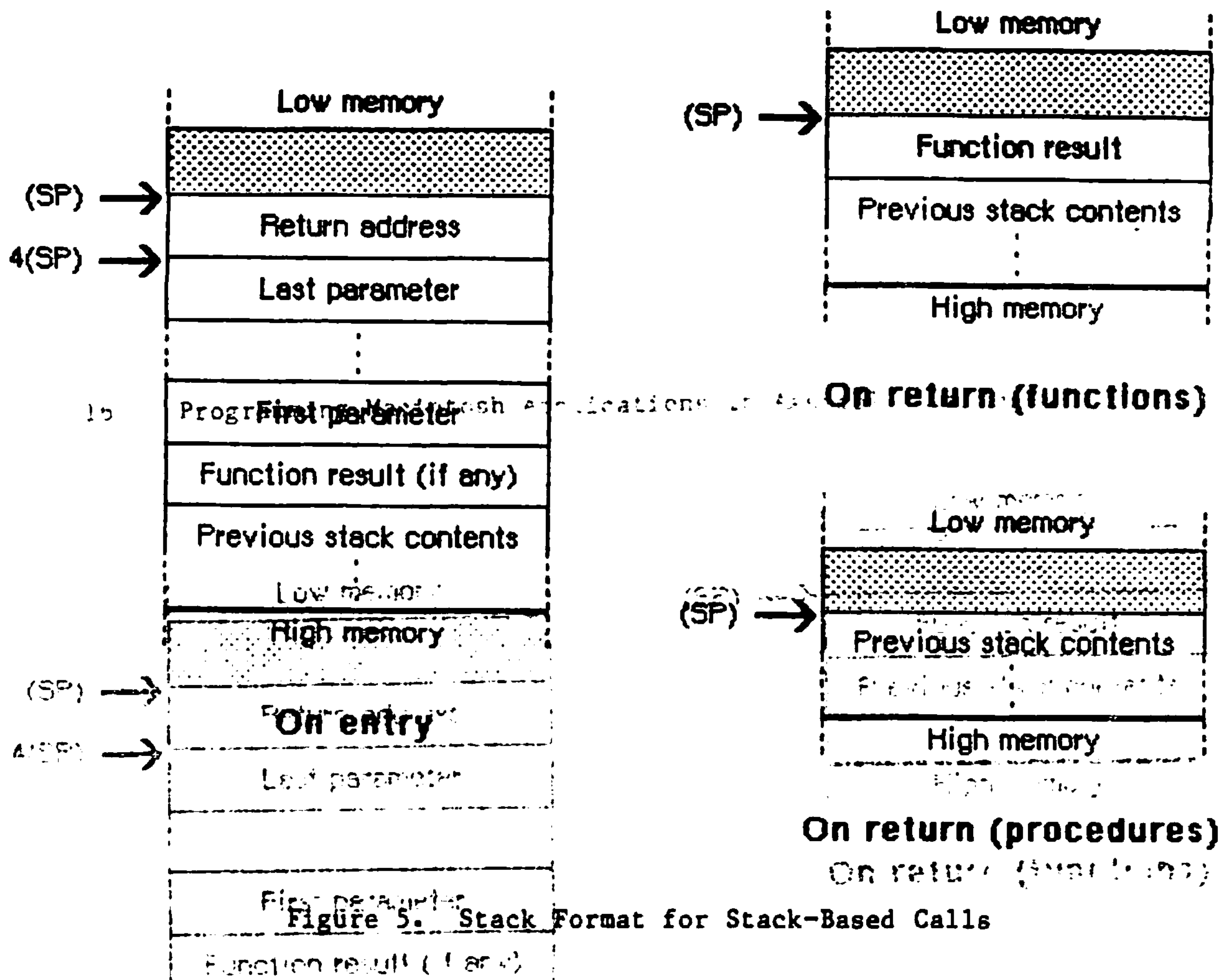


Figure 5. Stack Format for Stack-Based Calls

For example, the Window Manager function GrowWindow is defined in Pascal as follows:

```
FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point;
                     sizeRect: Rect) : LongInt;
```

To call this function from assembly language, you'd write something like the following:

```
SUBQ.L   #4,SP              ;make room for LongInt result
MOVE.L   theWindow,-(SP)    ;push window pointer
MOVE.L   startPt,-(SP)      ;a Point is a 4-byte record,
                            ;  so push actual contents
PEA      sizeRect           ;a Rect is an 8-byte record,
                            ;  so push a pointer to it
 _GrowWindow                ;trap to routine
MOVE.L   (SP)+,D3           ;pop result from stack
```

(warning)

Don't forget that the stack pointer must always be aligned on a word boundary (that is, at an even byte address). When pushing a value with an odd number of bytes (such as a Boolean or a character), you have to add a byte of "padding" to keep the stack pointer even. Because all Macintosh application code runs in the 68000's supervisor mode, an odd stack pointer will cause a "double bus fault": a catastrophic system failure from which the only escape is to turn the power off and restart the machine.

(note)

To keep the stack pointer properly aligned, the 68000 automatically adjusts the pointer by 2 instead of 1 when you move a byte-length value to or from the stack. This special case applies only when three conditions are met: a one-byte value is being transferred; either the source or the destination is specified by predecrement or postincrement addressing; and the register being decremented or incremented is the stack pointer (A7). For example, you can push the Boolean value TRUE onto the stack with the instruction

```
        ST.B    -(SP)           ;byte-length
                                ; predecrement to
                                ; stack pointer
```

and an extra, unused byte will automatically be added to keep the stack pointer even.

However, when you use any other method to manipulate the stack pointer, it's your responsibility to make sure the pointer stays properly aligned. For instance, to reserve space on the stack for a Boolean function result, you have to remember to decrement explicitly by two bytes instead of one:

```
        SUBQ.L  #2,SP           ;make room for
                                ; Boolean result
```

The function will return its result in the high-order (even-addressed) byte of the two; the other byte is just padding and should be ignored.

Register-Saving Conventions

All OS and Toolbox routines follow Lisa Pascal's register-saving conventions, which require the routine to preserve the contents of all registers except A0, A1, and D0-D2 (and of course A7, which is special). In addition, for register-based routines, the Trap Dispatcher saves some of the remaining registers before dispatching to the routine and restores them before returning to the calling program.

Registers A1, D1, and D2 are always saved and restored in this way, so their contents are unaffected by a register-based trap even though the routine itself is allowed to "trash" them. A7 and DØ are never restored: whatever the routine leaves in these registers is passed back unchanged to the calling program, allowing the routine to manipulate the stack pointer as appropriate and to return a result code.

Whether the Trap Dispatcher preserves register AØ depends on the setting of bit 8 of the trap word. If this bit is Ø, AØ is saved and restored; if it's 1, AØ is passed back from the routine unchanged. Thus bit 8 of the trap word should be set to 1 only for those routines that return a result in AØ, and to Ø for all other routines. The trap macros automatically set this bit correctly for each routine, so you never have to worry about it yourself.

Notice, however, that the Trap Dispatcher preserves these other registers only on register-based traps. Stack-based traps preserve only those registers required by the Pascal conventions (A2-A6, D3-D7). If you want to preserve any of the other registers, you have to save them yourself before trapping to the routine--typically on the stack with a MOVEM (Move Multiple) instruction--and restore them afterward.

Pascal Interface to the OS and Toolbox

Lisa Pascal doesn't know anything about the Macintosh trap mechanism. When you call an OS or Toolbox routine from Pascal, you're actually calling an interface routine that performs the trap for you. For register-based calls, the interface routine fetches the parameters from the stack where the Pascal calling program left them, puts them in the registers where the routine expects them, then traps to the routine. On return, it moves the routine's result, if any, from a register to the stack and then returns to the calling program. (For routines that return a result code, the interface routine also moves the result code to a global variable, where it can later be accessed with a special Pascal utility routine.) For stack-based calls, there's nothing for the interface routine to do except trap to the routine and then return to the calling program.

Ordinarily this would mean that each stack-based interface routine would be two instructions long: a trap word and an RTS (Return from Subroutine) instruction. However, to save code, the interface routines to the Toolbox dispense with the RTS and instead use the "auto-pop" bit, bit 1Ø of the trap word for Toolbox traps. When this bit is set to 1, the Trap Dispatcher doesn't return control to the interface routine after the trap. Instead, it just removes the trap's return address from the stack and returns directly to the calling program. This halves the amount of memory space taken up by the Toolbox interface routines--from two words per routine to only one, the trap word itself. When you trap to a Toolbox routine from assembly language, the trap macro sets the auto-pop bit to Ø, so that control will return normally.

MIXING PASCAL AND ASSEMBLY LANGUAGE

You can mix Pascal and assembly language freely in your own programs, calling routines written in either language from the other. The Pascal and assembly-language portions of the program have to be compiled and assembled separately, then combined with the Lisa Pascal Linker. For convenience in this discussion, we'll refer to such separately compiled or assembled portions of a program as "modules", although this term isn't actually used in Lisa Pascal. You can divide a program into any number of modules, each of which may be written in either Pascal or assembly language.

References in one module to routines defined in another are called external references. The Linker resolves external references by matching them up with their definitions in other modules. You have to identify all the external references in each module so they can be resolved properly. To call an assembly-language routine from Pascal, you name the routine in a .DEF, .PROC, or .FUNC directive in the module where it's defined and declare it with an EXTERNAL declaration in the Pascal module that refers to it. To call a Pascal routine from assembly language, you declare it in the INTERFACE section of a Pascal unit to make it available to other modules and name it in a .REF directive in the assembly-language module that uses it. The actual process of linking the modules together is covered in the document "Putting Together a Macintosh Application".

All calls from one language to the other, in either direction, must obey Pascal's stack-based calling conventions (see "Calling Toolbox Routines", above). To call a Pascal routine from assembly language, you push the parameters onto the stack before the call and (if the routine is a function) look for the result on the stack on return. In an assembly-language routine to be called from Pascal, you look for the parameters on the stack on entry and leave the result (if any) on the stack before returning.

Under stack-based calling conventions, a convenient way to access a routine's parameters on the stack is with a frame pointer, using the 68000's LINK and UNLK (Unlink) instructions. You can use any address register for the frame pointer (except A7, which is reserved for the stack pointer), but on the Macintosh register A6 is conventionally used for this purpose. The instruction

```
LINK    A6,#-12
```

at the beginning of a routine saves the previous contents of A6 on the stack and sets A6 to point to them. The second operand specifies the number of bytes of stack space to be reserved for the routine's local variables: in this case, 12 bytes. The LINK instruction offsets the stack pointer by this amount after copying it into A6.

(warning)
The offset is **added** to the stack pointer, not subtracted from it. So to allocate stack space for local variables,

you have to give a **negative** offset; the instruction won't work properly if the offset is positive. Also, to keep the stack pointer correctly aligned, be sure the offset is even. For a routine with no local variables on the stack, use an offset of #Ø.

Register A6 now points to the routine's stack frame; the routine can locate its parameters and local variables by indexing with respect to this register (see Figure 6). The register itself points to its own saved contents, which are often (but needn't necessarily be) the frame pointer of the calling routine. The parameters and return address are found at positive offsets from the frame pointer.

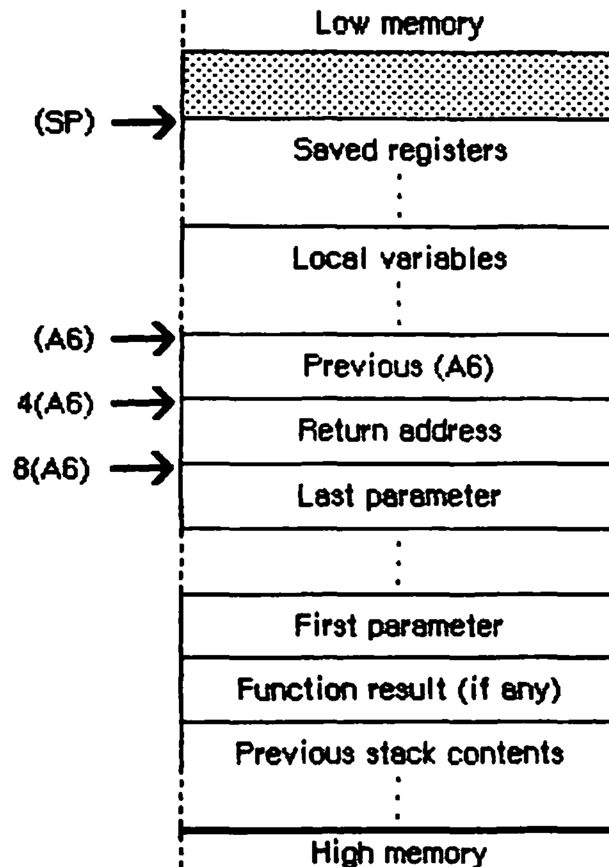


Figure 6. Frame Pointer

Since the saved contents of the frame pointer register occupy a long word (4 bytes) on the stack, the return address is located at 4(A6) and the last parameter at 8(A6). This is followed by the rest of the parameters in reverse order, and finally by the space reserved for the function result, if any. The proper offsets for these remaining parameters and for the function result depend on the number and types of the parameters, according to the table above under "Stack-Based Calls". If the LINK instruction allocated stack space for any local variables, they can be accessed at negative offsets from the frame pointer, again depending on their number and types.

At the end of the routine, the instruction

```
        UNLK    A6
```

reverses the process: first it releases the local variables by setting the stack pointer equal to the frame pointer (A6), then pops the saved contents back into register A6. This restores the register to its original state and leaves the stack pointer pointing to the routine's return address.

A routine with no parameters can now just return to the caller with an RTS (Return from Subroutine) instruction. But if there are any parameters, it's the routine's responsibility to "strip" them from the stack before returning. The usual way of doing this is to pop the return address into an address register, increment the stack pointer to remove the parameters, then exit with an indirect jump through the register.

Another point to remember is that any routine that's called from Pascal must observe Pascal register conventions and preserve registers A2-A6 and D3-D7. This is usually done by saving those registers the routine will be using on the stack with a MOVEM (Move Multiple) instruction, then restoring them before returning. Any routine you write that will be accessed via the trap mechanism--for instance, your own version of an OS or Toolbox routine that you've patched into the dispatch table--should observe the same conventions.

Putting all this together, the routine should begin with a sequence like

```
MyRoutine LINK     A6,#-dd             ;set up frame pointer--
                                       ; dd = number of bytes
                                       ; of local variables

          MOVEM.L A2-A5/D3-D7,-(SP)    ;...or whatever subset of
                                       ; these registers you use
```

and end with something like

```
          MOVEM.L (SP)+,A2-A5/D3-D7    ;restore registers
          UNLK     A6                  ;restore frame pointer

          MOVE.L   (SP)+,A1            ;save return address in a
                                       ; "trashable" register
          ADD.W    #pp,SP              ;strip parameters--
                                       ; pp = number of bytes
                                       ; of parameters
          JMP      (A1)                ;return to caller
```

Notice that A6 doesn't have to be included in the MOVEM instructions, since it's saved and restored by the LINK and UNLK.

(warning)

Recall that the Segment Loader, when it starts up an application, sets register A5 to point to the boundary between the application's globals and parameters. Certain parts of the system (notably QuickDraw and the File Manager) rely on finding A5 set up properly--so you have to be a bit more careful about preserving this register. The safest policy is never to touch A5 at all. If you must use it for your own purposes, just saving its contents at the beginning of a routine and restoring them before returning isn't enough: you have to be sure to restore it before any call that might depend on it. The correct setting of A5 is always available in the long-word global variable currentA5.

GLOSSARY

application heap: The portion of the heap available to the running application program for its own memory allocation.

dispatch table: A table in RAM containing the addresses of all Operating System and Toolbox routines in encoded form.

external reference: A reference to a routine or variable defined in a separate compilation or assembly.

frame pointer: A pointer to a routine's stack frame, held in an address register and manipulated with the LINK and UNLK instructions.

heap: The area of memory in which space is dynamically allocated and released on demand, using the Memory Manager.

interface routine: A routine called from Pascal whose purpose is to trap to a certain Operating System or Toolbox routine.

IWM ("Integrated Woz Machine"): The Macintosh's built-in custom disk interface.

parameter block: A table of parameter values to an Operating System routine, stored in memory and located by means of a pointer passed in an address register.

QuickDraw equates file: The file defining global constants and variables pertaining to QuickDraw.

QuickDraw macro file: The file defining trap macros for calling QuickDraw routines.

register-based: Said of an Operating System or Toolbox routine that receives its parameters and returns its results in the processor's registers.

result code: A code returned by an Operating System routine to report successful completion or failure due to some error condition.

SCC (Serial Communications Controller): The Macintosh's built-in 8530 serial communication interface.

stack: The area of memory in which space is allocated and released in LIFO (last-in-first-out) order, used primarily for routine parameters, return addresses, local variables, and temporary storage.

stack-based: Said of an Operating System or Toolbox routine that receives its parameters and returns its results on the stack.

stack frame: The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

system communication area: An area of memory containing global variables used by the Macintosh system software.

system equates file: The file defining global constants and variables pertaining to the Operating System.

system errors file: The file defining all result codes returned by Operating System routines.

system heap: The portion of the heap reserved for use by the Macintosh system software.

system macro file: The file defining trap macros for calling Operating System routines.

Toolbox equates file: The file defining global constants and variables pertaining to the User Interface Toolbox.

Toolbox macro file: The file defining trap macros for calling Toolbox routines.

trap macro: A macro that assembles into a trap word, used for calling an Operating System or Toolbox routine from assembly language.

trap number: The identifying number of an Operating System or Toolbox routine.

trap word: An unimplemented instruction representing a call to an Operating System or Toolbox routine.

unimplemented instruction: An instruction word that doesn't correspond to any valid machine-language instruction but instead causes a trap; used for calling Operating System and Toolbox routines via the 68000's trap mechanism.

VIA (Versatile Interface Adapter): The Macintosh's built-in 6522 parallel communication interface.

```
{$X-}
PROGRAM Boxes;

   USES {$U-}
      {$U obj/QuickDraw  } QuickDraw,
      {$U obj/OSIntf     } OSIntf,
      {$U obj/ToolIntf   } ToolIntf,
      {$U obj/Sane       } Sane,
      {$U obj/Elems      } Elems,
      {$U obj/Graf3D     } Graf3D;

   CONST
      boxCount = 15;

   TYPE
      Box3D = RECORD
                 pt1: Point3D;
                 pt2: Point3D;
                 dist: extended;
              END;

   VAR
      myPort: GrafPtr;
      myPort3D: Port3DPtr;
      boxArray: ARRAY [0..boxCount] OF Box3D;
      nBoxes: INTEGER;
      i: INTEGER;
      etop,ebottom,eleft,eright,temp: extended;

   PROCEDURE Distance(pt1,pt2: Point3D; VAR result: extended);

      VAR
         dx,dy,dz: extended;

      BEGIN
         dx := pt2.X; { dx:=pt2.X - pt1.X; }
         SubX(pt1.X,dx);

         dy := pt2.Y; { dy:=pt2.Y - pt1.Y; }
         SubX(pt1.Y,dy);

         dz := pt2.Z; { dz:=pt2.Z - pt1.Z; }
         SubX(pt1.Z,dz);

         MulX(dx,dx); { result:=SQRT(dx*dx + dy*dy + dz*dz); }
         MulX(dy,dy);
         MulX(dz,dz);
         AddX(dx,dy);
         AddX(dy,dz);
         SqrtX(dz);
         result := dz
      END;

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

      VAR
         tempRgn: RgnHandle;

      BEGIN
         tempRgn := NewRgn;
         OpenRgn;
         MoveTo3D(pt1.X,pt1.Y,pt1.Z); { front face, y=y1 }
         LineTo3D(pt1.X,pt1.Y,pt2.Z);
         LineTo3D(pt2.X,pt1.Y,pt2.Z);
         LineTo3D(pt2.X,pt1.Y,pt1.Z);
         LineTo3D(pt1.X,pt1.Y,pt1.Z);
```

```
      CloseRgn(tempRgn);
      FillRgn(tempRgn, white);

      OpenRgn;
      MoveTo3D(pt1.X,pt1.Y,pt2.Z); { top face, z=z2 }
      LineTo3D(pt1.X,pt2.Y,pt2.Z);
      LineTo3D(pt2.X,pt2.Y,pt2.Z);
      LineTo3D(pt2.X,pt1.Y,pt2.Z);
      LineTo3D(pt1.X,pt1.Y,pt2.Z);
      CloseRgn(tempRgn);
      FillRgn(tempRgn, gray);

      OpenRgn;
      MoveTo3D(pt2.X,pt1.Y,pt1.Z); { right face, x=x2 }
      LineTo3D(pt2.X,pt1.Y,pt2.Z);
      LineTo3D(pt2.X,pt2.Y,pt2.Z);
      LineTo3D(pt2.X,pt2.Y,pt1.Z);
      LineTo3D(pt2.X,pt1.Y,pt1.Z);
      CloseRgn(tempRgn);
      FillRgn(tempRgn, black);

      PenPat(white);
      MoveTo3D(pt2.X,pt2.Y,pt2.Z); { outline right }
      LineTo3D(pt2.X,pt2.Y,pt1.Z);
      LineTo3D(pt2.X,pt1.Y,pt1.Z);
      PenNormal;

      DisposeRgn(tempRgn);
    END;

  PROCEDURE MakeBox;

    VAR
      myBox: Box3D;
      i,j,h,v: INTEGER;
      p1,p2: Point3D;
      myRect: Rect;
      testRect: Rect;
      temp: extended;

    BEGIN
      I2X(Random,p1.X); {p1.x:=Random mod 70 -15;}
      I2X(140,temp);
      RemX(temp,p1.X,i);
      I2X(15,temp);
      SubX(temp,p1.X);

      I2X(Random,p1.Y); {p1.y:=Random mod 70 -10;}
      I2X(140,temp);
      RemX(temp,p1.Y,i);
      I2X(10,temp);
      SubX(temp,p1.Y);

      I2X(0,p1.Z); {p1.z:=0.0;}

      I2X(Random,p2.X); {p2.x:=p1.x + 10 + ABS(Random) MOD 30; }
      I2X(60,temp);
      RemX(temp,p2.X,i);
      AbsX(p2.X);
      I2X(10,temp);
      AddX(temp,p2.X);
      AddX(p1.X,p2.X);

      I2X(Random,p2.Y); {p2.y:=p1.y + 10 + ABS(Random) MOD 45; }
      I2X(90,temp);
      RemX(temp,p2.Y,i);
      AbsX(p2.Y);
      I2X(10,temp);
```

```
      AddX(temp,p2.Y);
      AddX(p1.Y,p2.Y);

      I2X(Random,p2.Z); {p2.z:=p1.z + 10 + ABS(Random) MOD 35; }
      I2X(70,temp);
      RemX(temp,p2.Z,i);
      AbsX(p2.Z);
      I2X(10,temp);
      AddX(temp,p2.Z);
      AddX(p1.Z,p2.Z);

      { reject box if it intersects one already in list }
      WITH myRect DO
         BEGIN {
              SetRect(myRect,ROUND(p1.x),ROUND(p1.y),ROUND(p2.x),ROUND(p2.y));
              }
         X2I(p1.X,left);
         X2I(p1.Y,top);
         X2I(p2.X,right);
         X2I(p2.Y,bottom)
         END;
      FOR i := 0 TO nBoxes-1 DO
         BEGIN
         WITH boxArray[i],testRect DO
            BEGIN { SetRect(myRect,ROUND(pt1.x),ROUND(pt1.y) }
            X2I(pt1.X,left); { ,ROUND(pt2.x),ROUND(pt2.y)); }
            X2I(pt1.Y,top);
            X2I(pt2.X,right);
            X2I(pt2.Y,bottom)
            END;
         IF SectRect(myRect,testRect,testRect) THEN EXIT(MakeBox)
         END;

      myBox.pt1 := p1;
      myBox.pt2 := p2;

      { calc midpoint of box and its distance from the eye }

      AddX(p2.X,p1.X); { p1.x:=(p1.x + p2.x)/2.0; }
      I2X(2,temp);
      DivX(temp,p1.X);

      AddX(p2.Y,p1.Y); { p1.y:=(p1.y + p2.y)/2.0; }
      I2X(2,temp);
      DivX(temp,p1.Y);

      AddX(p2.Z,p1.Z); { p1.z:=(p1.z + p2.z)/2.0; }
      I2X(2,temp);
      DivX(temp,p1.Z);

      Transform(p1,p2);
      Distance(p2,myPort3D^.eye,myBox.dist); { distance to eye }

      i := 0;

      boxArray[nBoxes].dist := myBox.dist; { sentinel }

      WHILE CmpX(myBox.dist,GT,boxArray[i].dist) { myBox.dist >
                                                  boxArray[i].dist }
           DO
         i := i+1; { insert in order of dist }
      FOR j := nBoxes DOWNTO i+1 DO boxArray[j] := boxArray[j-1];
      boxArray[i] := myBox;
      nBoxes := nBoxes+1;
   END;

BEGIN { main program }
```

```
   InitGraf(@thePort);

   HideCursor;
   NEW(myPort); OpenPort(myPort);
   NEW(myPort3D); Open3DPort(myPort3D);

   ViewPort(myPort^.portRect); { put the image in this rect }
   I2X(-100,eleft);
   I2X(75,etop);
   I2X(100,eright);
   I2X(-75,ebottom);
   LookAt(eleft,etop,eright,ebottom); { aim the camera into 3D space }
   I2X(30,temp);
   ViewAngle(temp); { choose lens focal length }
   Identity;
   I2X(20,temp);
   Roll(temp);
   I2X(70,temp);
   Pitch(temp); { roll and pitch the plane }

   REPEAT

      nBoxes := 0;
      REPEAT
         MakeBox
      UNTIL nBoxes=boxCount;

      PenPat(white);
      BackPat(black);
      EraseRect(myPort^.portRect);

      FOR i := -10 TO 10 DO
         BEGIN
         I2X(i*10,eleft);
         I2X(-100,etop);
         I2X(0,temp);
         MoveTo3D(eleft,etop,temp);
         I2X(100,ebottom);
         LineTo3D(eleft,ebottom,temp);
         END;

      FOR i := -10 TO 10 DO
         BEGIN
         I2X(i*10,eleft);
         MoveTo3D(etop,eleft,temp);
         LineTo3D(ebottom,eleft,temp);
         END;

      FOR i := nBoxes-1 DOWNTO 0 DO DrawBrick(boxArray[i].pt1,boxArray[i].pt2);

   UNTIL button
END.
```

```
{$X-}
PROGRAM Edit;

  { Edit -- A small sample application written in Pascal }
  {         by Macintosh User Education                  }

  USES {$U-}
     {$U Obj/QuickDraw    } QuickDraw,
     {$U Obj/OSIntf       } OSIntf,
     {$U Obj/ToolIntf     } ToolIntf;

  CONST
     lastMenu = 3; { number of menus }
     appleMenu = 1; { menu ID for desk accessory menu }
     fileMenu = 256; { menu ID for File menu }
     editMenu = 257; { menu ID for Edit menu }

  VAR
     myMenus: ARRAY [1..lastMenu] OF MenuHandle;
     screenRect, dragRect, pRect: Rect;
     doneFlag, temp: BOOLEAN;
     myEvent: EventRecord;
     code, refNum: INTEGER;
     wRecord: WindowRecord;
     myWindow, whichWindow: WindowPtr;
     theMenu, theItem: INTEGER;
     hTE: TEHandle;

  PROCEDURE SetUpMenus;
  { Once-only initialization for menus }

     VAR
        i: INTEGER;
        appleTitle: STRING[1];

     BEGIN
        InitMenus; { initialize Menu Manager }
        appleTitle := ' '; appleTitle[1] := CHR(appleSymbol);
        myMenus[1] := NewMenu(appleMenu, appleTitle);
        AddResMenu(myMenus[1], 'DRVR'); { desk accessories }
        myMenus[2] := GetMenu(fileMenu);
        myMenus[3] := GetMenu(editMenu);
        FOR i := 1 TO lastMenu DO InsertMenu(myMenus[i], 0);
        DrawMenuBar;
     END; { of SetUpMenus }

  PROCEDURE DoCommand(mResult: LongInt);

     VAR
        name: STR255;

     BEGIN
        theMenu := HiWord(mResult); theItem := LoWord(mResult);
        CASE theMenu OF

           appleMenu:
              BEGIN
              GetItem(myMenus[1], theItem, name);
              refNum := OpenDeskAcc(name);
              END;

           fileMenu: doneFlag := TRUE; { Quit }

           editMenu:
              BEGIN
              IF NOT SystemEdit(theItem-1) THEN
                 BEGIN
                 SetPort(myWindow);
```

```
            CASE theItem OF

               1: TECut(hTE);

               2: TECopy(hTE);

               3: TEPaste(hTE);

            END; { of item case }
            END;
         END; { of editMenu }

      END; { of menu case }
      HiliteMenu(0);

   END; { of DoCommand }

BEGIN { main program }
   InitGraf(@thePort);
   InitFonts;
   FlushEvents(everyEvent,0);
   InitWindows;
   SetUpMenus;
   TEInit;
   InitDialogs(NIL);
   InitCursor;

   screenRect := screenBits.bounds;
   SetRect(dragRect,4,24,screenRect.right-4,screenRect.bottom-4);
   doneFlag := FALSE;

   myWindow := GetNewWindow(256,@wRecord,POINTER(-1));
   SetPort(myWindow);

   pRect := thePort^.portRect;
   InsetRect(pRect,4,0);
   hTE := TENew(pRect,pRect);
   REPEAT
      SystemTask;
      TEIdle(hTE);
      temp := GetNextEvent(everyEvent,myEvent);
      CASE myEvent.what OF

         mouseDown:
            BEGIN
            code := FindWindow(myEvent.where,whichWindow);
            CASE code OF

               inMenuBar: DoCommand(MenuSelect(myEvent.where));

               inSysWindow: SystemClick(myEvent,whichWindow);

               inDrag: DragWindow(whichWindow,myEvent.where,dragRect);

               inGrow,inContent:
                  BEGIN
                  IF whichWindow<>FrontWindow THEN
                     SelectWindow(whichWindow)
                  ELSE
                     BEGIN
                     GlobalToLocal(myEvent.where);
                     TEClick(myEvent.where,FALSE,hTE);
                     END;
                  END;

            END; { of code case }
            END; { of mouseDown }
```

```
        keyDown, autoKey:
           IF myWindow=FrontWindow THEN
              TEKey(CHR(myEvent.message MOD 256),hTE);

        activateEvt:
           IF ODD(myEvent.modifiers) { window is becoming active }
              THEN
              TEActivate(hTE)
           ELSE
              TEDeactivate(hTE);

        updateEvt:
           BEGIN
           SetPort(myWindow);
           BeginUpdate(myWindow);
           TEUpdate(thePort^.portRect,hTE);
           EndUpdate(myWindow);
           END; { of updateEvt }

     END; { of event case }

  UNTIL doneFlag;
END.
```

```
{ File -- Example code for printing, reading and writing files, and Text Edit }
{      -- by Cary Clark, Macintosh Technical Support }

PROGRAM MyFile;
  { Please read 'more about File,' included on the Mac Master disk. }
  {$DECL BUG}
  {$SETC BUG := 0}
{One good way of debugging code is to write status information to one of the
 serial ports.  Even while debugging code which uses one of the ports, the other
 can be used for transmitting information to an external terminal.

 In this program, the compile time variable BUG is set to either -1, 0 or 1
 according to the extent of the debugging information required.  Since compile
 time variables or constants are used, setting a single flag should cause the
 resulting program to have no more code than is required by the debugging level
 requested.

 If BUG is set equal to -1, then no debugging information appears;  this is as you
 would want the end user to see your product.

 BUG set to 0 provides an additional menu bar called 'debug' that can display the
 amount of memory available, compact memory, and discard segments and resources
 resident in memory.  You can do something similar to display some debugging
 information on the Mac itself if you do not have a terminal, but the penalty here
 is that you may spend much of your time debugging the code which is intended to
 debug some other part of the program.  Obviously, creating and maintaining a
 window on a screen full of other windows in untested code is a difficult thing to
 do.

 BUG set to 1 adds an additional item to the 'debug' menu that writes various runtime
 information to an external terminal.  This is the preferred method of debugging,
 since it does not interfere with the Macintosh display.  Even if you do not have
 a separate terminal, you can use the LISA terminal program to act as one.  Since
 writing a lot of debugging information to a serial port can slow the program down,
 I would recommend a way of turning the information on and off.  In this program,
 the variable DEBUG is set to true or false in the beginning of one of the first
 procedures executed, SETUP, to provide debugging information.  The DEBUG variable
 may also be set by the bottom item on the rightmost menu.}

  {$U-} {Turn off the Lisa Libraries. This is required by Workshop.}
  {$X-} {Turn off stack expansion. This is a Lisa concept, not needed on Mac.}

  {$IFC BUG > -1}
  {$D+} {Put the procedures name just after it in the code, to help in debugging}
  {$R+} {Turn on range checking. Violating the range at runtime will produce a
         check exception.}
  {$ELSEC}
  {$D-} {Do not include the procedure name in the 'production' code}
  {$R-} {Turn off range checking.}
  {$ENDC}

   USES {$U Obj/QuickDraw } QuickDraw,
        {$U Obj/OSIntf    } OSIntf,
        {$U Obj/ToolIntf  } ToolIntf,
        {$U Obj/PackIntf  } PackIntf,
        {$U Obj/StdFile   } StdFile, {later, this will be part of PackIntf}
        {$U Obj/MacPrint  } MacPrint;

   CONST
      appleMenu = 1;
      FileMenu = 2;
      EditMenu = 3;
      DebugMenu = 4;

{See the file Misc:Fileasm about the constants below.
 In this example program, I only use the first two.}
      TEScrpLength = 0; {the length of the private TextEdit scrap}
      TEScrpHandle = 1; {the handle to the private TextEdit }
```

```
    dlgFont = 2; {the font used inside alerts and dialogs}
    ScrVRes = 3; {screen vertical resolution (dots/inch)}
    ScrHRes = 4; {screen horizontal resolution (dots/inch)}
    doubleTime = 5; {double click time in 4/60's of a second}
    caretTime = 6; {caret blink time in 4/60's of a second}
    ANumber = 7; {the active alert}
    ACount = 8; {the alert stage level}

    {$IFC BUG = -1}
    lastMenu = 3; { number of menus w/o debug}
    {$ELSEC}
    lastMenu = 4; { number of menus w/ debug}
    {$ENDC}

    {$IFC BUG < 1}
    debug = FALSE; { compiler will discard code after 'If debug ...'}
    {$ENDC}

TYPE
    ProcOrFunc = (proc,func,neither);
    edset = SET OF 1..9;
    appParms = RECORD { params set up by Finder at launch }
                  message: INTEGER;
                  count: INTEGER; { how many icons did the user select }
                  vRefNum: INTEGER; { for each, the volume reference #, }
                  fTYPE: resType; { the file type, }
                  vByte: INTEGER; { the version number (should be 0) }
                  fName: Str255; { and the name. See SetUp for use. }
               END;
    pAppParms = ^appParms;

    MyData = RECORD {each document window keeps a handle to this in WRefCon}
                TERecord: TEHandle; {the text associated with this document}
                changed: Boolean; {the document is 'dirty'}
                titled: Boolean; {the document has never been saved to disk}
             END;
    MyDataPointer = ^MyData;
    MyDataHandle = ^MyDataPointer;
```

```
  {<<< this little beauty does a form feed when you print this out.
       Copy and Paste it to move it to your source code}
{Here are a ton of global variables.  This is not a good programming example.
 You professionals, of course, will keep the number of globals in your own
 programs to a much smaller number than shown here.}

      {these first six values are changed as windows are activated}

  VAR
      MyWindow: WindowPtr;
      MyPeek: WindowPeek; {MyPeek is the same as MyWindow}
      WindowData: MyDataHandle; {this record is pointed to by the WRefCon.}
      hTE: TEHandle; {The active text edit handle}
      vScroll: ControlHandle; {The active vertical scroll bar.}
      topline: INTEGER; {the value of VScroll, also the visible top line.}

      printhdl: THPrint; {initialized in SetUp, used by MyPrint}
      myMenus: ARRAY [1..lastMenu] OF MenuHandle; {Handles to all of the menus}
      growRect, {contains how big and small the window can grow}
      dragRect: Rect; {contains where the window can be dragged}
      tempwindow: WindowPtr; {window referenced by GetNextEvent (bad pgming.)}
      theChar: CHAR; {keyboard input goes here}
      myPoint: Point; {the point where an event took place}
      laststate: INTEGER; {last scrap state, to see if it has changed}
      doneFlag: Boolean; {set when the user quits the program}
      myEvent: EventRecord; {returned by GetNextEvent}
      scrapwind: WindowPtr; {the ClipBoard.window, which contains the scrap}
      iBeamHdl: CursHandle; {the text editing cursor}
      watchHdl: CursHandle; {the wait cursor}
      windownum: LongInt; {the # of untitled windows opened}
      windowpos: LongInt; {the # of windows opened}
      txtfile: FInfo; {'TEXT', the type of My Editor's documents}
      MyFileTypes: SFTypeList; {same as txtfile, in a format for Standard File}
      typelistptr: SFTListPtr; {pointer to 'TEXT', as seen by Standard File}
      firstchar: INTEGER; {position of first character on top visible line}
      printflag: Boolean; {the user selected 'Print ' from the File menu}
      finderprint: Boolean; {the user selected 'Print' from the finder}

      {$IFC BUG > -1}
      FreeWind: WindowPtr; {the free memory window}
      oldmem: LongInt; {the last amount of free memory}
      {$ENDC}

      {$IFC BUG = 1}
      debug: Boolean;
      {$ENDC}
      debugger: text; {the external terminal file}
      extdebughdl: StringHandle; {the menu entry}
      lf: CHAR; {chr(10), linefeed}

{--------------------------------------------------------------------------------}

   FUNCTION GlobalAddr(routineAddr: INTEGER): Ptr;
      EXTERNAL;

   FUNCTION GlobalValue(valueAddr: INTEGER): LongInt;
      EXTERNAL;
{these routines, for now, allows us to retrieve where the TextEdit private scrap
 is, and allow us to set its size.  They are defined in Misc:FileAsn.}

   {$S Utilities}
{--------------------------------------------------------------------------------}

   PROCEDURE DebugInProc(prockind: ProcOrFunc; where: Str255; location: Ptr);
{This procedure writes the executing routine's name and location in memory on the
 external terminal.  The location is especially important in a program like this
 that has segments.}
```

```
      BEGIN
         {$IFC BUG = 1}
         Write(debugger, 'in ');
         IF prockind=proc THEN Write(debugger, 'Procedure ');
         IF prockind=func THEN Write(debugger, 'Function ');
         Writeln(debugger,where, ' @ ',ord4(location),lf)
         {$ENDC}
      END;

{-----------------------------------------------------------------------------}

   PROCEDURE CursorAdjust;

      VAR
         mousePt: Point;
         tempport: GrafPtr;

      BEGIN
   { Take care of application tasks which should be executed when the machine has }
{ nothing else to do, like changing the cursor from an arrow to an I-Beam when it
  is over text that can be edited. }
         {$IFC BUG >-1}
{ If the amount of free memory is being displayed in its own window, and if it has
  changed, then create an update event so that the correct value will be displayed.}
         IF (FreeWind<>NIL) AND (FreeMem<>oldmem) THEN
            BEGIN
            oldmem := FreeMem;
            GetPort(tempport);
            SetPort(FreeWind);
            InvalRect(FreeWind^.portrect);
            SetPort(tempport)
            END;
         {$ENDC}
         GetMouse(mousePt); {where the cursor is, currently (local to the topmost
                             window)}
         IF hTE<>NIL {if text edit is currently active, (document window is
                      topmost)}
            THEN
            BEGIN
            TEIdle(hTE);
            IF (PtInRect(mousePt,hTE^^.viewrect)) {In the text edit viewrect
                                                    area.}
               THEN
               SetCursor(iBeamHdl^^) { make the cursor an I-beam.}
            ELSE
               SetCursor(arrow)
            END
      END;

{-----------------------------------------------------------------------------}

   PROCEDURE InSystemWindow;

      VAR
         DScrap: PScrapstuff;
         tempport: GrafPtr;

      BEGIN
{for desk accessories, service them with a SystemClick.  Also, check to see if they
 have changed the scrap.  If so, create an update event to redraw the clipboard.}
         IF debug THEN DebugInProc(proc, 'InSystemWindow',@InSystemWindow);
         SystemClick(myEvent,tempwindow);
         DScrap := InfoScrap;
         IF (DScrap^.scrapState<>laststate) AND (scrapwind<>NIL) THEN
            BEGIN
            GetPort(tempport);
            SetPort(scrapwind);
            InvalRect(scrapwind^.portrect);
```

```
        SetPort(tempport)
        END
    END;

{---------------------------------------------------------------------------}

  PROCEDURE SetScrollMax;

    TYPE
      txt = PACKED ARRAY [0..32000] OF 0..255;

    VAR
      cr: INTEGER;
      txtptr: ^txt;
      max: INTEGER;

    BEGIN
{This adjusts the scroll value so that the scroll bar range is not allowed to exceed
 the end of the text.  Also, the scroll bar is disabled if the max is set equal to
 the min, which is zero.  The formula for determining the range is somewhat complex.
 Sorry.}
      IF debug THEN DebugInProc(proc, 'SetScrollMax', @SetScrollMax);
      WITH hTE^^, hTE^^.viewrect DO
        BEGIN
        txtptr := pointer(htext^);
        cr := 0;
        IF teLength>0 THEN IF txtptr^[teLength-1]=13 THEN cr := 1;
        max := nLines+cr-(bottom-top+1) DIV lineHeight;
        IF max<0 THEN max := 0;
        SetCtlMax(vScroll, max);
        IF debug THEN Writeln(debugger, 'vscrollmax = ', max, lf);
        topline := -destrect.top DIV lineHeight;
        SetCtlValue(vScroll, topline);
        IF debug THEN Writeln(debugger, 'topline = ', topline, lf)
        END;
    END;

{---------------------------------------------------------------------------}

  PROCEDURE ScrollText(showcaret: Boolean);
{called to either show the caret after an action like 'Copy';
 also called to adjust the text within the window after the window is resized.  The
 same formula used in SetScrollMax is used here as well.  Don't worry about how this
 works, too much.  This possibly could be made much simpler.}

    TYPE
      txt = PACKED ARRAY [0..32000] OF 0..255;

    VAR
      bottomline, viewlines, SelLine, scrlAmount, numlines, blanklines,
      newtop: INTEGER;
      txtptr: ^txt;

    BEGIN
      IF debug THEN DebugInProc(proc, 'ScrollText', @ScrollText);
      WITH hTE^^ DO
        BEGIN
        scrlAmount := 0;
        txtptr := pointer(htext^);
        numlines := nLines; {if the last character is a carriage return, add 1
                             to numlines}
        IF teLength>0 THEN
          IF txtptr^[teLength-1]=13 THEN numlines := numlines+1;
        WITH hTE^^.viewrect DO viewlines := (bottom-top+1) DIV lineHeight; {don't
          count partial lines}
        topline := -destrect.top DIV lineHeight;
        bottomline := topline+viewlines-1;
        IF debug THEN
```

```
      BEGIN
      Write(debugger, 'nlines=',nLines: 4,'; topline=',topline: 4);
      Writeln(debugger,'; numlines=',numlines: 4,'; bottom=',bottomline:
            4,lf);
      Writeln(debugger,'viewlines=',viewlines: 4,'; showcaret=',
            showcaret,lf)
      END;
   IF showcaret THEN
      BEGIN
      SelLine := 0;
      WHILE (SelLine+1<nLines) AND (selstart>=linestarts[SelLine+1]) DO
         SelLine := SelLine+1;
      {if selstart = selend is = a cr, then add 1 to selstline}
      IF (selstart=selend) AND (selstart>0) THEN
         IF (txtptr^[selstart-1]=13) THEN SelLine := SelLine+1;
      IF debug THEN
         BEGIN
         Write(debugger,'selstart=',selstart: 5,'; selLine=',SelLine: 5);
         IF selstart>0 THEN
            Writeln(debugger,'; txtptr^[selstart-1] = 13 is ',
                  txtptr^[selstart-1]=13,lf)
         END;
      IF SelLine>bottomline THEN
         BEGIN
         scrlAmount := bottomline-SelLine;
         IF numlines-SelLine>viewlines DIV 2 THEN
            scrlAmount := scrlAmount-viewlines DIV 2
         ELSE
            scrlAmount := scrlAmount-numlines+SelLine+1
         END;
      IF SelLine<topline THEN
         BEGIN
         scrlAmount := topline-SelLine;
         IF SelLine>viewlines DIV 2 THEN
            scrlAmount := scrlAmount+viewlines DIV 2
         ELSE
            scrlAmount := scrlAmount+SelLine
         END
      END;
   IF scrlAmount=0 THEN
      BEGIN
      blanklines := viewlines-numlines+topline;
      IF blanklines<0 THEN blanklines := 0;
      IF (blanklines>0) AND (topline>0) THEN
         BEGIN
         scrlAmount := blanklines;
         IF scrlAmount>topline THEN scrlAmount := topline
         END;
      IF NOT showcaret THEN
         BEGIN
         newtop := 0;
         WHILE (newtop+1<nLines) AND (firstchar>=linestarts[newtop+1]) DO
            newtop := newtop+1;
         IF (newtop<>topline) AND (ABS(newtop-topline)>
            ABS(scrlAmount)) THEN
            scrlAmount := topline-newtop
         END
      END;
   IF debug THEN
      BEGIN
      Write(debugger,'newtop=',newtop: 4,'; blanklines=',blanklines: 4);
      Writeln(debugger,'; newtop - topline=',newtop-topline,lf)
      END;
   IF scrlAmount<>0 THEN
      BEGIN
      IF selstart=selend THEN TEDeactivate(hTE);
      TEScroll(0,scrlAmount*lineHeight,hTE);
      IF selstart=selend THEN TEActivate(hTE)
```

```
            END;
          IF debug THEN Writeln(debugger, 'scrlAmount=',scrlAmount: 4,1f);
          SetScrollMax
          END
      END;

{-----------------------------------------------------------------------------}

  PROCEDURE ToggleScrap;

      VAR
         temppeek: WindowPeek;
         getwhich: INTEGER;
         showhidehdl: StringHandle;

      BEGIN
{The clipboard comes and goes, here.  The last item in the editmenu is alternately
 made to read, 'Show Clipboard' and 'Hide Clipboard'.}
         IF debug THEN DebugInProc(proc, 'ToggleScrap',@ToggleScrap);
         IF scrapwind=NIL THEN {make it appear}
            BEGIN
            scrapwind := GetNewWindow(257,NIL,pointer(-1));
            temppeek := pointer(scrapwind);
            temppeek^.windowkind := 9;
            SetPort(scrapwind);
            InvalRect(scrapwind^.portrect);
            getwhich := 263 {hide clipboard}
            END
         ELSE {make it disappear}
            BEGIN
            DisposeWindow(scrapwind);
            scrapwind := NIL;
            getwhich := 262 {show clipboard}
            END;
         showhidehdl := GetString(getwhich);
         Hlock(pointer(showhidehdl));
         SetItem(myMenus[EditMenu],9,showhidehdl^^);
         Hunlock(pointer(showhidehdl));
         ReleaseResource(pointer(showhidehdl))
      END;

   {$IFC BUG > -1}
{-----------------------------------------------------------------------------}

  PROCEDURE ToggleFree;

      VAR
         temppeek: WindowPeek;
         getwhich: INTEGER;
         showhidehdl: StringHandle;

      BEGIN
 {just about the same as ToggleClipboard, above.  This is just for debugging fun.}
         IF debug THEN DebugInProc(proc, 'ToggleFree',@ToggleFree);
         IF FreeWind=NIL THEN {make it appear}
            BEGIN
            FreeWind := GetNewWindow(258,NIL,pointer(-1));
            temppeek := pointer(FreeWind);
            temppeek^.windowkind := 10;
            SetPort(FreeWind);
            InvalRect(FreeWind^.portrect);
            getwhich := 265;
            END
         ELSE {make it disappear}
            BEGIN
            DisposeWindow(FreeWind);
            FreeWind := NIL;
            getwhich := 264
```

```
         END;
       showhidehdl := GetString(getwhich);
       Hlock(pointer(showhidehdl));
       SetItem(myMenus[DebugMenu],1,showhidehdl^^);
       Hunlock(pointer(showhidehdl));
       ReleaseResource(pointer(showhidehdl))
     END;
  {SENDC}

{----------------------------------------------------------------------------------}

  PROCEDURE SetViewRect;

     BEGIN
{text edit's view rect is inset in the content of the window, to prevent it from
 running into the lefthand side or the scroll bar.}
       IF debug THEN DebugInProc(proc,'SetViewRect',@SetViewRect);
       WITH hTE^^.viewrect DO
          BEGIN
          hTE^^.viewrect := MyWindow^.portrect;
          left := left+4;
          right := right-15
          END
     END;

{----------------------------------------------------------------------------------}

  PROCEDURE MoveScrollBar;

     BEGIN
       {When the window is resized, the scroll bar needs to be stretched to fit.}
       IF debug THEN DebugInProc(proc,'MoveScrollBar',@MoveScrollBar);
       WITH MyWindow^.portrect DO
          BEGIN
          HideControl(vScroll);
          MoveControl(vScroll,right-15,top-1);
          SizeControl(vScroll,16,bottom-top-13);
          ShowControl(vScroll)
          END
     END;

{----------------------------------------------------------------------------------}

  PROCEDURE GrowWnd;
  { Handles growing and sizing the window and manipulating the update region. }

     VAR
        longResult: LongInt;
        height,width,newvert,oldstart: INTEGER;
        tRect,oldportrect: Rect;

     BEGIN
        IF debug THEN DebugInProc(proc,'GrowWnd',@GrowWnd);
        longResult := GrowWindow(MyWindow,myEvent.where,growRect);
        IF longResult=0 THEN EXIT(GrowWnd);
        SetCursor(watchHdl^^); {because the word wrap could take a second or two}
        height := HiWord(longResult); width := LoWord(longResult);
        SizeWindow(MyWindow,width,height,TRUE); { Now draw the newly sized
                                                  window. }
        InvalRect(MyWindow^.portrect);
        IF MyPeek^.windowkind=8 THEN {a document (not the clipboard) is being
                                      resized}
           BEGIN
           MoveScrollBar;
           WITH MyWindow^.portrect DO
              BEGIN
              width := right-left-19;
              height := bottom-top
```

```
          END;
        WITH hTE^^ DO
          BEGIN
          destrect.right := destrect.left+width;
          viewrect.right := viewrect.left+width;
          viewrect.botton := viewrect.top+height;
          firstchar := hTE^^.linestarts[topline];
          TECalText(hTE); {re-wrap the text to fit the new screen.}
    {if the rectangle is grown such that there is now blank space on the botton
    of the screen, backpedal the screen to fill it back up, if there is enough
    scrolled off the screen to do so.  Otherwise, the first character in the top line on
    the screen should continue to be somewhere on the top line after resizing}
          ScrollText(FALSE);
          END
        END;
      SetCursor(arrow)
    END; { of GrowWnd }

{-------------------------------------------------------------------------------}

  PROCEDURE MyActivate;

    VAR
      tRect: Rect;

    BEGIN
{activate events occur when one window appears in front of another.  This takes care
 of hiliting the scroll bar and deactivating the insertion caret or the text
 selection.}
      IF debug THEN DebugInProc(proc, 'MyActivate',@MyActivate);
      MyWindow := pointer(myEvent.message);
      MyPeek := pointer(MyWindow);
      IF MyPeek^.windowkind IN [8,9] THEN
        BEGIN {redraw the scrollbar area, if a document or the clipboard}
        SetPort(MyWindow);
        tRect := MyWindow^.portrect; tRect.left := tRect.right-16;
        InvalRect(tRect)
        END;
      IF MyPeek^.windowkind=8 THEN
        BEGIN {make global variables point to the information associated with
               this window}
        WindowData := pointer(GetWRefCon(MyWindow));
        vScroll := pointer(MyPeek^.ControlList);
        hTE := WindowData^^.TERecord;
        IF ODD(myEvent.modifiers) THEN
          BEGIN {this window is now top most}
          TEActivate(hTE);
          ShowControl(vScroll);
          topline := GetCtlValue(vScroll)
          END
        ELSE
          BEGIN {this window is no longer top most}
          HideControl(vScroll);
          TEDeactivate(hTE);
          hTE := NIL {a document is no longer on top}
          END
        END
    END; { of activateEvt }

{-------------------------------------------------------------------------------}

  PROCEDURE DialogueDeactivate;

    VAR
      temprect: Rect;

    BEGIN
{This routine takes care of cases where, for instance, a modal dialog is about to
```

```
pop up in front of all the other windows.  Since the Dialog Manager handles all
activate events for you, you do not get a chance to 'turn off' the controls associated
with the window.  This routine is called just before the dialog box makes its
appearance, and takes care of the hiliting as if an activate event had occured.}
        IF debug THEN DebugInProc(proc,'DialogueDeactivate',@DialogueDeactivate);
        IF hTE<>NIL THEN {for documents, only}
           BEGIN
           TEDeactivate(hTE);
           HideControl(vScroll);
           SetCursor(arrow)
           END;
        IF (frontwindow<>NIL) AND (MyPeek^.windowkind IN [8,9]) THEN
           BEGIN {this is a little kludgy, but it works.}
           MyPeek^.hilited := FALSE; {DrawGrowIcon will now unhilite.}
           temprect := MyWindow^.portrect;
           temprect.left := temprect.right-15;
           Cliprect(temprect); {clipaway the horizontal scrollbar part}
           DrawGrowIcon(MyWindow);
           Cliprect(MyWindow^.portrect);
           MyPeek^.hilited := TRUE {fix things back}
           END
     END;

  {
```

```
}
  {SS READFILE}
{---------------------------------------------------------------------------}

  FUNCTION ReadFile(VrefNo: INTEGER; fName: Str255): Boolean;

     VAR
        refNo, io: INTEGER;
        logEOF: LongInt;
        errin: Str255;

{---------------------------------------------------------------------------}

     PROCEDURE DiskRErr(io: INTEGER);

        VAR
           str: Str255;
           readfromhdl, loadedhdl: StringHandle;
           dummy: INTEGER;

        BEGIN
{A generic error is reported to the user if something goes wrong.  Amazingly little can
 go wrong, since the user does not get the chance to do things like type file names,
 remove the disk himself, and so on.  About the only errors that could happen are:

 the disk is full (for the companion writing error handler.)
 an error occured while reading/writing the disk (damaged media or hardware)

 Can you think of anything else?  An almost identical routine further down handles
 writing to disk.  Note that in both reading and writing, the entire file is handled
 by a single read/write call, and no 'disk buffer' needs to be specified by the
 programmer.}

           IF debug THEN
              BEGIN
              DebugInProc(func, 'DiskRErr', @DiskRErr);
              Writeln(debugger, errin, ' err = ', io, lf)
              END;
           readfromhdl := GetString(267); {this says 'reading from'}
           loadedhdl := GetString(269); {this says 'loaded'}
           Hlock(pointer(readfromhdl));
           Hlock(pointer(loadedhdl));
           MakeNumString(io, str);
           Paramtext(readfromhdl^^, fName, loadedhdl^^, str);
           SetCursor(arrow);
           dummy := StopAlert(256, NIL); {discribe error to user in generic way.}
           Hunlock(pointer(readfromhdl));
           Hunlock(pointer(loadedhdl));
           EXIT(ReadFile)
        END;

     BEGIN
        IF debug THEN DebugInProc(func, 'ReadFile', @ReadFile);
        SetCursor(watchHdl^^);
        ReadFile := FALSE;
        io := FSOpen(fName, VrefNo, refNo);
        {$IFC BUG = 1} {these debugging statements are for the external terminal,
                         only}
        errin := 'FSOpen';
        {$ENDC}
        IF io<>0 THEN DiskRErr(io);
        io := GetEOF(refNo, logEOF);
        {$IFC BUG = 1}
        errin := 'GetEOF';
        {$ENDC}
        IF io<>0 THEN DiskRErr(io);
        {add code here:  if file is too large, then notify user and truncate}
        SetHandleSize(hTE^^.htext, logEOF);
```

```
      IF debug THEN
         IF memerror<>0 THEN Writeln(debugger,'memerr = ',memerror: 4);
      io := FSRead(refNo,logEOF,hTE^^.htext^);
      {SIFC BUG = 1}
      errin := 'FSRead';
      {SENDC}
      IF io<>0 THEN DiskRErr(io);
      io := FSClose(refNo);
      {SIFC BUG = 1}
      errin := 'FSClose';
      {SENDC}
      IF io<>0 THEN DiskRErr(io);
      hTE^^.teLength := logEOF;
      IF NOT finderprint THEN {if printing from the finder, no window or
                                  editing information is needed}
         BEGIN
         TESetSelect(0,0,hTE);
         TECalText(hTE);
         InvalRect(hTE^^.viewrect);
         SetScrollMax;
         WindowData^^.titled := TRUE;
         WindowData^^.changed := FALSE
         END;
      ReadFile := TRUE {everything worked out OK}
   END;

{-----------------------------------------------------------------------------}

  PROCEDURE MakeAWindow(str: Str255; disk: Boolean);

    VAR
      bounds: Rect;

    BEGIN
  {A window is created here, and all associated data structures are linked to it}
      IF debug THEN DebugInProc(proc,'MakeAWindow',@MakeAWindow);
      windowpos := windowpos+1; {this position it is created to on the screen}
      bounds.left := windowpos MOD 16*20+5;
      bounds.top := windowpos MOD 11*20+45;
      bounds.right := bounds.left+200;
      bounds.bottom := bounds.top+100;
      MyWindow := NewWindow(NIL,bounds,str,TRUE,0,pointer(-1),TRUE,0);
      SetPort(MyWindow);
      MyPeek := pointer(MyWindow);
      TextFont(2); {the good ole application font}
      MyPeek^.windowkind := 8; {an arbitrary number identify the type of
                                  window}
      hTE := TENew(MyWindow^.portrect,MyWindow^.portrect);
      WindowData := pointer(NewHandle(8)); {1 handle plus 2 booleans}
      SetWRefCon(MyWindow,ord4(WindowData));
      WindowData^^.TERecord := hTE;
      SetViewRect;
      hTE^^.destrect := hTE^^.viewrect;
      WindowData^^.changed := FALSE;
      vScroll := GetNewControl(256,MyWindow);
      MoveScrollBar;
      topline := 0
    END;

{-----------------------------------------------------------------------------}

  PROCEDURE MyGetFile;

    VAR
      reply: SFReply;
      wher: Point;
      NameHdl: StringHandle;
      temprect: Rect;
```

```
        tempport: GrafPtr;

     BEGIN
{This calls Standard File to allow the user to choose the document on disk that she
 wishes to edit.}
        IF debug THEN DebugInProc(proc, 'MyGetFile', @MyGetFile);
        wher.h := 90;
        wher.v := 100;
        DialogueDeactivate;
        SFGetFile(wher, '', NIL, 1, typelistptr, NIL, @reply);
        ReleaseResource(pointer(NameHdl));
        WITH reply DO
           IF good THEN
              BEGIN
              MakeAWindow(fName, TRUE);
              IF NOT ReadFile(vRefNum, fName) THEN
                 BEGIN
     { if nothing was read, then dispose of the window, TEdata, etc, but then again,
       you can't have everything in an example program that you would like.}
                 END
              END
     END;

{---------------------------------------------------------------------------}

   PROCEDURE OpenAWindow;

     VAR
        s: Str255;
        untitled: StringHandle;

     BEGIN
        {this creates a new window that is untitled and empty.}
        IF debug THEN DebugInProc(proc, 'OpenAWindow', @OpenAWindow);
        {see if enough mem exists to open a window}
        MakeNumString(windownum, s);
        windownum := windownum+1;
        untitled := GetString(256);
        Hlock(pointer(untitled));
        MakeAWindow(Concat(untitled^^, s), FALSE);
        Hunlock(pointer(untitled))
     END;

   {SS WRITFILE}
{---------------------------------------------------------------------------}

   FUNCTION WriteFile(VrefNo: INTEGER; fName: Str255): Boolean;

     VAR
        refNo, io: INTEGER;
        txtlength: LongInt;
        specialhdl: Charshandle;

        errin: Str255;

{---------------------------------------------------------------------------}

     PROCEDURE DiskWErr(io: INTEGER);

        VAR
           str: Str255;
           writetoHdl, savedHdl: StringHandle;
           dummy: INTEGER;

        BEGIN
  {this is just about the same as DiskRErr (read).  A good thing to add here would
   be a separate error message for disk full occurances.}
           IF debug THEN
```

```
            BEGIN
            DebugInProc(proc, 'DiskWErr',@DiskWErr);
            Writeln(debugger,errin,' err = ',io,lf)
            END;
         {read resource for writeto}
         writetoHdl := GetString(268);
         {read resource for saved}
         savedHdl := GetString(270);
         Hlock(pointer(writetoHdl));
         Hlock(pointer(savedHdl));
         MakeNumString(io,str);
         Paramtext(writetoHdl^^,fName,savedHdl^^,str);
         SetCursor(arrow);
         dummy := StopAlert(256,NIL);
         Hunlock(pointer(writetoHdl));
         Hunlock(pointer(savedHdl));
         EXIT(WriteFile)
      END;

    BEGIN
{this isn't very different from read file.  The only complication is finding out
 if the file exists.  If it doesn't, create it.  Also, assign the information that
 the finder needs to properly associate it with this application. One particularly
 bad thing here:  the volume reference number is not associated with the document.
 This means I do not know enough to write a file on the same disk from which it was
 read.  Oh well, you'll know better.}
      IF debug THEN DebugInProc(proc, 'WriteFile',@WriteFile);
      SetCursor(watchHdl^^);
      WriteFile := FALSE;
      io := FSOpen(fName,VrefNo,refNo);
      {$IFC BUG = 1}
      errin := 'FSOpen'; {once again, these only benefit the external
                          debugger.}
      {$ENDC}
      IF debug THEN Writeln(debugger,'file RefNum =',refNo,lf);
      IF io= {file not found Err} -43 THEN
         BEGIN
         io := Create(fName,VrefNo);
         {$IFC BUG = 1}
         errin := 'Create';
         {$ENDC}
         IF io<>0 THEN DiskWErr(io);
         io := SetFInfo(fName,VrefNo,txtfile);
         {$IFC BUG = 1}
         errin := 'SetFInfo';
         {$ENDC}
         IF io<>0 THEN DiskWErr(io);
         io := FSOpen(fName,VrefNo,refNo);
         {$IFC BUG = 1}
         errin := 'FSOpen';
         {$ENDC}
         IF debug THEN Writeln(debugger,'file RefNum = ',refNo,lf);
         IF io<>0 THEN DiskWErr(io)
         END {Create}
      ELSE IF io<>0 THEN DiskWErr(io);
      WITH hTE^^ DO
         BEGIN
         txtlength := ord4(teLength);
         Hlock(htext);
         io := FSWrite(refNo,txtlength,htext^);
         Hunlock(htext)
         END;
      IF debug THEN Write(debugger,'.');
      {$IFC BUG = 1}
      errin := 'FSWrite';
      {$ENDC}
      IF io<>0 THEN DiskWErr(io);
      io := SetEOF(refNo,txtlength);
```

```
        IF debug THEN Write(debugger,'.');
        {SIFC BUG = 1}
        errin := 'SetEOF';
        {SENDC}
        IF io<>0 THEN DiskWErr(io);
        io := FSClose(refNo);
        IF debug THEN Write(debugger,'.');
        {SIFC BUG = 1}
        errin := 'FSClose';
        {SENDC}
        IF io<>0 THEN DiskWErr(io);
        io := FlushVol(NIL,VrefNo); {this is important; without it, if the
                                     program died (not possible as a result of a
                                     programming mistake, of course), the
                                     directory information on the disk would not
                                     be accurate.}
        IF debug THEN Write(debugger,'.');
        {SIFC BUG = 1}
        errin := 'FlushVol';
        {SENDC}
        IF io<>0 THEN DiskWErr(io);
        IF NOT WindowData^^.titled THEN SetWTitle(MyWindow,fName);
        WindowData^^.titled := TRUE;
        WindowData^^.changed := FALSE;
        WriteFile := TRUE {everything is OK.}
     END;

{------------------------------------------------------------------------------}

  FUNCTION MyPutFile(Filename: Str255): Boolean;

     VAR
        reply: SFReply;
        wher: Point;
        NameHdl: StringHandle;
        temprect: Rect;
        tempport: GrafPtr;

     BEGIN
{The user can select the name of the file that they wish to save the document with.}
        IF debug THEN DebugInProc(func,'MyPutFile',@MyPutFile);
        MyPutFile := FALSE;
        NameHdl := GetString(257);
        wher.h := 100;
        wher.v := 100;
        Hlock(pointer(NameHdl));
        DialogueDeactivate;
        SFPutFile(wher,NameHdl^^,Filename,NIL,@reply);
        Hunlock(pointer(NameHdl));
        WITH reply DO
           BEGIN
           IF debug THEN Writeln(debugger,'reply.good = ',good,lf);
           IF good THEN MyPutFile := WriteFile(vRefNum,fName)
           END;
        ReleaseResource(pointer(NameHdl));
        IF debug THEN Writeln(debugger,'release reserror = ',reserror,lf)
     END;

{------------------------------------------------------------------------------}

  PROCEDURE CloseAWindow;

     VAR
        itemhit: INTEGER;
        DBoxPtr: DialogPtr;
        str,str1: Str255;
        Goodwrite: Boolean;
        temprect: Rect;
```

```
        NameHdl: Handle;
        NamePtr: ^Str255;
        typ: INTEGER;
        itemhdl: Handle;
        box: Rect;

     BEGIN
{All sorts of windows can be closed through this single routine, which is accessed
 by the user through the go-away box on the window, or the Close item in the File
 menu.}
        IF debug THEN DebugInProc(proc, 'CloseAWindow',@CloseAWindow);
        MyPeek := pointer(frontwindow);
        CASE MyPeek^.windowkind OF
           8:
              BEGIN
              GetWTitle(MyWindow,str);
              itemhit := 0;
              IF WindowData^^.changed THEN {give the user the chance to save his
                                              data before you throw it away.}
                 BEGIN
                 DialogueDeactivate;
                 IF doneFlag THEN
                    BEGIN
                    NameHdl := Getresource('STR ',266);
                    Hlock(NameHdl);
                    NamePtr := pointer(NameHdl^);
                    str1 := NamePtr^;
                    Hunlock(NameHdl);
                    IF debug THEN Writeln(debugger,'err = ',reserror,lf);
                    END
                 ELSE
                    str1 := '';
                 Paramtext(str,str1,'','');
                 DBoxPtr := GetNewDialog(256,NIL,pointer(-1));
                 REPEAT
                    ModalDialog(NIL,itemhit) {this could have been an alert.}
                 UNTIL itemhit IN [OK {Yes} ,Cancel,3 {No} ];
                 DisposDialog(DBoxPtr)
                 END;
              IF debug THEN Writeln(debugger,'itemhit = ',itemhit,lf);
              Goodwrite := FALSE;
              IF NOT WindowData^^.titled THEN str := '';
              IF itemhit=OK {save} THEN Goodwrite := MyPutFile(str);
              IF Goodwrite OR (itemhit IN [0,3] {discard} ) THEN
                 BEGIN
                 TEDispose(hTE);
                 hTE := NIL;
                 DisposHandle(pointer(WindowData));
                 IF debug THEN
                    Writeln(debugger,'dispose WindowData; memerr = ',memerror,
                         lf);
                 KillControls(MyWindow); {do I need this? Why am I asking you?}
                 DisposeWindow(MyWindow)
                 END;
              IF itemhit=Cancel THEN doneFlag := FALSE
              END;
           9: ToggleScrap;
           {$IFC BUG > -1}
           10: ToggleFree;
           {$ENDC}
           OTHERWISE
              CloseDeskAcc(MyPeek^.windowkind) {can't be anything else}
        END {Case}
     END;

   {$S AboutMyPgm}
{--------------------------------------------------------------------------}
```

```
  PROCEDURE AboutMyEditor;

     VAR
        str1hdl, str2hdl: StringHandle;
        MyWindow: WindowPtr;
        width, height, counter: INTEGER;
        newcount: LongInt;
        quit: Boolean;
        txtinfo: fontinfo;
        temprect, trect1: Rect;
        tempbits: bitmap;
        sz: size;

     BEGIN
{this bit of fluff shows a totally wrong method of telling the user something about
 my program, but it was fun to do.}
        IF debug THEN DebugInProc(proc, 'AboutMyEditor', @AboutMyEditor);
        DialogueDeactivate;
        str1hdl := GetString(259);
        IF debug THEN Writeln(debugger, 'err = ', reserror, 1f);
        str2hdl := GetString(260);
        IF debug THEN Writeln(debugger, 'err = ', reserror, 1f);
        Hlock(pointer(str1hdl));
        Hlock(pointer(str2hdl));
        MyWindow := GetNewWindow(256, NIL, pointer(-1));
        SetPort(MyWindow);
        counter := 1;
        width := MyWindow^.portrect.right-MyWindow^.portrect.left;
        height := MyWindow^.portrect.bottom-MyWindow^.portrect.top;
        TextFont(2);
        TextMode(srcCopy);
        quit := FALSE;
        REPEAT
           SystemTask;
           newcount := tickcount+6;
           TextSize(counter);
           GetFontInfo(txtinfo);
           WITH txtinfo DO
              BEGIN
              MoveTo((width-StringWidth(str1hdl^^)) DIV 2, height DIV
                     2-descent-leading);
              DrawString(str1hdl^^);
              MoveTo((width-StringWidth(str2hdl^^)) DIV 2, height DIV 2+ascent);
              DrawString(str2hdl^^)
              END;
           IF EventAvail(10, myEvent) THEN quit := TRUE;
           counter := counter+1;
           WHILE newcount>tickcount DO;
        UNTIL quit OR (counter=12);
        newcount := tickcount+300; {5 seconds}
        WHILE NOT quit AND (tickcount<newcount) DO
           BEGIN
           SystemTask;
           IF EventAvail(10, myEvent) THEN quit := TRUE;
           END;
        temprect := MyWindow^.portrect;
        WITH txtinfo DO
           BEGIN
           temprect.top := height DIV 2-ascent-descent-leading;
           temprect.bottom := height DIV 2+ascent+descent
           END;
        trect1 := temprect;
        OffsetRect(trect1, 0, -trect1.top);
        tempbits.rowbytes := (width+7) DIV 8;
        tempbits.bounds := trect1;
        WITH txtinfo DO
           sz := ord4(tempbits.rowbytes*(ascent*2+descent*2+leading));
        tempbits.baseaddr := pointer(NewPtr(sz));
```

```
        IF debug THEN Writeln(debugger, 'err = ',memerror,lf);
        CopyBits(MyWindow^.portbits,tempbits,temprect,trectl,srcCopy,NIL);
        insetrect(trectl,8,0);
        temprect.top := temprect.top-2;
        temprect.bottom := temprect.bottom+2;
        WHILE NOT quit AND (trectl.right>width DIV 2) DO
           BEGIN
           SystemTask; {the clock still ticks!}
           CopyBits(tempbits,MyWindow^.portbits,trectl,temprect,srcCopy,NIL);
           IF temprect.top>MyWindow^.portrect.top THEN
              BEGIN
              insetrect(trectl,8,0);
              insetrect(temprect,0,-2)
              END
           ELSE
              insetrect(trectl,8,2);
           IF EventAvail(10,myEvent) THEN quit := TRUE
           END;
        Hunlock(pointer(str1hdl));
        Hunlock(pointer(str2hdl));
        ReleaseResource(pointer(str1hdl));
        ReleaseResource(pointer(str2hdl));
        DisposPtr(pointer(tempbits.baseaddr));
        IF debug THEN Writeln(debugger, 'err = ',memerror,lf);
        DisposeWindow(MyWindow)
     END;

   {SS MyPrint }
{-------------------------------------------------------------------------------}

   PROCEDURE MyPrint(finderfirst: Boolean; Filename: Str255);

      CONST
         bottommargin = 20; {amount of space on the margins of the page in pixels}
         leftmargin = 30;
         rightmargin = 10;

      VAR
         tempport: GrafPtr;
         MyPPort: TPPrPort;
         txt: Ptr;
         pglen,MyLngth,start,finish,counter,loop: INTEGER;
         temprect,tmprect2,pagerect: Rect;
         status: TPrStatus;
         userOK: Boolean;
         s: string[1];
         str: Str255;
         dlogptr: DialogPtr;

      BEGIN
{For heavyweight programmers only. All modes of printing are handled by Macprint.  The
 only things you have to do are:
   image each page, using QuickDraw (or something that uses QuickDraw);
   Do it once for the number of copies the user specified in draft mode only.

   You do not have to worry with:
     copies in normal or high res.
     which pages the user chose to print.
     tall, wide, etc.

  Remember, these Page Setup dialog is printer specific.  It will not always be the
  same, so don't write any code around it.

  The reason this program is heavily segmented is that printing normal or high-res
  on line takes gobs of memory (in this example, up to 25K.)  You may minimize the
  by omitting 1 line below and creating a spooled file instead.

  One more thing.  The dialog shown here (press command-period to stop) is not the
```

```
thing to do.  You may choose to either:
  run your program in the background.  This is not necessarily a hard thing to do.
  put up a dialog with a button so the user may press the button to stop.  Then
  the printing idle proc only needs to monitor that button.

Printing is not re-entrant.  If your main program loop is to be the
print idle proc, disable the Page Setup and Print items in the File menu.}

       IF debug THEN DebugInProc(proc, 'MyPrint',@MyPrint);
       printflag := FALSE;
       IF debug THEN
          Writeln(debugger, 'finderprint =',finderprint, '; finderfirst =',
                finderfirst,lf);
       IF NOT finderprint THEN DialogueDeactivate;
       userOK := TRUE;
       IF finderfirst THEN
          BEGIN
          SetCursor(arrow);
          userOK := PrJobDialog(printhdl)
          END;
       IF userOK THEN
          BEGIN
          SetCursor(watchHdl^^);
          s := 'x';
          s[1] := chr(cmdsymbol); {this is terrible, terrible, terrible. Don't
                                  do it.}
          Paramtext(Filename,s,'','');
          dlogptr := GetNewDialog(257,NIL,pointer(-1));
          DrawDialog(dlogptr);
          {for now, approximate a full page}
          GetPort(tempport);
          MyPPort := PrOpenDoc(printhdl,NIL,NIL);
          SetPort(pointer(MyPPort));
          TextFont(2);
          WITH printhdl^^.prinfo DO
             BEGIN
             HLock(pointer(printhdl));
             pagerect := rpage;
             pagerect.left := pagerect.left+leftmargin;
             pagerect.right := pagerect.right-rightmargin;
             pagerect.bottom := pagerect.bottom-bottommargin-(pagerect.bottom-
                               bottommargin) MOD hTE^^.lineHeight {get rid of
                                  partial line} ;
             hTE^^.destrect := pagerect;
             TECalText(hTE);
             WITH hTE^^ DO
                BEGIN
                HLock(pointer(hTE));
                HLock(htext);
                txt := htext^;
                temprect := destrect;
                tmprect2 := viewrect;
                pglen := (rpage.bottom-rpage.top-bottommargin) DIV lineHeight;
                finish := nLines;
                IF debug THEN
                   Writeln(debugger, 'BJDocLoop = ',printhdl^^.prjob.BJDocLoop,
                         lf);
                IF printhdl^^.prjob.BJDocLoop=BSpoolLoop THEN
                   loop := 1
                ELSE
                   loop := printhdl^^.prjob.iCopies;
                FOR counter := 1 TO loop DO
                   BEGIN
                   start := 0;
                   WHILE start<finish DO
                      BEGIN
                      IF finish-start>pglen THEN
                         MyLngth := linestarts[start+pglen]-linestarts[start]
```

```
                    ELSE
                       MyLngth := teLength-linestarts[start];
                    IF debug THEN
                       BEGIN
                       Writeln(debugger, 'MyLngth = ',MyLngth: 5,'; start = ',
                               start: 5,'; pglen = ',pglen: 5,lf);
                       Writeln(debugger, 'finish = ',finish: 5,'; teLength = ',
                               teLength: 5,'; ord4(txt) = ',ord4(txt),lf)
                       END;
                    PrOpenPage(MyPPort,NIL);
                    TextBox(txt,MyLngth,pagerect,0);
                    PrClosePage(MyPPort);
                    txt := pointer(ord4(txt)+MyLngth);
                    start := start+pglen
                    END {While start < finish}
                 END {For counter := 1 to loop}
              END {with hTE}
           END; {with PrintHdl^^.prjob}
        PrCloseDoc(MyPPort);
        Hunlock(pointer(hTE));
        Hunlock(hTE^^.htext);
        Hunlock(pointer(printhdl));
        IF printhdl^^.prjob.BJDocLoop=BSpoolLoop THEN
           PRPicFile(printhdl,NIL,NIL,NIL,status); {omit this for spooled
                                                     files.}
        SetPort(tempport);
        hTE^^.destrect := temprect;
        hTE^^.viewrect := tmprect2;
        TECalText(hTE);
        DisposDialog(dlogptr);
        IF NOT finderprint THEN SetCursor(arrow)
        END
    END;

  {SS EditMenu}
{-----------------------------------------------------------------------------}

  PROCEDURE EditMain(theItem: INTEGER; commandkey: Boolean);

    CONST
      undo = 1;
      cut = 3;
      kopy = 4; {copy is a Pascal string function}
      paste = 5;
      clear = 6;
      selectAll = 7;
      clipboard = 9;

    VAR
      DeskAccUp,dummy: Boolean;
      DScrap: PScrapstuff;
      off: LongInt;
      ticks: LongInt;
      tempport: GrafPtr;
      box: Rect;
      itemhdl,hdl: Handle;
      typ,io,tempstart,tempend: INTEGER;
      tempptr: Ptr;
      TextScrap: Handle;
      TextLength: INTEGER;
      Ptr2ScrapLength: ^INTEGER;

    BEGIN
{Since the Edit menu does so much, it has been broken up into a separate procedure.
 It does not yet support undo, but does support Cutting, Copying and Pasting between
 the Desk Scrap and the TextEdit Scrap.}
      DeskAccUp := FALSE;
      IF theItem<selectAll THEN DeskAccUp := SystemEdit(theItem-1);
```

```
IF ((theItem IN [undo,cut,kopy]) OR DeskAccUp) AND (scrapwind<>NIL) THEN
   BEGIN {invalidate clipboard}
   GetPort(tempport);
   SetPort(scrapwind);
   InvalRect(scrapwind^.portrect);
   SetPort(tempport)
   END;
IF theItem IN [cut,kopy] THEN
   BEGIN
   tempend := hTE^^.selend;
   tempstart := hTE^^.selstart
   END;
IF (theItem>clear) OR NOT DeskAccUp THEN
   BEGIN
   IF debug THEN Writeln(debugger,'not system edit',lf);
   { Delay so menu title will stay lit a little only if Command key }
   { equivalent was typed. }
   IF commandkey THEN
      BEGIN
      ticks := tickcount+10;
      REPEAT
      UNTIL ticks<=tickcount
      END;
   {** see if enough memory exists for move}
   CASE theItem OF
      undo: ; { no Undo/Z in this example }
      cut: TECut(hTE); { Cut/X }
      kopy: TECopy(hTE); { Copy/C }
      paste:
         BEGIN { Paste/V }
         DScrap := InfoScrap;
         IF DScrap^.scrapState<>laststate THEN
            BEGIN
            laststate := DScrap^.scrapState;
            hdl := NewHandle(0);
            io := GetScrap(hdl,'TEXT',off);
            IF debug THEN Writeln(debugger,'io = ',io);
            IF io>0 THEN
               BEGIN
               TextScrap := pointer(GlobalValue(TEScrpHandle));
               SetHandleSize(TextScrap,io);
               Ptr2ScrapLength := pointer(GlobalAddr(TEScrpLength));
               Ptr2ScrapLength^ := io;
               Hlock(hdl);
               Hlock(TextScrap);
               BlockMove(hdl^,TextScrap^,io);
               Hunlock(hdl);
               Hunlock(TextScrap)
               END;
            DisposHandle(hdl)
            END;
         TEPaste(hTE);
         END;
      clear: TEDelete(hTE); { Clear }
      selectAll: TESetSelect(0,65535,hTE); { Select All/A }
      clipboard:
         ToggleScrap { Show, Hide Clipboard }
   END; { of item case }
   IF theItem IN [cut,kopy] THEN
      BEGIN
      io := ZeroScrap;
      IF debug THEN Writeln(debugger,'zero scrap err =',io,lf);
      TextScrap := pointer(GlobalValue(TEScrpHandle));
      TextLength := GlobalValue(TEScrpLength);
      Hlock(TextScrap);
      io := PutScrap(TextLength,'TEXT',TextScrap^);
      IF debug THEN Writeln(debugger,'put scrap err =',io,lf);
      Hunlock(TextScrap)
```

```
            END;
         IF theItem IN [cut,clear,paste] THEN WindowData^^.changed := TRUE;
         IF (theItem IN [cut..clear]) THEN ScrollText(TRUE)
         END {not systemedit}
      END; { of editMain }

   {$S Command }
{-----------------------------------------------------------------------------}

   PROCEDURE MyDisable;

      VAR
         counter: INTEGER;
         DScrap: PScrapstuff;
         temppeek: WindowPeek;
         stycount: styleitem;

{-----------------------------------------------------------------------------}

      PROCEDURE KillFE(fileitems,edititems: edset);

         VAR
            counter: INTEGER;

         BEGIN
{This guy disables the items in the File and Edit menus.  This approach has a real
 disadvantage:  If an entire menu should be disabled at some given time, there is
 no convenient way to do a DrawMenuBar here to disable the item in the bar itself.}
            IF debug THEN
               BEGIN
               DebugInProc(proc,'KillFE',@KillFE);
               Write(debugger,'file:');
               FOR counter := 1 TO 9 DO
                  IF counter IN fileitems THEN Write(debugger,counter: 2,',');
               Write(debugger,';  edit:');
               FOR counter := 1 TO 9 DO
                  IF counter IN edititems THEN Write(debugger,counter: 2,',');
               Writeln(debugger,lf)
               END;
            FOR counter := 1 TO 9 DO
               BEGIN
               IF counter IN fileitems THEN
                  DisableItem(myMenus[FileMenu],counter);
               IF counter IN edititems THEN
                  DisableItem(myMenus[EditMenu],counter);
               END
         END;

      BEGIN
{This part goes through all of the applicable elements of the frontmost window, if any
 and from that decides what operations are allowable at this time.}
         IF debug THEN DebugInProc(proc,'MyDisable',@MyDisable);
         FOR counter := 1 TO 9 DO
            BEGIN
            EnableItem(myMenus[FileMenu],counter);
            IF counter IN [1,3..7,9] THEN EnableItem(myMenus[EditMenu],counter)
            END;
         IF frontwindow=NIL THEN
            KillFE([3..8],[1..7])
         ELSE
            BEGIN
            MyPeek := pointer(frontwindow);
            CASE MyPeek^.windowkind OF
               8:
                  BEGIN
                  KillFE([],[1]);
                  IF NOT WindowData^^.titled THEN KillFE([4,6],[]);
                  IF NOT WindowData^^.changed THEN KillFE([4,6],[]);
```

```
            IF hTE^^.teLength=0 THEN KillFE([4,5,7,8],[6,7]);
            IF hTE^^.selstart=hTE^^.selend THEN KillFE([],[3,4,6]);
            DScrap := InfoScrap;
            IF DScrap^.scrapSize=0 THEN KillFE([],[5]);
            END;
         9,10: KillFE([4..8],[1,3..7]);
         OTHERWISE
            KillFE([4..8],[7]) {system window}
      END {Case}
      END
   END;

{--------------------------------------------------------------------------}

 PROCEDURE DoCommand(commandkey: Boolean);

   VAR
      name,s,str: Str255;
      bstr: string[5];
      dummy: size;
      err: Boolean;
      num,refnum,theMenu,theItem: INTEGER;
      temppeek: WindowPeek;
      mresult,ticks: LongInt;
      dipeek: DialogPeek;
      box: Rect;
      itemhdl: Handle;
      typ: INTEGER;

   BEGIN
      {This handles the actions that are initiated through the Menu Manager}
      IF debug THEN DebugInProc(proc,'DoCommand',@DoCommand);
      MyDisable;
      IF commandkey THEN
         mresult := MenuKey(theChar)
      ELSE
         mresult := MenuSelect(myEvent.where);
      theMenu := HiWord(mresult); theItem := LoWord(mresult);
      CASE theMenu OF
         1: {enough memory to allow desk accessory to open?}
            BEGIN
            {unload all segments}
            IF theItem=1 THEN
               BEGIN
               AboutMyEditor;
               UnloadSeg(@AboutMyEditor)
               END
            ELSE
               GetItem(myMenus[appleMenu],theItem,name);
            refnum := OpenDeskAcc(name)
            END;
         2:
            BEGIN
            IF frontwindow<>NIL THEN
               IF MyPeek^.windowkind=8 THEN
                  IF WindowData^^.titled THEN
                     GetWTitle(frontwindow,str)
                  ELSE
                     str := '';
            CASE theItem OF
               1: OpenAWindow; { New }
               2: MyGetFile; { Open  }
               3: CloseAWindow; { Close }
               4: err := WriteFile(0,str); { Save }
               5: err := MyPutFile(str); { Save As  }
               6: IF CautionAlert(257,NIL)=OK THEN err := ReadFile(0,str); {
                  Revert to Saved }
               7:
```

```
                  IF PrStlDialog(printhdl) { Page Setup  }
                     THEN;
                {eventually, store info in document resource fork}
                8: printflag := TRUE; { Print }
                9: doneFlag := TRUE; { Quit }
             END;
             UnloadSeg(@ReadFile);
             UnloadSeg(@WriteFile);
             UnloadSeg(@MyPrint)
             END;
          3: EditMain(theItem, commandkey);

          {$IFC BUG > -1}
          100:
             CASE theItem OF
                1: ToggleFree;
                2: dummy := MaxMem(dummy);
                {$IFC BUG = 1}
                3:
                   BEGIN
                   debug := NOT debug;
                   CheckItem(myMenus[DebugMenu], 3, debug)
                   END
                   {$ENDC}
             END { of debug }
                {$ENDC}

       END; { of menu case }
       HiliteMenu(0)
    END; { of DoCommand }

{----------------------------------------------------------------------------------}

  PROCEDURE DrawWindow;

    VAR
       tempport: GrafPtr;
       tempscrap: Handle;
       scraplength, off: LongInt;
       temprect, rectToErase: Rect;
       str: Str255;
       temppeek: WindowPeek;
       whichwindow: WindowPtr;
       temphTE: TEHandle;
       tempdata: MyDataHandle;

    BEGIN
{ Draws the content region of the given window, after erasing whatever
  was there before. }
       IF debug THEN DebugInProc(proc, 'DrawWindow', @DrawWindow);
       whichwindow := pointer(myEvent.message);
       IF whichwindow<>NIL THEN {** why is this here **}
          BEGIN
          BeginUpdate(whichwindow);
          GetPort(tempport);
          SetPort(whichwindow);
          temppeek := pointer(whichwindow);
          CASE temppeek^.windowkind OF
             8:
                BEGIN
                temprect := whichwindow^.portrect;
                tempdata := pointer(GetWRefCon(whichwindow));
                temphTE := tempdata^^.TERecord;
                IF temppeek^.hilited THEN temprect.top := temprect.bottom-15;
                temprect.left := temprect.right-15;
                Cliprect(temprect);
                DrawGrowIcon(whichwindow);
                Cliprect(whichwindow^.portrect);
```

```
                DrawControls(whichwindow);
                {this only erases the window past the end of text, if any}
                WITH temphTE^^ DO
                   IF nLines-topline<(viewrect.bottom-viewrect.top+
                      lineHeight) DIV lineHeight THEN
                      BEGIN
                      rectToErase := viewrect;
                      rectToErase.top := (nLines-topline)*lineHeight;
                      EraseRect(rectToErase)
                      END;
                TEUpdate(whichwindow^.visRgn^^.rgnBBox,temphTE)
                END;
             9:
                BEGIN
                tempscrap := NewHandle(0);
                scraplength := GetScrap(tempscrap,'TEXT',off);
                EraseRect(whichwindow^.portrect);
                temprect := whichwindow^.portrect;
                temprect.left := temprect.left+4;
                temprect.right := temprect.right-15;
                IF scraplength>0 THEN
                   BEGIN
                   Hlock(tempscrap);
                   TextBox(tempscrap^,scraplength,temprect,0);
                   Hunlock(tempscrap)
                   END;
                DisposHandle(tempscrap);
                temprect := whichwindow^.portrect;
                temprect.left := temprect.right-15;
                Cliprect(temprect);
                DrawGrowIcon(whichwindow);
                Cliprect(whichwindow^.portrect)
                END;
             {$IFC BUG > -1}
             10:
                BEGIN
                EraseRect(whichwindow^.portrect);
                MoveTo(5,12);
                MakeNumString(FreeMem,str);
                DrawString(str)
                END;
             {$ENDC}
          END; {Case}
          SetPort(tempport);
          EndUpdate(whichwindow)
          END
    END; { of DrawWindow }

  {$S CONTROL}
{---------------------------------------------------------------------------}

  PROCEDURE ScrollBits;

    VAR
       oldvert: INTEGER;

    BEGIN
{if the visible information has changed as a because of the scrollbar,
 scroll the window here.}
       IF debug THEN DebugInProc(proc,'ScrollBits',@ScrollBits);
       oldvert := topline;
       topline := GetCtlValue(vScroll);
       TEScroll(0,(oldvert-topline)*hTE^^.lineHeight,hTE)
    END;

{---------------------------------------------------------------------------}

  PROCEDURE ScrollUp(theControl: ControlHandle; partCode: INTEGER);
```

```
    BEGIN
       {This function is called by TrackControl in the Up button}
       IF debug THEN DebugInProc(proc, 'ScrollUp',@ScrollUp);
       IF partCode=inUpButton THEN
          BEGIN
          SetCtlValue(theControl,GetCtlValue(theControl)-1); {VScroll}
          ScrollBits
          END
    END;

{---------------------------------------------------------------------------}

  PROCEDURE ScrollDown(theControl: ControlHandle; partCode: INTEGER);

    BEGIN
       {This function is called by TrackControl in the Down button}
       IF debug THEN DebugInProc(proc, 'ScrollDown',@ScrollDown);
       IF partCode=inDownButton THEN
          BEGIN
          SetCtlValue(theControl,GetCtlValue(theControl)+1); {VScroll}
          ScrollBits
          END
    END;

{---------------------------------------------------------------------------}

  PROCEDURE PageScroll(which: INTEGER);

    VAR
       myPt: Point;
       amount: INTEGER;

    BEGIN
      {This function is called by TrackControl in the Grey part of the scrollbar}
       IF debug THEN DebugInProc(proc, 'PageScroll',@PageScroll);
       IF which=InPageUp THEN
          amount := -1
       ELSE
          amount := 1;
       REPEAT
          GetMouse(myPt);
          IF TestControl(vScroll,myPt)=which THEN
             BEGIN
             WITH hTE^^.viewrect DO
                SetCtlValue(vScroll,GetCtlValue(vScroll)+amount*(bottom-
                        top) DIV hTE^^.lineHeight);
             ScrollBits
             END
       UNTIL NOT StillDown;
    END;

{---------------------------------------------------------------------------}

  PROCEDURE MyControls;

    VAR
       t,code,whichpart: INTEGER;
       AControl: ControlHandle;

    BEGIN {controls}
       {This routine handles the scrollbar}
       IF debug THEN DebugInProc(proc,'MyControls',@MyControls);
       whichpart := FindControl(myPoint,MyWindow,AControl);
       IF debug THEN Writeln(debugger,'whichpart = ',whichpart,lf);
       IF debug THEN Writeln(debugger,'ord( AControl = ',ord4(AControl),lf);
       {adjust scrollbar range}
       IF AControl<>NIL THEN
```

```
          BEGIN
          vScroll := AControl;
          CASE whichpart OF
             inUpButton: t := TrackControl(vScroll, myPoint, @ScrollUp);
             inDownButton: t := TrackControl(vScroll, myPoint, @ScrollDown);
             InPageUp: PageScroll(whichpart);
             inPageDown: PageScroll(whichpart);
             inThumb:
                BEGIN
                t := TrackControl(vScroll, myPoint, NIL);
                ScrollBits
                END
          END {Case MyControl}
          END {AControl <> NIL}
    END; {controls}

  {$S Initial }
{------------------------------------------------------------------------------------}

  PROCEDURE SetUp;

    VAR
       counter, vRefNum, numfiles: INTEGER;
       DScrap: PScrapstuff;
       hdl, hAppparms: Handle;
       off: LongInt;
       apName: Str255;
       NameHdl: Handle;
       strhdl: StringHandle;
       dummyrect: Rect;
       tempptr: pAppParms;
       dummy: Boolean;

    BEGIN
{Initialization for a variety of things is done here.  This code is 'discarded'
 after it is executed by an UnLoadSeg.}
       {$IFC BUG = 1}
       debug := TRUE; {if you want debugging on as soon as the program starts,
                       set it here}
       lf := chr(10);
       Rewrite(debugger, '.BOUT'); {the serial port not used for downloading from
                                    Lisa}
       {$ENDC}
       IF debug THEN
          BEGIN
          Writeln(debugger, lf, lf);
          DebugInProc(proc, 'SetUp', @SetUp)
          END;
       InitGraf(@thePort);
       InitWindows;
       InitFonts;
       FlushEvents(everyEvent, 0);
       TEInit;
       InitDialogs(NIL);
       NameHdl := NewHandle(1000000); {force MemMgr to allocate all 'grow' to
                                       app.}
       DisposHandle(NameHdl);
       printhdl := pointer(NewHandle(120));
       PrOpen;
       PrintDefault(printhdl);
       getAppParms(apName, vRefNum, hAppparms);
      {** sometime, get file info for apName, to use folder info as appropriate}
       tempptr := pointer(hAppparms^);
       iBeamHdl := pointer(GetCursor(1));
       watchHdl := pointer(GetCursor(4));
       numfiles := tempptr^.count;
       IF debug THEN Writeln(debugger, 'numfiles=', numfiles, lf);
       finderprint := (tempptr^.message=1);
```

```
      IF finderprint THEN
         BEGIN
  {put something meaningful on menu bar; use TextBox to say the ap name perhaps?}
         Hlock(hApppararms);
         FOR counter := 1 TO numfiles DO
            WITH tempptr^ DO
               BEGIN
               IF fTYPE='TEXT' THEN
                  BEGIN
                  SetRect(dummyrect,0,0,100,100);
                  hTE := TENew(dummyrect,dummyrect);
                  dummy := ReadFile(vRefNum,fName); {assume that page setup is
                                                     read in as well}
                  UnloadSeg(@ReadFile);
                  IF counter=1 THEN
                     MyPrint(TRUE,fName)
                  ELSE
                     MyPrint(FALSE,fName);
                  TEDispose(hTE); {dispose of text edit stuff}
                  tempptr := pointer(ord4(tempptr)+length(fName)+
                                    10-length(fName) MOD 2)
                  END
                  {ELSE clear the proper bytes in the appParms handle?}
               END;
         Hunlock(hApppararms);
         hTE := NIL;
         doneFlag := TRUE;
         END
      ELSE
         BEGIN
         InitMenus; { initialize Menu Manager }
         myMenus[appleMenu] := GetMenu(appleMenu);
         myMenus[appleMenu]^^.menudata[1] := chr(Applesymbol);
         AddResMenu(myMenus[1],'DRVR'); { desk accessories }
         FOR counter := FileMenu TO EditMenu DO
            myMenus[counter] := GetMenu(counter);
         {$IFC BUG > -1}
         myMenus[DebugMenu] := GetMenu(100); { temporary debug menu }
         {$ENDC}
         {$IFC BUG = 1}
         extdebughdl := GetString(261);
         Hlock(pointer(extdebughdl));
         AppendMenu(myMenus[DebugMenu],extdebughdl^^);
         Hunlock(pointer(extdebughdl));
         ReleaseResource(pointer(extdebughdl));
         CheckItem(myMenus[DebugMenu],3,debug);
         {$ENDC}
         FOR counter := 1 TO lastMenu DO InsertMenu(myMenus[counter],0);
         DrawMenuBar;
         dragRect := screenbits.bounds;
         dragRect.top := dragRect.top+20; {leave room for menu bar}
         growRect := dragRect;
         insetrect(dragRect,4,4); {leave some of dragged rectangle on screen}
         growRect.left := {replace this with the max font width + constant} 80;
         growRect.top := 80 {18 + 16*3 + slop?} ;
         doneFlag := FALSE;
         printflag := FALSE;
         windownum := 1;
         windowpos := 0;
         typelistptr := @MyFileTypes;
         typelistptr^[0] := 'TEXT';
         txtfile.fdType := 'TEXT';
         txtfile.fdCreator := 'CARY';
         SetPt(txtfile.fdLocation,0,0);
         txtfile.fdFlags := 0;
         txtfile.fdFldr := 0;
         laststate := 0; {eventually, init laststate to scrapstate - 1?}
         Hlock(hApppararms);
```

```
          FOR counter := 1 TO numfiles DO
             WITH tempptr^ DO
                BEGIN
                IF fTYPE='TEXT' THEN
                   BEGIN
                   MakeAWindow(fName,TRUE); {**could async open while this is
                                              going on}
                   IF counter<numfiles THEN DialogueDeactivate;
                   IF NOT ReadFile(vRefNum,fName) THEN
                      BEGIN
         {if nothing was read, then
          dispose of the window, TEdata, etc, depending on how far we got}
                      END;
                   tempptr := pointer(ord4(tempptr)+length(fName)+
                                      10-length(fName) MOD 2)
                   END
                END;
          Hunlock(hApppparms);
          IF frontwindow=NIL THEN
             BEGIN
             OpenAWindow;
             END;
          {if something 'TEXT' is in deskscrap then allow paste}
          DScrap := InfoScrap;
          laststate := DScrap^.scrapState;
          IF DScrap^.scrapSize>0 THEN laststate := laststate-1;
          {what about when scrapsize is too big?}
          scrapwind := NIL;
          {SIFC BUG > -1}
          FreeWind := NIL
          {SENDC}
          END
      END; { of SetUp}

   {SS          }
{-----------------------------------------------------------------------------}

   PROCEDURE MainEventLoop;

      VAR
         code: INTEGER; {the type of mousedown event}
         dummy: Boolean;
         str: Str255;

      BEGIN
{This event loop handles most of the communications between this program and events
taking place in the outside world.  This procedure could also be called as the printer
idle procedure so that the program appears to be doing background printing.}
         IF debug THEN DebugInProc(proc,'MainEventLoop',@MainEventLoop);
         REPEAT
            CursorAdjust;
            SystemTask;
            IF printflag THEN
               BEGIN {unload the world}
               UnloadSeg(@CursorAdjust);
               UnloadSeg(@ReadFile);
               UnloadSeg(@WriteFile);
               UnloadSeg(@AboutMyEditor);
               UnloadSeg(@MyDisable);
               UnloadSeg(@ScrollBits); {*** segmenting badly out of date}
               GetWTitle(MyWindow,str);
               MyPrint(TRUE,str)
               END;
            dummy := GetNextEvent(everyEvent,myEvent);
            CASE myEvent.what OF
               mouseDown:
                  BEGIN
                  code := FindWindow(myEvent.where,tempwindow);
```

```
          CASE code OF
            inMenuBar: DoCommand(FALSE);
            inSysWindow: InSystemWindow;
            inDrag: DragWindow(tempwindow,myEvent.where,dragRect);
            inGoAway:
              IF TrackGoAway(tempwindow,myEvent.where) THEN
                CloseAWindow;

            inGrow:
              IF MyPeek^.windowkind IN [8,9] THEN
                BEGIN
                GrowWnd;
                UnloadSeg(@GrowWnd)
                END;

            inContent:
              BEGIN
              IF tempwindow<>frontwindow THEN
                SelectWindow(tempwindow)
              ELSE IF hTE<>NIL THEN
                BEGIN
                myPoint := myEvent.where;
                GlobalToLocal(myPoint);
                IF PtInRect(myPoint,hTE^^.viewrect) THEN
                  BEGIN
                  IF debug THEN
                    Writeln(debugger,'point in HTE viewrect',lf);
                  IF (BitAnd(myEvent.modifiers,512)<>0) { Shift key
                      pressed }
                    THEN
                    TEClick(myPoint,TRUE,hTE)
                  ELSE
                    TEClick(myPoint,FALSE,hTE)
                  END
                ELSE
                  MyControls
                END {hTE <> NIL}
              END {in Content}
          END { of code case }
          END; { of mouseDown }

        keyDown,autoKey:
          BEGIN
          theChar := chr(myEvent.message MOD 256);
          IF BitAnd(myEvent.modifiers,256)<>0 { Command key pressed }
            THEN
            DoCommand(TRUE)
          ELSE IF hTE<>NIL THEN
            BEGIN
            TEKey(theChar,hTE);
            WindowData^^.changed := TRUE;
            ScrollText(TRUE)
            END
          END; { of keyDown }

        activateEvt:
          BEGIN
          MyActivate;
          UnloadSeg(@MyActivate)
          END;

        updateEvt: DrawWindow;
        nullEvent: IF doneFlag AND (frontwindow<>NIL) THEN CloseAWindow
      END; { of event case }
    UNTIL doneFlag AND (frontwindow=NIL);
  END;
```

```
   BEGIN { main program }
{Please don't look at this program as the the last word in example programming, and
be very cautious about porting some portion of this program over to your own code.}
      SetUp;
      UnloadSeg(@SetUp);
      IF NOT finderprint THEN MainEventLoop;
      SetCursor(watchHdl^^);
      PrClose
   END.
```

```
{$X-}
PROGRAM Grow;

  { Grow -- Scroll bars and a resizable window added to Edit }
  {         by Cary Clark, Macintosh Technical Support       }

  USES {$U-}
     {$U Obj/QuickDraw   } QuickDraw,
     {$U Obj/OSIntf      } OSIntf,
     {$U Obj/ToolIntf     } ToolIntf;

  CONST
     lastMenu = 3; { number of menus }
     appleMenu = 1; { menu ID for desk accessory menu }
     fileMenu = 256; { menu ID for File menu }
     editMenu = 257; { menu ID for Edit menu }

  VAR
     myMenus: ARRAY [1..lastMenu] OF MenuHandle;
     growRect, dragRect, pRect, tRect: Rect;
     doneFlag, temp: BOOLEAN;
     myEvent: EventRecord;
     code, refNum, MyControl, t: INTEGER;
     wRecord: WindowRecord;
     theWindow, whichWindow: WindowPtr;
     theMenu, theItem: INTEGER;
     theChar: CHAR;
     ticks: LongInt;
     hTE: TEHandle;
     hCurs: CursHandle;
     iBeam: Cursor;
     hScroll, vScroll, whichControl: ControlHandle;
     TheOrigin: point;

  PROCEDURE SetUpMenus;
  { Once-only initialization for menus }

     VAR
        i: INTEGER;
        appleTitle: STRING[1];

     BEGIN
        InitMenus; { initialize Menu Manager }
        appleTitle := ' '; appleTitle[1] := CHR(20);
        myMenus[1] := NewMenu(appleMenu, appleTitle);
        AddResMenu(myMenus[1], 'DRVR'); { desk accessories }
        myMenus[2] := GetMenu(fileMenu);
        myMenus[3] := GetMenu(editMenu);
        FOR i := 1 TO lastMenu DO InsertMenu(myMenus[i], 0);
        DrawMenuBar;
     END; { of SetUpMenus }

  PROCEDURE CursorAdjust;
  { Makes cursor be I-beam inside the (active) application window's }
  { content region (except for size box and scroll bar areas). }

     VAR
        mousePt: point;

     BEGIN
        GetMouse(mousePt);
        IF theWindow=FrontWindow THEN
           BEGIN
           IF (PtInRect(mousePt, pRect)) THEN
              SetCursor(iBeam)
           ELSE
              SetCursor(arrow);
           END;
```

```
   END;

PROCEDURE DoCommand(mResult: LongInt);

   VAR
      name: STR255;

   BEGIN
      theMenu := HiWord(mResult); theItem := LoWord(mResult);
      CASE theMenu OF

         appleMenu:
            BEGIN
            GetItem(myMenus[1],theItem,name);
            refNum := OpenDeskAcc(name);
            END;

         fileMenu: doneFlag := TRUE; { Quit }

         editMenu:
            BEGIN
            IF NOT SystemEdit(theItem-1) THEN
               BEGIN
               SetPort(theWindow);
               ClipRect(pRect);

            { Delay so menu title will stay lit a little while if Command key }
               { equivalent was typed. }
               ticks := TickCount+30;
               REPEAT
               UNTIL ticks<=TickCount;

               CASE theItem OF

                  1: TECut(hTE);

                  2: TECopy(hTE);

                  3: TEPaste(hTE);

               END; { of item case }
               END;
            END; { of editMenu }

      END; { of menu case }
      HiliteMenu(0);

   END; { of DoCommand }

PROCEDURE MoveScrollBars;

   BEGIN
      WITH theWindow^.portRect DO
         BEGIN
         HideControl(vScroll);
         MoveControl(vScroll,right-15,top-1);
         SizeControl(vScroll,16,bottom-top-13);
         ShowControl(vScroll);
         HideControl(hScroll);
         MoveControl(hScroll,left-1,bottom-15);
         SizeControl(hScroll,right-left-13,16);
         ShowControl(hScroll)
         END
   END;

PROCEDURE ResizePRect;

   BEGIN
```

```
      pRect := thePort^.portRect;
      pRect.left := pRect.left+4; pRect.right := pRect.right-15;
      pRect.bottom := pRect.bottom-15
   END;

PROCEDURE GrowWnd(whichWindow: WindowPtr);
{ Handles growing and sizing the window and manipulating }
{ the update region. }

   VAR
      longResult: LongInt;
      height,width: INTEGER;
      tRect: Rect;

   BEGIN
      longResult := GrowWindow(whichWindow,myEvent.where,growRect);
      IF longResult=0 THEN EXIT(GrowWnd);
      height := HiWord(longResult); width := LoWord(longResult);

      { Add the old "scroll bar area" to the update region so it will }
      { be redrawn (for when the window is enlarged). }
      tRect := whichWindow^.portRect; tRect.left := tRect.right-16;
      InvalRect(tRect);
      tRect := whichWindow^.portRect; tRect.top := tRect.bottom-16;
      InvalRect(tRect);

      { Now draw the newly sized window. }
      SizeWindow(whichWindow,width,height,TRUE);
      MoveScrollBars;
      ResizePRect;

      { Adjust the view rectangle for TextEdit. }
      hTE^^.viewRect := pRect;

      { Add the new "scroll bar area" to the update region so it will }
      { be redrawn (for when the window is made smaller). }
      tRect := whichWindow^.portRect; tRect.left := tRect.right-16;
      InvalRect(tRect);
      tRect := whichWindow^.portRect; tRect.top := tRect.bottom-16;
      InvalRect(tRect);
   END; { of GrowWnd }

PROCEDURE DrawWindow(whichWindow: WindowPtr);
{ Draws the content region of the given window, after erasing whatever }
{ was there before. }

   VAR
      i: INTEGER;

   BEGIN
      ClipRect(whichWindow^.portRect);
      EraseRect(whichWindow^.portRect);
      DrawGrowIcon(whichWindow);
      DrawControls(whichWindow);
      TEUpdate(pRect,hTE)
   END; { of DrawWindow }

PROCEDURE ScrollBits;

   VAR
      oldOrigin: point;
      dh,dv: INTEGER;

   BEGIN
      WITH theWindow^ DO
         BEGIN
         oldOrigin := TheOrigin;
         TheOrigin.h := 4*GetCtlValue(hScroll);
```

```
         TheOrigin.v := 4*GetCtlValue(vScroll);
         dh := oldOrigin.h-TheOrigin.h;
         dv := oldOrigin.v-TheOrigin.v;
         TEScroll(dh,dv,hTE)
         END
   END;

PROCEDURE ScrollUp(whichControl: ControlHandle; theCode: INTEGER);

   BEGIN
      IF theCode=inUpButton THEN
         BEGIN
         SetCtlValue(whichControl,GetCtlValue(whichControl)-1);
         ScrollBits
         END
   END;

PROCEDURE ScrollDown(whichControl: ControlHandle; theCode: INTEGER);

   BEGIN
      IF theCode=inDownButton THEN
         BEGIN
         SetCtlValue(whichControl,GetCtlValue(whichControl)+1);
         ScrollBits
         END
   END;

PROCEDURE PageScroll(code,amount: INTEGER);

   VAR
      myPt: point;

   BEGIN
      REPEAT
         GetMouse(myPt);
         IF TestControl(whichControl,myPt)=code THEN
            BEGIN
            SetCtlValue(whichControl,GetCtlValue(whichControl)+amount);
            ScrollBits
            END
      UNTIL NOT StillDown;
   END;

BEGIN { main program }
   InitGraf(@thePort);
   InitFonts;
   FlushEvents(everyEvent,0);
   InitWindows;
   SetUpMenus;
   TEInit;
   InitDialogs(NIL);
   SetCursor(arrow);
   SetRect(dragRect,4,24,508,338);
   SetRect(growRect,100,60,512,302);
   doneFlag := FALSE;

   theWindow := GetNewWindow(256,@wRecord,POINTER(-1));
   SetPort(theWindow);
   theWindow^.txFont := 2;

   ResizePRect;
   hTE := TENew(pRect,pRect);
   hCurs := POINTER(ORD(GetCursor(256))); iBeam := hCurs^^;

   vScroll := GetNewControl(256,theWindow);
   hScroll := GetNewControl(257,theWindow);
   TheOrigin.h := 0; TheOrigin.v := 0;
```

```
REPEAT
   CursorAdjust;
   SystemTask;
   TEIdle(hTE);
   temp := GetNextEvent(everyEvent,myEvent);
   CASE myEvent.what OF

      mouseDown:
         BEGIN
         code := FindWindow(myEvent.where,whichWindow);
         CASE code OF

            inMenuBar: DoCommand(MenuSelect(myEvent.where));

            inSysWindow: SystemClick(myEvent,whichWindow);

            inDrag: DragWindow(whichWindow,myEvent.where,dragRect);

            inGoAway:
               IF TrackGoAway(whichWindow,myEvent.where) THEN
                  doneFlag := TRUE;

            inGrow:
               IF whichWindow=FrontWindow THEN
                  GrowWnd(whichWindow)
               ELSE
                  SelectWindow(whichWindow);

            inContent:
               BEGIN
               IF whichWindow<>FrontWindow THEN
                  SelectWindow(whichWindow)
               ELSE
                  BEGIN {front}
                  GlobalToLocal(myEvent.where);
                  IF PtInRect(myEvent.where,pRect) THEN
                     IF BitAnd(myEvent.modifiers,512)<>0 { Shift key pressed
                          }
                        THEN
                        TEClick(myEvent.where,TRUE,hTE)
                     ELSE
                        TEClick(myEvent.where,FALSE,hTE)
                  ELSE
                     BEGIN {controls}
                     MyControl := FindControl(myEvent.where,whichWindow,
                                              whichControl);
                     CASE MyControl OF
                        inUpButton:
                           t := TrackControl(whichControl,myEvent.where,
                                             @ScrollUp);
                        inDownButton:
                           t := TrackControl(whichControl,myEvent.where,
                                             @ScrollDown);
                        inPageUP: PageScroll(MyControl,-10);
                        inPageDown: PageScroll(MyControl,10);
                        inThumb:
                           BEGIN
                           t := TrackControl(whichControl,myEvent.where,
                                 NIL);
                           ScrollBits
                           END
                     END {Case MyControl}
                     END {controls}
                  END {front}
               END {in Content}
         END { of code case }
         END; { of mouseDown }
```

```
      keyDown, autoKey:
         BEGIN
         IF theWindow=FrontWindow THEN
            BEGIN
            theChar := CHR(myEvent.message MOD 256);
            IF BitAnd(myEvent.modifiers,256)<>0 { Command key pressed }
               THEN
               DoCommand(MenuKey(theChar))
            ELSE
               TEKey(theChar,hTE)
            END
         END; { of keyDown }

      activateEvt:
         BEGIN
         DrawGrowIcon(theWindow);
         IF ODD(myEvent.modifiers) THEN { window is becoming active }
            BEGIN
            TEActivate(hTE);
            ShowControl(vScroll);
            ShowControl(hScroll)
            END
         ELSE
            BEGIN
            TEDeactivate(hTE);
            HideControl(vScroll);
            HideControl(hScroll)
            END
         END; { of activateEvt }

      updateEvt:
         BEGIN
         BeginUpdate(theWindow);
         DrawWindow(theWindow);
         EndUpdate(theWindow)
         END { of updateEvt }

    END { of event case }

  UNTIL doneFlag
END.
```

```
$EXEC
Psample/soundlab


GSM+
sample/soundlab

Lsample/soundlab
obj/quickDraw
obj/ToolTraps
obj/OSTraps
obj/macpaslib


sample/soundlabL

RRMaker
sample/soundlabR

RSendone
Mac/soundlab.RSRC=Sound Lab.rsrc

Fdsample/soundlab.OBJ
Ydsample/soundlab.i
Ydsample/soundlabL.obj
YQ
$ENDEXEC
```

MACINTOSH CONTROL MANAGER
PROGRAMMER'S GUIDE

TOOLBOX/CONTROLS

See also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide (*** To be written ***)

Modification History:			
	First Draft	C. Espinosa	8/13/82
	Interim release (inaccurate)	C. Espinosa	9/ 7/82
	Second Draft	S. Chernicoff	3/16/83

ABSTRACT

Controls are special objects on the Macintosh screen with which the user, using the mouse, can manipulate information or control the way it is displayed. The Macintosh Control Manager is a subroutine package, part of the User Interface Toolbox, that enables application programs to create and manipulate controls in a way that is consistent with the User Interface Guidelines. This document describes the program interface to version 2.1 of the Control Manager.

Summary of significant changes and additions since last version:

- Control definition functions are now treated as resources and accessed through the Resource Manager.

- Control types are now identified with a <u>control definition ID</u>, which includes both the resource ID of the definition function and a 4-bit <u>variation code</u>. The variation code allows the same definition function to implement several related control types as "variations on a theme". Built-in constants are provided for the definition IDs of the standard control types.

- Templates for individual controls can be accessed as resources with the new function GetNewControl.

- The contrlHilite field of a control record is now a one-byte <u>part code</u> specifying the part of the control that is highlighted. A code of 255 marks the control as <u>inactive</u>; it is displayed on the screen in some distinctive way and will not respond to the mouse.

TABLE OF CONTENTS

xx About This Manual
xx About the Control Manager
xx Controls and Windows
xx Controls and Resources
xx Part Codes
xx Control Records
xx   Control Handles
xx   The ControlRecord Data Type
xx Using the Control Manager
xx Control Manager Routines
xx   Initialization and Allocation
xx   Control Display
xx   Mouse Location
xx   Control Movement and Sizing
xx   Setting and Range of a Control
xx   Miscellaneous Utilities
xx Format of a Control Template
xx Defining Your Own Controls
xx   Format of a Control Definition Function
xx   The Draw Routine
xx   The Test Routine
xx   The Routine to Calculate Regions
xx   The Initialize Routine
xx   The Dispose Routine
xx   The Position Routine
xx   The Thumb Routine
xx   The Drag Routine
xx Notes for Assembly-Language Programmers
xx Summary of the Control Manager
xx Glossary

- Control records have an additional field, contrlAction, containing a default action procedure for use by TrackControl. There are two new Control Manager routines, SetCtlAction and GetCtlAction, for accessing this field.

- The contrlTitle field has been moved to the end of the control record and is now a variable-length string instead of a pointer. Title strings of three characters or fewer are no longer handled specially.

- The FindWindow function is now in the Window Manager; FindWindow in the Control Manager has been replaced by FindControl.

- Controls are kept in a separate linked list for each window, not a single list for the entire system.

- Dragging of a control's indicator with the mouse is now handled by TrackControl instead of DragControl.

- DragControl now takes an additional parameter, slopRect, to allow some "slop" in the user's mouse movements.

- SetCtlMin and SetCtlMax now do range checking against the control's current setting and "pin" the setting, if necessary, to the new endpoint of the range.

- There are two new control messages: thumbCntl, to calculate the constraint parameters for dragging the indicator, and dragCntl, to perform custom dragging.

- The control message calcCRgns requests the indicator region instead of the whole control's region if the high bit of the parameter is set.

- The part codes used by the standard control definition functions have been modified and somewhat expanded; part codes > 127 (high bit on) now denote moving indicators.

ABOUT THIS MANUAL

This manual describes version 2.1 of the Macintosh Control Manager. *** It will eventually become a chapter in the Macintosh User Interface Toolbox Programmer's Guide. *** The Control Manager is the part of the Toolbox that deals with controls, as defined in the Macintosh User Interface Guidelines. Using it, your programs can create, manipulate, and dispose of controls in a way that is consistent with the Guidelines.

(eye)

> This document applies specifically to the version of the Control Manager in version 2.1 of the Macintosh ROM. Earlier versions will not work exactly as described here.

Like all Toolbox documentation, this document assumes you are familiar with the Macintosh User Interface Guidelines (particularly the section on controls), the Lisa Pascal programming language and system, and the memory management mechanism of the Macintosh Operating System. To understand and use the information presented here, you should also be familiar with:

- The basic principles of the QuickDraw graphics package, particularly rectangles, regions, and grafPorts. (You don't need a detailed knowledge of QuickDraw, since programs that implement controls through the Control Manager need not interface directly with QuickDraw.)

- The Window Manager. Every control you create with the Control Manager "belongs" to some window. The Window Manager and Control Manager are designed to be used together, and their structure and operation are parallel in many ways.

- The Event Manager. The essence of a control is to respond to the user's actions with the mouse. Your program finds out about those actions (such as when and where the user pressed the mouse button) by calling the Event Manager; it can then call various Control Manager routines to find out whether the button was pressed inside a control and, if so, to respond in whatever way is appropriate.

- The basics of the Resource Manager. You'll need this only if you're defining your own "custom" controls or using predefined templates for individual controls. If you use only controls of the standard types and don't create them from templates, you won't need to know any details about resources; the Control Manager itself will handle all dealings with the Resource Manager for you.

It would also be helpful to have some familiarity with a Macintosh application program that uses controls, as an illustration of the concepts presented here.

The manual begins with an introduction to the Control Manager and what you can do with it. It then discusses some basic concepts about

controls: the relationship between controls and windows; that between controls and resources; and how the various parts of a control are identified. Following this is a discussion of control records, where the Control Manager keeps all the information it needs about a control.

Next, a section on using the Control Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Control Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not be of interest to all readers. Special information is given for programmers who want to define their own controls and for those who will use the Control Manager routines from assembly language.

Finally, there are a summary of the Control Manager data structures and routines, for quick reference, and a glossary of terms used in this manual.

ABOUT THE CONTROL MANAGER

The Control Manager is the part of the Macintosh User Interface Toolbox that deals with controls. A <u>control</u> is a special object on the Macintosh screen with which the user, using the mouse, can manipulate information or control the way it is presented. Using the Control Manager, your application program can:

- Create and dispose of controls;

- Display or hide controls on the screen;

- Change the size, position, or appearance of a control;

- Read or change the setting or other properties of a control; and

- Monitor the user's operation of a control with the mouse and respond accordingly.

Your program performs these actions by calling the appropriate Control Manager routines. The Control Manager carries out the actual operations, but it's up to your program to decide when, where, and how.

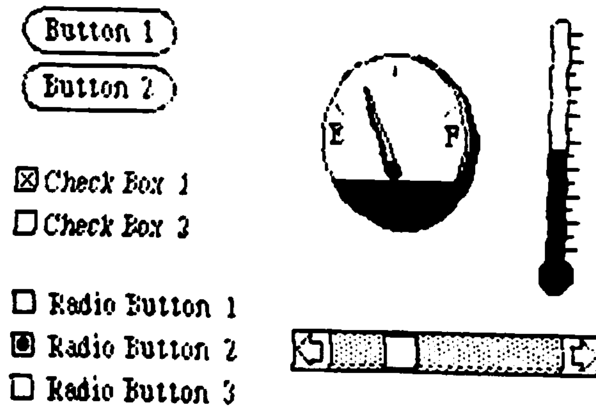


Figure 1. Controls

Controls may be of various types (see Figure 1), each with its own characteristic appearance on the screen and responses to the mouse. Each individual control has its own specific properties--such as a title, setting, location, and size--but controls of the same type behave in the same general way.

Certain standard types of control are predefined for you by the Toolbox. Your program can easily create and use controls of these standard types; you can also define your own "custom" control types for your program's special needs. Among the standard control types are the following:

- Buttons cause an immediate or continuous action when clicked or pressed with the mouse. They appear on the screen as rounded-corner rectangles with a title centered inside.

- Check boxes retain and display a setting, either checked (on) or unchecked (off); clicking with the mouse reverses the setting. On the screen, a check box appears as a small square with a title alongside it; the box is either filled in with an "X" (checked) or empty (unchecked). Check boxes are frequently used to control or modify some future action, instead of causing an immediate action of their own.

- Radio buttons also retain an on-or-off setting. They're organized into groups, with the property that only one button in the group can be on at a time: clicking any button on turns off all the others in the group, like the buttons on a car radio. Radio buttons are used to offer the user a "multiple choice" among

several alternatives. On the screen, they look just like check boxes, except that the button that's on is marked with a round, black dot instead of an "X".

(hand)
The Control Manager doesn't know which radio buttons are "connected", and doesn't automatically turn one off when the user clicks another one on: it's up to your program to handle this.

(hand)
It's a good idea to group radio buttons visually on the screen to make it clear to the user which ones are related. Each such group should be clearly labeled "Choose one of the following", or something similar.

Another important category of controls are dials. These display a quantitative setting or value, typically in some pseudoanalog form such as the position of a sliding switch, the reading on a thermometer scale, or the angle of a needle on a gauge; the setting may be displayed digitally as well. The moving part of the control that displays the current setting is called the indicator. A dial may allow the user to change its setting by dragging the indicator with the mouse, or it may simply display a value not under the user's direct control, such as the amount of free space remaining on a disk.

The Toolbox predefines one type of dial for you: the scroll bars of the standard document window, which represent the visible portion of the document by the vertical or horizontal position of the scroll bar's thumb within its shaft. A scroll bar has five parts, as shown in Figure 2: the up and down arrows scroll the window's contents a line at a time, the two paging regions scroll a "page" (windowful) at a time, and the thumb can be dragged to any desired position within the document. You can define other types of dial for yourself if your program needs them.

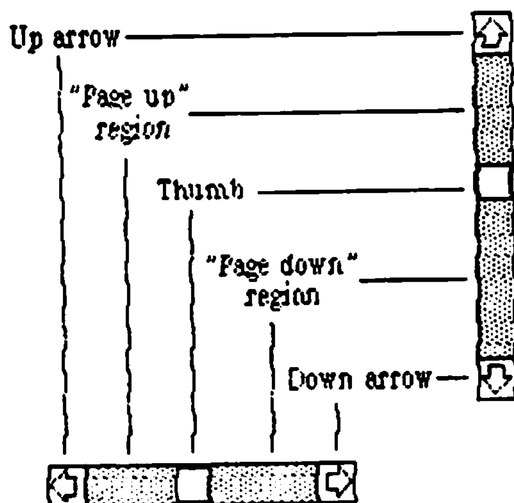


Figure 2. Parts of a Scroll Bar

(hand)
The terms "up" and "down" are used even when referring to horizontal scroll bars. In this case, "up" really means "left" and "down" means "right".

(hand)
Although they behave like controls, a document window's close box and size box are not actually implemented as controls, because the Window Manager can handle them with greater efficiency and flexibility than the Control Manager.

A control may be visible or invisible. As with windows, these terms refer only to whether the control is drawn within its own plane. A control may be "visible" and still not appear on the screen, because it is partially or completely obscured by overlapping windows or other objects. Conversely, an invisible control never appears on the screen, even if it's completely exposed to view on the desk top.

A visible control may or may not be highlighted. A highlighted control is displayed in some distinctive visual way, depending on its type (see Figure 3). A common way of highlighting a control is to invert it (change white to black and vice versa), but some control types may use other forms of highlighting, such as shading the control in gray or making its outline heavier. It's also possible for just a part of a control to be highlighted: for example, when the user presses the mouse button inside the up or down arrow of a scroll bar, the arrow (not the whole scroll bar) becomes highlighted until the button is released.

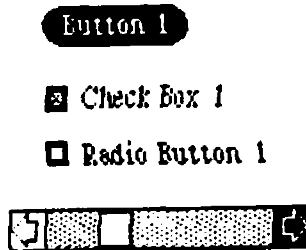


Figure 3. Highlighted Controls

A control can also be active or inactive. Active controls respond to the user's actions with the mouse; inactive controls don't. An inactive control remains visible, but is highlighted in some special way, depending on its control type (see Figure 4). For example, an inactive button, check box, or radio button is "dimmed" with light gray shading; an inactive scroll bar has no thumb.

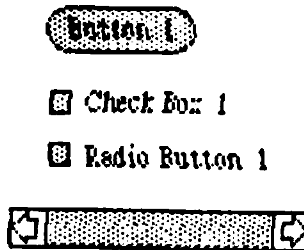


Figure 4. Inactive Controls

CONTROLS AND WINDOWS

Every control "belongs" to a particular window. When displayed, the control appears within that window's content region; when manipulated with the mouse, it acts on the contents of that window. All coordinates pertaining to the control (such as those describing its location) are expressed in its window's local coordinate system.

(eye)

> In order for the Control Manager to draw a control properly, the control's window must have the top left corner of its boundary rectangle aligned at coordinates (0,0). If your program changes a window's local coordinate system for any reason, be sure to realign its top left corner at (0,0) before drawing any of its controls. Since almost all of the Control Manager routines can (at least potentially) redraw a control, the safest policy is simply to realign the window's top left corner at (0,0) before calling any Control Manager routine.

CONTROLS AND RESOURCES

Each control type has a control definition function that determines how controls of that type look and behave. The control definition function performs all those actions that differ from one control type to another, such as initializing or disposing of a control, drawing it on the screen, testing whether the mouse button has been pressed inside it, and responding to the user's dragging of the mouse. The Control Manager calls the control definition function whenever it needs to perform one of these type-dependent actions.

Like menus, fonts, or icons, control definition functions are considered resources of your application program: they're kept in resource files and accessed through the Resource Manager. The system resource file includes definition functions for the standard control types (buttons, check boxes, radio buttons, and scroll bars). In most cases, these standard control types will be all your program will need, and you can just use the built-in definition functions. If you want to define your own, nonstandard control types, you'll have to write your own definition functions for them, as described later in the section "Defining Your Own Controls".

When you create a control, you specify its type with a control definition ID, which tells the Control Manager the resource ID of the definition function for that control type. The Control Manager provides built-in constants for the definition IDs of the standard control types:

```
CONST PushButProc   = 0;   {simple button}
      CheckBoxProc  = 1;   {check box}
      RadioButProc  = 2;   {radio button}
      ScrollBarProc = 16;  {scroll bar}
```

(hand)

The control definition ID includes some other information in addition to the resource ID of the control definition function. Details on this other information and how it's combined with the resource ID are given later under "Defining Your Own Controls". If you're using only the standard control types, you don't need to know the details; you can just use the predefined constants listed above.

To create a new control, you have to supply not only a control definition ID, but also a lot of other information, such as the control's title (if any), the window it belongs to, its location within the window, and so forth. If you're creating lots of controls with the same general characteristics, you may want to simplify the process by defining a control template. This is a single resource, stored in a resource file, that contains all the information needed to create a control of a particular type. Instead of giving all the specifics every time you create a control, you can just supply the resource ID of the template. Control templates also allow you to isolate individual control descriptions from the code of your program itself. Then if you need to change the characteristics of a control--for example, to translate its title into a foreign language--you can just change the template in the resource file, instead of modifying and recompiling your whole program.

(hand)

You can create control templates and store them in resource files with the aid *** (eventually) *** of the Resource Editor. *** In the meantime, you can use the interim Resource Compiler; see your Macintosh software coordinator for more information. *** The Resource Editor relieves you of having to know the exact format of a control template, but if you're curious *** (or until the Resource Editor is available) ***, you'll find details in the section "Format of a Control Template".

PART CODES

Some controls, such as buttons, are simple and straightforward. Others can be complex objects with many parts: for example, a scroll bar has two scroll arrows, two paging regions, and a thumb (see Figure 2). To allow different parts of a control to respond to the mouse in different ways, many of the Control Manager routines accept a part code as a parameter or return one as a result.

A part code is an integer between Ø and 255 that stands for a particular part of a control. Each type of control has its own set of part codes, assigned by the control definition function for that type. A simple control such as a button or check box might have just one "part" that encompasses the entire control; a more complex control such as a scroll bar can have as many parts as are needed to define how the control operates. Some of the Control Manager routines need to give special treatment to the moving indicator of a dial (such as the thumb of a scroll bar). To allow the Control Manager to recognize such indicators, they always have part codes of 128 or greater.

The part codes for the standard control types are built into the Control Manager as predefined constants:

```
CONST inButton     = 1Ø;   {simple button}
      inCheckBox   = 11;   {check box or radio button}

      inUpButton   = 2Ø;   {up arrow of a scroll bar}
      inDownButton = 21;   {down arrow of a scroll bar}
      inPageUp     = 22;   {"page up" region of a scroll bar}
      inPageDown   = 23;   {"page down" region of a scroll bar}
      inThumb      = 129;  {thumb of a scroll bar}
```

(hand)

Notice that the Control Manager considers a radio button to be a kind of check box. The part code inCheckBox applies to both check boxes and radio buttons.

CONTROL RECORDS

Every control is represented internally by a control record containing all pertinent information about that control. The control record contains:

- A pointer to the window the control belongs to.
- A handle to the next control in the window's control list.
- A handle to the control definition function.
- The control's title, if any.
- The control's position within its window.
- An indication of whether the control is currently visible.
- An indication of whether the control is currently active.
- An indication of which part of the control, if any, is currently highlighted.

For controls that retain a setting, either a simple on-or-off (such as a check box or radio button) or a quantitative value (such as a dial), the current setting is kept in a field of the control record. The control record also contains the minimum and maximum values the setting can assume.

The control record also includes a 32-bit reference value field, which is reserved for use by your application program. You specify an initial reference value when you create a new control, and can then access or change the reference value whenever you wish. The Control Manager completely ignores the contents of this field; your program can use it in any way you like.

A control record is a dynamic data structure and is referred to by a handle, as discussed further under "Control Handles" below. You can access and store into most of its fields with Control Manager routines, so normally you don't have to know the exact field names. However, if you want more information about the exact structure of a control record --for instance, if you're defining your own control types--you'll find it below under "The ControlRecord Data Type".

Control Handles

Storage space for control records is allocated from your program's relocatable heap zone. To allow the Operating System's memory management routines to move them as needed without creating dangling pointers, they're normally referred to by double indirection, through a control handle (a pointer to a master pointer):

```
TYPE ControlPtr    = ^ControlRecord;
     ControlHandle = ^ControlPtr;
```

(eye)

To maintain the integrity of the storage allocation system, always create and dispose of control records with the Control Manager routines provided for this purpose, rather than the Pascal standard procedures NEW and DISPOSE. The Control Manager functions for creating a new control return a handle to a newly allocated control record; thereafter, your program should normally refer to the control by this handle. Most of the Control Manager routines expect a control handle as their first parameter.

For purposes of efficiency (for example, inside a loop that your program executes many times), you may sometimes want to refer to a control by single indirection, using a pointer instead of a handle. For example,

```
VAR aPointer: ControlPtr;
    aHandle:  ControlHandle;
    . . . ;
BEGIN
    . . . ;
    aHandle := NewControl( . . . );
    aPointer := aHandle^;
    . . .
END.
```

But BE CAREFUL! Any operation that allocates storage from the heap may trigger a heap compaction, which would move (relocate) the underlying control record and leave the pointer dangling. Not only is this type of error usually disastrous, it's also very difficult to diagnose and correct. So you can safely use single indirection to refer to a control record only if you're sure you're not doing anything that may cause fresh storage to be allocated from the heap.

Handles don't suffer from this problem: the handle points to a master pointer, which in turn points to the control record. When the record is moved during a heap compaction, the master pointer is updated to point to the record at its new location; the master pointer itself is never moved. Thus you can rely on the handle not to dangle, even after a compaction.

The ControlRecord Data Type

This section contains detailed information on the structure of control records, for those who need it (for example, to define their own control types). The type ControlRecord is defined as follows:

```
TYPE ControlRecord = RECORD
                        nextControl:  ControlHandle;
                        contrlOwner:  WindowPtr;
                        contrlRect:   Rect;
                        contrlVis:    BOOLEAN;
                        contrlHilite: Byte;
                        contrlValue:  INTEGER;
                        contrlMin:    INTEGER;
                        contrlMax:    INTEGER;
                        contrlProc:   Handle;
                        contrlData:   Handle;
                        contrlAction: ProcPtr;
                        contrlRfCon:  LongInt;
                        contrlTitle:  Str255
                     END;
```

NextControl is a handle to the next control associated with this control's window. All the controls belonging to a given window are kept in a linked list, beginning in the controlList field of the window record and chained together through the nextControl fields of the individual control records. The end of the list is marked by a NIL value; as new controls are created, they are added to the beginning of the list.

ContrlOwner is a pointer to the window to which this control belongs. Notice that the contrlOwner field contains a pointer to the window, not a handle. This is because a window record is actually a grafPort with some extra fields added. Since the QuickDraw graphics package refers to grafPorts by pointers rather than handles, the Toolbox follows the same convention.

ContrlRect is the rectangle that completely encloses the control, expressed in the local coordinates of the control's window. You define this rectangle when you create the control, and can change its size or position at any time. When drawn, the control may be either scaled or clipped to this rectangle, depending on its control type; the choice is up to the control definition function.

When contrlVis is TRUE, the control is currently visible.

ContrlHilite is an integer between Ø and 255 that specifies whether and how the control is to be highlighted on the screen. A value of Ø means no highlighting; 255 means that the control is inactive and should be highlighted accordingly. Any other value is interpreted as a part code designating the part of the control that is highlighted.

ContrlValue is the control's current setting. For two-state controls such as check boxes and radio buttons, a value of Ø means the control is off and 1 means it's on. For dials, the fields contrlMin and contrlMax define the range of possible settings; contrlValue may take on any value within that range. Other (custom) control types can use these three fields as they see fit.

ContrlProc is a handle to the control definition function for this type of control. When you create a new control, you identify its type with a control definition ID; this is converted into a handle to the control definition function and stored into the contrlProc field. Thereafter, the Control Manager uses this handle to access the definition function; your program should never need to refer to this field directly.

(hand)

> The high-order byte of the contrlProc field contains some additional information that the Control Manager gets from the control definition ID; for details, see the section "Defining Your Own Controls".

(hand)

> If you write your own control definition function and will not be sharing it with other programs, you can include it as part of your application program (instead

of putting it in a resource file) and just store a handle to it in the contrlProc field. See "Defining Your Own Controls" for further information.

ContrlAction is a pointer to the control's default action procedure, used by the Control Manager function TrackControl to respond to the user's dragging the mouse inside the control. For more information on action procedures, see the description of the TrackControl function, below.

ContrlRfCon is the control's reference value. This field is provided strictly for the convenience of the application program, and you can use it for any purpose you wish.

ContrlData is a utility field reserved for use by the control definition function, typically to hold additional information specific to a particular control type. For example, the standard definition function for scroll bars uses this field for a handle to the region containing the scroll bar's thumb. If no more than four bytes of additional information are needed, the definition function can store the information directly in the contrlData field instead of using a handle.

ContrlTitle is the control's title, a variable-length string with a maximum length of 255 characters. The title is optional; some control types (such as scroll bars) don't display one. Notice that the title is given as a plain ASCII string, without CoreEdit-style formatting; the control definition function determines the type font, type size, and character style to use in displaying the title.

USING THE CONTROL MANAGER

This section discusses how the Control Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

To use the Control Manager, you must have previously called the QuickDraw routine InitGraf to initialize QuickDraw. You should also have called the Resource Manager routine OpenResFile to open any resource files that you'll be using (other than the system resource file, which is opened automatically).

Where appropriate in your program, use NewControl or GetNewControl to create any controls you need. NewControl takes descriptive information about the new control from its parameters; GetNewControl gets the information from a control template in a resource file. When you no longer need a control, call DisposeControl to remove it from its window's control list and free the memory it occupies. To dispose of all of a given window's controls at once, use KillControls.

(hand)

The Window Manager routines DisposeWindow and CloseWindow automatically dispose of all the controls associated with the given window.

When the Event Manager reports that an update event has occurred for a window, your program should call DrawControls to redraw the window's controls as part of the process of updating the window.

After receiving a mouse-down event from GetNextEvent,

1. First call FindWindow to determine in which part of which window the mouse button was pressed.

2. If it was in the content region of the active window, next call FindControl for that window to find out whether it was in an active control, and if so, in which part of which control.

3. Finally, take whatever action is appropriate when the user presses the mouse button in that part of the control, using routines such as TrackControl (to perform some action repeatedly for as long as the mouse button is down, or to allow the user to drag the control's moving indicator with the mouse), DragControl (to allow the user to drag the entire control with the mouse), and HiliteControl (to change the way the control is highlighted on the screen).

Wherever needed in your program, you can call HideControl to make a control invisible or ShowControl to make it visible. Similarly, MoveControl, which simply changes a control's location without pulling around an outline of it, can be called at any time, as can SizeControl, which changes its size--though you shouldn't surprise the user by taking these actions unexpectedly.

Whenever necessary, you can read the current setting of a control with GetCtlValue, or other attributes with GetCTitle, GetCtlMin, GetCtlMax, GetCRefCon, or GetCtlAction; you can change them with SetCtlValue, SetCTitle, SetCtlMin, SetCtlMax, SetCRefCon, or SetCtlAction.

CONTROL MANAGER ROUTINES

This section describes the routines (procedures and functions) that make up the Control Manager.

Initialization and Allocation

```
FUNCTION NewControl (theWindow: WindowPtr; boundsRect: Rect; title:
            Str255; visible: BOOLEAN; value: INTEGER; min: INTEGER;
            max: INTEGER; procID: INTEGER; refCon: LongInt) :
            ControlHandle;
```

NewControl creates a new control record, links it to the beginning of theWindow's control list, and returns a handle to the new record. It initializes the new record's fields to the values passed as parameters, setting the contrlHilite field to 0 (no highlighting) and contrlAction to NIL (no default action procedure; see TrackControl under "Mouse Location", below). It also calls the control definition function to perform any type-specific initialization that may be needed, such as setting the contrlData field.

TheWindow is the window the new control will belong to. All coordinates pertaining to the control will be interpreted in this window's local coordinate system.

BoundsRect, a rectangle expressed in theWindow's local coordinates, determines the control's size and location.

Title is the control's title. The string you supply as the value of this parameter will be stored in the control's contrlTitle field, but some types of control will never use it. In this case, you can just pass an empty string as the title.

If the visible parameter is TRUE, NewControl calls the control definition function to draw the control.

The min and max parameters define the control's range of possible settings; the value parameter gives the initial setting, and must fall within the specified range. For controls that don't retain a setting, such as simple buttons, the values you supply for these parameters will be stored into the corresponding fields of the control record, but will never be used. So it doesn't matter what values you give--0 for all three parameters will do. For controls that just retain an on-or-off setting, such as check boxes or radio buttons, min should be 0 (meaning the control is off) and max should be 1 (meaning it's on). For dials, you can specify whatever numerical values are appropriate for min, max, and value.

ProcID is the control definition ID, which leads to the control definition function for this type of control. The control definition

IDs for the standard control types are listed above under "Controls and Resources". Control definition IDs for custom control types are discussed under "Defining Your Own Controls", below.

RefCon is the control's reference value, set and used only by your application program.

FUNCTION GetNewControl (controlID: INTEGER; theWindow: WindowPtr) :
ControlHandle;

GetNewControl creates a new control record from a control template stored in a resource file, links it to the beginning of theWindow's control list, and returns a handle to the new record. ControlID is the resource ID of the template in the resource file. GetNewControl works exactly the same as NewControl (see above), except that it gets the initial values for the new control's fields from the specified control template instead of accepting them as parameters.

PROCEDURE DisposeControl (theControl: ControlHandle);

DisposeControl erases theControl from the screen, deletes it from its window's control list, and disposes of its storage. It returns to the heap all data structures associated with the control. It also calls the control definition function to do any type-specific housekeeping that may be needed, such as disposing of a data structure whose handle is kept in the contrlData field.

PROCEDURE KillControls (theWindow: WindowPtr);

KillControls disposes of all controls associated with theWindow by calling DisposeControl (see above) for each.

Control Display

The routines in this section affect the appearance of a control but not its size or location.

PROCEDURE SetCTitle (theControl: ControlHandle; theTitle: Str255);

SetCTitle sets theControl's title to theTitle. The control definition function determines the type font, type size, and character style to use in displaying the title; it may use the system font, that of the control's window, or any other font it chooses, or it may choose not to display the title at all.

(hand)
Buttons, check boxes, and radio buttons all display their titles in the standard system font; scroll bars don't display a title.

PROCEDURE GetCTitle (theControl: ControlHandle; VAR theTitle: Str255);

GetCTitle returns theControl's current title string as the value of the parameter theTitle, regardless of whether the definition function for this control type actually uses the title.

PROCEDURE HideControl (theControl: ControlHandle);

HideControl makes theControl invisible. It sets the contrlVis field to FALSE and fills the region the control occupies within its window with the window's background pattern. It also adds the control's enclosing rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the control is already invisible, HideControl has no effect.

PROCEDURE ShowControl (theControl: ControlHandle);

ShowControl makes theControl visible. It sets the contrlVis field to TRUE and calls the control definition function to do the actual drawing. The control is drawn in its proper plane on the screen, and may be completely or partially obscured by overlapping windows or other objects. If the control is already visible, ShowControl has no effect.

PROCEDURE DrawControls (theWindow: WindowPtr);

DrawControls draws all controls currently visible in theWindow. The controls are drawn in reverse order of creation; thus in case of overlap the earliest-created controls appear frontmost in the window.

(hand)

> Window Manager routines such as SelectWindow, ShowWindow, and BringToFront do not automatically call DrawControls to display the window's controls. They just add the appropriate regions to the window's update region, generating an update event. Your program should always call DrawControls explicitly on receiving an update event for a window.

PROCEDURE HiliteControl (theControl: ControlHandle; hiliteState: INTEGER);

HiliteControl changes the way theControl is highlighted on the screen. HiliteState is an integer between Ø and 255. A value of Ø means no highlighting; 255 means that the control is to be made inactive and highlighted accordingly. Any other value is interpreted as a part code designating the part of the control to be highlighted. HiliteControl sets the contrlHilite field to the designated value, then calls the control definition function to redraw the control with its new highlighting.

Mouse Location

```
FUNCTION TestControl (theControl: ControlHandle; thePoint: Point) :
          INTEGER;
```

TestControl tests which part of theControl contains thePoint and returns the corresponding part code, or Ø if the point is outside the control. If the control is invisible or inactive, no test is performed and TestControl returns a result of Ø.

```
FUNCTION FindControl (thePoint: Point; theWindow: WindowPtr; VAR
          theControl: ControlHandle) : INTEGER;
```

FindControl finds which of theWindow's active controls, if any, contains thePoint. It returns a handle to the control as the value of the parameter theControl; the function result is a part code identifying the part of the control that contains the given point. The point must be expressed in the window's local coordinate system.

When a mouse down event occurs, you should normally call the Window Manager function FindWindow to find out in which window, if any, the mouse button was pressed. Next, if it was pressed in the window's content region, call FindControl to see whether it was in any of the window's controls. If so, you can then do whatever is appropriate for a mouse down event in that control (for example, call TrackControl or DragControl).

(eye)

> Notice that FindControl expects the mouse point in local (window) coordinates, whereas FindWindow expects it in global coordinates. Always be sure to convert the point to local coordinates with the QuickDraw procedure GlobalToLocal before calling FindControl.

FindControl calls TestControl (see above) for each of the window's active controls to see whether it contains the given point. In the event of overlap, FindControl returns the frontmost control containing the point. If the point doesn't lie within any active control, it returns NIL for the control and Ø for the part code. (It also returns these values if the window is invisible or doesn't contain the given point. In these cases, however, FindWindow wouldn't have returned this window in the first place, so the situation should never arise.)

```
FUNCTION TrackControl (theControl: ControlHandle; startPt: Point;
          actionProc: ProcPtr) : INTEGER;
```

TrackControl is the routine that does the actual work of a control. When called with the mouse button down, it keeps control until the button is released, following the movements of the mouse and responding

in whatever way is appropriate, depending on the type of control and the part of the control in which the button was pressed.

The actionProc parameter is a pointer to an action procedure; it defines some action to be performed repeatedly for as long as the user holds down the mouse button. For example, when the mouse button is pressed in the up or down arrow of a scroll bar, the action procedure should scroll the contents of the window one line in the indicated direction. This will cause the window's contents to scroll continuously, one line at a time, for as long as the button is held down.

If the actionProc parameter is NIL, TrackControl simply retains control until the mouse button is released, performing no action while the button is down beyond highlighting the control or dragging its indicator. If actionProc is POINTER(-1), TrackControl uses the control's default action procedure (if any), stored in the contrlAction field of the control record.

(hand)

> Actually, the default action procedure is used whenever the value of the actionProc parameter is odd. This causes no conflict, since genuine procedure pointers are always even (aligned on a word boundary).

The parameter startPt is assumed to be the screen location where the mouse button was pressed, expressed in local window coordinates. TrackControl finds which part of the control contains the given point, then focuses its attention only on that part. Its behavior depends on whether the part is the indicator of a dial (that is, whether it has a part code > 127).

If the part is an indicator, TrackControl drags a flickering outline of the indicator to follow the mouse until the button is released. (The process is similar to that described below under DragControl, except that only the indicator is dragged and not the whole control. The control definition function calculates the limiting rectangle, slop rectangle, and axis parameter for this operation.) In this case, the action procedure passed to TrackControl, if any, should take no parameters. For example, if the name of the action procedure is Action, it should be declared simply as

```
PROCEDURE Action;
```

When the user releases the mouse button, TrackControl calls the control definition function to reposition the control's indicator, passing the vertical and horizontal offset through which the mouse was dragged. It's up to the definition function to adjust the control's setting, redraw the control, or take whatever other action is appropriate. For example, the standard definition function for scroll bars redraws the scroll bar's thumb, calculates its new relative position within the shaft, and scrolls the window to the corresponding relative position in the document.

If the control is not a dial, or if the mouse button was initially pressed in a part of a dial other than the indicator, the action procedure (if any) should be of the form

```
PROCEDURE Action (theControl: ControlHandle; partCode: INTEGER);
```

In this case, TrackControl repeatedly reads the position of the mouse for as long as the button remains down, testing whether it's still in the original part of the control. If so, TrackControl highlights the part and passes its part code to the action procedure, along with a handle to the control itself. If the mouse is outside the original control part--that is, if the user has moved out of the part while still holding down the button--TrackControl unhighlights the part and passes a part code of Ø to the action procedure. In either case, TrackControl reads the mouse's position again and repeats the process until the mouse button is released.

When the user finally releases the button, TrackControl unhighlights the control. If the button is released inside the same part of the control in which it was originally pressed, TrackControl returns the part code for that part; if not, it returns Ø. You can use this information, for example, to allow the user to "back out" of an operation by moving the mouse out of the control before releasing the button.

Control Movement and Sizing

```
PROCEDURE MoveControl (theControl: ControlHandle; h, v: INTEGER);
```

MoveControl moves theControl to a new location within its window. The top left corner of the control's enclosing rectangle is moved to the new horizontal and vertical coordinates h and v; the bottom right corner is adjusted accordingly, to keep the size of the rectangle the same as before. If the control is currently visible, it is hidden and then redrawn at its new location.

```
PROCEDURE DragControl (theControl: ControlHandle; startPt: Point;
           limitRect, slopRect: Rect; axis: INTEGER);
```

When called with the mouse button down, DragControl allows the user to drag a flickering outline of theControl around the screen with the mouse. It follows the movements of the mouse for as long as the button is held down, then calls MoveControl (see above) to move the control to the position where the button was released.

(hand)

> Before beginning to follow the mouse, DragControl calls the control definition function to allow it to do its own "custom dragging" if it chooses. If the definition function doesn't choose to do any custom dragging,

DragControl uses the default method of dragging described here.

The startPt parameter is assumed to be the point where the mouse button was originally pressed, expressed in the local coordinates of the control's window. The limitRect rectangle limits the travel of the control, and should normally coincide with or be contained within the window's content region. DragControl will never move the top left corner of the control outside this rectangle, regardless of where the user drags the mouse. The second rectangle, slopRect, allows the user some "slop" in moving the mouse; it should completely enclose the limiting rectangle. DragControl's behavior while tracking depends on the position of the mouse with respect to these two rectangles:

- When the mouse is inside limitRect, the control's flickering outline follows it normally; if the button is released, the control will be moved to the mouse position.

- When the mouse is outside limitRect but inside slopRect, the control's outline "pins" at the edge of limitRect; if the button is released, the control will be moved to this "pinned" location.

- When the mouse is outside slopRect, the control's outline disappears from the screen, but DragControl continues to follow the mouse; if it moves back into slopRect, the outline reappears. If the button is released outside slopRect, the control will not be moved from its original position.

The axis parameter allows you to constrain the control's motion to only one axis:

Axis Parameter	Meaning
Ø	No constraint
1	Horizontal motion only
2	Vertical motion only

If an axis constraint is in effect, the control will follow the mouse's movements along the specified axis only, ignoring motion along the other axis. With or without an axis constraint, the mouse must still be inside the slop rectangle for the control to move at all.

PROCEDURE SizeControl (theControl: ControlHandle; w, h: INTEGER);

SizeControl changes the size of theControl's enclosing rectangle. The bottom right corner of the rectangle is adjusted to set the rectangle's width and height to w and h; the position of the top left corner is not changed. If the control is currently visible, it is hidden and then redrawn in its new size. The actual drawing is done by the control definition function, which may either scale or clip the control to its new enclosing rectangle.

Setting and Range of a Control

PROCEDURE SetCtlValue (theControl: ControlHandle; theValue: INTEGER);

SetCtlValue sets theControl's current setting (contrlValue) to theValue and redraws the control to reflect the new setting. If the specified value is out of range, it is forced to the nearest endpoint of the current range. That is, if theValue < contrlMin, contrlValue is set to contrlMin; if theValue > contrlMax, contrlValue is set to contrlMax.

FUNCTION GetCtlValue (theControl: ControlHandle) : INTEGER;

GetCtlValue returns theControl's current setting (contrlValue).

PROCEDURE SetCtlMin (theControl: ControlHandle; minValue: INTEGER);

SetCtlMin sets theControl's minimum setting (contrlMin) to minValue and redraws the control to reflect the new range. If minValue is greater than the control's current setting (contrlValue), the setting is changed to the new minimum value.

FUNCTION GetCtlMin (theControl: ControlHandle) : INTEGER;

GetCtlMin returns theControl's current minimum value (contrlMin).

PROCEDURE SetCtlMax (theControl: ControlHandle; maxValue: INTEGER);

SetCtlMax sets theControl's maximum setting (contrlMax) to maxValue and redraws the control to reflect the new range. If maxValue is less than the control's current setting (contrlValue), the setting is changed to the new maximum value.

FUNCTION GetCtlMax (theControl: ControlHandle) : INTEGER;

GetCtlMax returns theControl's current maximum value (contrlMax).

Miscellaneous Utilities

PROCEDURE SetCRefCon (theControl: ControlHandle; refVal: LongInt);

SetCRefCon sets theControl's reference value to refVal. The reference value is reserved for use by your program, which can use it in any way you wish; it is ignored by the Control Manager itself.

FUNCTION GetCRefCon (theControl: ControlHandle) : LongInt;

GetCRefCon returns theControl's current reference value.

PROCEDURE SetCtlAction (theControl: ControlHandle; actionProc:
ProcPtr);

SetCtlAction sets theControl's default action procedure to actionProc. TrackControl uses this procedure to respond to the user's dragging the mouse inside the control; for more information, see TrackControl under "Mouse Location", above.

FUNCTION GetCtlAction (theControl: ControlHandle) : ProcPtr;

GetCtlAction returns a pointer to theControl's default action procedure. TrackControl uses this procedure to respond to the user's dragging the mouse inside the control; for more information, see TrackControl under "Mouse Location", above.

FORMAT OF A CONTROL TEMPLATE

As described above, you can use the GetNewControl function to create a new control from a template stored in a resource file. Such a template contains the same information that the NewControl function gets from eight of its parameters. The resource type for a control template is 'CTRL', and the resource data has the following format:

Number of bytes	Contents
8 bytes	Same as boundsRect parameter to NewControl
2 bytes	Same as value parameter to NewControl
2 bytes	Same as visible parameter to NewControl
2 bytes	Same as max parameter to NewControl
2 bytes	Same as min parameter to NewControl
4 bytes	Same as procID parameter to NewControl
4 bytes	Same as refCon parameter to NewControl
n bytes	Same as title parameter to NewControl (1-byte length in bytes, followed by the characters of the title)

DEFINING YOUR OWN CONTROLS

In addition to the standard, built-in control types (buttons, check boxes, radio buttons, and scroll bars), the Control Manager allows you to define "custom" control types of your own. Maybe you need a three-way selector switch, a disk-space indicator that looks like a thermometer, or a thruster control for a spacecraft simulator--whatever

your particular application calls for. This section contains the information you need to define your own control types to meet your program's special needs.

(hand)

For the convenience of your program's user, remember to conform to the Macintosh User Interface Guidelines for controls as much as possible.

Every control type is defined by a control definition function, which is normally stored in a resource file; its resource type is 'CDEF'. To define a control type of your own, you write a control definition function and (usually) store it in a resource file, with a resource type of 'CDEF' and a resource ID of your own choosing. The resource data is simply the compiled or assembled code of the control definition function, which may be written in Pascal or assembly language; the only requirement is that its entry point must be at the beginning.

(eye)

Resource IDs Ø through 8 are reserved for predefined control definition functions in the system resource file. Unless you want to override one of the built-in functions, the resource ID you choose for your own control definition function should be greater than 8.

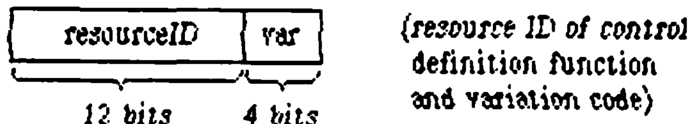
Whenever you create a new control, you specify its type by giving a control definition ID. This is a 16-bit integer that contains the resource ID of the control definition function in its upper 12 bits, along with a variation code in the lower four bits. Thus, for a given resource ID and variation code, the control definition ID is:

16 * resource ID + variation code

The variation code allows a single control definition function to implement several related control types as "variations on a theme". For example, buttons, check boxes, and radio buttons all use the standard definition function whose resource ID is Ø, but they have variation codes of Ø, 1, and 2, respectively.

The Control Manager calls the Resource Manager to find the resource of type 'CDEF' with the given resource ID. The Resource Manager searches first in any application resource files, in the reverse order they were opened, and last in the system resource file. When it finds the requested resource, it reads the resource's data (the code of the control definition function) into memory and returns a handle to it. The Control Manager stores this handle in the contrlProc field of the new control record, along with the variation code in the high-order byte of the field. Later, when it needs to perform a type-dependent action on the control, it uses the handle to find the control definition function and passes it the variation code as a parameter. Figure 5 illustrates this process.

Control definition ID supplied when control is created:



Resource Manager call made by Control Manager:

defHandle = GetResource ('CDEF', resourceID);

Field in control record:

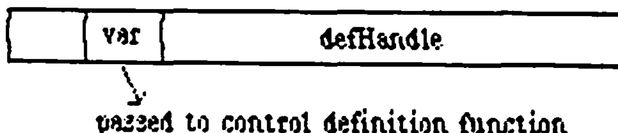


Figure 5. Control Definition Handling

(hand)

If you won't be sharing your control definition function with other application programs, you may find it more convenient to include it with the code of your program instead of placing it in a resource file. If you do this, you have to supply a dummy control definition ID when you create a new control of this type, pointing to a definition function that IS stored in a resource file--for example, the definition ID of one of the standard control types--and specify that the control initially be made invisible. Once the control is created, you can replace the contents of the contrlProc field with a handle to the actual control definition function (along with a variation code, if needed, in the high-order byte of the field). You can then call ShowControl, if necessary, to make the control visible within its window.

Format of a Control Definition Function

You can give your control definition function any name you like. Here's how you would declare one named MyControl:

```
FUNCTION MyControl (varCode: INTEGER; theControl: ControlHandle;
         theMessage: ControlMessage; param: LongInt) : LongInt;
```

VarCode is the variation code, as described above.

TheControl is a handle to the control that the operation will affect.

TheMessage is a control message identifying the desired operation:

```
TYPE ControlMessage = (drawCntl, testCntl, calcCRgns, initCntl,
                       dispCntl, posCntl, thumbCntl, dragCntl);
```

Message	Operation
drawCntl	Draw the control in its window
testCntl	Test in what part of the control (if any) the mouse button was pressed
calcCRgns	Calculate the control's region (or that of its indicator) within its window
initCntl	Do any special control initialization
dispCntl	Take any special actions when the control is disposed of
posCntl	Reposition the control's indicator and update its value accordingly
thumbCntl	Calculate the parameters for dragging the control's indicator with the mouse
dragCntl	Drag the control (or its indicator) with the mouse

As described below in the discussions of the routines that perform these operations, the value passed for param, the last parameter of the control definition function, depends on the operation. Where it is not mentioned below, this parameter is ignored. Similarly, the control definition function is expected to return a function result only where indicated; in other cases, the function should return Ø.

(hand)

"Routine" here does not necessarily mean a procedure or function. While it's a good idea to set these up as subprograms inside the window definition function, you are not required to do so.

The Draw Routine

The message drawCntl asks the control definition function to draw all or part of a control within its window. The value of param is a part code specifying which part of the control to draw, or Ø for the entire control. If the control is invisible (that is, if its contrlVis field is FALSE), there's nothing to do; if it's visible, the definition function should draw it (or the requested part), taking into account the current values of its contrlHilite and contrlValue fields.

(eye)

The Control Manager procedures SetCtlValue, SetCtlMin, and SetCtlMax all send the message drawCntl with a part code parameter of 128, asking the control definition function to redraw a control's moving indicator. For control types using other part codes to represent indicators, the definition function must detect a param

value of 128 as a special case and redraw all indicators, regardless of part code.

The Test Routine

The message testCntl asks in which part of a control, if any, a given point lies. The point is passed as the value of param, expressed as a four-byte record of type Point (not a pointer or a handle) in the local coordinates of the control's window. The control definition function should return the part code for the part of the control that contains the point; it should return Ø if the point is outside the control's region or if the control is inactive (contrlHilite = 255).

The Routine to Calculate Regions

The control definition function should respond to the message calcCRgns by calculating the region a control occupies within its window. Param is a QuickDraw region handle; the definition function should update this region to the shape, size, and position of the control, expressed in the local coordinate system of its window.

If the high-order bit of param is set, the region requested is that of the control's indicator, rather than that of the control as a whole. The definition function should clear the high BYTE (not just the high bit) of the region handle before attempting to update the region.

(hand)

> Notice that the control and its indicator aren't limited to rectangular boxes, but may occupy regions of any shape, in the full generality permitted by QuickDraw.

The Initialize Routine

When it creates a new control, the Control Manager sends the message initCntl to the control definition function. This gives the definition function a chance to perform any type-specific initialization it may require. For example, the standard definition function for scroll bars allocates space for a region to hold the scroll bar's thumb location and stores the region handle in the contrlData field of the new control record. The initialization routine for buttons, check boxes, and radio buttons does nothing.

The Dispose Routine

The Control Manager's DisposeControl procedure sends the message dispCntl to the control definition function, telling it to carry out any special "housekeeping" associated with disposing of a control. For example, the standard definition function for scroll bars deallocates the space occupied by the thumb region, whose handle is kept in the

control's contrlData field. The dispose routine for buttons, check boxes, and radio buttons does nothing.

The Position Routine

The message posCntl tells the control definition function to reposition a control's moving indicator and update the control's setting accordingly. The value of param is a point giving the vertical and horizontal offset, in screen pixels, by which the indicator is to be moved relative to its current position. (Typically, this is the offset between the points where the user pressed and released the mouse button while dragging the indicator.) The vertical offset is given in the high-order word of the LongInt and the horizontal offset in the low-order word. The definition function should calculate the control's new setting based on the given offset, update the contrlValue field, and redraw the control within its window to reflect the new setting.

(hand)

> If you use the Control Manager procedure SetCtlValue to update the contrlValue field, the control will be redrawn automatically.

The Thumb Routine

The control definition function should respond to the message thumbCntl by calculating the limiting rectangle, slop rectangle, and axis constraint for dragging a control's indicator with the mouse (see the descriptions of DragControl and TrackControl, above). Param is a pointer to a data structure of type

```
RECORD
   limitRect, slopRect: Rect;
   axis: INTEGER
END;
```

On entry, param^.limitRect.topLeft contains the point where the mouse button was first pressed. The definition function should store the appropriate values into the fields of the record pointed to by param.

The Drag Routine

The message dragCntl asks the control definition function to drag a control or its indicator around on the screen to follow the mouse until the user releases the mouse button. Param is a Boolean value specifying whether to drag the indicator or the whole control: TRUE means just drag the indicator.

The control definition function need not implement any form of "custom dragging"; if it returns a result of Ø, the Control Manager will use its own default method of dragging (see the description of DragControl above). Conversely, if the control definition function chooses to do

its own custom dragging, it should signal the Control Manager not to use the default method by returning a nonzero result.

If the whole control is being dragged, the definition function should call MoveControl to reposition the control to its new location after the user releases the mouse button. If just the indicator is being dragged, the definition function should execute its own position routine (see above) to update the control's setting and redraw it in its window.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

Information about how to use the User Interface Toolbox from assembly language is given elsewhere. *** For now, see the QuickDraw manual. *** This section contains special notes of interest to progrmmers who will be using the Control Manager from assembly language.

The primary aid to assembly-language programmers is a file named TOOLEQU.TEXT. If you name this file in an .INCLUDE statement when you assemble your program, all the Control Manager constants, offsets to locations of global variables, and offsets into the fields of structured types will be available in symbolic form.

SUMMARY OF THE CONTROL MANAGER

```
CONST PushButProc   = 0;   {simple button}
      CheckBoxProc  = 1;   {check box}
      RadioButProc  = 2;   {radio button}
      ScrollBarProc = 16;  {scroll bar}

      inButton      = 10;  {simple button}
      inCheckBox    = 11;  {check box or radio button}

      inUpButton    = 20;  {up arrow of a scroll bar}
      inDownButton  = 21;  {down arrow of a scroll bar}
      inPageUp      = 22;  {"page up" region of a scroll bar}
      inPageDown    = 23;  {"page down" region of a scroll bar}
      inThumb       = 129; {thumb of a scroll bar}

TYPE ControlHandle = ^ControlPtr;
     ControlPtr    = ^ControlRecord;

     ControlRecord = RECORD
                       nextControl:  ControlHandle;
                       contrlOwner:  WindowPtr;
                       contrlRect:   Rect;
                       contrlVis:    BOOLEAN;
                       contrlHilite: Byte;
                       contrlValue:  INTEGER;
                       contrlMin:    INTEGER;
                       contrlMax:    INTEGER;
                       contrlProc:   Handle;
                       contrlData:   Handle;
                       contrlAction: ProcPtr;
                       contrlRfCon:  LongInt;
                       contrlTitle:  Str255
                     END;

     ControlMessage = (drawCntl, testCntl, calcCRgns, initCntl,
                       dispCntl, posCntl, thumbCntl, dragCntl);
```

Initialization and Allocation

```
FUNCTION NewControl       (theWindow: WindowPtr; boundsRect: Rect;
                           title: Str255; visible: BOOLEAN; value:
                           INTEGER; min: INTEGER; max: INTEGER;
                           procID: INTEGER; refCon: LongInt) :
                           ControlHandle;
FUNCTION GetNewControl    (controlID: INTEGER; theWindow: WindowPtr) :
                           ControlHandle;
PROCEDURE DisposeControl (theControl: ControlHandle);
PROCEDURE KillControls   (theWindow: WindowPtr);
```

Control Display

```
PROCEDURE SetCTitle     (theControl: ControlHandle; theTitle: Str255);
PROCEDURE GetCTitle     (theControl: ControlHandle; VAR theTitle:
                         Str255);
PROCEDURE HideControl   (theControl: ControlHandle);
PROCEDURE ShowControl   (theControl: ControlHandle);
PROCEDURE DrawControls  (theWindow: WindowPtr);
PROCEDURE HiliteControl (theControl: ControlHandle; hiliteState:
                         INTEGER);
```

Mouse Location

```
FUNCTION TestControl  (theControl: ControlHandle; thePoint: Point) :
                       INTEGER;
FUNCTION FindControl  (thePoint: Point; theWindow: WindowPtr; VAR
                       theControl: ControlHandle) : INTEGER;
FUNCTION TrackControl (theControl: ControlHandle; startPt: Point;
                       actionProc: ProcPtr) : INTEGER;
```

Control Movement and Sizing

```
PROCEDURE MoveControl (theControl: ControlHandle; h, v: INTEGER);
PROCEDURE DragControl (theControl: ControlHandle; startPt: Point;
                       limitRect, slopRect: Rect; axis: INTEGER);
PROCEDURE SizeControl (theControl: ControlHandle; w, h: INTEGER);
```

Setting and Range of a Control

```
PROCEDURE SetCtlValue (theControl: ControlHandle; theValue: INTEGER);
FUNCTION GetCtlValue  (theControl: ControlHandle) : INTEGER;
PROCEDURE SetCtlMin   (theControl: ControlHandle; minValue: INTEGER);
FUNCTION GetCtlMin    (theControl: ControlHandle) : INTEGER;
PROCEDURE SetCtlMax   (theControl: ControlHandle; maxValue: INTEGER);
FUNCTION GetCtlMax    (theControl: ControlHandle) : INTEGER;
```

Miscellaneous Utilities

```
PROCEDURE SetCRefCon    (theControl: ControlHandle; refVal: LongInt);
FUNCTION GetCRefCon     (theControl: ControlHandle) : LongInt;
PROCEDURE SetCtlAction  (theControl: ControlHandle; actionProc: ProcPtr);
FUNCTION GetCtlAction   (theControl: ControlHandle) : ProcPtr;
```

GLOSSARY

action procedure: A procedure passed as a parameter to the Control Manager routine TrackControl, defining an action to be performed repeatedly for as long as the mouse button is held down.

active control: A control that will respond to the user's actions with the mouse.

button: A standard Macintosh control that causes some immediate or continuous action when clicked or pressed with the mouse.

check box: A standard Macintosh control that retains and displays a setting, either checked (on) or unchecked (off). Clicking inside the check box with the mouse reverses the setting.

control: An object in a window on the Macintosh screen with which the user, using the mouse, can manipulate the information in the window or control the way it is presented.

control definition function: A function called by the Control Manager when it needs to perform certain basic operations on a particular type of control, such as drawing the control in its window.

control definition ID: A number passed to control-creation routines to indicate the type of control; it consists of the control definition function's resource ID and a variation code.

control handle: A reference to a control record by double indirection; a pointer to the master pointer to the record.

control list: A linked list of the controls associated with a given window.

control message: A parameter passed to a control definition function to identify the operation desired.

control record: The internal representation of a control, where the Control Manager stores all the information it needs for its operations on that control.

control template: A resource that contains information from which the Control Manager can create a control.

dial: A control with a moving indicator that displays a quantitative setting or value. Depending on the type of dial, the user may or may not be able to change the setting by dragging the indicator with the mouse.

highlight: To display a control or part of a control in some distinctive visual way, such as inverting it or making its outline heavier.

inactive control: A control that will not respond to the user's actions with the mouse. An inactive control is highlighted in some special way, such as "dimming" it with light gray shading.

indicator: The moving part of a dial that displays its current setting.

invisible control: A control that is not drawn in its window.

part code: An integer code, defined by the control definition function, that stands for a particular part of a control.

radio button: A standard Macintosh control that retains and displays a setting, either on or off, and is part of a group with the property that only one button in the group can be on at a time. Clicking a radio button on turns off all the others in the group, like the buttons on a car radio.

reference value: In a control record, a 32-bit field which the application program may stor into and access for any purpose.

variation code: A number that distinguishes closely related types of controls and is passed as part of a control definition ID when a control is created.

visible control: A control that is drawn in its window (but may be completely overlapped by another window or other object on the screen).

MACINTOSH USER EDUCATION

The Desk Manager: A Programmer's Guide /DSKMGR/DESK

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide

Modification History:	First Draft (ROM 2.0)	C. Rose	2/3/83
	Erratum Added	C. Rose	2/28/83
	Second Draft (ROM 4)	C. Rose	6/14/83
	Third Draft (ROM 7)	C. Rose	9/26/83

This manual introduces you to the Desk Manager, the part of the Macintosh User Interface Toolbox that handles desk accessories such as the Calculator. It describes the simple programmatic interface to the Desk Manager and tells you how to define your own desk accessories.

Summary of significant changes and additions since last version:

- OpenDeskAcc is now a Desk Manager routine, as is the new procedure CloseDeskAcc (page 7).

- A new function, SystemEdit, processes standard editing commands in desk accessories (page 8). Four new messages are passed to a desk accessory's control routine to handle this (page 13).

- Storing the window pointer in the Device Control Entry is now optional for a desk accessory's open routine, and setting the windowKind field to the driver's reference number is required (page 13).

- A desk accessory may be displayed in a window created by the Dialog Manager; if so, its control routine must respond to the "cursor" message in a special way (page 14). Applications allowing access to desk accessories must initialize TextEdit and the Dialog Manager.

TABLE OF CONTENTS

3 About This Manual
3 About the Desk Manager
5 Using the Desk Manager
6 Desk Manager Routines
7 Opening and Closing Desk Accessories
7 Handling Events in Desk Accessories
8 Performing Periodic Actions
9 Advanced Routines
10 Defining Your Own Desk Accessories
12 The Device Control Entry
12 The Driver Routines
15 A Sample Desk Accessory
16 Summary of the Desk Manager
17 Glossary

Copyright (c) 1983 Apple Computer, Inc. All rights reserved. Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Desk Manager, the part of the Macintosh User Interface Toolbox that supports the use of desk accessories from an application; the Calculator, for example, is a standard desk accessory available to any application. *** Eventually this will become part of a large manual describing the entire Toolbox. *** You'll learn how to use the Desk Manager routines and how to define your own accessories.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Desk Manager may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The Toolbox Event Manager, the Window Manager, the Menu Manager, and the Dialog Manager.

- The basic concepts behind the Resource Manager.

- I/O drivers, as discussed in the Macintosh Operating System Reference Manual.

This manual begins with an introduction to the Desk Manager and desk accessories. Next, a section on using the Desk Manager introduces you to its routines and tells how they fit into the flow of your application. This is followed by the detailed descriptions of all Desk Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions is a section for programmers who want to define their own desk accessories.

Finally, there's a summary of the Desk Manager routine calls, for quick reference, and a glossary of terms used in this manual. *** The glossary will eventually be merged with the glossaries from the other Toolbox documentation. The many Operating System terms have not been included in the glossary in this manual. ***

ABOUT THE DESK MANAGER

The Desk Manager enables your application to support desk accessories, which are "mini-applications" that can be run at the same time as a Macintosh application. The standard Calculator desk accessory is shown in Figure 1. *** The method of highlighting an active desk accessory is currently different from what's shown here and will probably change. ***

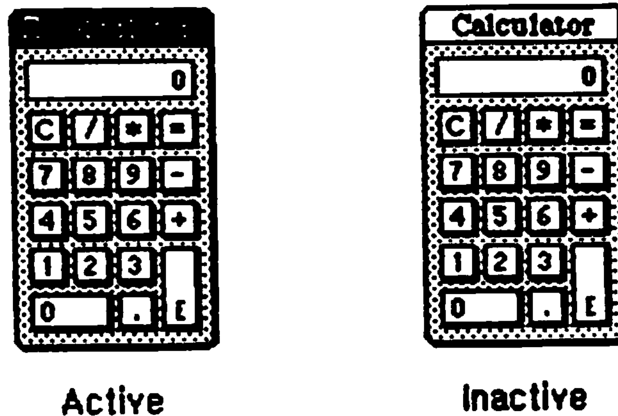


Figure 1. The Calculator Desk Accessory

The Macintosh user opens desk accessories by choosing them from the standard Apple menu (the menu whose title is an Apple symbol), which by convention is the first menu in the menu bar. When a desk accessory is chosen from this menu, it's usually displayed in a window on the desktop, and that window becomes the active window. (See Figure 2.)

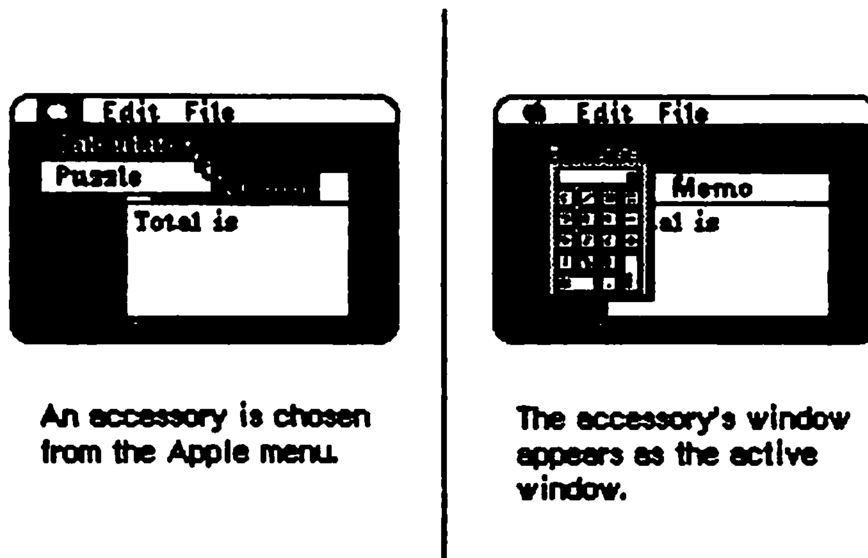


Figure 2. Opening a Desk Accessory

After being selected, the accessory may be used as long as it's active. The user can activate other windows and then reactivate the desk accessory by clicking inside it. Whenever a standard desk accessory is active, it has a close box in its title bar. Clicking the close box makes the accessory disappear, and the window that's then the frontmost becomes active.

The window associated with a desk accessory usually resembles a rounded-corner document window, as shown above. It also may look and behave like a dialog window; the accessory can call on the Dialog Manager to create the window and then use Dialog Manager routines to

operate on it. In either case, the window will be a system window, as indicated by its window class.

Many applications will have an Edit menu that includes the standard commands Cut, Copy, Paste, and Undo, which may be useful in desk accessories as well as in the application's windows. The Desk Manager provides a mechanism that lets those commands be applied to a desk accessory when it's active. Even if the commands aren't particularly useful for editing within the accessory, they may be useful for cutting and pasting between the accessory and the application or even another accessory. For example, the result of a calculation made with the Calculator desk accessory can be copied into a document prepared in MacWrite *** eventually ***.

A desk accessory may also have its own menu. When the accessory becomes active, the title of its menu is added to the menu bar and menu items may be chosen from it. Any of the application's menus or menu items that no longer apply are disabled. A desk accessory can even have an entire menu bar full of its own menus, which will completely replace the menus already in the menu bar. When an accessory that has its own menu or menus becomes inactive, the menu bar is restored to normal.

Although desk accessories are usually displayed in windows (one per accessory), this is not necessarily so. It's possible for an accessory to have only a menu (or menus) and not a window. The menu includes a command to close the accessory. Also, a desk accessory that's displayed in a window may create any number of additional windows while it's open.

You can define your own desk accessories. A desk accessory is actually a special type of I/O driver--special in that it may have its own windows and menus for interacting with the user. Desk accessories and other I/O drivers used by Macintosh applications are stored in resource files.

USING THE DESK MANAGER

This section introduces you to the Desk Manager routines and how they fit into the general flow of an application program. The routines themselves are described in detail in the next section.

To allow access to desk accessories, your application must do the following:

- Initialize TextEdit and the Dialog Manager, in case any desk accessories are displayed in windows created by the Dialog Manager (which uses TextEdit).

- Set up the Apple menu as the first menu in the menu bar. You can put the names of all currently available desk accessories in a

menu by using the Menu Manager routine AddResMenu (see the Menu Manager manual for details).

When the user chooses a menu item from the Apple menu, you should call the Menu Manager procedure GetItem to get the name of the corresponding desk accessory, and then the Desk Manager function OpenDeskAcc to open and display the accessory. You can close the desk accessory with the CloseDeskAcc procedure.

When the Toolbox Event Manager function GetNextEvent reports that a mouse down event has occurred, the application calls the Window Manager function FindWindow to find out where the mouse button was pressed. If FindWindow returns the predefined constant inSysWindow, which means that the mouse button was pressed in a system window, you should call the Desk Manager procedure SystemClick. SystemClick handles mouse down events in system windows, routing them to desk accessories where appropriate.

(hand)

> The application need not be concerned with exactly which desk accessories are currently open, except when it wants to use the accessory directly itself (such as the Mini-Finder accessory).

When the active window changes from an application window to a system window, the application should disable any of its menus or menu items that don't apply while an accessory is active. It should enable them again when one of its own windows becomes active.

When a mouse down event occurs in the menu bar, or a key down event occurs when the Command key is held down, and the application determines that one of the four standard editing commands Cut, Copy, Paste, and Undo has been invoked, it should call SystemEdit. Only if SystemEdit returns FALSE should the application process the editing command itself; if the active window belongs to a desk accessory, SystemEdit passes the editing command on to that accessory and returns TRUE.

Certain periodic actions may be defined for desk accessories. To see that they're performed, you need to call the SystemTask procedure at least once every time through your main event loop.

The two remaining Desk Manager routines--SystemEvent and SystemMenu--are never called by the application, but are described in this manual because they reveal inner mechanisms of the Toolbox that may be of interest to advanced Macintosh programmers.

DESK MANAGER ROUTINES

This section describes all the Desk Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language"

*** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Opening and Closing Desk Accessories

FUNCTION OpenDeskAcc (theAcc: Str255) : INTEGER;

OpenDeskAcc opens the desk accessory having the given name, displays its window (if any) as the active window, and returns its reference number (or 0 if the accessory can't be opened). The name is the accessory's resource name, which you get from the Apple menu by calling the Menu Manager procedure GetItem. OpenDeskAcc calls the Resource Manager to read the desk accessory from the resource file.

PROCEDURE CloseDeskAcc (refNum: INTEGER);

CloseDeskAcc closes the desk accessory having the given reference number. Usually, though, the application won't close the desk accessory; instead, it will be closed when the user clicks its close box (or, if there's a menu instead of a window, when the user chooses the command to close the accessory). Also, since the application heap is deallocated when the application terminates, every desk accessory goes away at that time.

Handling Events in Desk Accessories

PROCEDURE SystemClick (theEvent: EventRecord; theWindow: WindowPtr);

When a mouse down event occurs and the Window Manager routine FindWindow reports that the mouse button was pressed in a system window, the application should call SystemClick with the event record and the window pointer. If the given window belongs to a desk accessory, SystemClick sees that the event gets handled properly.

SystemClick determines which part of the desk accessory's window the mouse button was pressed in, and responds accordingly (similar to the way your application responds to mouse activities in its own windows).

- If the mouse button was pressed in the content region of the window and the window was active, SystemClick sends the mouse down event to the desk accessory, which processes it as appropriate.

- If the mouse button was pressed in the content region and the window was inactive, SystemClick makes it the active window.

- If the mouse button was pressed in the drag region, SystemClick calls the Window Manager routine DragWindow to pull an outline of

the window across the screen and move the window to a new location. If the window was inactive, DragWindow also makes it the active window (unless the Command key was pressed along with the mouse button).

- If the mouse button was pressed in the go-away region, SystemClick calls the Window Manager routine TrackGoAway to determine whether the mouse is still inside the go-away region when the click is completed: if so, it tells the desk accessory to close itself; otherwise, it does nothing.

FUNCTION SystemEdit (editCmd: INTEGER) : BOOLEAN;

Call SystemEdit when the user invokes the editing command specified by editCmd, which may be one of the following predefined constants:

```
CONST cutCmd   = 0;   {Cut command}
      copyCmd  = 1;   {Copy command}
      pasteCmd = 2;   {Paste command}
      undoCmd  = 3;   {Undo command}
```

If the active window doesn't belong to a desk accessory, SystemEdit returns FALSE; the application should then process the editing command as usual. If the active window does belong to a desk accessory, SystemEdit asks that accessory to process the command and returns TRUE; in this case, the application should ignore the command.

(hand)
It's up to the application to make sure desk accessories get their editing commands. In particular, make sure your application doesn't disable the Edit menu or any of the four commands when a desk accessory is activated.

Performing Periodic Actions

PROCEDURE SystemTask;

For each open desk accessory, SystemTask causes the accessory to perform the periodic action defined for it, if any such action has been defined and if the proper time period has passed since the action was last performed. For example, a clock accessory can be defined such that the second hand is to move once every second; the periodic action for the accessory will be to move the second hand to the next position, and SystemTask will alert the accessory every second to perform that action.

You should call SystemTask as often as possible, usually once every time through your main event loop. Call it more than once if your application does an unusually large amount of processing each time through the loop.

(hand)
Preferably SystemTask would be called at least every 60th of a second.

Advanced Routines

FUNCTION SystemEvent (theEvent: EventRecord) : BOOLEAN;

SystemEvent is called only by the Toolbox Event Manager routine GetNextEvent when it receives an event, to determine whether the event should be handled by the application or by the system. If the given event should be handled by the application, SystemEvent returns FALSE; otherwise, it calls the appropriate system code to handle the event and returns TRUE.

In the case of a null, abort, or mouse down event, SystemEvent does nothing but return FALSE. Notice that it responds this way to a mouse down event even though the event may in fact have occurred in a system window (and therefore may have to be handled by the system). The reason for this is that the check for exactly where the event occurred (via the Window Manager routine FindWindow) is made later by the application and so would be made twice if SystemEvent were also to do it. To avoid this duplication, SystemEvent passes the event on to the application and lets it make the sole call to FindWindow. Should FindWindow reveal that the mouse down event did occur in a system window, the application can then call SystemClick, as described above, to get the system to handle it.

If the given event is a mouse up, key down, key up, or auto-key event, SystemEvent checks whether the active window belongs to a desk accessory and whether that accessory can handle this type of event. If so, it sends the event to the desk accessory and returns TRUE; otherwise, it returns FALSE.

If SystemEvent is passed an activate or update event, it checks whether the window it occurred in is a system window belonging to a desk accessory and whether that accessory can handle this type of event. If so, it sends the event to the desk accessory and returns TRUE; otherwise, it returns FALSE.

(hand)
It's unlikely that a desk accessory would not be set up to handle activate and update events.

Finally, if the given event is a disk inserted event, SystemEvent does some low-level processing (by calling the Operating System routine MountVolume) but passes the event on to the application by returning FALSE, in case the application wants to do further processing.

PROCEDURE SystemMenu (menuResult: LongInt);

SystemMenu is called only by the Menu Manager routines MenuSelect and MenuKey, when an item in a menu belonging to a desk accessory has been chosen. The menuResult parameter has the same format as the value returned by MenuSelect and MenuKey: the menu ID in the high-order word and the menu item number in the low-order word. (The menu ID will be negative.) SystemMenu directs the desk accessory to perform the appropriate action for the given menu item.

DEFINING YOUR OWN DESK ACCESSORIES

To define your own desk accessories, you must create the corresponding I/O driver and include it in a resource file. Standard or shared desk accessories are stored in the system resource file. Accessories specific to an application are rare; if there are any, they're stored in the application's resource file.

The resource type for I/O drivers is 'DRVR'. The resource ID for a desk accessory is the driver's unit number and should be between 12 and 31 inclusive. The resource name should be whatever you want to appear in the Apple menu, but should also include a nonprinting character; by convention, the name should begin with a NUL character (ASCII code Ø). The nonprinting character is needed to avoid conflict with file names that are the same as the names of desk accessories.

The structure of an I/O driver is described in the Macintosh Operating System Reference Manual. The rest of this section reviews some of that information and presents additional details pertaining specifically to I/O drivers that are desk accessories.

(hand)

> Usually drivers are created entirely from assembly language, but you can use an assembly language-to-Pascal interface that will enable you to write the body of the driver routines in Pascal. An interface named ProtoOrn has been created for this purpose at Apple; for more information, see your Macintosh software coordinator.

As illustrated in Figure 3, the I/O driver begins with a few words of flags and other data for the driver, followed by offsets to the routines that do the work of the driver, an optional title, and finally the routines themselves.

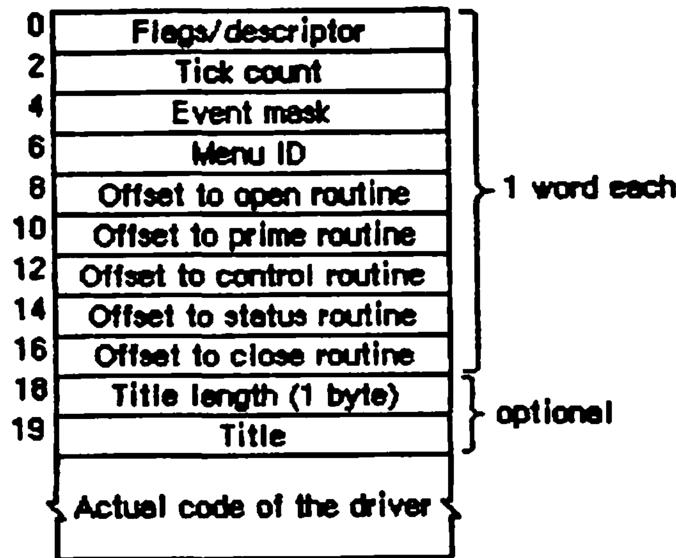


Figure 3. Desk Accessory I/O Driver

The first four words of the driver for a desk accessory contain the following:

1. A flags/descriptor word. Bits 0 through 7 and bit 12 are relevant only to ROM-based drivers; they're ignored for desk accessories. Bits 8 through 11 are the enable flags for the driver routines. The following flags are especially for desk accessories:

Flag	Name	Meaning if set
bit 13	dNeedTime	Driver needs time for performing a periodic action for the desk accessory
bit 14	dNeedLock	Driver will be locked in memory as soon as it's opened

   If you want to test one of these flags with the assembly-language instruction BTST, remember that when the destination of BTST is a memory location, the operation is performed on a **byte** read from that location.

2. If the dNeedTime flag is set, a tick count indicating how often the periodic action should occur. A tick count of 0 means it should happen as often as possible, 1 means it should happen every 60th of a second, 2 means every 30th of a second, and so on. The action itself is performed by the control routine in the driver when it's called by the SystemTask procedure.

3. An event mask specifying which events the desk accessory can handle. This should especially include update and activate events and usually will include mouse down events.

4. If the desk accessory has its own menu (or menus), the ID of the menu (or of any of the menus); otherwise, 0. The menu ID will be negative. For menus defined in resource files, it's the resource

ID; for menus created by the desk accessory, it's any negative number (between -1 and -32767) that you choose to identify this accessory's menu. It must be different from the menu ID stored here for other desk accessories.

Following these four words are the offsets to the driver routines and, optionally, a title for the desk accessory (preceded by its length in bytes). You can use the title in the driver as the title of the accessory's window, or just as a way of identifying the driver in memory.

The Device Control Entry

When any of the routines in the I/O driver is called, a pointer to the driver's Device Control Entry is passed in A1. Most of the data in the Device Control Entry is stored and accessed only by the Operating System, but in some cases the driver routines themselves must store into it. The structure of the Device Control Entry, which is discussed in detail in the Operating System manual, is illustrated in Figure 4. Notice that some of the data is taken from the first four words of the I/O driver.

Offset	Field	Size
0	Pointer to start of driver	long
4	Flags (from driver, plus some dynamic flags)	word
6	Driver input queue header: flags	word
8	Driver input queue header: QHead	long
12	Driver input queue header: QTail	long
16	Position pointer (position in file)	long
20	Handle to driver's private storage (optional)	long
24	Reference number for this driver	word
26	Counter for SystemTask timing	long
30	Pointer to driver's window (optional)	long
34	Tick count (from driver)	word
36	Event mask (from driver)	word
38	Menu ID (from driver)	word

Figure 4. Device Control Entry

The Driver Routines

Of the five possible driver routines, only three need to exist for desk accessories: the open, close, and control routines. The other routines (prime and status) may be used if desired for a particular accessory.

The open routine opens the desk accessory.

- It creates the window to be displayed when the accessory is opened, if any, specifying that it be invisible (since OpenDeskAcc will display it). The window can be created with the Dialog Manager routine NewDialog (or GetNewDialog) if desired; the accessory will look and respond like a dialog box, and subsequent operations may be performed on it with Dialog Manager routines. In any case, the open routine sets the windowKind field in the window record to the reference number for the driver, which it gets from the Device Control Entry. (The reference number will be negative.) It also may store the window pointer in the Device Control Entry if desired.

- If the driver has any private storage, it allocates the storage, stores a handle to it in the Device Control Entry, and initializes any local variables. It might, for example, create a menu or menus for the accessory.

The close routine closes the desk accessory, disposing of its window (if any) and replacing the window pointer in the Device Control Entry with NIL (if one was stored there by the open routine). If the driver has any private storage, the close routine also disposes of that storage.

The action taken by the control routine depends on information passed in the parameter block pointed to by A0. A message is passed in the "op code" field (a word located at 26(A0)); this message is simply a number that tells the routine what action to take. There are eight such messages:

Message	Name	Action to be taken by control routine
64	accEvent	Handle a given event
65	accRun	Take the periodic action, if any, for this desk accessory
66	accCursor	Change the cursor shape if appropriate; generate a null event if the window was created by the Dialog Manager
67	accMenu	Handle a given menu item
68	accCut	Handle the Cut command
69	accCopy	Handle the Copy command
70	accPaste	Handle the Paste command
71	accUndo	Handle the Undo command

Along with the accEvent message, the control routine receives as a parameter a pointer to an event record (a long integer located at 28(A0)). It responds by handling the given event in whatever way is appropriate for this desk accessory. SystemClick and SystemEvent call the control routine with this message to send the driver an event that it should handle--for example, an activate event that makes the desk accessory active or inactive. When a desk accessory becomes active, its control routine might install a menu in the menu bar. If the accessory becoming active has more than one menu, the control routine should respond as follows:

- Store the accessory's unique menu ID in the system global mBarEnable. (This is the negative menu ID in the I/O driver and the Device Control Entry.)

- Call the Menu Manager routines GetMenuBar to save the current menu list and ClearMenuBar to clear the menu bar.

- Install the accessory's own menus in the menu bar.

Then, when the desk accessory becomes inactive, the control routine should call SetMenuBar to restore the former menu list, call DrawMenuBar to draw the menu bar, and set mBarEnable to Ø.

The accRun message tells the control routine to perform the periodic action for this desk accessory. For every open driver that has the dNeedTime flag set, the SystemTask procedure calls the control routine with this message if the proper time period has passed since the action was last performed.

The accCursor message makes it possible for the cursor to have a special shape when it's inside an active desk accessory. The control routine is called repeatedly with this message as long as the desk accessory is active. If desired, the control routine may respond by checking whether the mouse position is in the desk accessory's window and then changing the shape of the cursor if so. Furthermore, if the desk accessory is displayed in window created by the Dialog Manager, the control routine should respond to the accCursor message by generating a null event (storing the event code for a null event in an event record) and passing it to DialogSelect. This enables the Dialog Manager to blink the vertical bar in editText items.

(hand)

In assembly language, the code might look like this:

```
CLR.L    -SP          ; event code for null event is Ø
PEA      2(SP)        ; pass null event
CLR.L    -SP          ; pass NIL dialog pointer
CLR.L    -SP          ; pass NIL pointer
_DialogSelect         ; invoke DialogSelect
ADDQ.L   #4,SP        ; pop off result and null event
```

When the accMenu message is sent to the control routine, the following information is passed in the parameter block: the menu ID of the desk accessory's menu in a word at 28(AØ), and a menu item number in a word at 3Ø(AØ). The control routine takes the appropriate action for when the given menu item is chosen from the menu, and then makes the Menu Manager call HiliteMenu(Ø) to remove the highlighting from the menu bar.

Finally, the control routine should respond to one of the last four messages--accCut through accUndo--by processing the corresponding editing command in the desk accessory window if appropriate. SystemEdit calls the control routine with these messages. For information on cutting and pasting between a desk accessory and the

application, or between two desk accessories, see the *** forthcoming *** Scrap Manager manual.

(hand)

> If you use .INCLUDE to include a file named SysEqu.Text when you assemble your program, the messages sent to the driver's control routine will be available in symbolic form, as will offsets into the fields of the I/O driver and Device Control Entry.

A Sample Desk Accessory

*** to be supplied; meanwhile, see your Macintosh software coordinator ***

SUMMARY OF THE DESK MANAGER

```
CONST cutCmd   = 0;   {Cut command}
      copyCmd  = 1;   {Copy command}
      pasteCmd = 2;   {Paste command}
      undoCmd  = 3;   {Undo command}
```

Opening and Closing Desk Accessories

```
FUNCTION  OpenDeskAcc  (theAcc: Str255) : INTEGER;
PROCEDURE CloseDeskAcc (refNum: INTEGER);
```

Handling Events in Desk Accessories

```
PROCEDURE SystemClick (theEvent: EventRecord; theWindow: WindowPtr);
FUNCTION  SystemEdit  (editCmd: INTEGER) : BOOLEAN;
```

Performing Periodic Actions

```
PROCEDURE SystemTask;
```

Advanced Routines

```
FUNCTION  SystemEvent (theEvent: EventRecord) : BOOLEAN;
PROCEDURE SystemMenu  (menuResult: LongInt);
```

GLOSSARY

desk accessory: A "mini-application", implemented as an I/O driver, that can be run at the same time as a Macintosh application.

tick: A 60th of a second.

MACINTOSH USER EDUCATION

The Device Manager: A Programmer's Guide

/DMGR/DEVICE

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Inside Macintosh: A Road Map

Modification History: First Draft (ROM 7) B. Hacker 2/dd/84

*** Review Draft. Not for distribution ***

ABSTRACT

This manual describes the Device Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and devices.

TABLE OF CONTENTS

3 About This Manual
4 About the Device Manager
7 Using the Device Manager
8 Device Manager Routines
8 Routines For Opening and Closing Drivers
9 High-Level Device Manager Routines
11 Low-Level Device Manager Routines
12 Routine Parameters
14 Routine Descriptions
19 The Structure of a Device Driver
20 A Device Control Entry
22 The Unit Table
23 Writing Your Own Device Drivers
24 Routines for Writing Drivers
66 Interrupts
66 Level-1 (VIA) Interrupts
66 Level-2 (SCC) Interrupts
66 Writing Your Own Interrupt Handlers
66 A Sample Driver
68 Summary of the Device Manager
75 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Device Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and devices. *** Eventually it will become part of a larger manual describing the entire Toolbox and Operating System. *** General information about using device drivers can be found in this manual; specific information about the standard Macintosh drivers is contained in separate manuals.

(eye)

> This manual describes version 7 of the ROM. If you're using a different version, the Device Manager may not work as discussed here.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager
- application data buffers, as described in the Macintosh Operating System's File Manager manual

The manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

This manual begins with an introduction to the Device Manager and what you can do with it. It then discusses some basic concepts behind the Device Manager: what devices and drivers are, and how they are used.

A section on using the Device Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all the procedures and functions used to call device drivers, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that provide information for programmers who want to write their own drivers, including a discussion of interrupts and a sample driver.

Finally, there's a summary of the Device Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE DEVICE MANAGER

The Device Manager is the part of the Operating System that handles communication between applications and devices. A device is a part of the Macintosh, or a piece of external equipment, that can transfer information into or out of the Macintosh. Macintosh devices include the keyboard, screen, disk drives, two asynchronous serial ports, the sound generator, the mouse, and printers.

There are two kinds of devices: character devices and block devices. A character device reads or writes a stream of characters, one at a time: it can neither skip characters nor go back to a previous character. A character device is used to get information from or send information to the world outside of the Macintosh Operating System and memory: it can be an input device, an output device, or an input/output device. The mouse, keyboard, screen, sound generator, and printers, are all character devices.

A block device reads and writes blocks of 512 characters at a time; it can read or write any accessible block on demand. A block device is used to store and retrieve information: it's always an input/output device. Disk drives are block devices.

Applications communicate with devices by calling Device Manager routines. The Device Manager routines don't manipulate devices, but they call device drivers that do. Device drivers are programs that take streams or blocks of characters coming from the Device Manager and convert them into actions of devices, and convert device actions into streams or blocks of characters for the Device Manager to process.

All information exchange between the Device Manager and devices occurs via drivers; the Device Manager never communicates directly with a device (see Figure 1).

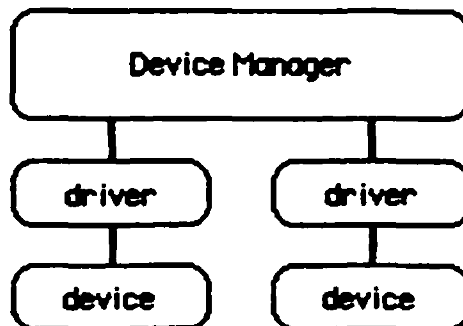


Figure 1. Communication with Devices

The Operating System includes three standard device drivers in ROM: the Disk Driver, the Sound Driver, and the Serial Driver. There are also a number of standard RAM drivers that are read from the system resource file when the system starts up: the Printer Driver and desk accessories. The keyboard and mouse don't have drivers, and are

handled via the Keyboard/Mouse Handler. Other drivers can be added independently or built on the existing drivers (for example, the Printer Driver is built on top of the Serial Driver); the section "Writing Your Own Device Drivers" describes how to do this. Desk accessories are a special type of device driver, in that they have windows, their name should appear in the Apple menu, and they are manipulated via the specialized routines of the Desk Manager. Information about desk accessories covered in the Desk Manager manual will not be repeated here.

A driver can be either open or closed. After a driver has been opened, an application can read information from and write information to the driver. Drivers that are no longer in use can be closed, and the memory used by them recovered. The standard Macintosh drivers are opened when the system starts up. Up to 32 drivers may be open at any one time.

A driver is identified by its driver name and, after it's opened, by its reference number. A driver name consists of a period (.) followed by any sequence of 1 to 255 printing characters. You can use uppercase and lowercase letters when naming drivers, but the Device Manager ignores case when comparing names (it doesn't ignore diacritical marks).

(hand)

> Although driver names can be quite long, there's little reason for them to be more than a few characters in length. Normally the user will never see a driver name unless it's displayed in a menu, and names in menus should be short enough that the menu doesn't become excessively wide.

The Device Manager assigns each open driver a driver reference number, from -1 to -32, that is used instead of its driver name to refer to it.

In addition to data that's read from or written to drivers, drivers may require or provide other information. Required information transmitted to a driver by an application is called control information; information provided by a driver is called status information. Control information may select modes of operation, start or stop processes, enable buffers, choose protocols, and so on. Status information may indicate the current mode of operation, the readiness of the device, the occurrence of errors, and so on.

Each driver may respond to a number of different types of control information and may provide a number of different types of status information. The standard Macintosh drivers receive control information and provide status information via a predefined data structure, of type OpParamType:

```
TYPE OpParamPtr = ^OpParamType;

     OpParamType = RECORD
           CASE OpVariant OF
                {control information}
                     sound:                     {Sound Driver}
                       (sndVal: INTEGER);
                     asyncRst:                  {Async Driver}
                       (asncConfig: INTEGER);
                     asyncInBuff:
                       (asncBPtr: Ptr;
                        asncBLen: INTEGER);
                     asyncShk:
                       (asncHndShk: LongInt;
                        asncMisc: LongInt);
                     printer:                   {Printer Driver}
                       (param1: LongInt;
                        param2: LongInt;
                        param3: LongInt);
                     fontMgr:                   {Font Manager}
                       (fontRecPtr: Ptr;
                        fontCurDev: INTEGER);
                     diskDrv:                   {Disk Driver}
                       (diskBuff: Ptr);
                {status information}
                     asyncBuffBytes:            {Async Driver}
                       (asyncNBytes: LongInt);
                     asyncStatus:
                       (asncS1: INTEGER;
                        asncS2: INTEGER;
                        asncS3: INTEGER);
                     diskStat:                  {Disk Driver}
                       (dskTrackLock: INTEGER;
                        dskInfoBits:  LongInt;
                        dskQElem:     drvrQElRec;
                        dskPrime:     INTEGER;
                        dskErrCnt:    INTEGER);
                 END;
```

The CASE statement selects which field(s) of the record will be used, based on the OpVariant data type:

```
TYPE OpVariant = (sound, asyncRst, asyncInBuff, asyncShk, printer,
                  fontMgr, diskDrv, asyncBuffBytes, asyncStatus,
                  diskStat);
```

The maximum size of the OpParamType variant record is 22 bytes. Explanations of the fields can be found in the manuals describing the

different drivers.

USING THE DEVICE MANAGER

This section discusses how the Device Manager routines fit into the general flow of an application program and gives an idea of what routines you'll need to use. The routines themselves are described in detail in the next section.

The Device Manager routines can be called via three different methods: high-level Pascal calls, low-level Pascal calls, and assembly language. The high-level Pascal calls are designed for Pascal programmers interested in using the Device Manager in a simple manner; they provide adequate device I/O and don't require much special knowledge to use. The low-level Pascal and assembly-language calls are designed for advanced Pascal programmers and assembly-language programmers interested in using the Device Manager to its fullest capacity; they require some special knowledge to be used most effectively.

(hand)

> The names used to refer to routines here are actually assembly-language macro names, but the Pascal routine names are very similar.

The Device Manager is automatically initialized each time the system is started up.

Before an application exchanges information with a driver, the driver must be opened. ROM drivers are opened when the system starts up; for RAM drivers, call Open. (Desk accessories use OpenDeskAcc.) The Device Manager will return the driver reference number that you'll use every time you want to refer to that driver.

You can transfer data from an open driver to an application's data buffer with Read, and send data from an application's data buffer to a driver with Write. An application passes control information to a driver by calling Control, and receives status information from a driver by calling Status.

Whenever you want to stop a driver from completing I/O initiated by a Read, Write, Control, or Status call, call KillIO. KillIO halts the currently executing I/O, and ignores any pending I/O (if any).

When you're through using a driver, call Close. (Desk accessories use CloseDeskAcc.) Close forces the driver to complete any pending I/O, and then deallocates all the memory used by the driver. *** Currently, you shouldn't close the Serial Driver. ***

Advanced programmers who write their own device drivers may find the Desk Manager routines SystemClick, SystemEdit, and SystemTask to be of use.

DEVICE MANAGER ROUTINES

This section is divided into three parts that describe routines used to call drivers. The first presents the two routines used to open and close drivers; this part must be read by all programmers. The second describes all the high-level Pascal routines of the Device Manager, and the third presents information about calling the low-level Pascal and assembly-language routines.

All Device Manager routines return a result code of type OSErr. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Routines For Opening and Closing Drivers

FUNCTION OpenDriver (name: OSStr255; VAR refNum: INTEGER) : OSErr;

OpenDriver opens the driver specified by name and returns its reference number in refNum.

Result codes		
	noErr	No error
	resErr	Resource Manager error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	openErr	Driver cannot perform reading or writing
	unitEmptyErr	Bad reference number

FUNCTION CloseDriver (refNum: INTEGER) : OSErr;

CloseDriver closes the driver having the reference number refNum. Any pending I/O is completed, and the memory used by the driver is deallocated.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	resErr	Resource Manager error
	unitEmptyErr	Bad reference number

High-Level Device Manager Routines

```
FUNCTION FSRead (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
         OSErr;
```

FSRead attempts to read the number of bytes specified by the count parameter from the driver having the reference number refNum, and transfer them to the data buffer pointed to by buffPtr. After the read is completed, the number of bytes actually read is returned in the count parameter.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	readErr	Driver isn't enabled for read calls

```
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
         OSErr;
```

FSWrite attempts to take the number of bytes specified by the count parameter from the buffer pointed to by buffPtr and write them to the open driver having the reference number refNum. After the write is completed, the number of bytes actually written is returned in the count parameter.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	writErr	Driver isn't enabled for write calls

FUNCTION FSControl (refNum: INTEGER; opCode: INTEGER; opParams:
OpParamPtr) : OSErr;

FSControl sends control information to the driver having the reference number refNum. The type of information sent is specified by opCode, and the information itself is pointed to by opParams. The values passed in opCode and pointed to by opParams depend on the driver being called.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	controlErr	Driver isn't enabled for control calls

FUNCTION FSStatus (refNum: INTEGER; opCode: INTEGER; opParams:
OpParamPtr) : OSErr;

FSStatus returns status information about the driver having the reference number refNum. The type of information returned is specified by opCode, and the information itself is pointed to by opParams. The values passed in opCode and pointed to by opParams depend on the driver being called.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	statusErr	Driver isn't enabled for status calls

FUNCTION FSKillIO (refNum: INTEGER) : OSErr;

FSKillIO terminates all I/O with the driver having the reference number refNum.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dSIOCoreErr	Device control entry was purged
	unitEmptyErr	Bad reference number
	controlErr	Driver isn't enabled for control calls

Low-Level Device Manager Routines

This section contains special information for programmers using the low-level Pascal or assembly-language routines of the Device Manager, and then describes the routines in detail.

All Device Manager routines described in this section can be executed either synchronously (meaning that the application must wait until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is executing).

When a Device Manager routine is called asynchronously, an I/O request is placed in the driver's I/O queue, and control returns to the calling application--even before the actual I/O is completed. Requests are taken from the queue one at a time (in the same order that they were entered), and processed. Only one request may be processed at any given time.

The calling application may specify a completion routine to be executed as soon as the I/O operation has been completed.

Routine parameters passed by an application to the Device Manager and returned by the Device Manager to an application are contained in a parameter block, which is memory space in the heap or stack. All low-level Pascal calls to the Device Manager are of the form

```
PBCallName (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

PBCallName is the name of the routine. ParamBlock points to the parameter block containing the parameters for the routine. If async is TRUE, the call will be executed asynchronously; if FALSE, it will be executed synchronously.

Assembly-language note: All Device Manager routines are called with AØ pointing to a parameter block containing the parameters for the routine, and A1 pointing to the driver's device control entry. All routines return with DØ containing a result code.

You specify whether a routine will be executed synchronously or asynchronously by clearing or setting bit 1Ø of the routine trap instruction, as described in the Using Assembly Language manual *** doesn't exist yet ***.

Routine Parameters

The lengthy, variable-length data structure of a parameter block is given below. The ~~Device Manager and~~ File Manager use this same data structure, but only the parts relevant to the Device Manager are discussed here. Each kind of parameter block contains eight fields of standard information and two to nine fields of additional information:

```
TYPE ParamBlkType = (ioParam, fileParam, volumeParam, controlParam);

     ParamBlockRec = RECORD
                       ioLink:       Ptr;      {next queue entry}
                       ioType:       INTEGER;  {always 5}
                       ioTrap:       INTEGER;  {routine trap}
                       ioCmdAddr:    Ptr;      {routine address}
                       ioCompletion: ProcPtr;  {completion routine}
                       ioResult:     OSErr;    {result code}
                       ioNamePtr:    OSStrPtr; {driver name}
                       ioVRefNum:    INTEGER;  {not used}
                       CASE ParamBlkType OF
                         ioParam:
                         . . .  {I/O routine parameters}
                         fileParam:
                         . . .  {file information routine parameters}
                         volumeParam:
                         . . .  {volume information routine parameters}
                         controlParam:
                         . . .  {Control and Status routine parameters}
                     END;

     ParmBlkPtr = ^ParamBlockRec;
```

The first four fields in each parameter block are handled entirely by the Device Manager, and most programmers needn't be concerned with them; programmers who are interested in them should see the section "The Structure of a Driver".

IOCompletion contains the address of a completion routine to be executed at the end of an asynchronous call; it should be NIL for asynchronous calls with no completion routine, and is automatically set to NIL for all synchronous calls. For asynchronous calls, ioResult is positive while the routine is executing, and returns the result code.

IONamePtr is a pointer to the name of a driver.

An 8-field parameter block is adequate for opening a driver, but most of the Device Manager routines require longer parameter blocks, as described below. The parameters used with file and volume information routines are described in the File Manager manual.

Control and Status routines use two additional fields:

```
controlParam:
  csCode:  INTEGER;     {type of Control or Status call}
  csParam: OpParamType; {control or status information}
```

CSCode contains a number identifying the type of call. This number may be interpreted differently by each driver. CSParam contains the control or status information for the call.

I/O routines use seven additional fields:

```
ioParam:
  ioRefNum:    INTEGER;    {driver reference number}
  ioVersNum:   SignedByte; {not used}
  ioPermssn:   SignedByte; {read/write permission}
  ioMisc:      Ptr;        {not used}
  ioBuffer:    Ptr;        {data buffer}
  ioReqCount:  LongInt;    {requested number of bytes}
  ioActCount:  LongInt;    {actual number of bytes}
  ioPosMode:   INTEGER;    {type of positioning operation}
  ioPosOffset: LongInt;    {size of positioning offset}
```

IOPermssn requests permission to read from or write to a driver:

IOPermssn	I/O operation
Ø	Whatever the driver is capable of doing
1	Reading only
2	Writing only
3	Reading and writing

This request is compared with the capabilities of the driver (some drivers are read-only, some are write-only). If the driver is incapable of performing as requested, an error will be returned.

IOBuffer points to an application's data buffer into which data is written by Read calls and from which data is read by Write calls. IOReqCount specifies the requested number of bytes to be read or written. IOActCount contains the number of bytes actually read or written.

Advanced programmers: IOPosMode and ioPosOffset contain positioning information used for Read and Write calls by drivers of block devices. Bits Ø and 1 of ioPosMode indicate a byte position beyond the physical beginning of the block-formatted medium (such as a disk):

IOPosMode	Offset
Ø	None
1	Relative to beginning of device
2	None
3	Relative to current position

IOPosOffset specifies the byte offset beyond ioPosMode where the operation is to be performed.

Routine Descriptions

This section describes the procedures and functions. Each routine description includes the low-level Pascal form of the call and the routine's assembly-language macro. A list of the fields in the parameter block affected by the call is also given.

Assembly-language note: The field names given in these descriptions are those of the ParamBlockRec data type; see "Summary of the Device Manager" for the equivalent assembly-language equates.

The number next to each parameter name indicates the byte offset of the parameter from the start of the parameter block pointed to by A0; only assembly-language programmers need be concerned with it. An arrow drawn next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
<--	Parameter is passed to the routine
-->	Parameter is returned by the routine
<->	Parameter is passed to and returned by the routine

FUNCTION PBRead (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Read

Parameter block

<--	12	ioCompletion	pointer
-->	16	ioResult	word
<--	24	ioRefNum	word
<--	32	ioBuffer	pointer
<--	36	ioReqCount	long word
-->	40	ioActCount	long word
<--	44	ioPosMode	word
<->	46	ioPosOffset	long word

Result codes

noErr	No error
badUnitErr	Bad reference number
dSIOCoreErr	Device control entry was purged
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
readErr	Driver isn't enabled for read calls

PBRead attempts to read ioReqCount bytes from the driver having the reference number ioRefNum, and transfer them to the data buffer pointed to by ioBuffer. After the read operation is completed, the number of bytes actually read is returned in ioActCount.

Advanced programmers: If the driver is reading from a block device, the byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset.

FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Write

Parameter block

<--	12	ioCompletion	pointer
-->	16	ioResult	word
<--	24	ioRefNum	word
<--	32	ioBuffer	pointer
<--	36	ioReqCount	long word
-->	40	ioActCount	long word
<--	44	ioPosMode	word
<--	46	ioPosOffset	long word

Result codes

noErr	No error
badUnitErr	Bad reference number
dSIOCoreErr	Device control entry was purged
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
writErr	Driver isn't enabled for write calls

PBWrite attempts to take ioReqCount bytes from the buffer pointed to by ioBuffer and write them to the driver having the reference number ioRefNum. After the write operation is completed, the number of bytes actually written is returned in ioActCount.

Advanced programmers: If the driver is writing to a block device, ioPosMode indicates whether the write should begin relative to the beginning of the device or the current position. The byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset.

FUNCTION PBControl (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Control

Parameter block

<--	12	ioCompletion	pointer
-->	16	ioResult	word
<--	24	ioRefNum	word
<--	26	csCode	word
<--	28	csParam	record

Result codes

noErr	No error
badUnitErr	Bad reference number
dSIOCoreErr	Device control entry was purged
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
controlErr	Driver isn't enabled for control calls

PBControl sends control information to the driver having the reference number refNum. The type of information sent is specified by csCode, and the information itself is pointed to by csParam. The values passed in csCode and pointed to by csParam depend on the driver being called.

FUNCTION PBStatus (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Status

Parameter block

<--	12	ioCompletion	pointer
-->	16	ioResult	word
<--	24	ioRefNum	word
<--	26	csCode	word
<--	28	csParam	variable

Result codes

noErr	No error
badUnitErr	Bad reference number
dSIOCoreErr	Device control entry was purged
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
statusErr	Driver isn't enabled for status calls

PBStatus returns status information about the driver having the reference number refNum. The type of information returned is specified by csCode, and the information itself is pointed to by csParam. The values passed in csCode and pointed to by csParam depend on the driver being called.

FUNCTION PBKillIO (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _KillIO

Parameter block

<--	12	ioCompletion	pointer
-->	16	ioResult	word
<--	24	ioRefNum	word
<--	26	csCode	word
<--	28	csParam	variable

Result codes

noErr	No error
badUnitErr	Bad reference number
dSIOCoreErr	Device control entry was purged
unitEmptyErr	Bad reference number
controlErr	Driver isn't enabled for control calls

FSKillIO stops any current I/O request being processed, and removes all pending I/O requests from the I/O queue of the driver having the reference number refNum. The completion routine of each pending I/O request is executed.

THE STRUCTURE OF A DRIVER

This section and the next describe the structure of drivers and how to write device drivers. If this information doesn't interest you, skip ahead to the summary.

RAM drivers are stored in resource files. Drivers that will be used by more than one application should be stored in the system resource file, while those specific to an application should be stored in the application's resource file.

The resource type for drivers is 'DRVR'. The resource ID for a driver is its unit number (explained below) and should be between Ø and 31 inclusive. (The resource ID for a desk accessory must be greater than 11.) Don't use numbers of existing drivers unless you want the existing driver to be replaced. The resource name should match the driver name. (The resource name for a desk accessory must contain a nonprinting character.)

As illustrated in Figure 2, a driver begins with a few words of flags and other data, followed by offsets to the routines that do the work of the driver, an optional title, and finally the routines themselves.

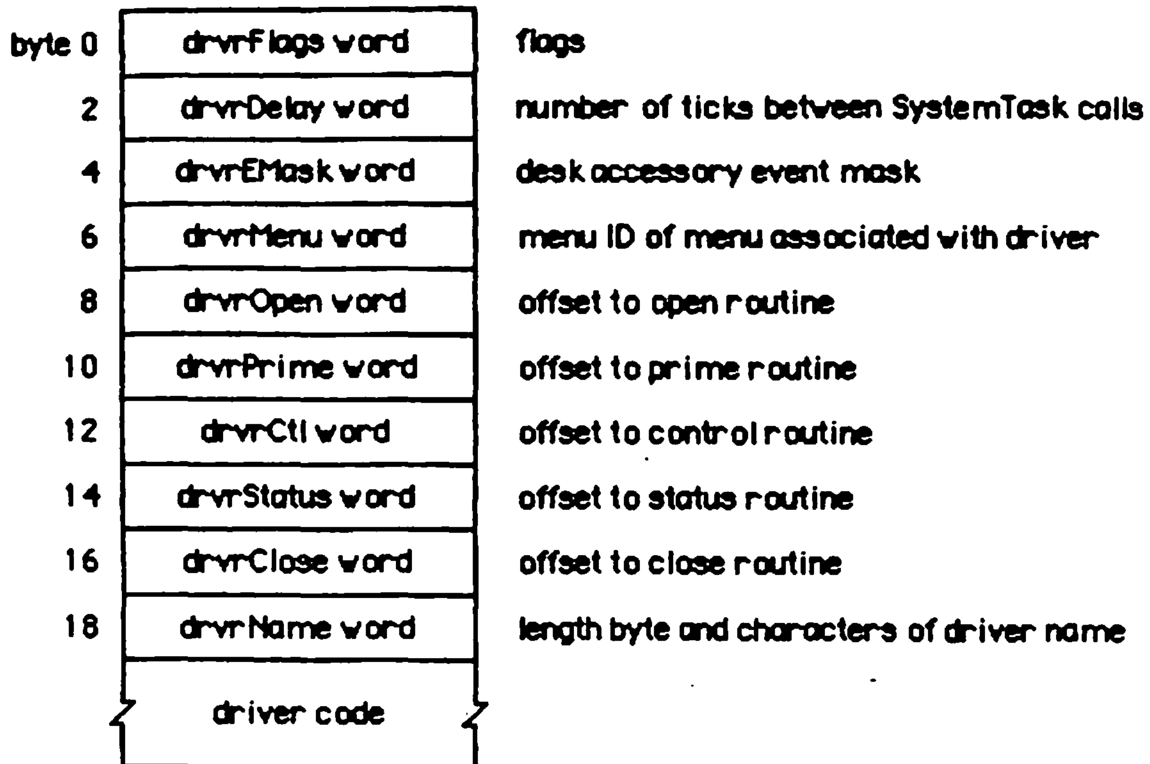


Figure 2. Driver Structure

The drvrFlags word contains the following:

Flag	Name	Meaning if set
bit 8	dReadEnable	Driver enabled for Read calls
bit 9	dWriteEnable	Driver enabled for Write calls
bit 1Ø	dCtlEnable	Driver enabled for Control calls
bit 11	dStatEnable	Driver enabled for Status calls
bit 12	dNeedGoodBye	Driver needs to be called prior to application heap compactions
bit 13	dNeedTime	Driver needs time for performing a periodic action
bit 14	dNeedLock	Driver will be locked in memory as soon as it's opened (always set for ROM drivers)

Bits 8 through 11 are the enable flags for the driver routines. Each flag that corresponds to a Device Manager call that the driver can respond to must be set.

RAM drivers that exist on the application heap will be destroyed every time the heap is compacted (when an application starts up, for example). If dNeedGoodBye is set, the control routine of the driver will be called before the heap is compacted, and the driver can perform any "clean-up" actions it needs to. The driver's control routine can identify this "good-bye" call by checking the csCode parameter--it will be -1.

If the dNeedTime flag is set, the drvrDelay word contains a tick count indicating how often the periodic action should occur. A tick count of Ø means it should happen as often as possible, 1 means it should happen every 6Øth of a second, 2 means every 3Øth of a second, and so on. The action itself is performed by the control routine of the driver when it's called by the Device Manager procedure SystemTask. The driver's control routine can identify this periodic-action call by checking the csCode parameter—it will be accRun. Normally only desk accessories will use dNeedTime and drvrDelay.

DrvrEMask is used only for desk accessories and is discussed in the Desk Manager manual. If the driver has its own menu (or menus), drvrMenu contains the ID of the menu (or one of the menus); otherwise it contains Ø. Normally only desk accessories have menus.

Following these four words are the offsets to the driver routines, a title for the driver (preceded by its length in bytes), and the routines that do the work of the driver.

A Device Control Entry

The first time a driver is opened, information about it is read into a structure in memory called a device control entry. A device control entry tells the Device Manager the location of the driver's routines, the location of the driver's I/O queue, and other information. A device control entry is a 4Ø-byte relocatable block located in the system communication area of the heap. It's locked while the driver is open, and unlocked and purgeable while the driver is closed.

The structure of a device control entry is illustrated in Figure 3. Notice that some of the data is taken from the first four words of the driver. Most of the data in the device control entry is stored and accessed only by the Device Manager, but in some cases the driver itself must store into it.

Byte	Field	Description
byte 0	dCtlDriver long word	pointer to ROM driver or handle to RAM driver
4	dCtlFlags word	flags
6	dCtlQueue word	not used
8	dCtlQHead pointer	pointer to first entry in driver's I/O queue
12	dCtlQTail pointer	pointer to last entry in driver's I/O queue
16	dCtlPosition long word	byte position used by Read and Write calls
20	dCtlStorage handle	handle to driver's private storage
24	dCtlRefNum word	driver's reference number
26	dCtlCurTicks long word	counter for timing SystemTask calls
30	dCtlWindow pointer	pointer to driver's window record (if any)
34	dCtlDelay word	number of ticks between SystemTask calls
36	dCtlEMask word	desk accessory event mask
38	dCtlMenu word	menu ID of menu associated with driver

Figure 3. Device Control Entry

The dCtlFlags word contains the following (bits 8 through 14 are copied from the drvrFlags word of the driver):

Flag	Name	Meaning if set
bit 5	dOpened	Driver is open
bit 6	dRAMBased	Driver is RAM-based
bit 7	drvrActive	Driver is currently executing
bit 8	dReadEnable	Driver enabled for Read calls
bit 9	dWritEnable	Driver enabled for Write calls
bit 1Ø	dCtlEnable	Driver enabled for Control calls
bit 11	dStatEnable	Driver enabled for Status calls
bit 12	dNeedGoodBye	Driver needs to be called prior to application heap compactions
bit 13	dNeedTime	Driver needs time for performing a periodic action
bit 14	dNeedLock	Driver will be locked in memory as soon as it's opened (always set for ROM drivers)

DCtlPosition is used only by drivers of block devices, and indicates the current source or destination position of a Read or Write call.

The position is given in number of bytes beyond the physical beginning of the medium used by the device. For example, if one logical block of data has just been read from a 3 1/2-inch disk via the Disk Driver, dCtlPosition would be 512.

ROM drivers generally use low-memory reserved locations for their local storage. RAM drivers may reserve space within their code space, or allocate a relocatable block and keep a handle to it in dCtlStorage (this memory is locked when the driver is opened, and unlocked when the driver is closed).

DCtlCurTicks is used by the Device Manager to time SystemTask calls (if any were indicated by the dNeedTime flag in the driver).

The Unit Table

The location of each device control entry is maintained in a list called the unit table. The unit table is a 128-byte relocatable block containing 32 4-byte entries. Each entry has a number, from Ø to 31, called the unit number, and contains a handle to the device control entry for a driver. The unit number can be used as an index into the unit table to locate the handle to a specific driver's device control entry; it's equal to minus (the driver's reference number - 1). For example, the Sound Driver's reference number is -4, its resource ID is 3.

Figure 4 shows the layout of the unit table created at startup time with the standard Macintosh drivers.

byte		unit number
0	not used	0
4	not used	1
8	Printer Driver	2
12	Sound Driver	3
16	Disk Driver	4
20	Serial Driver port A input	5
24	Serial Driver port A output	6
28	Serial Driver port B input	7
32	Serial Driver port B output	8
	not used	
48	Calculator	12
52	Alarm Clock	13
56	Key Caps	14
60	Puzzle	15
64	Note Pad	16
68	Scrapbook	17
72	Control Panel	18
	not used	
124	not used	31

Figure 4. The Unit Table

Assembly-language note: The system global uTableBase points to the unit table.

Each driver contains an I/O queue with a list of routines to be executed by the driver. There's one I/O queue for each device driver

(Figure 5). The queue's header is located in the device control entry for the driver.

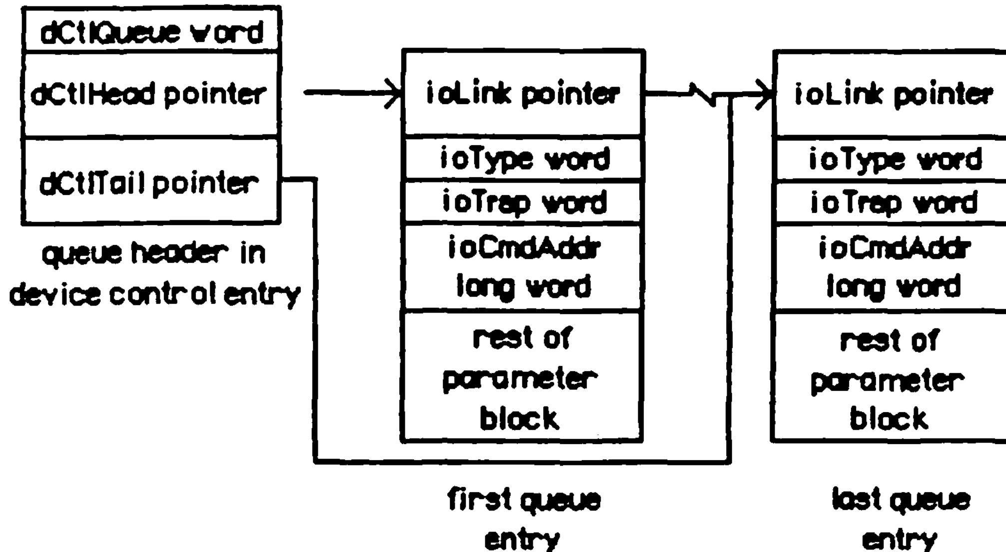


Figure 5. I/O Queue Structure

DCtlHead points to the first entry in the queue, and dCtlTail points to the last entry in the queue. Each queue entry consists of a parameter block for the routine called--an abbreviation of which is given below:

```
TYPE ParamBlockRec = RECORD
                       ioLink:     Ptr;      {next entry}
                       ioType:     INTEGER;  {always ioQType}
                       ioTrap:     INTEGER;  {routine trap}
                       ioCmdAddr:  Ptr;      {rest of block}
                         . . .
                     END;
```

IOLink points to the next entry in the queue, and ioType indicates the queue type, which must be the value of the system global ioQType or 2. IOCmdAddr contains the address of the Device Manager routine called. IOTrap contains the trap (of the form $AXnn) of the routine called. The following system globals identify Device Manager traps:

Name	Value	Trap	Routine
aRdCmd	2	$A002	Read
aWrCmd	3	$A003	Write
aCtlCmd	4	$A004	Control
aStsCmd	5	$A005	Status

WRITING YOUR OWN DEVICE DRIVERS

This section describes what you'll need to do to write your own device driver. The structure of the driver must match that shown in the previous section. The routines that do the work of the driver should be written to operate the device in whatever way you require.

Your driver must contain routines to handle Open and Close calls, and may choose to handle Read, Write, Control, Status, and KillIO calls as well. The driver routines that the Device Manager will execute when one of these calls is made are as follows:

Device Manager call	Driver routine
Open	Open
Read	Prime
Write	Prime
Control	Control
KillIO	Control
Status	Status
Close	Close

When the Device Manager executes a driver routine to handle an application call, it passes a pointer to the call's parameter block in AØ, a pointer to the driver's device control entry in A1, and Ø in DØ. From this information, the driver can determine exactly what operations are required to fulfill the call's requests, and do them.

Open and close routines must be executed synchronously. They needn't preserve any registers that they use. Open and Close routines should place a result code in DØ and return via an RTS.

The open routine must allocate any private storage required by the driver, store a handle to it in the device control entry (in the dCtlStorage field), initialize any local variables, and then be ready to receive a Read, Write, Status, Control, or KillIO call. It might also install interrupt handlers, change interrupt vectors, and store a pointer to the device control entry somewhere in its local storage for its interrupt handlers to use. The close routine must reverse the effects of the open routine, by deallocating all used memory, removing interrupt handlers, and replacing changed interrupt vectors. If anything about the operational state of the driver should be saved until the next time the driver is opened, it should be kept in the relocatable block of memory pointed to by dCtlStorage.

Prime, control, and status routines are queueable (in other words, they can be executed asynchronously), and should be interrupt-driven. They can use registers AØ to A3 and DØ to D3, but must preserve any other registers used. Prime, control, and status routines should place a result code in DØ and return via an RTS, unless the device completes the I/O request immediately, in which case they should JMP to the IODone routine (explained below).

(eye)

Because they can be called as the result of an interrupt during a previous I/O request, these routines should never call Memory Manager routines that cause heap compactions.

The prime routine must implement all Read and Write calls made to the driver. You may want to use the Fetch and Stash routines described below to read and write characters. If the driver is for a block device, it should update the dCtlPosition position after each read or write. The control routine must accept the control information passed to it, and manipulate the device as requested. The status routine must return requested status information. As both the control and status routines may be subjected to Control and Status calls sending and requesting a variety of information, they must be prepared to respond correctly to all types.

Routines For Writing Drivers

The Device Manager includes three routines that provide low-level functions for drivers: Fetch, Stash, and IODone. Include them in the code of your device driver if they're useful to you. Fetch, Stash, and IODone are invoked via jump vectors rather than macros (in the interest of speed). These routines don't return a result code, as the only result possible is dSIOCoreErr, which invokes the System Error Handler.

FUNCTION Fetch (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Jump vector	jFetch	
On entry	AØ:	pointer to device control entry
On exit	DØ:	character fetched; bit 15=1 if it's the last character in the data buffer

Parameter block

←	12	ioCompletion	pointer
→	16	ioResult	word
←	24	ioRefNum	word
←	32	ioBuffer	pointer
←	36	ioReqCount	long word
←	4Ø	ioActCount	long word

Fetch gets the next character from the data buffer pointed to by ioBuffer and places it in DØ. IOActCount is incremented by 1. If ioActCount equals ioReqCount, bit 15 of DØ is set. After receiving the last byte requested, the driver should call IODone.

FUNCTION Stash (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Jump vector	jStash	
On entry	AØ:	pointer to device control entry
	DØ:	character to stash
On exit	DØ:	bit 15=1 if it's the last character requested

Parameter block

←	12	ioCompletion	pointer
→	16	ioResult	word
←	24	ioRefNum	word
←	32	ioBuffer	pointer
←	36	ioReqCount	long word
←	4Ø	ioActCount	long word

Stash places the character in DØ into the data buffer pointed to by ioBuffer, and increments ioActCount by 1. If ioActCount equals ioReqCount, bit 15 of DØ is set. After stashing the last byte requested, the driver should call IODone.

FUNCTION IODone (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Jump vector		jIODone	
On entry		AØ:	pointer to device control entry
On exit		DØ:	bit 15=1 if it's the last character in the buffer

Parameter block

<--	12	ioCompletion	pointer
-->	16	ioResult	word
<--	24	ioRefNum	word
<--	32	ioBuffer	pointer
<--	36	ioReqCount	long word
<--	4Ø	ioActCount	long word

Result codes		
	noErr	No error
	resErr	Can't load driver from resource file
	unitEmptyErr	Reference number specifies NIL handle in unit table

IODone removes the current I/O request from the driver's I/O queue, and executes the completion routine (if there's one). It marks the driver inactive, and unlocks it and its device control entry (if it's allowed to by the dNeedLock bit of the dCtlFlags word). Then it begins executing the next I/O request in the I/O queue.

Interrupts

This section discusses interrupts: how the Macintosh uses them, and how you can use them if you're writing your own device driver. Only programmers who want to write their own interrupt-driven device drivers need read this section. Programmers who want to build their own driver on top of a built-in Macintosh driver may be interested in some of the information presented here.

An interrupt is a form of exception: an error or abnormal condition detected by the processor in the course of program execution. Specifically, an interrupt is an exception that is signaled to the processor by a device, as distinct from a trap, which arises directly from the execution of an instruction. Interrupts are used by devices to notify the processor of a change in condition of the device, such as the completion of an I/O request. An interrupt causes the processor to suspend normal execution, save the address of the next instruction and the processor's internal status on the stack, and execute an interrupt handler.

The 68ØØØ recognizes seven different levels of interrupts, each with its own interrupt handler. The addresses of the various handlers, called interrupt vectors, are kept in a vector table in the system

communication area. Each level of interrupt has its own vector found at a definite fixed location in the vector table. When an interrupt occurs, the processor fetches the proper vector from the table, uses it to locate the interrupt handler for that level of interrupt, and jumps to the handler. On completion, the handler exits with an RTE instruction, which restores the internal state of the processor from the stack and resumes normal execution from the point of suspension.

There are three devices that can create interrupts: the Synertek 6522 Versatile Interface Adapter (VIA), the Zilog 8530 Serial Communications Controller, and the debugging switch. They send a 3-bit number, from 0 to 7, called the interrupt priority level, to the processor. The interrupt level indicates which device is interrupting, and indicates which interrupt handler should be executed:

Level	Interrupting device
0	None
1	VIA
2	SCC
3	Spurious
4-7	Debugging button

A level-3 interrupt occurs when both the VIA and SCC interrupt at the same time; the interrupt handler for a level-3 interrupt is simply an RTE instruction. Debugging interrupts shouldn't occur during the normal execution of an application.

The interrupt priority level is compared with the processor priority in bits 8, 9, and 10 of the status register. If the interrupt priority level is greater than the processor priority, the 68000 acknowledges the interrupt and initiates interrupt processing. The processor priority determines which interrupting devices are ignored, and which are serviced:

Level	Services
0	All interrupts
1	SCC and debugging interrupts only
3-6	Debugging interrupts only
7	No interrupts

When an interrupt is acknowledged, the processor priority is set to the interrupt priority level, to prevent additional interrupts of equal or lower priority, until the interrupt handler has finished servicing the interrupt.

The interrupt priority level is used as an index into the primary interrupt vector table. This table contains 7 long words beginning at address $64. Each long word contains the starting address of an interrupt handler (Figure 6).

$64	autoInt1	pointer to level-1 interrupt handler
$68	autoInt2	pointer to level-2 interrupt handler
$6C	autoInt3	pointer to level-3 interrupt handler
$70	autoInt4	pointer to level-4 interrupt handler
$74	autoInt5	pointer to level-5 interrupt handler
$78	autoInt6	pointer to level-6 interrupt handler
$7C	autoInt7	pointer to level-7 interrupt handler

Figure 6. Primary Interrupt Vector Table

6. Execution jumps to the interrupt handler at the address specified in the table.

The interrupt handler then must identify and service the interrupt, and restore the processor priority, status register, and program counter to the values they contained before the interrupt occurred.

Level-1 (VIA) Interrupts

Level-1 interrupts are generated by the VIA. You'll need to read the Synertek manual describing the VIA to use most of the information provided in this section. The level-1 interrupt handler determines the source of the interrupt (via the VIA's IFR and IER registers) and then uses a table of secondary vectors in the system communication area to determine which interrupt handler to call (Figure 7).

byte		
0	one-second interrupt	VIA CA2 control line
4	vertical-retrace interrupt	VIA CA1 control line
8	shift-register interrupt	VIA shift register
12	not used	
16	not used	
20	T2 timer: Disk Driver	VIA timer 1
24	T1 timer: Sound Driver	VIA timer 2
28	spurious (shouldn't occur)	

Figure 7. Level-1 Secondary Interrupt Vector Table

The level-1 secondary interrupt vector table is pointed to by the system global lvl1DT. Each vector in the table points to the interrupt handler for a different source of interrupt. The interrupts are handled in order of their entry in the table, and only one interrupt handler is called per level-1 interrupt (even if two or more sources are interrupting). This allows the level-1 interrupt handler to be reentrant, and interrupt handlers should lower the processor priority as soon as possible in order to enable other pending interrupts to be processed.

One-second interrupts occur every second, and simply update the system global time. Vertical retrace interrupts are generated once every vertical retrace interval; control is passed to the Vertical Retrace Manager, which updates the system global ticks, handles changes in the state of the cursor, keyboard, and mouse button, and executes tasks installed in the vertical retrace queue.

Whenever the Disk Driver or Sound Driver aren't being used, you can use the T1 and T2 timers for your own needs.

If the cumulative elapsed time for all tasks on a level-1 interrupt exceed 16msec (one video frame), a level-1 interrupt may itself be interrupted by a vertical retrace interrupt. In this case, the vertical retrace interrupt is cleared, and the vertical retrace tasks are ignored.

The base address of the VIA (the system global vBase) is passed to each interrupt handler in A1.

Level-2 (SCC) Interrupts

Level-2 interrupts are generated by the SCC. You'll need to use the Zilog manual describing the VIA to effectively use the information provided in this section. The level-2 interrupt handler determines the source of the interrupt, and then uses a table of secondary vectors in the system communication area to determine which interrupt handler to call (Figure 8).

byte		
0	channel B transmit buffer empty	
4	channel B external/status change	mouse vertical
8	channel B receive character available	
12	channel B special receive condition	
16	channel A transmit buffer empty	
20	channel A external/status change	mouse horizontal
24	channel A receive character available	
28	channel A special receive condition	

Figure 8. Level-2 Secondary Interrupt Vector Table

The level-2 secondary interrupt vector table is pointed to by the system global lvl2DT. Each vector in the table points to the interrupt handler for a different source of interrupt. The interrupts are handled according to the following fixed priority:

channel A receive character available and special receive
channel A transmit buffer empty
channel A external/status change
channel B receive character available and special receive
channel B transmit buffer empty
channel B external/status change

Only one interrupt handler is called per level-2 interrupt (even if two or more sources are interrupting). This allows the level-2 interrupt handler to be reentrant, and interrupt handlers should lower the processor priority as soon as possible in order to enable other pending interrupts to be processed.

External/status interrupts pass through a tertiary vector table (Figure 9) in the system communication area to determine which interrupt handler to call (Figure 9).

byte 0	channel B nonmouse interrupt
4	mouse vertical interrupt
8	channel A nonmouse interrupt
12	mouse horizontal interrupt

Figure 9. Level-2 External/Status Interrupt Vector Table

The external/status interrupt vector table is pointed to by the system global extStsDT. Each vector in the table points to the interrupt handler for a different source of interrupt. Nonmouse interrupts (break/abort, for example) always handled before mouse interrupts.

When a level-2 interrupt handler is called, DØ points to the SCC read register Ø (external/status interrupts only), and D1 points to the SCC read register Ø containing the changed bits since the last external/status interrupt. AØ points to the SCC channel A or channel B control read address and A1 points to SCC channel A or channel B control write address, depending on which channel is interrupting. The SCC's data read address and data write address are located 4 bytes beyond AØ and A1, respectively. The following system globals can be used to refer to these locations:

System global	Value	Refers to
sccRBase	$9FFFF8	Base read address
sccWBase	$BFFFF9	Base write address
bCtl	Ø	Offset for channel B control
aCtl	2	Offset for channel A control
bData	4	Offset for channel B data
aData	6	Offset for channel A data

Writing Your Own Interrupt Handlers

You can write your own interrupt handlers to replace any of the standard interrupt handlers just described. Be sure to place an interrupt vector that points to your interrupt handler in one of the interrupt vector tables.

Both the level-1 and level-2 interrupt handlers preserve AØ through A3 and DØ through D3. Every interrupt handler (except for external/status interrupt handlers) is responsible for clearing the source of the interrupt, and for saving and restoring any additional registers used. Interrupt handlers should return directly via an RTS, or, if the I/O requested by the handler is completed immediately, via a JMP to IODone.

(hand)

Any software action indicating that interrupts are being enabled should be taken before the corresponding hardware action, lest an interrupt occur before the software has been told such an event is possible.

Any software action indicating that interrupts are being disabled should be taken after the corresponding hardware action, lest one interrupt slip in with the software thinking that interrupts are off.

A Sample Driver

Here's the skeleton of the Disk Driver, as an example of how a driver should be constructed.

```
                .
                .
                .

SonyDrvr
                .WORD   $4F00               ;read, write, control...
                .WORD   0,0                 ;no delay or event mask
                .WORD   0                   ;no menu

;Entry-point offset table

                .WORD   DiskOpen-DiskDrvr   ;open
                .WORD   DiskPrime-DiskDrvr  ;prime
                .WORD   DiskControl-DiskDrvr ;control
                .WORD   DiskStatus-DiskDrvr ;status
                .WORD   DiskRTS-DiskDrvr    ;close (just RTS)
                   .
                   .
                   .

;Disk Driver routines

DiskOpen        MOVEQ    #<DiskVarLth/2>,D0
                ...
DiskRTS         RTS
                ...
DiskDone        JMP     IODone
                ...
DiskControl     MOVE.L  JControl,-(SP)
                ...
DiskStatus      MOVEQL  #StatusErr,D0
                ...
DiskPrime       MOVE.L  JDiskPrime,(-SP)
                ...
```

SUMMARY OF THE DEVICE MANAGER

Constants

```
CONST goodByeCode = -1;
```

Data Structures

```
TYPE ParmBlkPtr    = ^ParamBlockRec;

     ParamBlkType  = (ioParam, fileParam, volumeParam, controlParam);

     ParamBlockRec = RECORD
                       ioLink:       Ptr;
                       ioType:       INTEGER;
                       ioTrap:       INTEGER;
                       ioCmdAddr:    Ptr;
                       ioCompletion: ProcPtr;
                       ioResult:     INTEGER;
                       ioNamePtr:    OSStrPtr;
                       ioVRefNum:    INTEGER;
                     CASE ParamBlkType OF
                       ioParam:
                       fileParam:
                       volumeParam:
                       controlParam:
                     END;

TYPE OpVariant = (sound, asyncRst, asyncInBuff, asyncShk, printer,
                  fontMgr, diskDrv, asyncBuffBytes, asyncStatus,
                  diskStat);
```

```
TYPE OpParamPtr = ^OpParamType;

     OpParamType = RECORD
            CASE OpVariant OF
                 {control information}
                       sound:                     {Sound Driver}
                         (sndVal: INTEGER);
                       asyncRst:                  {Async Driver}
                         (asncConfig: INTEGER);
                       asyncInBuff:
                         (asncBPtr: Ptr;
                          asncBLen: INTEGER);
                       asyncShk:
                         (asncHndShk: LongInt;
                          asncMisc: LongInt);
                       printer:                   {Printer Driver}
                         (param1: LongInt;
                          param2: LongInt;
                          param3: LongInt);
                       fontMgr:                   {Font Manager}
                         (fontRecPtr: Ptr;
                          fontCurDev: INTEGER);
                       diskDrv:                   {Disk Driver}
                         (diskBuff: Ptr);
                 {status information}
                       asyncBuffBytes:            {Async Driver}
                         (asyncNBytes: LongInt);
                       asyncStatus:
                         (asncS1: INTEGER;
                          asncS2: INTEGER;
                          asncS3: INTEGER);
                       diskStat:                  {Disk Driver}
                         (dskTrackLock: INTEGER;
                          dskInfoBits:  LongInt;
                          dskQElem:     drvrQElRec;
                          dskPrime:     INTEGER;
                          dskErrCnt:    INTEGER);
                   END;
```

Routines For Opening and Closing Drivers

```
FUNCTION OpenDriver  (fileName: OSStr255; VAR refNum: INTEGER) : OSErr;
FUNCTION CloseDriver (refNum: INTEGER) : OSErr;
```

High-Level Routines

```
FUNCTION FSRead    (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                    : OSErr;
FUNCTION FSWrite   (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                    : OSErr;
FUNCTION FSControl (refNum: INTEGER; opCode: INTEGER; opParams:
                    OpParamPtr) : OSErr;
FUNCTION FSStatus  (refNum: INTEGER; opCode: INTEGER; VAR opParams:
                    OpParamPtr) : OSErr;
FUNCTION FSKillIO  (refNum: INTEGER) : OSErr;
```

Low-Level Routines

```
FUNCTION PBRead    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBWrite   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBControl (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBStatus  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBKillIO  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Routines For Writing Drivers

```
FUNCTION Fetch  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION Stash  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION IODone (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Assembly-Language Information

Constants

```
ioQType     .EQU   2        ;I/O request queue entry type
aRdCmd      .EQU   2        ;ioTrap type for Read call
aWrCmd      .EQU   3        ;ioTrap type for Write call
aCtlCmd     .EQU   4        ;ioTrap type for Control call
aStsCmd     .EQU   5        ;ioTrap type for Status call
sccRBase    .EQU   $9FFFF8  ;SCC base read address
sccWBase    .EQU   $BFFFF9  ;SCC base write address
bCtl        .EQU   0        ;Offset for SCC channel B control
aCtl        .EQU   2        ;Offset for SCC channel A control
bData       .EQU   4        ;Offset for SCC channel B data
aData       .EQU   6        ;Offset for SCC channel A data
```

Standard Parameter Block Data Structure

ioLink	Next queue entry
ioType	Always fsQType
ioTrap	Routine trap
ioCmdAddr	Routine address
ioCompletion	Completion routine
ioResult	Result code
ioFileName	File name (and possibly volume name too)
ioVNPtr	Volume name
ioVRefNum	Volume reference number
ioDrvNum	Drive number

Control and Status Parameter Block Data Structure

csCode	Type of Control or Status call
csParam	Parameters for Control or Status call

I/O Parameter Block Data Structure

ioRefNum	Driver reference number
ioFileType	Not used
ioPermssn	Open permission
ioBuffer	Data buffer
ioReqCount	Requested number of bytes
ioActCount	Actual number of bytes
ioPosMode	Type of positioning operation
ioPosOffset	Size of positioning offset

Macro Names

Routine name	Macro name
PBRead	_Read
PBWrite	_Write
PBControl	_Control
PBStatus	_Status
PBKillIO	_KillIO

System Globals

Name	Size	Contents
uTableBase	4 bytes	Pointer to the unit table
unitNtryCnt	2 bytes	Maximum number of entries in the unit table

Result Codes

Name	Value	Meaning
abortErr	-27	IO call aborted by KillIO
badUnitErr	-21	Reference number doesn't match unit table
controlErr	-17	Driver isn't enabled for control calls
dInstErr	-26	Couldn't find driver in resource file
dRemoveErr	-25	Tried to remove an open driver
dSIOCoreErr	14	Device control entry was purged
memFullErr	-1Ø8	Memory full
noErr	Ø	No error
notOpenErr	-28	Driver isn't open
openErr	-23	Requested read/write permission doesn't match driver's open permission
readErr	-19	Driver isn't enabled for read calls
resErr		Resource Manager error
statusErr	-18	Driver isn't enabled for status calls
unitEmptyErr	-22	Reference number specifies NIL handle in unit table
writErr	-2Ø	Driver isn't enabled for write calls

GLOSSARY

asynchronous execution: During asynchronous execution of a routine, the Device Manager is free to perform other tasks.

block device: A device that reads and writes blocks of 512 characters at a time; it can read or write any accessible block on demand.

character device: A device that reads or writes a stream of characters, one at a time: it can neither skip characters nor go back to a previous character.

closed driver: A driver that cannot be read from or written to.

close routine: The part of a driver's code that implements Device Manager Close calls.

completion routine: Any application-defined code to be executed when an asynchronous call to a Device Manager routine is completed.

control information: Information transmitted by an application to a driver; it can typically select modes of operation, start or stop processes, enable buffers, choose protocols, and so on.

control routine: The part of a driver's code that implements Device Manager Control and KillIO calls.

data buffer: Heap space containing information to be written to a driver from an application, or read from a driver to an application.

device: A part of the Macintosh or a piece of external equipment, that can transfer information into or out of the Macintosh.

device control entry: a 40-byte relocatable block of heap space that tells the Device Manager the location of a driver's routines, the location of a driver's I/O queue, and other information.

device driver: a program that exchanges information between an application and a device.

driver name: A sequence of up to 255 printing characters; driver names are always prefixed by a period (.).

driver reference number: A number that uniquely identifies an individual driver.

exception: An error or abnormal condition detected by the processor in the course of program execution.

interrupt: An exception that is signaled to the processor by a device, to notify the processor of a change in condition of the device, such as the completion of an I/O

interrupt handler: A routine that services interrupts.

interrupt priority level: A number identifying the importance of the interrupt. It indicates which device is interrupting, and which interrupt handler should be executed.

interrupt vector: A pointer to an interrupt handler.

I/O queue: A queue containing the parameter blocks of all I/O requests for one driver.

I/O request: A request for input from or output to a driver; caused by calling a Device Manager routine asynchronously.

open driver: A driver that can be read from and written to.

open routine: The part of a driver's code that implements Device Manager Open calls.

parameter block: An area of heap space used to transfer information between applications and the Device Manager.

prime routine: The part of a driver's code that implements Device Manager Read and Write calls.

processor priority: Bits 8, 9, and 1Ø of the status register, that indicate which interrupts will be processed and which will be ignored.

status information: Information transmitted to an application by a driver; it may indicate the current mode of operation, the readiness of the device, the occurrence of errors, and so on.

status routine: The part of a driver's code that implements Device Manager Status calls.

synchronous execution: During synchronous execution of a routine, the Device Manager must devote all of its attention to the routine, and isn't free to perform any other task.

unit number: The number of each driver's entry in the unit table.

unit table: A 128-byte relocatable block containing a handle to the device control entry for each device driver.

vector table: A table of vectors in the system communication area.

The Dialog Manager: A Programmer's Guide /DMGR/DIALOG

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Desk Manager: A Programmer's Guide
CoreEdit: A Programmer's Guide
TextEdit: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:			
	Preliminary Draft	C. Rose	12/8/82
	Preliminary Draft	C. Rose	1/7/83
	First Draft (ROM 2.1)	C. Rose	3/22/83
	Second Draft (ROM 4)	C. Rose	6/13/83
	Third Draft (ROM 7)	C. Rose	11/16/83

ABSTRACT

The Dialog Manager is the part of the Macintosh User Interface Toolbox that supports dialog boxes and the alert mechanism. This manual tells you how to manipulate dialogs and alerts with Dialog Manager routines.

Summary of significant changes and additions since last version:

- The Return key (like Enter) now has the same effect as clicking the default button in an alert box. Return and Enter also have this effect in a modal dialog; the default button is the first button in the item list, normally the OK button (pages 5, 11, 22).

- Changes have been made to the structure of a dialog record (page 14) and a dialog template (page 28).

- The standard sound procedure now exists. Sounds 1 through 3 are the corresponding number of short beeps (page 15).

- The discussion of filterProcs for ModalDialog has changed (page 22).

- Assembly-language programmers can change the font used in dialogs and alerts (page 19).

TABLE OF CONTENTS

3 About This Manual
4 About the Dialog Manager
6 Dialog and Alert Windows
8 Dialogs, Alerts, and Resources
9 Item Lists in Memory
9 Item Types
11 Item Handle or Procedure Pointer
12 Display Rectangle
12 Item Numbers
13 Dialog Records
13 Dialog Pointers
14 The DialogRecord Data Type
15 Alerts
16 Using the Dialog Manager
17 Dialog Manager Routines
17 Initialization
18 Creating and Disposing of Dialogs
20 Handling Dialog Events
23 Invoking Alerts
25 Manipulating Items in Dialogs and Alerts
27 Modifying Templates in Memory
28 Dialog Templates in Memory
29 Alert Templates in Memory
30 Formats of Resources for Dialogs and Alerts
30 Dialog Templates in a Resource File
31 Alert Templates in a Resource File
32 Items Lists in a Resource File
34 Summary of the Dialog Manager
38 Glossary

Copyright (c) 1983 Apple Computer, Inc. All rights reserved. Distribution in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Dialog Manager of the Macintosh User Interface Toolbox. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. *** The Dialog Manager provides Macintosh programmers with routines for implementing dialog boxes and the alert mechanism, two means of communication between the application and the end user.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Dialog Manager may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly rectangles, grafPorts, and pictures.
- The basic concepts behind TextEdit or CoreEdit, to understand editing text in dialog boxes.
- Resources, as discussed in the Resource Manager manual.
- The Toolbox Event Manager, the Window Manager, and the Control Manager.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it. *** Some of that information refers to the "Toolbox equates" file (ToolEqu.Text), which the reader will have learned about in an earlier chapter of the final comprehensive manual. ***

The manual begins with an introduction to the Dialog Manager and what you can do with it. It then discusses the basics of dialogs and alerts: their relationship to windows, their relationship to resources, and the information stored in memory for the items in a dialog or alert. Following this is a discussion of the dialog record, where the Dialog Manager keeps all the information it needs about a dialog, and an overview of how alerts are handled.

Next, a section on using the Dialog Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Dialog Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. There's a discussion of how to modify definitions of dialogs and alerts after they've been read from a resource file, and a section that gives the exact formats of resources related to dialogs and alerts.

Finally, there's a summary of the Dialog Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE DIALOG MANAGER

The Dialog Manager is a tool for handling dialogs and alerts in a way that's consistent with the Macintosh User Interface Guidelines.

A dialog box appears on the screen when a Macintosh application needs more information to carry out a command. As shown in Figure 1, it typically resembles a form on which the user checks boxes and fills in blanks.

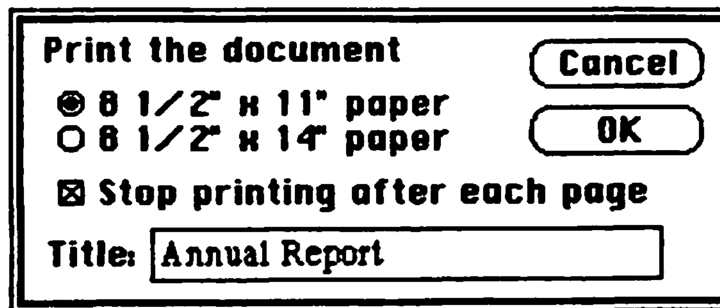


Figure 1. A Typical Dialog Box

By convention, a dialog box comes up slightly below the menu bar, is a bit narrower than the screen, and is centered between the left and right edges of the screen. It may contain any or all of the following:

- Informative or instructional text
- Rectangles in which text may be entered (initially blank or containing default text that can be edited)
- Controls of any kind
- Graphics (icons or QuickDraw pictures)
- Anything else, as defined by the application

The user provides the necessary information in the dialog box, such as by entering text or clicking a check box. There's usually a button marked "OK" to tell the application to accept the information provided and perform the command, and a button marked "Cancel" to cancel the

command. Some dialog boxes contain more than one button that will perform the command, each in a different way.

Most dialog boxes require the user to respond before doing anything else. Clicking a button to perform or cancel the command makes the box go away; clicking outside the dialog box only causes a beep from the Macintosh's speaker. This type is called a modal dialog box because it puts the user in the state or "mode" of being able to work only inside the dialog box. It usually has the same general appearance as shown in Figure 1. One of the buttons in the dialog box may be outlined boldly. Pressing the Return key or the Enter key has the same effect as clicking the outlined button or, if none, the OK button; the particular button whose effect occurs is called the dialog's default button and is the preferred ("safest") button to use in the current situation. If there's no boldly outlined or OK button, pressing Return or Enter will by convention have no effect.

Other dialog boxes do not require the user to respond before doing anything else; these are called modeless dialog boxes. The user can, for example, do work in document windows on the desktop before clicking the appropriate button in the dialog box. Clicking the Cancel button in this type of dialog box always makes the box go away, but clicking the OK button may not: it may keep the box around so that the command can be performed again. A modeless dialog box looks exactly like a document window, as illustrated in Figure 2.

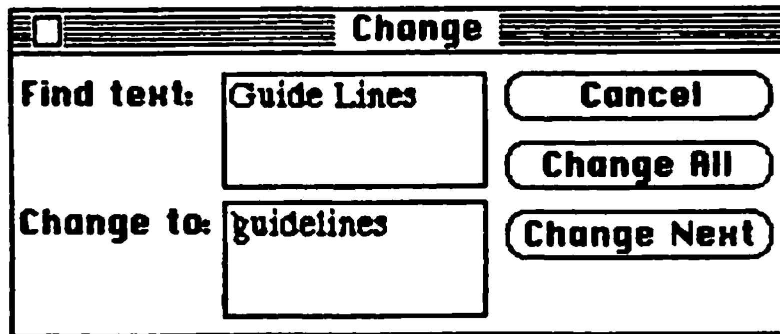


Figure 2. A Modeless Dialog Box

Dialog boxes may in fact require no response at all. For example, while an application is performing a time-consuming process, it can display a dialog box that contains only a message telling what it's doing; then, when the process is complete, it can simply remove the dialog box.

The alert mechanism provides applications with a means of reporting errors or giving warnings. An alert box is similar to a modal dialog box, but it appears only when something has gone wrong or must be brought to the user's attention. Its conventional placement is slightly farther below the menu bar than a dialog box. To assist the user who isn't sure how to proceed when an alert box appears, the

preferred button to use in the current situation is outlined boldly so it stands out from the other buttons in the alert box (see Figure 3). The outlined button is also the alert's default button; if the user presses the Return key or the Enter key, the effect is the same as clicking this button.

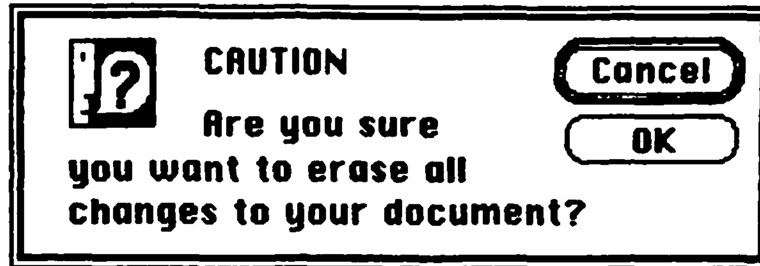


Figure 3. A Typical Alert Box

There are three standard kinds of alert--Stop, Note, and Caution--each indicated by a particular icon in the top left corner of the alert box. Figure 3 illustrates a Caution alert. The icons identifying Stop and Note alerts are similar; instead of a question mark, they show an exclamation point and an asterisk, respectively. Other alerts can have anything in the the top left corner, including blank space if desired.

The alert mechanism also provides another type of signal: sound from the Macintosh's speaker. The application can base its response on the number of consecutive times an alert occurs; the first time, it might simply beep, and thereafter it may present an alert box. The sound is not limited to a single beep but may be any sequence of tones, and may occur either alone or along with an alert box. As an error is repeated, there can also be a change in which button is the default button (perhaps from OK to Cancel). You can specify different responses for up to four occurrences of the same alert.

With Dialog Manager routines, you can create dialog boxes or invoke alerts. The Dialog Manager gets most of the descriptive information about the dialogs and alerts from resources in a resource file. You use a program such as the Resource Editor *** eventually *** to store the necessary information in the resource file. The Dialog Manager calls the Resource Manager to read what it needs from the resource file into memory as necessary. In some cases you can modify the information after it's been read into memory.

DIALOG AND ALERT WINDOWS

A dialog box appears in a <u>dialog window</u>. When you call a Dialog Manager routine to create a dialog, you supply the same information as when you create a window with a Window Manager routine. For example, you supply the window definition ID, which determines how the window

looks and behaves, and a rectangle that becomes the portRect of the window's grafPort. You specify the window's plane (which, by convention, should initially be the frontmost) and whether the window is visible or invisible. The dialog window is created as specified.

You can manipulate a dialog window just like any other with Window Manager or QuickDraw routines, showing it, hiding it, moving it, changing its size or plane, or whatever--all, of course, in conformance with the Macintosh User Interface Guidelines. The Dialog Manager observes the clipping region of the dialog window's grafPort, so if you want clipping to occur, you can set this region with a QuickDraw routine.

Similarly, an alert box appears in an alert window. You don't have the same flexibility in defining and manipulating an alert window, however. The Dialog Manager chooses the window definition ID, so that all alert windows will have the standard appearance and behavior. The size and location of the box are supplied as part of the definition of the alert and are not easily changed. You don't specify the alert window's plane; it always comes up in front of all other windows. Since an alert box requires the user to respond before doing anything else, and the response makes the box go away, the application doesn't do any manipulation of the alert window.

Figure 4 illustrates a document window, dialog window, and alert window, all overlapping on the desktop. Note that if a dialog or alert window comes up while a document window is active, the document window becomes inactive; as shown in Figure 4, any scroll bars or size box in the document window disappear until the window becomes active again.

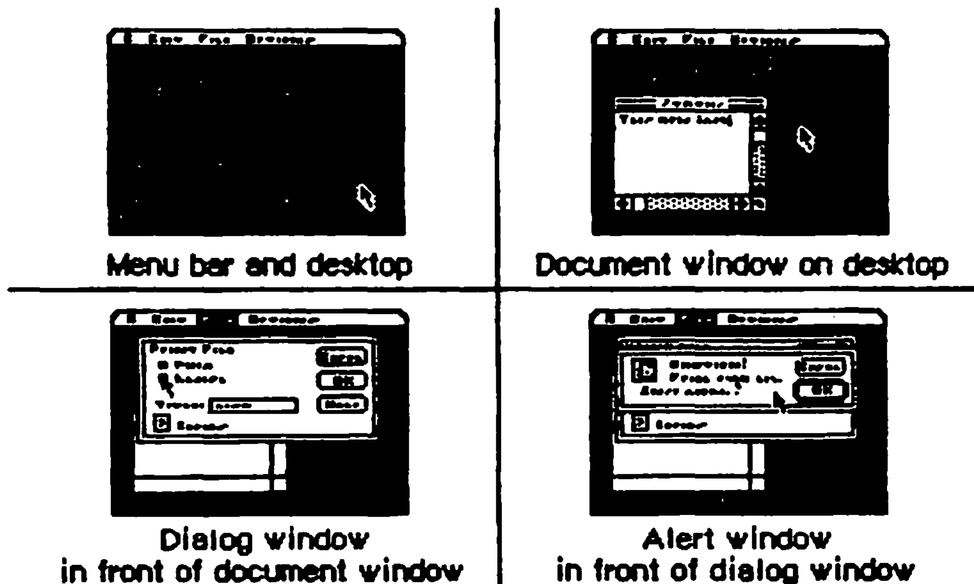


Figure 4. Dialog and Alert Windows

DIALOGS, ALERTS, AND RESOURCES

To create a dialog, the Dialog Manager needs the same information about the dialog window as the Window Manager needs when it creates a new window: the window definition ID along with other information specific to this window. The Dialog Manager also needs to know what items the dialog box contains. You can store the needed information as a resource in a resource file and pass the resource ID to a function that will create the dialog. This type of resource, which is called a dialog template, is analogous to a window template, and the function, GetNewDialog, is similar to the Window Manager function GetNewWindow. The Dialog Manager calls the Resource Manager to read the dialog template from the resource file. It then incorporates the information in the template into a dialog data structure in memory, called a dialog record.

Similarly, the data that the Dialog Manager needs to create an alert is stored in an alert template in a resource file. The various routines for invoking alerts require the resource ID of the alert template as a parameter.

The information about all the items (text, controls, or graphics) in a dialog or alert box is stored in an item list in a resource file. The resource ID of the item list is included in the dialog or alert template. The item list in turn contains the resource IDs of any icons or QuickDraw pictures in the dialog or alert box, and possibly the resource IDs of control templates for controls in the box. After calling the Resource Manager to read a dialog or alert template into memory, the Dialog Manager calls it again to read in the item list, and again to read in any individual items as necessary.

(hand)

> To create dialog or alert templates and item lists and store them in resource files, you can use the Resource Editor *** eventually (for now, the Resource Compiler, as described in the manual "Putting Together a Macintosh Application") ***. The Resource Editor relieves you of having to know the exact format of these resources, but for interested programmers this information is given in the section "Formats of Resources for Dialogs and Alerts".

If desired, the application can gain some additional flexibility by calling the Resource Manager directly to read templates, item lists, or items from a resource file. For example, you can read in a dialog or alert template directly and modify some of the information in it before calling the routine to create the dialog or alert. Or, as an alternative to using a dialog template, you can read in a dialog's item list directly and then pass a handle to it along with other information to a function that will create the dialog (NewDialog, analogous to the Window Manager function NewWindow).

(hand)
The use of dialog templates is recommended wherever possible; like window templates, they isolate descriptive information from your application code for ease of modification or translation to foreign languages.

ITEM LISTS IN MEMORY

This section discusses the contents of an item list once it's been read into memory from a resource file and the Dialog Manager has set it up as necessary to be able to work with it.

An item list in memory contains the following information for each item:

- The type of item. This includes not only whether the item is a control, text, or whatever, but also whether the Dialog Manager should return to the application when the item is clicked.
- A handle to the item or, for special application-defined items, a pointer to a procedure that draws the item.
- A display rectangle, which determines the location of the item within the dialog or alert box.

These are discussed below along with item numbers, which identify particular items in the item list.

There's a Dialog Manager procedure that, given a pointer to a dialog record and an item number, sets or returns that item's type, handle (or procedure pointer), and display rectangle.

Item Types

The item type is specified by a predefined constant or combination of constants, as listed below. Figure 5 illustrates some of these item types.

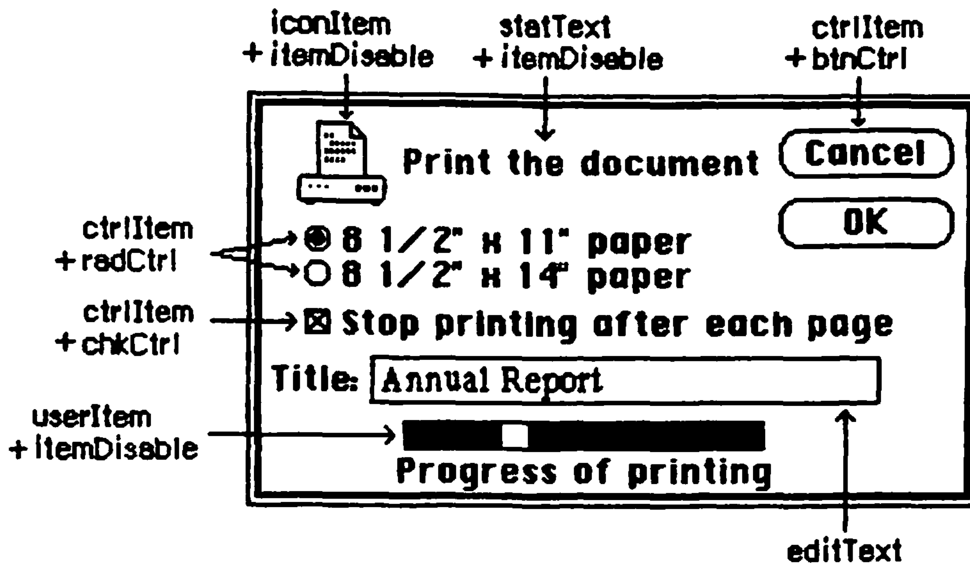


Figure 5. Item Types

Item type	Meaning
ctrlItem+btnCtrl	A standard button control.
ctrlItem+chkCtrl	A standard check box control.
ctrlItem+radCtrl	A standard "radio button" control.
ctrlItem+resCtrl	A control defined in a control template in a resource file.
statText	Static text; text that cannot be edited.
editText	(Dialogs only) Text that can be edited; the Dialog Manager accepts text typed by the user and allows editing.
iconItem	An icon (a 32-by-32 bit image).
picItem	A QuickDraw picture.
userItem	(Dialogs only) An application-defined item, such as a picture whose appearance changes.
itemDisable+<any of the above>	The item is <u>disabled</u> (the Dialog Manager doesn't report events involving this item).

The text of an editText item may initially be either default text or empty. Text entry and editing is handled in the conventional way, as in TextEdit and CoreEdit (in fact, the Dialog Manager calls TextEdit to handle it):

- Clicking in the item displays a blinking vertical bar, indicating an insertion point where text may be entered.

- Dragging over text in the item selects that text, and double-clicking selects a word; the selection is inversely highlighted and is replaced by what the user then types.

- Clicking or dragging while holding down the Shift key extends or shortens the current selection.

- The Backspace key deletes the current selection or the character preceding the insertion point.

The Tab key advances to the next editText item in the item list (wrapping around to the first if there aren't any more). In an alert box or a modal dialog box (regardless of whether it contains an editText item), the Return key or Enter key has the same effect as clicking the default button; for alerts, the default button is identified in the alert template, whereas for modal dialogs it's always the first item in the item list.

If itemDisable is specified for an item, the Dialog Manager doesn't let the application know about events involving that item. For example, you may not have to be informed every time the user enters or edits text in an editText item, but may only need to look at the text when the OK button is clicked. In this case, the editText item would be disabled. Standard buttons and check boxes should always be enabled, so your application will know when they've been clicked.

(eye)

> Don't confuse disabling a control with making one "inactive" with the Control Manager procedure HiliteControl: When you want a control not to respond at all to being clicked, you make it inactive.

Item Handle or Procedure Pointer

The item list contains the following information for the various types of items:

Item type	Contents
any ctrlItem	A control handle
statText	A handle to the text
editText	A handle to the current text
iconItem	A handle to the icon
picItem	A picture handle
userItem	A procedure pointer

The procedure for a userItem draws the item; for example, if the item is a clock, it will draw the clock with the current time displayed. When this procedure is called, the current port is the dialog window's grafPort. The procedure has two parameters:

- A windowPtr to the dialog window. In case the procedure draws in more than one dialog window, this parameter tells it which one to draw in.

- The item number. In case the procedure draws more than one item, this parameter tells it which one to draw.

Display Rectangle

Each item in the item list is displayed within its display rectangle. Icons and QuickDraw pictures are scaled to fit the display rectangle. If the procedure for a userItem draws outside the item's display rectangle, the drawing is clipped to the display rectangle.

(eye)

Clicking anywhere within the display rectangle is considered a click of that item.

A rectangle is drawn just outside the display rectangle around each editText item. When a statText or editText item is displayed, the text is clipped to the display rectangle and word wrap occurs as in TextEdit.

Item Numbers

Each item in an item list is identified by an item number, which is simply the index of the item in the list (starting from 1). By convention, the first item in an alert's item list should be the OK button (or, if none, then one of the buttons that will perform the command) and the second item should be the Cancel button. The Dialog Manager provides predefined constants equal to the item numbers for OK and Cancel:

```
CONST OK     = 1;
      Cancel = 2;
```

In a modal dialog's item list, the first item is assumed to be the dialog's default button; if the user presses the Return key or Enter key, the Dialog Manager normally returns item number 1, just as when that item is actually clicked. To conform to the Macintosh User Interface Guidelines, the application should boldly outline the dialog's default button if it isn't the OK button. The best way to do this is with a userItem. To allow for changes in the default button's size or location, the userItem should identify which button to outline by its item number and then use that number to get the button's display rectangle.

(eye)

If the first item in a modal dialog's item list isn't an OK button and you don't boldly outline it, you should set up the dialog to ignore Return and Enter. To learn how to do this, see ModalDialog under "Handling Dialog

Events" in the "Dialog Manager Routines" section.

DIALOG RECORDS

To create a dialog, you pass information to the Dialog Manager in a dialog template and in individual parameters, or only in parameters; in either case, the Dialog Manager incorporates the information into a dialog record. The dialog record contains the window record for the dialog window, a handle to the dialog's item list, and some additional fields. The Dialog Manager creates the dialog window by calling the Window Manager function NewWindow and then setting the window class in the window record to indicate that it's a dialog window. The routine that creates the dialog returns a pointer to the dialog record, which you use thereafter to refer to the dialog in Dialog Manager routines or even in Window Manager or QuickDraw routines (see "Dialog Pointers" below). The Dialog Manager provides routines for handling events in the dialog window and disposing of the dialog when you're done.

The data type for a dialog record is called DialogRecord. You can do all the necessary operations on a dialog without accessing the fields of the dialog record directly; for advanced programmers, however, the exact structure of a dialog record is given under "The DialogRecord Data Type" below.

Dialog Pointers

There are two types of dialog pointer, DialogPtr and DialogPeek, analogous to the window pointer types WindowPtr and WindowPeek. Most users will only need to use DialogPtr.

The Dialog Manager defines the following type of dialog pointer:

```
TYPE DialogPtr = WindowPtr;
```

It can do this because the first thing stored in a dialog record is the window record for the dialog window. This type of pointer can be used to access fields of the window record or can be passed to Window Manager routines that expect window pointers as parameters. Since the WindowPtr data type is itself defined as GrafPtr, this type of dialog pointer can also be used to access fields of the dialog window's grafPort or passed to QuickDraw routines that expect pointers to grafPorts as parameters.

In some cases, a more direct way of accessing the dialog record may be desired. For this reason, the Dialog Manager also defines the following type of dialog pointer:

```
TYPE DialogPeek = ^DialogRecord;
```

Programmers who want to access the dialog record fields directly must use this type of pointer.

Assembly-language note: From assembly language, of course, there's no type checking on pointers, and the two types of pointer are equal.

The DialogRecord Data Type

For those who want to know more about the data structure of a dialog record, the exact structure is given here.

```
TYPE DialogRecord = RECORD
                      window:    WindowRecord;
                      items:     Handle;
                      textH:     TEHandle;
                      editField: INTEGER;
                      editOpen:  INTEGER;
                      aDefItem:  INTEGER
                    END;
```

The window field contains the window record for the dialog window. The items field contains a handle to the item list for the dialog.

(hand)

Remember that to get or change information about an item in a dialog, you pass the dialog pointer and the item number to a Dialog Manager procedure. You will never access information directly through the handle to the item list.

The Dialog Manager uses the next three fields when text is being entered or edited in an editText item. The textH field contains the handle TextEdit uses; the data type TEHandle is defined in TextEdit. EditField is 1 less than the item number of the editText item. The editOpen field is used internally by the Dialog Manager.

The aDefItem field is used for modal dialogs and alerts, which are treated internally as special modal dialogs. It contains the item number of the default button. The default button for a modal dialog is the first item in the item list, so this field contains 1 for modal dialogs. The default button for an alert is specified in the alert template; see the following section for more information.

Assembly-language note: The Toolbox equates file includes dWindLen, the length of a dialog record in bytes.

ALERTS

When you call a Dialog Manager routine to invoke an alert, you pass it the resource ID of the alert template, which contains the following:

- A rectangle, given in global coordinates, which determines the alert window's size and location. It becomes the portRect of the window's grafPort. To allow for the menu bar and the border around the portRect, the top of the rectangle should be at least 25 pixels below the top of the screen.

- The resource ID of the item list for the alert.

- Information about exactly what should happen at each stage of the alert.

There are four stages to every alert: the first three stages correspond to the first three (consecutive) occurrences of the alert, and the fourth stage corresponds to the fourth occurrence and any beyond the fourth. The actions for each stage are specified by the following three pieces of information:

- Which is the default button--the OK button (or, if none, a button that will perform the command) or the Cancel button

- Whether the alert box is to be drawn

- Which of four sounds should be emitted at this stage of the alert

The alert sounds are determined by a sound procedure that emits one of up to four tones or sequences of tones. The sound procedure has one parameter, an integer from Ø to 3. It can emit any sound for each of these numbers, which identify the sounds in the alert template. If you don't write your own sound procedure, sound number Ø represents no sound and sound numbers 1 through 3 represent the corresponding number of short beeps, each of the same pitch and duration. For example, if the second stage of an alert is to cause a beep and no alert box, you can just specify boxDrawn=FALSE and sound=1 for that stage in the alert template. If instead you want two successive beeps of different pitch, for example, you need to write a procedure that will emit that sound for a particular sound number, and specify that number in the alert template. See the Sound Manager manual *** (doesn't yet exist) *** for information about how to write a procedure that emits sound.

(hand)
> When the Dialog Manager detects a click outside an alert box or a modal dialog box, it emits sound number 1; thus, for consistency with the Macintosh User Interface Guidelines, sound number 1 should always be a single beep.

Internally, alerts are treated as special dialogs. The alert routine creates the alert window by calling NewDialog. The Dialog Manager

works from the dialog record created by NewDialog, just as when it operates on a dialog window, but it disposes of the window before returning to the application. Normally your application will not access the dialog record for an alert; however, there is a way that this can happen: for any alert, you can specify a procedure that will be executed repeatedly during the alert, and this procedure may access the dialog record. For details, see the alert routines under "Invoking Alerts" in the "Dialog Manager Routines" section.

USING THE DIALOG MANAGER

This section discusses how the Dialog Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Before using the Dialog Manager, you should initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. The first Dialog Manager routine to call is InitDialogs, which initializes the Dialog Manager.

Where appropriate in your program, call NewDialog or GetNewDialog to create any dialogs you need. Usually you'll call GetNewDialog, which takes descriptive information about the dialog from a dialog template in a resource file. You can instead pass the information in individual parameters to NewDialog. In either case, you can supply a pointer to the storage for the dialog record or let it be allocated by the Dialog Manager. When you no longer need a dialog, you'll usually call CloseDialog if you supplied the storage, or DisposDialog if not.

In most cases, you probably won't have to make any changes to the dialogs from the way they're defined in the resource file. However, if you should want to modify an item in a dialog, you can call GetDItem to get the information about the item and SetDItem to change it. In particular, SetDItem is the routine to use for installing an application-defined item. There are also two procedures specifically for accessing or setting the content of a text item in a dialog box: GetIText and SetIText.

If your application includes any modeless dialog boxes, call IsDialogEvent to learn whether an event has occurred that needs to be handled as part of a dialog, and then call DialogSelect if so. After putting up a modal dialog box, just call ModalDialog.

Mouse activity in an editText item causes an insertion point to be displayed or text to be selected accordingly. Your application may want to bring up a dialog box with an editText item already selected, or to cause an insertion point or text selection to appear after the user has made an error in entering text. The SelIText procedure lets the application do this.

For alerts, if you want other sounds besides the standard ones (up to three short beeps), write your own sound procedure and call ErrorSound to make it the current sound procedure. To invoke a particular alert, call one of the alert routines: StopAlert, NoteAlert, or CautionAlert for one of the standard kinds of alert, or Alert for an alert defined to have something other than a standard icon (or nothing at all) in its top left corner. If you're going to invoke an alert when the resource file might not be accessible, first call CouldAlert, which will make the alert template and related resources unable to be purged from memory; you can later make them purgeable again by calling FreeAlert.

Finally, in either dialogs or alerts, you can substitute text in statText items with text that you specify in the ParamText procedure. This means, for example, that a document name supplied at execution time can appear in an error message.

DIALOG MANAGER ROUTINES

This section describes all the Dialog Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Initialization

PROCEDURE InitDialogs (restartProc: ProcPtr);

Call InitDialogs once before all other Dialog Manager routines, to initialize the Dialog Manager.

- It sets a pointer to a fail-safe procedure as specified by restartProc; this pointer will be accessed when a system error (such as running out of memory) occurs. RestartProc should point to a procedure that will restart the application after a system error. If no such procedure is desired, pass NIL as the parameter.

<u>Assembly-language note</u>: The Dialog Manager stores the address of the fail-safe procedure in a system global named restProc.

- It installs the standard sound procedure, which associates sound number Ø with no sound and sound numbers 1 through 3 with the corresponding number of short beeps.

- It passes empty strings to ParamText (described below under "Manipulating Items in Dialogs and Alerts").

PROCEDURE ErrorSound (soundProc: ProcPtr);

ErrorSound sets the sound procedure for dialogs and alerts to the procedure pointed to by soundProc. The sound procedure should have one parameter, an integer from Ø to 3; these numbers identify the sounds (such as in the stages field of an alert template).

(hand)
So that the dialog and alert routines will respond to mouse activity in a way that conforms to the Macintosh User Interface Guidelines (as described below), sound number 1 should always be a single beep.

If you don't call ErrorSound, the Dialog Manager uses a standard sound procedure that causes sound number Ø to represent no sound and sound numbers 1 through 3 to be the corresponding number of short beeps. If you pass NIL for soundProc, there will be no sound at all for sound numbers Ø through 3.

Assembly-language note: The address of the sound procedure being used is stored in the system global daBeeper.

Creating and Disposing of Dialogs

FUNCTION NewDialog (dStorage: Ptr; boundsRect: Rect; title: Str255; visible: BOOLEAN; procID: INTEGER; behind: WindowPtr; goAwayFlag: BOOLEAN; refCon: LongInt; items: Handle) : DialogPtr;

NewDialog creates a dialog as specified by its parameters and returns a pointer to the new dialog. The first eight parameters (dStorage through refCon) are passed to the Window Manager function NewWindow, which creates the dialog window; the meanings of these parameters are summarized below. The items parameter is a handle to the dialog's item list. You can get the items handle by calling the Resource Manager to read the item list from the resource file into memory.

(hand)
Advanced programmers can create their own item lists in memory rather than have them read from a resource file.

DStorage is analogous to the wStorage parameter of NewWindow; it's a pointer to the storage to use for the dialog record. If you pass NIL

for dStorage, the dialog record will be allocated on the heap.

BoundsRect, a rectangle given in global coordinates, which determines the dialog window's size and location. It becomes the portRect of the window's grafPort. Remember that the top of this rectangle should be at least 25 pixels below the top of the screen for a modal dialog, to allow for the menu bar and the border around the portRect, and at least 40 pixels below the top of the screen for a modeless dialog, to allow for the menu bar and the window's title bar.

Title is the dialog window's title. If the window has a title bar, this title appears in it, centered and in the system font and system font size.

If the visible parameter is TRUE, the dialog window is drawn on the screen. If it's FALSE, the window is initially invisible and may later be shown with a call to the Window Manager procedure ShowWindow.

ProcID is the window definition ID, which leads to the window definition function for this type of window. The window definition IDs for the standard types of dialog window are dBoxProc for the modal type and documentProc for the modeless type.

The behind parameter specifies the window behind which the dialog window is to be placed on the desktop. You should pass POINTER(-1) for this parameter to bring up the dialog window in front of all other windows.

If goAwayFlag is TRUE, the dialog window has a close box in its title bar (if any) when the window is active.

RefCon is the dialog window's reference value, which the application may store into and access for any purpose.

NewDialog also sets the font of the dialog window's grafPort to the system font and sets the window class in the window record to indicate a dialog window.

> Assembly-language note: NewDialog actually sets the font to the font number stored in the system global dlgFont. If you want a different font to be used in a dialog box, you can set dlgFont to the desired font number before creating the dialog.

```
FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr; behind:
            WindowPtr) : DialogPtr;
```

Like NewDialog (above), GetNewDialog creates a dialog as specified by its parameters and returns a pointer to the new dialog. Instead of having the parameters boundsRect, title, visible, procID, goAwayFlag,

and refCon, GetNewDialog has a single dialogID parameter, where dialogID is the resource ID of a dialog template that supplies the same information as those parameters. The dialog template also contains the resource ID of the dialog's item list. After calling the Resource Manager to read the item list into memory (if it's not already in memory), GetNewDialog makes a copy of the item list and uses that copy; thus you may have multiple independent dialogs whose items have the same types, locations, and initial contents. The dStorage and behind parameters of GetNewDialog have the same meaning as in NewDialog.

PROCEDURE CloseDialog (theDialog: DialogPtr);

CloseDialog removes theDialog's window from the screen and deletes it from the window list, just as when the Window Manager procedure CloseWindow is called. It returns to the heap the storage used by all data structures associated with the dialog window (such as the window regions) and all the items in the dialog's item list (except for pictures and icons, which might be shared resources). It does not dispose of the dialog record or the item list itself. Call this procedure when you're done with a dialog if you supplied NewDialog or GetNewDialog with a pointer to the dialog storage (in the dStorage parameter) when you created the dialog.

(hand)

> Even if you didn't supply a pointer to the dialog storage, you may want to call CloseDialog if you created the dialog with NewDialog. You would call CloseDialog if you wanted to keep the item list around (since, unlike GetNewDialog, NewDialog does not use a copy of the item list).

PROCEDURE DisposDialog (theDialog: DialogPtr);

DisposDialog calls CloseDialog (above) and then disposes of the dialog's item list and dialog record. Call this procedure when you're done with a dialog if you let the dialog record be allocated on the heap when you called NewDialog or GetNewDialog (by passing NIL as the dStorage parameter).

Handling Dialog Events

FUNCTION IsDialogEvent (theEvent: EventRecord) : BOOLEAN;

If your application includes any modeless dialogs, call IsDialogEvent after calling the Toolbox Event Manager function GetNextEvent. Pass the current event in theEvent. IsDialogEvent determines whether theEvent needs to be handled as part of a dialog. If theEvent is an update or activate event in a dialog window, a mouse down event in the content region of an active dialog window, or any other type of event

when a dialog window is active, IsDialogEvent returns TRUE; otherwise, it returns FALSE. When TRUE is returned, the application should check whether the event is one that should not in fact be handled as part of a dialog, such as a key down event with the Command key held down: if so, it should ignore the event; otherwise, it should pass the event to DialogSelect (below).

```
FUNCTION DialogSelect (theEvent: EventRecord; VAR theDialog: DialogPtr;
            VAR itemHit: INTEGER) : BOOLEAN;
```

After learning from IsDialogEvent that the current event needs to be handled as part of a modeless dialog, pass the event to DialogSelect. DialogSelect handles the event as described below. If the event involves an enabled dialog item, DialogSelect returns a function result of TRUE with the dialog pointer in theDialog and the item number in itemHit; otherwise, it returns FALSE with theDialog and itemHit undefined. Normally when DialogSelect returns TRUE, you'll do whatever is appropriate as a response to the event, and when it returns FALSE you'll do nothing.

If the event is an activate or update event in a dialog window, DialogSelect activates or updates the window and returns FALSE.

If the mouse button is pressed in an editText item, DialogSelect responds to the mouse activity as appropriate (displaying an insertion point or selecting text). If a key down event occurs and there's an editText item, text entry and editing are handled in the standard way for such items. In either case, DialogSelect returns TRUE if the item is enabled or FALSE if it's disabled. If a key down event occurs when there's no editText item, DialogSelect returns FALSE.

(hand)

> To treat a typed character in a special way (such as ignore it, or make it have the same effect as another character or as clicking a button), the application should test for a key down event with that character before calling DialogSelect.

If the mouse button is pressed in a control, DialogSelect calls the Control Manager function TrackControl. If the mouse button is released inside the control and the control is enabled, DialogSelect returns TRUE; otherwise, it returns FALSE.

If the mouse button is pressed in any other enabled item, DialogSelect returns TRUE. If it's pressed in any other disabled item or in no item, or if any other event occurs, DialogSelect returns FALSE.

```
PROCEDURE ModalDialog (filterProc: ProcPtr; VAR itemHit: INTEGER);
```

Call ModalDialog after creating a modal dialog and bringing up its window in the frontmost plane. ModalDialog repeatedly gets and handles events in the dialog's window; after handling an event involving an

enabled dialog item, it returns with the item number in itemHit. Normally you'll then do whatever is appropriate as a response to an event in that item.

ModalDialog gets each event by calling the Toolbox Event Manager function GetNextEvent. If the event is a mouse down event outside the content region of the dialog window, ModalDialog emits sound number 1 (which should be a single beep) and gets the next event; otherwise, it filters and handles the event as described below.

(hand)

> Once before getting each event, ModalDialog calls SystemTask, a Desk Manager procedure that needs to be called regularly if the application is to support the use of desk accessories.

The filterProc parameter determines how events are filtered. If it's NIL, the standard filterProc is executed; this causes ModalDialog to return 1 in itemHit if the Return key or Enter key is pressed. If filterProc isn't NIL, ModalDialog filters events by executing the function it points to. The filterProc should have three parameters and should return a boolean value. For example, this is how it would be declared if it were named MyFilter:

```
FUNCTION MyFilter (theDialog: DialogPtr; VAR theEvent:
            EventRecord; VAR item: itemHit) : BOOLEAN;
```

A function result of FALSE tells ModalDialog to go ahead and handle the event, which either can be sent through unchanged or can be changed to simulate a different event. A function result of TRUE tells ModalDialog to return immediately rather than handle the event; in this case, the filterProc sets itemHit to the item number that ModalDialog should return.

You can use the filterProc, for example, to treat a typed character in a special way (such as ignore it, or make it have the same effect as another character or as clicking a button); in this case, the filterProc would test for a key down event with that character. If you want it to be consistent with the standard filterProc, your filterProc should at least check whether the Return key or Enter key was pressed and, if so, return 1 in itemHit and a function result of TRUE.

As another example, suppose the dialog box contains a userItem whose procedure draws a clock with the current time displayed. The filterProc can call that procedure and return FALSE without altering the current event.

ModalDialog handles the events returned by the filterProc as follows:

- If the mouse button is pressed in an editText item, ModalDialog responds to the mouse activity as appropriate (displaying an insertion point or selecting text). If a key down event occurs and there's an editText item, text entry and editing are handled in the standard way for such items. In either case, ModalDialog

returns TRUE if the item is enabled or FALSE if it's disabled. If a key down event occurs when there's no editText item, ModalDialog does nothing.

- If the mouse button is pressed in a control, ModalDialog calls the Control Manager function TrackControl. If the mouse button is released inside the control and the control is enabled, ModalDialog returns; otherwise, it does nothing.

- If the mouse button is pressed in any other enabled item in the dialog box, ModalDialog returns. If the mouse button is pressed in any other disabled item or in no item, or if any other event occurs, ModalDialog does nothing.

PROCEDURE DrawDialog (theDialog: DialogPtr);

DrawDialog draws the contents of the given dialog box. Since DialogSelect and ModalDialog handle dialog window updating, this procedure is useful only in unusual situations.

Invoking Alerts

FUNCTION Alert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;

This function invokes the alert defined by the alert template that has the given resource ID. It calls the current sound procedure, if any, passing it the sound number specified in the alert template for this stage of the alert. If no alert box is to be drawn at this stage, Alert returns a function result of -1; otherwise, it creates and displays the alert window for this alert and draws the alert box.

(hand)

It creates the alert window by calling NewDialog, and does the rest of its processing by calling ModalDialog.

Alert repeatedly gets and handles events in the alert window until an enabled item is clicked, at which time it returns the item number. Normally you'll then do whatever is appropriate in response to a click of that item.

Alert gets each event by calling the Toolbox Event Manager function GetNextEvent. If the event is a mouse down event outside the content region of the alert window, Alert emits sound number 1 (which should be a single beep) and gets the next event; otherwise, it filters and handles the event as described below.

The filterProc parameter has the same meaning as in ModalDialog (see above). If it's NIL, the standard filterProc is executed, which makes the Return key or the Enter key have the same effect as clicking the default button. If you specify your own filterProc and want to retain

this feature, you must include it in your filterProc. You can find out what the current default button is by looking at the aDefItem field of the dialog record for the alert (via the dialog pointer passed to the filterProc).

Alert handles the events returned by the filterProc as follows:

- If the mouse button is pressed in a control, Alert calls the Control Manager procedure TrackControl. If the mouse button is released inside the control and the control is enabled, Alert returns; otherwise, it does nothing.
- If the mouse button is pressed in any other enabled item, Alert simply returns. If it's pressed in any other disabled item or in no item, or if any other event occurs, Alert does nothing.

Before returning to the application with the item number, Alert removes the alert box from the screen. (It disposes of the alert window and its associated data structures, the item list, and the items.)

(hand)

> The Alert function's removal of the alert box would not be the desired result if the user clicked a check box or radio button; however, normally alerts contain only static text, icons, pictures, and buttons that are supposed to make the alert box go away. If your alert contains other items besides these, consider whether it might be more appropriate as a dialog.

```
FUNCTION StopAlert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
```

StopAlert is the same as the Alert function (above) except that before drawing the items of the alert in the alert box, it draws the Stop icon in the top left corner of the box (within the rectangle (10,20,42,52)). The Stop icon is the icon having the resource ID 0. If the application's resource file doesn't include an icon with that ID number, the standard Stop icon in the system resource file is used.

```
FUNCTION NoteAlert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
```

NoteAlert is the same as the Alert function (above) except that before drawing the items of the alert in the alert box, it draws the Note icon in the top left corner of the box (within the rectangle (10,20,42,52)). The Note icon is the icon having the resource ID 1. If the application's resource file doesn't include an icon with that ID number, the standard Note icon in the system resource file is used.

```
FUNCTION CautionAlert (alertID: INTEGER; filterProc: ProcPtr) :
          INTEGER;
```

CautionAlert is the same as the Alert function (above) except that before drawing the items of the alert in the alert box, it draws the Caution icon in the top left corner of the box (within the rectangle (10,20,42,52)). The Caution icon is the icon having the resource ID 2. If the application's resource file doesn't include an icon with that ID number, the standard Caution icon in the system resource file is used.

```
PROCEDURE CouldAlert (alertID: INTEGER);
```

CouldAlert ensures that the alert template having the given resource ID is in memory and makes it unable to be purged. It does the same for the alert window's definition function, the alert's item list, and any items defined as resources. This is useful if the alert may occur when the resource file isn't accessible, such as during a disk copy.

```
PROCEDURE FreeAlert (alertID: INTEGER);
```

Given the resource ID of an alert template previously specified in a call to CouldAlert (above), FreeAlert undoes the effect of CouldAlert. It should be called when there's no longer a need to keep the resources in memory.

Manipulating Items in Dialogs and Alerts

```
PROCEDURE ParamText (param0,param1,param2,param3: Str255);
```

ParamText provides a means of substituting text in statText items: param0 through param3 will replace the special strings "^0" through "^3" in all statText items in all subsequent dialog or alert boxes. Pass empty strings for parameters not used.

> <u>Assembly-language note</u>: Assembly-language programmers may pass NIL for parameters not used or for strings that are not to be changed.

For example, if the text is defined as "Cannot open document ^0" and docName is a string variable containing a document name that the user typed, you can call ParamText(docName,'','','').

(eye)

> All strings that will need to be translated to foreign languages should be stored in resource files.

Assembly-language note: The Dialog Manager stores handles to the four ParamText parameters in a system global array named daStrings.

PROCEDURE GetDItem (theDialog: DialogPtr; itemNo: INTEGER; VAR type: INTEGER; VAR item: Handle; VAR box: Rect);

GetDItem returns in its VAR parameters the following information about the item numbered itemNo in the given dialog's item list: in the type parameter, the item type; in the item parameter, a handle to the item (or, for item type userItem, the procedure pointer); and in the box parameter, the display rectangle for the item.

Suppose, for example, that you want to change the title of a control in a dialog box. You can get the item handle with GetDItem, convert it to a control handle, and call the Control Manager procedure SetCTitle to change the title.

(hand)

To access the text of a statText or editText item, pass the handle returned by GetDItem to GetIText or SetIText (see below).

PROCEDURE SetDItem (theDialog: DialogPtr; itemNo: INTEGER; type: INTEGER; item: Handle; box: Rect);

SetDItem sets the item numbered itemNo in the given dialog's item list, as specified by the parameters (without drawing the item). The type parameter is the item type; the item parameter is a handle to the item (or, for item type userItem, the procedure pointer); and the box parameter is the display rectangle for the item.

Consider, for example, how to install an item of type userItem in a dialog: In the item list in the resource file, define an item in which the type is set to userItem and everything else is set to Ø. Specify that the dialog window be invisible (in either the dialog template or the NewDialog call). After creating the dialog, convert the item's procedure pointer to a handle; then call SetDItem, passing that handle and the display rectangle for the item. Finally, call the Window Manager procedure ShowWindow to display the dialog window.

(hand)

Do not use SetDItem to change the text of a statText or editText item; call GetDItem to get a handle to the item and then call SetIText (see below).

PROCEDURE GetIText (item: Handle; VAR text: Str255);

Given a handle to a statText or editText item in a dialog box, as returned by GetDItem, GetIText returns the text of the item in the text parameter.

PROCEDURE SetIText (item: Handle; text: Str255);

Given a handle to a statText or editText item in a dialog box, as returned by GetDItem, SetIText sets the text of the item to the specified text and draws the item. For example, suppose the exact content of a dialog's text item cannot be determined until the application is running, but the display rectangle is defined in the resource file: Call GetDItem to get a handle to the item, and call SetIText with the desired text.

```
PROCEDURE SelIText (theDialog: DialogPtr; itemNo: INTEGER;
            strtSel,endSel: INTEGER);
```

Given a pointer to a dialog and the item number of an editText item in the dialog box, SelIText does the following:

- If the item contains text, SelIText sets the selection range to extend from character position strtSel up to but not including character position endSel. The selection range is inversely highlighted unless strtSel equals endSel, in which case a blinking vertical bar is displayed to indicate an insertion point at that position.

- If the item doesn't contain text, SelIText simply displays the insertion point.

For example, if the user makes an unacceptable entry in the editText item, the application can put up an alert box reporting the problem and then select the entire text of the item so it can be replaced by a new entry. (Without this procedure, the user would have to select the item by dragging with the mouse before making the new entry.)

(hand)
You can select the entire text by specifying 0 for strtSel and a very large number for endSel. For details about selection range and character position, see the TextEdit manual or the CoreEdit manual.

MODIFYING TEMPLATES IN MEMORY

When you call GetNewDialog or one of the routines that invokes an alert, the Dialog Manager calls the Resource Manager to read the dialog or alert template from the resource file and return a handle to it. If the template is already in memory, the Resource Manager just returns a

handle to it. If you want, you can call the Resource Manager yourself to read the template into memory (and make it unpurgeable), and then make changes to it before calling the dialog or alert routine. When called by the Dialog Manager, the Resource Manager will return a handle to the template as you modified it.

To modify a template in memory, you need to know its exact structure and the data type of the handle through which it may be accessed. These are discussed below for dialogs and alerts.

Dialog Templates in Memory

The data structure of a dialog template is as follows:

```
TYPE DialogTemplate = RECORD
                        boundsRect: Rect;
                        procID:     INTEGER;
                        visible:    BOOLEAN;
                        filler1:    BOOLEAN;
                        goAwayFlag: BOOLEAN;
                        filler2:    BOOLEAN;
                        refCon:     LongInt;
                        itemsID:    INTEGER;
                        title:       Str255
                      END;
```

The filler1 and filler2 fields are not used; they're there only to ensure that the goAwayFlag and refCon fields begin on a word boundary. The itemsID field contains the resource ID of the dialog's item list. The other fields are the same as the parameters of the same name in the NewDialog function.

You access the dialog template by converting the handle returned by the Resource Manager to a template handle.

```
TYPE DialogTPtr  = ^DialogTemplate;
     DialogTHndl = ^DialogTPtr;
```

For example, if dHandle is a variable of type DialogTHndl, you can do the following:

```
dHandle := POINTER(ORD(GetResource('DLOG',3)));
dHandle^^.visible := FALSE
```

The Resource Manager function GetResource takes the resource type and resource ID as parameters and returns a handle to the resource. You use ORD and POINTER to make it have the data type DialogTHndl.

Alert Templates in Memory

The data structure of an alert template is as follows:

```
TYPE AlertTemplate = RECORD
                       boundsRect: Rect;
                       itemsID:    INTEGER;
                       stages:     StageList
                     END;
```

BoundsRect is the rectangle that becomes the portRect of the window's grafPort. The itemsID field contains the resource ID of the item list for the alert.

The information in the stages field determines exactly what should happen at each stage of the alert. It's packed into a word that has the following structure:

```
TYPE StageList = PACKED ARRAY [1..4] OF RECORD
                                          boldItem:  Ø..1;
                                          boxDrawn:  BOOLEAN;
                                          sound:     Ø..3
                                        END;
```

The elements of the StageList array are stored in reverse order of the stages: element 1 is for the fourth stage, and element 4 is for the first stage.

BoldItem indicates which button should be the default button (and therefore boldly outlined in the alert box). If the first two items in the alert's item list are the OK button and the Cancel button, respectively, Ø will refer to the OK button and 1 to the Cancel button. The reason for this is that the value of boldItem plus 1 is interpreted as an item number, and normally items 1 and 2 are the OK and Cancel buttons, respectively. Whatever the item having the corresponding item number happens to be, a bold rounded-corner rectangle will be drawn around its display rectangle.

(eye)

> When deciding where to place items in an alert box, be sure to allow room for any bold outlines that may be drawn.

BoxDrawn is TRUE if the alert box is to be drawn.

The sound field specifies which sound should be emitted at this stage of the alert, with a number from Ø to 3 that's passed to the current sound procedure. You can call ErrorSound to specify your own sound procedure; if you don't, sound number Ø will represent no sound, and sound numbers 1 through 3 will be the corresponding number of short beeps.

You access the alert template by converting the handle returned by the Resource Manager to a template handle.

```
TYPE AlertTPtr  = ^AlertTemplate;
     AlertTHndl = ^AlertTPtr;
```

For example, if aHandle is a variable of type AlertTHndl, you can do the following:

```
aHandle := POINTER(ORD(GetResource('ALRT',1)));
aHandle^^.boxHeight := 50
```

Assembly-language note: Rather than offsets into the fields of the StageList data structure, the Toolbox equates file contains masks for accessing the information stored for an alert stage in a stages word. It also contains the system globals aNumber and aCount, which provide information about the last occurrence of an alert: its resource ID and its stage (as a number from 0 to 3).

FORMATS OF RESOURCES FOR DIALOGS AND ALERTS

Every dialog template, alert template, and item list must be stored in a resource file, as must any icons or QuickDraw pictures in item lists and any control templates for items of type ctrlItem+resCtrl. The exact formats of a dialog template, alert template, and item list in a resource file are given below. For icons and pictures, the resource type is 'ICON' or 'PICT' and the resource data is simply the icon or the picture. The format of a control template is discussed in the Control Manager manual.

Dialog Templates in a Resource File

The resource type for a dialog template is 'DLOG', and the resource data has the same format as a dialog template in memory.

Number of bytes	Contents
8 bytes	Same as boundsRect parameter to NewDialog
2 bytes	Same as procID parameter to NewDialog
1 byte	Same as visible parameter to NewDialog
1 byte	Ignored
1 byte	Same as goAwayFlag parameter to NewDialog
1 byte	Ignored
4 bytes	Same as refCon parameter to NewDialog
2 bytes	Resource ID of item list
n bytes	Same as title parameter to NewDialog (1-byte length in bytes, followed by the characters of the title)

Alert Templates in a Resource File

The resource type for an alert template is 'ALRT', and the resource data has the same format as an alert template in memory.

Number of bytes	Contents
8 bytes	Rectangle enclosing alert window
2 bytes	Resource ID of item list
2 bytes	Stages

The resource data ends with a word of information about stages. As illustrated in Figure 6, there are four bits of stage information for each of the four stages, from the four low-order bits for the first stage to the four high-order bits for the fourth stage. Each set of four bits is as follows:

Number of bits	Contents
1 bit	Item number minus 1 of default button; normally Ø is OK and 1 is Cancel
1 bit	1 if alert box is to be drawn; Ø if not
2 bits	Sound number (Ø through 3)

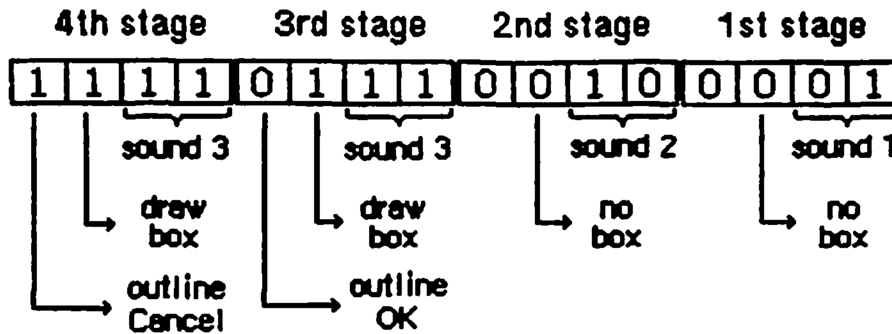


Figure 6. Sample Stages Word

(hand)

So that the disk won't be accessed just for an alert that beeps, you may want to set the resPreLoad attribute of the alert's template in the resource file. For more information, see the Resource Manager manual.

Item Lists in a Resource File

The resource type for an item list is 'DITL'. The resource data begins with a word containing the number of items in the list minus 1. This is what follows for each item:

Number of bytes	Contents	
4 bytes	Ø (placeholder for handle or procedure pointer)	
8 bytes	Display rectangle (local coordinates)	
1 byte	Item type	
1 byte	Length of following data in bytes	
n bytes	If item type is:	Content is:
(n is even)	ctrlItem+resCtrl	Resource ID (length 2)
	any other ctrlItem	Title of the control
	statText, editText	The text
	iconItem, picItem	Resource ID (length 2)
	userItem	Empty (length Ø)

As shown here, the first four bytes serve as a placeholder for the item's handle or, for item type userItem, its procedure pointer; the handle or pointer is stored after the item list is read into memory. The next eight bytes define the display rectangle for the item, and the next byte gives the length of the data that follows: for a text item, it's the text itself; for an icon, picture, or control of type ctrlItem+resCtrl, it's the two-byte resource ID for the item; and for

any other type of control, it's the title of the control. For userItems, no data follows the item type. When the data is text or a control title, the number of bytes it occupies must be even to ensure word alignment of the next item.

Assembly-language note: The Toolbox equates file contains offsets into the fields of an item list.

SUMMARY OF THE DIALOG MANAGER

Constants

```
CONST ctrlItem    = 4;    {add to following four constants}
      btnCtrl     = 0;    {standard button control}
      chkCtrl     = 1;    {standard check box control}
      radCtrl     = 2;    {standard "radio button" control}
      resCtrl     = 3;    {control defined in control template}
      statText    = 8;    {static text}
      editText    = 16;   {editable text (dialog only)}
      iconItem    = 32;   {icon}
      picItem     = 64;   {QuickDraw picture}
      userItem    = 0;    {application-defined item (dialog only)}
      itemDisable = 128;  {add to any of above to disable}

      OK     = 1;
      Cancel = 2;
```

Data Structures

```
TYPE DialogPtr    = WindowPtr;
     DialogPeek   = ^DialogRecord;
     DialogRecord = RECORD
                      window:    WindowRecord;
                      items:     Handle;
                      textH:     TEHandle;
                      editField: INTEGER;
                      editOpen:  INTEGER;
                      aDefItem:  INTEGER
                    END;

     DialogTHndl    = ^DialogTPtr;
     DialogTPtr     = ^DialogTemplate;
     DialogTemplate = RECORD
                        boundsRect: Rect;
                        procID:     INTEGER;
                        visible:    BOOLEAN;
                        filler1:    BOOLEAN;
                        goAwayFlag: BOOLEAN;
                        filler2:    BOOLEAN;
                        refCon:     LongInt;
                        itemsID:    INTEGER;
                        title:      Str255
                      END;

     AlertTHndl     = ^AlertTPtr;
     AlertTPtr      = ^AlertTemplate;
```

```
    AlertTemplate = RECORD
                        boundsRect: Rect;
                        itemsID:    INTEGER;
                        stages:     StageList
                    END;
    StageList = PACKED ARRAY [1..4] OF RECORD
                                          boldItem:  Ø..1;
                                          boxDrawn:  BOOLEAN;
                                          sound:     Ø..3
                                       END;
```

Routines

Initialization

```
PROCEDURE InitDialogs (restartProc: ProcPtr);
PROCEDURE ErrorSound  (soundProc: ProcPtr);
```

Creating and Disposing of Dialogs

```
FUNCTION  NewDialog    (dStorage: Ptr; boundsRect: Rect; title: Str255;
                        visible: BOOLEAN; procID: INTEGER; behind:
                        WindowPtr; goAwayFlag: BOOLEAN; refCon: LongInt;
                        items: Handle) : DialogPtr;
FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr; behind:
                        WindowPtr) : DialogPtr;
PROCEDURE CloseDialog  (theDialog: DialogPtr);
PROCEDURE DisposDialog (theDialog: DialogPtr);
```

Handling Dialog Events

```
FUNCTION  IsDialogEvent (theEvent: EventRecord) : BOOLEAN;
FUNCTION  DialogSelect  (theEvent: EventRecord; VAR theDialog: DialogPtr;
                         VAR itemHit: INTEGER) : BOOLEAN;
PROCEDURE ModalDialog   (filterProc: ProcPtr; VAR itemHit: INTEGER);
PROCEDURE DrawDialog    (theDialog: DialogPtr);
```

Invoking Alerts

```
FUNCTION  Alert        (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
FUNCTION  StopAlert    (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
FUNCTION  NoteAlert    (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
FUNCTION  CautionAlert (alertID: INTEGER; filterProc: ProcPtr) : INTEGER;
PROCEDURE CouldAlert   (alertID: INTEGER);
PROCEDURE FreeAlert    (alertID: INTEGER);
```

Manipulating Items in Dialogs and Alerts

```
PROCEDURE ParamText (param0,param1,param2,param3: Str255);
PROCEDURE GetDItem  (theDialog: DialogPtr; itemNo: INTEGER; VAR type:
                     INTEGER; VAR item: Handle; VAR box: Rect);
PROCEDURE SetDItem  (theDialog: DialogPtr; itemNo: INTEGER; type:
                     INTEGER; item: Handle; box: Rect);
PROCEDURE GetIText  (item: Handle; VAR text: Str255);
PROCEDURE SetIText  (item: Handle; text: Str255);
PROCEDURE SelIText  (theDialog: DialogPtr; itemNo: INTEGER; strtSel,
                     endSel: INTEGER);
```

FilterProc for Modal Dialogs and Alerts

```
FUNCTION MyFilter (theDialog: DialogPtr; VAR theEvent: EventRecord;
                   VAR item: itemHit) : BOOLEAN;
```

Assembly-Language Information

Dialog Record Data Structure

dWindow	Dialog window
items	Resource ID of dialog's item list
teHandle	Handle to editable text for TextEdit
editField	Item number minus 1 of editText item
editOpen	Used internally
aDefItem	Item number of default button
dWindLen	Length of dialog record

Dialog Template Data Structure

dBounds	Rectangle that becomes portRect of alert window's grafPort
dWindProc	Window definition ID
dVisible	Whether dialog window is visible
dGoAway	Whether dialog window has a close box
dRefCon	Dialog window's reference value
dItems	Resource ID of dialog's item list
dTitle	Dialog window's title

Alert Template Data Structure

aBounds	Rectangle that becomes portRect of dialog window's grafPort
aItems	Resource ID of alert's item list
aStages	Stages word; information for alert stages

Item List Data Structure

dlgMaxIndex	Number of items minus 1
itmHndl	Handle or procedure pointer for this item
itmRect	Display rectangle for this item
itmType	Item type for this item
itmData	Length byte followed by that many bytes of data for this item (must be even length)

Masks for Alert Stages Word

```
volBits       .EQU   3    ;sound number
alBit         .EQU   4    ;whether to draw box
okDismissal   .EQU   8    ;item number minus 1 of default button
```

System Globals

Name	Size	Contents
restProc	4 bytes	Address of restart fail-safe procedure
daStrings	16 bytes	Handles to ParamText strings
daBeeper	4 bytes	Address of current sound procedure
dlgFont	2 bytes	Font number for NewDialog
aNumber	2 bytes	Resource ID of last alert
aCount	2 bytes	Stage number of last alert (Ø through 3)

GLOSSARY

alert: A warning or report of an error, in the form of an alert box, sound from the Macintosh's speaker, or both.

alert box: A box that appears on the screen to give a warning or report an error during a Macintosh application.

alert template: A resource that contains information from which the Dialog Manager can create an alert.

alert window: The window in which an alert box is displayed.

default button: In an alert box or modal dialog, the button whose effect will occur if the user presses the Return key or the Enter key. In an alert box, it's boldly outlined; in a modal dialog, it's boldly outlined or the OK button.

dialog: Same as dialog box.

dialog box: A box that a Macintosh application displays to request information it needs to complete a command, or to report that it's waiting for a process to complete.

dialog record: The internal representation of a dialog, where the Dialog Manager stores all the information it needs for its operations on that dialog.

dialog template: A resource that contains information from which the Dialog Manager can create a dialog.

dialog window: The window in which a dialog box is displayed.

disabled: A disabled item in a dialog or alert box has no effect when clicked.

display rectangle: A rectangle that determines where an item is displayed within a dialog or alert box.

icon: A 32-by-32 bit image that graphically represents an object, concept, or message.

item: In dialog and alert boxes, a control, icon, picture, or piece of text, each displayed inside its own display rectangle.

item list: A list of information about all the items in a dialog or alert box.

item number: The index, starting from 1, of an item in an item list.

modal dialog: A dialog that requires the user to respond before doing any other work on the desktop.

modeless dialog: A dialog that allows the user to work elsewhere on the desktop before responding.

sound procedure: A procedure that will emit one of up to four sounds from the Macintosh's speaker. Its integer parameter ranges from Ø to 3 and specifies which sound.

stage: Every alert has four stages, corresponding to consecutive occurrences of the alert, and a different response may be specified for each stage.

MACINTOSH USER EDUCATION

The Event Manager: A Programmer's Guide /EMGR/EVENTS

See also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide

Modification History: First Draft (ROM 4) S. Chernicoff 6/20/83

ABSTRACT

The Macintosh Event Manager is your program's link to its human user, allowing it to monitor the user's actions with the mouse, keyboard, and keypad. A typical Macintosh application program is event-driven: it decides what to do from moment to moment by asking the Event Manager for events and responding to them one by one, in whatever way is appropriate. The Event Manager is also used for various purposes within the Toolbox itself, such as to coordinate the ordering and display of windows on the screen. Finally, you can use the Event Manager as a means of communication between parts of your own program.

TABLE OF CONTENTS

3 About This Manual
4 About the Event Manager
5 Event Types
6 Priority of Events
7 Keyboard Events
9 Event Records
12 Event Masks
14 Using the Event Manager
17 Event Manager Routines
17 Accessing Events
18 Posting and Removing Events
19 Reading the Mouse
20 Reading the Keyboard and Keypad
22 Miscellaneous Utilities
22 Journaling
23 Resource Format for Keyboard Configurations
24 Notes for Assembly-Language Programmers
25 Appendix: Standard Key and Character Codes
35 Summary of the Event Manager
37 Glossary

ABOUT THIS MANUAL

This manual describes the Event Manager, the part of the Macintosh User Interface Toolbox that allows your program to monitor the user's actions with the mouse, keyboard, and keypad. *** Eventually it will become part of a larger manual describing the entire Toolbox. *** The Event Manager is also used for various purposes within the Toolbox itself, such as to coordinate the ordering and display of windows on the screen. Finally, you can use the Event Manager as a means of communication between parts of your own program.

(eye)

> This manual describes version 4 of the Macintosh ROM. If you're using a different version, the Event Manager may not work exactly as described here.

Actually, there are two Event Managers: one in the Operating System and one in the Toolbox. The Toolbox Event Manager calls the one in the Operating System and serves as an interface between it and your application program; it also adds some features that aren't present at the Operating System level, such as the window management facilities mentioned above. This manual describes the Toolbox Event Manager, which is ordinarily the one your program will be dealing with. All references to "the Event Manager" should be understood to refer to the Toolbox Event Manager. For information on the Operating System's Event Manager, see the Macintosh Operating System Reference Manual.

Like all Toolbox documentation, this manual assumes you are familiar with the Macintosh User Interface Guidelines and with Lisa Pascal. You should also have at least a general notion of what the Window Manager, Desk Manager, Menu Manager, Control Manager, and Resource Manager do. It would also be helpful to have some familiarity with a Macintosh application program as an illustration of the concepts presented here.

The manual begins with an introduction to the Event Manager and what you can do with it. It then discusses the various types of event, their relative priority, and how the user's keyboard actions, in particular, are reported in the form of events. Next come sections on the structure of event records, which contain all the pertinent information about each event, and event masks, which some of the Event Manager routines expect as parameters.

A section on using the Event Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Event Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not be of interest to all readers. Special information is given on the Event Manager's journaling mechanism, which allows your program's interactions with the user to be recorded and played back later; on the format used in resource files for storing a keyboard configuration, which determines

what character each key on the keyboard stands for; and on how to use the Event Manager routines from assembly language.

Finally, there are an appendix containing detailed information on the standard Macintosh character set and keyboard configuration, a quick-reference summary of the Event Manager data structures and routines, and a glossary of terms used in this manual.

ABOUT THE EVENT MANAGER

The Macintosh Event Manager is your program's link to its human user. Whenever the user presses the mouse button, types on the keyboard or keypad, or inserts a disk in a disk drive, your program is notified by means of an event. A typical Macintosh application progam is event-driven: it decides what to do from moment to moment by asking the Event Manager for events and responding to them one by one, in whatever way is appropriate.

Although the Event Manager's primary purpose is to monitor the user's actions and pass them to your program in an orderly way, it also serves as a convenient mechanism for sending signals from one part of a program to another. For instance, the Window Manager uses events to coordinate the ordering and display of windows as the user activates and deactivates them and moves them around on the Macintosh screen. You can also define your own types of event and use them in any way your application calls for.

Events waiting to be processed are kept in the event queue. In principle, the event queue is a FIFO (first-in-first-out) list: events are added to the queue (posted) at one end and retrieved from the other. You can think of the queue as a funnel that collects events from a variety of sources and feeds them to your program on demand, in the order they occurred. (There are a few exceptions to the strict FIFO ordering, which will be discussed later.)

(eye)

> The event queue has a limited capacity *** (currently 30 events, but may change) ***. When the queue becomes full, the Event Manager begins throwing out old events to make room for new ones as they're posted. The event thrown out is always the oldest one in the queue.

Using the Event Manager, your program can:

- Retrieve events one at a time from the event queue
- Control which types of event get posted and which are ignored
- Post events of its own
- Read the current state of the keyboard, keypad, and mouse button

- Monitor the location of the mouse

- Read the system clock to find out how much time has elapsed since the system was last started up

Another important service provided by the Event Manager is journaling. This feature enables your program to record all its interactions with the Event Manager and play them back later.

EVENT TYPES

Events are of various types, depending on their origin and meaning. Some report actions by the user, some are generated by the Window Manager, some *** (not yet implemented) *** arise in the Macintosh's low-level input/output drivers, and some may be generated by your program itself for its own purposes. Some events are handled by the Desk Manager before your program ever sees them; others are left for your program to handle in its own way.

The most important event types, the ones the Event Manager was created to handle, are those that record actions by the user:

- Mouse down and mouse up events occur when the user presses or releases the mouse button.

- Key down and key up events occur when the user presses or releases a key on the keyboard or keypad. The Event Manager also automatically generates auto-key events when the user presses and holds down a repeating key. Together, these three event types are called keyboard events.

- Disk inserted events occur when the user inserts a disk into a disk drive.

- Abort events occur when the user presses a special combination of keys. *** Tentatively the combination is Command-period (Command-.), but this may change; there's also some possibility that more than one key combination will be provided to interrupt a running program in different ways or for different purposes. *** An abort event signals the program to stop whatever it's doing and return control directly to the user, allowing the user to interrupt a time-consuming process or regain control of a runaway program. An abort event can also be generated by the Event Manager's own journaling mechanism, signaling the program to reset itself to some standard initial state before replaying a journal.

(hand)
Mere movements of the mouse are not reported as events. If necessary, your program can keep track of them by periodically asking the Event Manager for the current location of the mouse.

The following event types are used by the Window Manager to coordinate the display of windows on the screen:

- Activate events are generated whenever an inactive window becomes active or vice versa. They generally occur in pairs (that is, one window is deactivated and another activated at the same time).

- Update events occur when a window's contents need to be redrawn, usually as a result of the user's opening, closing, activating, or moving a window.

Two more event types (I/O driver events and network events) are reserved for use by the low-level input/output system. *** At present, these types are not used at all. *** In addition, your program can define as many as four event types of its own and use them for whatever purposes you like.

One final type of event is the null event, which is what the Event Manager returns if it has no other events to report.

PRIORITY OF EVENTS

It was stated earlier that in principle the event queue is a FIFO list--that is, events are retrieved from the queue in the order they were originally posted. Actually, the way in which various types of event are generated and detected causes some to have higher priority than others. Furthermore, when you ask the Event Manager for an event, you can specify a particular type or types that are of interest. This can also alter the strict FIFO order, by causing some events to be passed over in favor of others that were actually posted later. Everything said in the following discussion is understood to be limited to the event types you've specifically requested in your Event Manager call.

The Event Manager always returns the highest-priority event available of the requested type(s). The priority ranking is as follows:

1. Activate (window becoming inactive before window becoming active)

2. Mouse down, mouse up, key down, key up, disk inserted, abort, network, I/O driver, application-defined (all in FIFO order)

3. Auto-key

4. Update (in front-to-back order)

5. Null

Activate events take priority over all others; they are detected in a special way, and are never actually placed in the event queue. The Event Manager checks for pending activate events before looking in the event queue, so it will always return such an event if one is available. Because of the special way activate events are detected,

there can never be more than two such events pending at the same time: one for a window becoming inactive and another for a window becoming active. If there's one of each, the event for the window becoming inactive is reported first.

Category 2 includes most of the possible event types. Within this category, events are normally retrieved from the queue in the order they were posted.

If no event is available in categories 1 and 2, the Event Manager next checks to see whether the appropriate conditions hold for an auto-key event. (These conditions are described in detail in the next section.) If so, it generates one and returns it to your program.

Next in priority are update events. Like activate events, these are not placed in the event queue, but are detected in another way. If no higher-priority event is available, the Event Manager checks for windows whose contents need to be redrawn. If it finds one, it generates and returns an update event for that window. Windows are checked in the order in which they're displayed on the screen, from front to back, so if two or more windows need to be updated, an update event will be generated for the frontmost such window.

Finally, if no other event is available, the Event Manager returns a null event.

KEYBOARD EVENTS

Every key on the Macintosh keyboard and the optional keypad generates key down and key up events when pressed and released. (Exceptions are the modifier keys--Shift, Caps Lock, Command *** name may change ***, and Option. These keys are treated specially, as described below, and generate no keyboard events of their own.) In addition, the Event Manager itself generates auto-key events whenever you request an event and all of the following conditions apply:

- No higher-priority event of the requested type(s) is available
- The user is currently holding down a key other than a modifier key
- The appropriate time interval (see below) has elapsed since the last keyboard event
- Auto-key events are one of the types you've requested
- Auto-key events are one of the types currently being posted into the event queue

Two different time intervals are taken into account. Auto-key events begin to be generated after a certain initial delay has elapsed since the original key down event (that is, since the key was originally pressed). Thereafter, they are generated each time a certain repeat

interval has elapsed since the last auto-key event. The initial settings for these two intervals are 16 ticks (sixtieths of a second) for the initial delay and 4 ticks for the repeat interval. The user can adjust these settings to individual preference with the control panel desk accessory.

When the user presses, holds down, or releases a key, the resulting keyboard event identifies the key in two different ways: with a key code designating the key itself and a character code designating the character the key stands for. Character codes are given in the extended version of ASCII (the American Standard Code for Information Interchange) used by Macintosh and Lisa; see the Appendix for further information.

The association between keys and characters is defined by a keyboard configuration. The particular character a key generates depends on three things:

- The key itself

- The keyboard configuration currently in effect

- Which, if any, of the modifier keys were held down when the key was pressed

As mentioned earlier, the modifier keys don't generate keyboard events of their own. Instead, they modify the meaning of the other keys by changing the character codes that those keys generate. For example, under the standard Macintosh keyboard configuration, the "C" key generates a lowercase letter c when pressed by itself; when pressed with the Shift or Caps Lock key down, it generates a capital C; with the Option key down, a lowercase c with a cedilla (ç), used in French, Portuguese, and a few other foreign languages; and with Option and Shift or Option and Caps Lock down, a capital C with a cedilla (Ç). The state of each of the option keys is also reported in a field of the event record (see next section), where your program can examine it directly.

Keyboard configurations are handled as resources and stored in resource files. The standard keyboard configuration gives each key its normal ASCII character code according to the standard Macintosh keyboard layout, as shown in the Appendix. When the Option key is held down, most keys generate special characters with codes between 128 and 255 ($80 and $FF), included in the extended character set for business, scientific, and international use.

(hand)

> Notice that under the standard keyboard configuration only the Shift, Caps Lock, and Option keys actually modify the character a key stands for: the Command key has no effect on the character code generated. (Keyboard configurations other than the standard may take the Command key into account.) Similarly, character codes for the keypad are affected only by the Shift key. To

find out whether the Command key was down at the time of an event (or Caps Lock or Option in the case of one generated from the keypad), you have to examine the appropriate field of the event record.

Normally you'll just want to use the standard keyboard configuration, which is read from the system resource file every time the Macintosh is started up. Other keyboard configurations can be used to reconfigure the keyboard for foreign use or for nonstandard layouts such as the Dvorak arrangement. In rare cases, you may want to define your own keyboard configuration to suit your program's special needs. For information on how to install an alternate keyboard configuration or define one of your own, see "Resource Format for Keyboard Configurations" and "Notes for Assembly-Language Programmers", below.

EVENT RECORDS

Every event is represented internally by an event record containing all pertinent information about that event. The event record includes the following information:

- The type of event
- The time the event was posted
- The location of the mouse at the time the event was posted
- The state of the mouse button and modifier keys at the time the event was posted
- Any additional information required for a particular type of event, such as which key the user pressed or which window is being activated

This information is filled into the event record for every event--even for null events, which just mean that nothing special has happened.

Event records are defined as follows:

```
TYPE EventRecord = RECORD
                     what:      INTEGER;
                     message:   LongInt;
                     when:      LongInt;
                     where:     Point;
                     modifiers: INTEGER
                   END;
```

The what field contains an event code identifying the type of the event. The Event Manager can handle a maximum of 16 different event types, denoted by event codes from Ø to 15. The following standard event codes are built into the Event Manager as predefined constants:

```
CONST nullEvent     = Ø;  {null}
      mouseDown     = 1;  {mouse down}
      mouseUp       = 2;  {mouse up}
      keyDown       = 3;  {key down}
      keyUp         = 4;  {key up}
      autoKey       = 5;  {auto-key}
      updateEvt     = 6;  {update}
      diskEvt       = 7;  {disk inserted}
      activateEvt   = 8;  {activate}
      abortEvt      = 9;  {abort}
      networkEvt    = 1Ø; {network}
      driverEvt     = 11; {I/O driver}
      app1Evt       = 12; {application-defined}
      app2Evt       = 13; {application-defined}
      app3Evt       = 14; {application-defined}
      app4Evt       = 15; {application-defined}
```

The when field contains the time the event was posted, in ticks (sixtieths of a second) since the system was last started up.

The where field gives the location of the mouse at the time the event was posted, expressed in global coordinates.

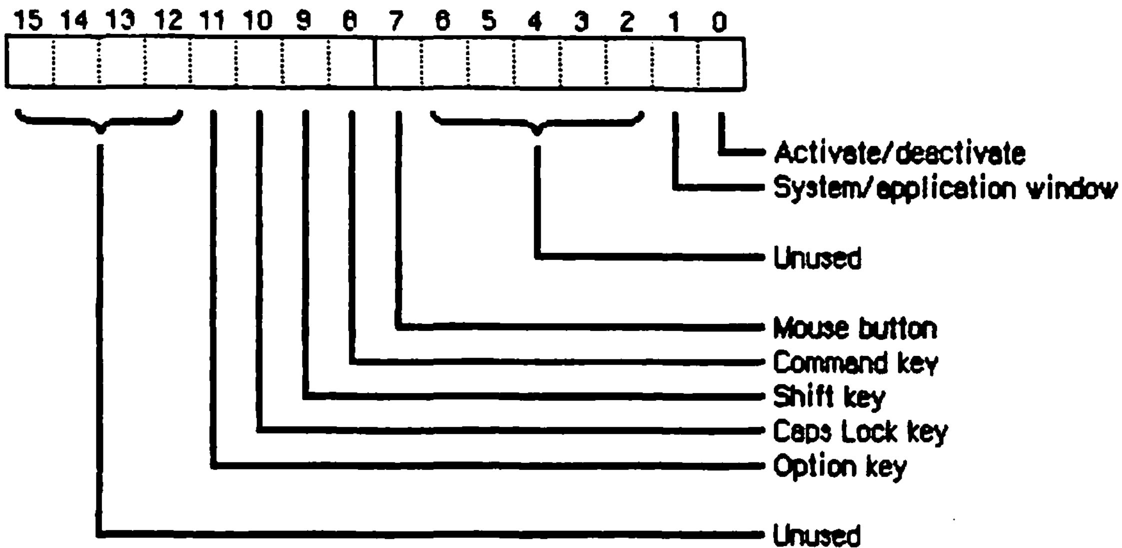


Figure 1. Modifier Bits

The modifiers field gives the state of the mouse button and the modifier keys at the time the event was posted, as shown below and in Figure 1. A 1 in any bit position means that that key or button was down; Ø means it was up. (Following the customary convention, the bit positions are numbered from right to left, starting from Ø at the low-order end; see Figure 1.)

Bit	Meaning
15-12	Unused
11	Option key
10	Caps Lock key
9	Shift key
8	Command key *** (name may change) ***
7	Mouse button
6-2	Unused
1-0	Used only by activate events (see below)

For activate events, the low-order bit of the modifiers field (bit 0) is set to 1 if a window is being activated, or to 0 if it is being deactivated. When one window is deactivated and another is activated at the same time (as is usually the case), bit 1 of the modifiers field is set to 1 if one of the windows involved belongs to your application program and the other is a system window (a window not created by your program, such as one containing a desk accessory); if they're both system or both application windows, this bit is set to 0. You can use this information to take some special action when the active window changes from an application window to a system window or vice versa: for example, you might want to hide a menu or dim some of its items when a system window becomes active and restore them when control returns to one of your program's own windows.

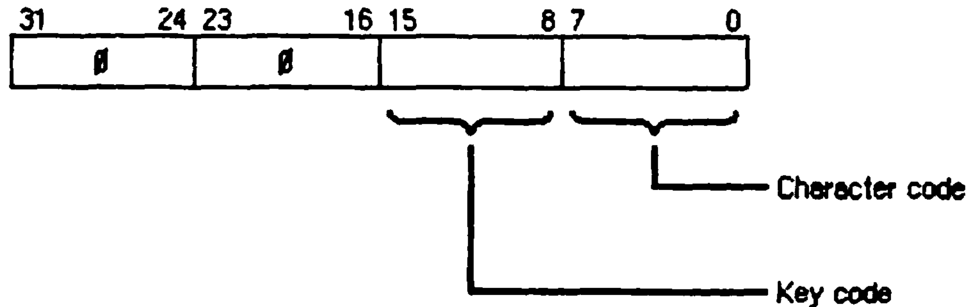


Figure 2. Event Message Format for Keyboard Events

The message field contains the event message, which conveys extra information specific to a particular event type:

- For keyboard events, the event message identifies the key that was pressed or released, as shown in Figure 2. The low-order byte (message MOD 256) contains the character code for the key, depending on the keyboard configuration currently in effect and on which, if any, of the modifier keys were held down. Under the

standard keyboard configuration this is just the normal ASCII code associated with the key, which is usually the information your program needs. The third byte (message DIV 256) gives the key code, useful in special cases (a music generator, for example) where you want to treat the keyboard as a set of buttons unrelated to specific characters. Detailed information on key and character codes for the standard Macintosh keyboard configuration is given in the Appendix. The first two bytes of the message are set to 0.

- For disk inserted events, the event message gives the drive number of the disk drive: 1 for the Macintosh's built-in drive, 2 for the external drive, if any. Numbers greater than 2 denote additional disk drives connected through the serial port. By the time your program receives a disk inserted event, the system will already have attempted to mount the volume that was inserted. If for any reason the attempt was unsuccessful (for example, if the user has inserted an unformatted disk), the high-order word of the event message will contain the error code returned by the Operating System; see the Operating System manual for further details.

- For activate and update events, the event message is a pointer to the window affected.

- For abort events, the event message identifies the key that the user pressed in order to interrupt the program. The format is the same as described above for keyboard events. For abort events generated by the Event Manager's own journaling mechanism, the message field is set to 0.

- For application-defined event types, you can use the event message for whatever information your application calls for.

- For mouse down, mouse up, and null events, the event message is meaningless and should be ignored.

EVENT MASKS

Several of the Event Manager routines can be restricted to a specific event type or group of types. For instance, instead of just requesting the next available event, you can ask specifically for the next keyboard event.

You specify which event types a particular Event Manager call applies to by supplying an event mask as a parameter. This is an integer in which each of the 16 bit positions stands for an event type, as shown in Figure 3. Notice that the bit position representing a given type corresponds to the event code for that type. For example, update events (type code 6) are specified by bit 6 of the mask, counting from 0 at the right (low-order) end. A 1 bit at that position means that this Event Manager call applies to update events; a 0 means it doesn't.

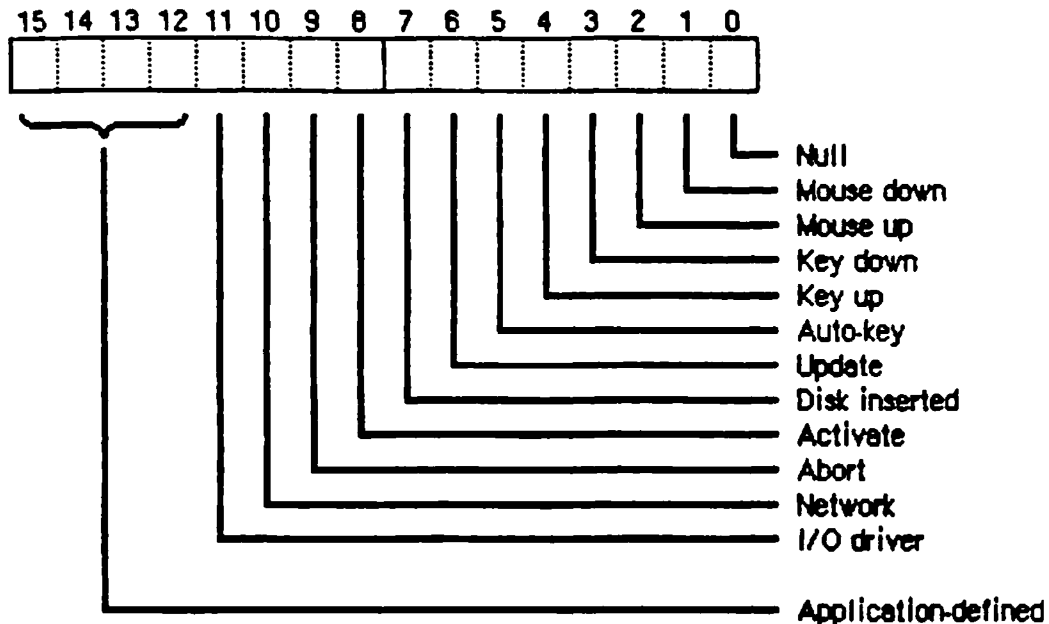


Figure 3. Event Mask

Masks for each single event type are built into the Event Manager as predefined constants:

```
CONST nullMask    = 1;       {null}
      mDownMask   = 2;       {mouse down}
      mUpMask     = 4;       {mouse up}
      keyDownMask = 8;       {key down}
      keyUpMask   = 16;      {key up}
      autoKeyMask = 32;      {auto-key}
      updateMask  = 64;      {update}
      diskMask    = 128;     {disk inserted}
      activMask   = 256;     {activate}
      abortMask   = 512;     {abort}
      networkMask = 1024;    {network}
      driverMask  = 2048;    {I/O driver}
      app1Mask    = 4096;    {application-defined}
      app2Mask    = 8192;    {application-defined}
      app3Mask    = 16384;   {application-defined}
      app4Mask    = -32768;  {application-defined}
```

There's also a predefined mask consisting of all 1 bits, to designate every event type:

```
CONST everyEvent = -1;
```

You can form any mask you need by combining these mask constants with integer addition and subtraction. For example, to specify any keyboard event, you can use a mask of

```
keyDownMask + keyUpMask + autoKeyMask
```

For any event except an update, you can use

    everyEvent - updateMask

(hand)

> Recommended programming practice is always to use an event mask of everyEvent unless there is a specific reason not to. This ensures that all events will be processed in their natural order.

In addition to the mask parameters to individual Event Manager routines, there's also a global <u>system event mask</u>, which controls which event types get posted into the event queue. Only those events corresponding to 1 bits in the system event mask are posted; those with Ø bits are ignored. When the system is started up, the system event mask is initially set to post all except key up events--that is, it is initialized to

    everyEvent - keyUpMask

(Key up events are meaningless for most applications, and your program will usually want to ignore them anyway.) If necessary for your particular application, you can change the setting of the system event mask with the Event Manager procedure SetEventMask.

USING THE EVENT MANAGER

This section discusses how the Event Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Before using the Event Manager, you should call the Window Manager procedure InitWindows: parts of the Event Manager rely on the Window Manager's data structures and will not work properly unless those structures have been properly initialized. It's also usually a good idea to call FlushEvents(everyEvent,Ø), to empty the event queue of any stray events left over from before your program was started up (such as keystrokes typed to the Finder).

As noted earlier, most application programs are event-driven. Such programs typically have a main loop that repeatedly calls GetNextEvent to retrieve the next available event, then uses a CASE statement to decide what type of event it is and take whatever action is appropriate.

Your program is only expected to respond to those events that are directly related to its own operations. Events that are of interest only to the system, or that pertain only to system windows, are intercepted and handled by the Desk Manager, but are still reported back to your program by GetNextEvent. After calling GetNextEvent, you

should test its Boolean result to find out whether your program needs to respond to the event: TRUE means the event is of interest to your program, FALSE means you can ignore it.

(hand)

Events handled by the system include activate and update events for system windows; all keyboard and mouse up events when a system window is active, if the window contains a desk accessory that is prepared to handle the event; and network events if there's a desk accessory present that will handle them. Further details are given in the Desk Manager manual.

On receiving a mouse down event, you should first call the Window Manager function FindWindow to find out where on the screen the mouse button was pressed; you can then respond in whatever way is appropriate. Depending on the part of the screen the button was pressed in, this may involve calls to Toolbox routines such as the Menu Manager function MenuSelect, the Desk Manager procedure SystemClick, the Window Manager routines SelectWindow, DragWindow, GrowWindow, and TrackGoAway, and the Control Manager routines FindControl, TrackControl, and DragControl. See the relevant Toolbox manuals for details.

(hand)

If your program attaches some special significance to double mouse clicks, you can detect them by comparing the time and location of each mouse down event with those of the previous such event. If the two events are sufficiently close to each other in time and space--separated by not more than, say, half a second (3Ø ticks) and three pixels--you can consider them a double click and respond accordingly.

When one of your own windows is active, you should respond to keyboard and mouse up events in whatever way your application calls for. For example, when the user types a character on the keyboard, you might want to insert that character into the document displayed in an active document window. For keyboard events, you should first check the modifiers field to see whether the character was typed with the Command key held down: if so, the user may have been choosing a menu item by typing its keyboard equivalent. To find out, pass the character that was typed to the Menu Manager function MenuKey. If that character, combined with the Command key, stands for a menu item, MenuKey will return a nonzero result identifying the item. You can then do whatever is appropriate to respond to that menu item, just as if the user had chosen it with the mouse. If MenuKey's result is Ø, the user has typed a key combination that has no menu equivalent; your program may then want to respond in some other way.

(hand)

Under the Macintosh User Interface Guidelines, the keyboard's usual auto-repeat property doesn't apply to Command-key combinations that stand for menu items. When

> you receive a nonzero result from MenuKey, you should execute the corresponding menu command only if the event you're responding to was a mouse down event; if it was an auto-key event, just ignore it and go on to the next event.

When you receive an activate event for one of your own windows, the Window Manager will already have done all of the normal "housekeeping" associated with the event, such as highlighting or unhighlighting the window. You can then take any further action of your own that your application may require, such as showing or hiding a scroll bar or highlighting or unhighlighting a selection.

On receiving an update event for one of your own windows, you should usually call the Window Manager procedure BeginUpdate, redraw the window's contents, then call EndUpdate.

When you receive a disk inserted event, the Desk Manager will already have responded to the event by attempting to mount the new volume just inserted in the disk drive. Usually there's nothing more for your program to do, but GetNextEvent returns TRUE anyway, giving you an opportunity to take some further action if your application demands it. If the attempt to mount the volume was unsuccessful, there will be a nonzero error code in the high-order word of the event message; in this case you might want to take some special action, such as displaying an alert box containing an error message.

If the event you receive is an abort event, first check to see whether it was generated by the user or by the Event Manager's own journaling mechanism. For user-generated abort events, your program should stop whatever it's doing and return to its main loop to process the next available event; for those that originate in the journaling mechanism, it should reset its internal state as appropriate to prepare for replaying a journal.

(hand)

> During any particularly time-consuming operation, your program should check for abort events periodically to allow the user to interrupt the operation from the keyboard.

Network events are handled by the Desk Manager as long as there's a desk accessory present that can respond to them. If GetNextEvent returns a TRUE result for a network event, then no such desk accessory is present; your program should normally just ignore the event. *** The exact meaning and use of I/O driver events is not yet specified, so (for the time being) you needn't worry about how to respond to them. ***

If you're using your own event types for internal communication between parts of your program, you can use PostEvent to post them into the event queue. When you receive them back from GetNextEvent, you can respond to them in whatever way is appropriate for your application.

To "peek" at pending events without removing them from the event queue, use EventAvail instead of GetNextEvent. To remove all events of a given type or types from the queue, use FlushEvents. To control which event types get posted into the queue, or to cause certain types to be ignored, use SetEventMask.

In addition to receiving the user's mouse and keyboard actions in the form of events, you can directly read the keyboard (and keypad), mouse location, and state of the mouse button by calling GetKeys, GetMouse, and Button, respectively. To follow the mouse when the user drags it with the button down, use StillDown or WaitMouseUp.

Finally, you can read the current setting of the system clock at any time by calling TickCount.

EVENT MANAGER ROUTINES

This section describes all the Event Manager procedures and functions. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** (doesn't exist, but see QuickDraw manual) *** and also "Notes for Assembly-Language Programmers" in this manual.

Accessing Events

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
          BOOLEAN;
```

GetNextEvent returns the next available event of a specified type or types and removes it from the event queue. The event is returned as the value of the parameter theEvent; eventMask specifies which event types are of interest. GetNextEvent will return the next available event of any type designated by a 1 bit in the mask, subject to the priority rules discussed above under "Priority of Events". Event types corresponding to 0 bits in the mask are ignored. If no event of any of the designated types is available, GetNextEvent returns a null event, regardless of the setting of the eventMask bit for null events.

(eye)

> Since update events are never actually placed in the event queue, GetNextEvent can't remove them from the queue before returning them, as it does with other events. If your program doesn't take some explicit action to "clear" the update event, it will keep getting the same event back again. The normal way of clearing an update event is with BeginUpdate and EndUpdate; further explanation can be found in the Window Manager manual.

Before reporting an event to your program, GetNextEvent first calls the Desk Manager function SystemEvent to see whether the system wants to intercept and respond to the event. If so (or if the event being reported is a null event), GetNextEvent returns a function result of FALSE to notify your program that it can ignore this event; a function result of TRUE means that your program should handle the event itself. The Desk Manager normally intercepts the following events:

- All activate and update events directed to a system window

- All keyboard and mouse up events if the currently active window is a system window and contains a desk accessory that is prepared to handle the event

- All network events if there is a desk accessory present that can handle them

The Desk Manager also responds to disk inserted events by attempting to mount the volume that has just been inserted; but in this case GetNextEvent returns TRUE to allow your program to take some further action if appropriate. All other events (including all mouse down events, regardless of which window is active) are left for your program to handle. See the Desk Manager manual for further details.

```
FUNCTION EventAvail (eventMask: INTEGER; VAR theEvent: EventRecord) :
            BOOLEAN;
```

EventAvail returns in theEvent the next available event of the type or types specified by eventMask, but does not remove the event from the event queue. This allows you to "peek" at pending events while still leaving them in the queue for later processing. In all other respects, EventAvail works exactly the same as GetNextEvent (see above).

Posting and Removing Events

```
PROCEDURE PostEvent (eventCode: INTEGER; eventMsg: LongInt);
```

PostEvent places in the event queue an event of the type designated by eventCode, with the event message specified by eventMsg. The main use of this procedure is for posting events of your own application-defined types. It's also sometimes useful for placing an event back in the queue after you've removed it with GetNextEvent. Notice, however, that in this case the system clock time, mouse location, and state of the mouse button and modifier keys will be changed from their original values to those in effect at the time the event is reposted.

(eye)

> Be very careful about posting any but your own application-defined events into the queue. For example, attempting to post an activate or update event will

interfere with the internal operation of the Event Manager, since such events are detected in other ways and are not normally placed in the queue at all. If you repost a mouse event, the mouse location associated with it will be changed, possibly altering its meaning; reposting a keyboard event may cause modifier information to be lost or characters to be transposed from the order in which the user originally typed them. In general, you should avoid using PostEvent for any but your own events unless you're sure you know what you're doing.

PROCEDURE FlushEvents (eventMask,stopMask: INTEGER);

FlushEvents removes from the event queue all events of the type(s) specified by eventMask, up to, but not including, the first event of any type specified by stopMask. To remove all events of a particular type or types, use a stopMask value of Ø. You might use FlushEvents, for example, on receiving an abort event, to remove any mouse or keyboard events that may have occurred before the program was interrupted.

(hand)
When your program is first started up, it's usually a good idea to call FlushEvents(everyEvent,Ø) to empty the event queue of any stray events that may have been left lying around, such as unprocessed keystrokes typed to the Finder.

Reading the Mouse

PROCEDURE GetMouse (VAR mouseLoc: Point);

GetMouse returns the current mouse location as the value of the parameter mouseLoc. The location is expressed in the local coordinate system of the current grafPort (which might be, for example, the currently active window). Notice that this differs from the mouse location stored in the where field of an event record, which is given in global coordinates.

FUNCTION Button : BOOLEAN;

The Button function returns the current state of the mouse button: TRUE if the button is down, FALSE if it isn't.

FUNCTION StillDown : BOOLEAN;

Called after a mouse down event, StillDown tests whether the mouse button is still down. It returns TRUE if the button is currently down

and there are no more mouse events (mouse ups or later mouse downs) pending in the event queue. This is a true test of whether the button is still down from the original press--unlike Button (see above), which returns TRUE whenever the button is currently down, even if it has been released and pressed again since the original mouse down event.

```
FUNCTION WaitMouseUp : BOOLEAN;
```

WaitMouseUp works exactly the same as StillDown (see above), except that if the button is not still down from the original press, WaitMouseUp removes the corresponding mouse up event before returning FALSE.

Reading the Keyboard and Keypad

```
PROCEDURE GetKeys (VAR theKeys: KeyMap);
```

GetKeys reads the current state of the keyboard (and keypad, if any) and returns it in the form of a keyMap:

```
TYPE KeyMap = PACKED ARRAY [1..128] OF BOOLEAN;
```

Each element of the keyMap is TRUE if the corresponding key is down, FALSE if it isn't. The correspondence between elements of the keyMap and keys on the keyboard and keypad is shown in Table 1. KeyMap elements corresponding to blank entries in the table are unused. Notice that GetKeys doesn't distinguish between the two Shift keys or the two Option keys.

Element	Key	Element	Key
0	A	48	Tab
1	S	49	Space bar
2	D	50	`
3	F	51	Backspace
4	H	52	Enter
5	G	53	
6	Z	54	
7	X	55	Command *** (name may change) ***
8	C	56	Shift
9	V	57	Caps Lock
10		58	Option
11	B	59	
12	Q	60	
13	W	61	
14	E	62	
15	R	63	
16	Y	64	
17	T	65	. (keypad)
18	1	66	* (keypad)
19	2	67	
20	3	68	
21	4	69	
22	6	70	+ (keypad)
23	5	71	Clear (keypad)
24	=	72	, (keypad)
25	9	73	
26	7	74	
27	-	75	
28	8	76	Enter (keypad)
29	0	77	/ (keypad)
30	]	78	- (keypad)
31	O	79	
32	U	80	
33	[	81	
34	I	82	0 (keypad)
35	P	83	1 (keypad)
36	Return	84	2 (keypad)
37	L	85	3 (keypad)
38	J	86	4 (keypad)
39	'	87	5 (keypad)
40	K	88	6 (keypad)
41	;	89	7 (keypad)
42	\	90	
43	,	91	8 (keypad)
44	/	92	9 (keypad)
45	N	93	
46	M	94	
47	.	95	
		96-127	(Unused)

Table 1. KeyMap Elements

Miscellaneous Utilities

PROCEDURE SetEventMask (theMask: INTEGER);

SetEventMask sets the system event mask to the specified value. This mask controls the posting of events into the event queue. Only event types corresponding to 1 bits in the mask are posted; all others are ignored. The initial setting for the system event mask is to post all except key up events.

SetEventMask is useful if for some reason you want to know when keys are released as well as when they're pressed, or if you know that some other event type is of no interest to your program and needn't be posted. For example, if your program attaches no special meaning to mouse up events, you may want to dispense with them; or you might want to eliminate keyboard repeat by preventing auto-key events from being posted.

(hand)

> Since space in the event queue is limited, it's generally a good idea to disable any event type that you know your program has no use for.

The system event mask has no effect on activate or update events, since these events are detected in other ways and are never actually posted into the event queue.

FUNCTION TickCount : LongInt;

TickCount returns the current value of the system clock, which gives the elapsed time in ticks (sixtieths of a second) since the system was last started up.

JOURNALING

Using the Event Manager's journaling mechanism, all of a program's interactions with the Event Manager can be recorded and later played back, just as if they were happening for the first time. A journal is a record of all calls to the Event Manager routines GetNextEvent, EventAvail, GetMouse, Button, GetKeys, and TickCount. When a journal is being recorded, every call to any of these routines is sent to a special input/output driver and recorded in the journal, along with the result returned.

When the journal is played back, the same Event Manager calls read their results back from the journal instead of directly from the mouse, keyboard, keypad, and system clock. To the application program, the results it receives from the Event Manager in response to these calls

look exactly as if they were coming directly from the user. Since the program is event-driven, its behavior is completely determined by this stream of results. In particular, the sequence of calls the program issues to the Event Manager while replaying the journal will exactly match those that occurred when the journal was originally recorded. Since the results the Event Manager sends back are taken from the journal, the same sequence of events that occurred when the journal was recorded will be reproduced when the journal is played back.

(eye)

> Null events are not fully recorded in the journal: the fact that a null event was generated is recorded, but not the contents of the event record's fields. When the journal is played back, this information--the time the event was posted, the mouse location, and the state of the mouse button and modifier keys--is lost; the contents of the when, where, and modifiers fields are meaningless. If there's any chance your program may be executed from a journal instead of by direct interaction with the user, it should not rely in any way on the contents of a null event's fields.

The user can control journal recording and playback with the journaling desk accessory. It can also be controlled by the application program itself, but only from the assembly-language level: see "Notes for Assembly-Language Programmers", below, for details. *** The exact method of controlling the journaling mechanism has not been finally determined and will probably change. ***

RESOURCE FORMAT FOR KEYBOARD CONFIGURATIONS

The keyboard configuration, which translates the keys the user presses on the keyboard and keypad into the characters they represent, is treated as a resource and read from a resource file. The standard Macintosh keyboard configuration is stored in the system resource file and is read automatically when the Macintosh is started up. One way to substitute an alternate keyboard configuration--for example, for foreign use--is to use the Resource Editor *** (which doesn't yet exist) *** to replace the standard configuration with the new one in the system resource file. Then the next time the Macintosh is restarted, it will read the new keyboard configuration instead of the standard one.

(hand)

> It's also possible for a running program to install a new keyboard configuration "on the fly". This can only be done in assembly language; details are given in the next section.

Actually, the keyboard configuration is a pair of machine-language <u>configuration routines</u>, one for the keyboard and one for the keypad. These routines accept a key code, along with the state of the modifier

keys, as input and return the corresponding character code as output. The arguments and result are passed directly in machine registers, so the routines must be written in assembly language, not in Pascal.

The keyMap index (see Table 1) for the key to be translated is passed to the configuration routine in register D2. Register D1 contains the fourth word (indices 48 to 63) of the current keyMap, which includes the status bits for the four modifier keys at the positions shown in Figure 4. All other bits in this word should be ignored. The configuration routine is expected to return a character code in register DØ; it should preserve the contents of all other registers. If the specified key combination doesn't correspond to any character, the configuration routine should return Ø: in this case, no keyboard event will be generated.

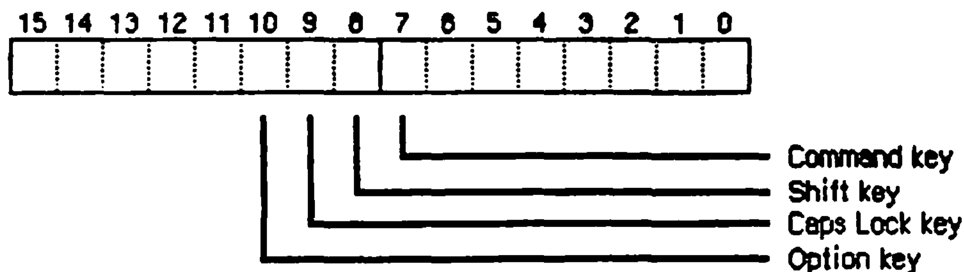


Figure 4. Modifier Bits for Configuration Routines

When the Macintosh is started up, two configuration routines are read from the system resource file. Both have a resource type of 'KEYC'; the resource ID is 1 for the keyboard routine and 2 for the keypad routine. The resource data for a resource of this type is just the machine code for the routine. The first byte of code is assumed to be the entry point for executing the routine.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

Information about how to use the User Interface Toolbox from assembly language is given elsewhere. *** For now, see the QuickDraw manual. *** This section contains special notes of interest to programmers who will be using the Event Manager from assembly language.

The primary aid to assembly-language programmers is a file named TOOLEQU.TEXT. If you use .INCLUDE to include this file when you assemble your program, all the Event Manager constants, offsets to locations of global variables, and offsets into fields of structured types will be available in symbolic form.

In assembly language, you can control the operation of the journaling mechanism by setting the global variable JournalFlag. Setting this variable to a positive, nonzero value turns on journal recording; setting it negative turns on playback; setting it to Ø turns journaling off.

The global variables Key1Trans and Key2Trans are used to hold pointers to the keyboard and keypad configuration routines, respectively. You can replace either or both of these routines "on the fly" by the following steps:

1. Call the Resource Manager function GetResource (or GetNamedResource) to find the new configuration routine in its resource file, read it into memory, and get a handle to it.

2. Use the Operating System call RecoverHandle to convert the existing routine pointer from Key1Trans or Key2Trans into a handle.

3. Use the Operating System call DisposHandle to free the storage occupied by the old routine.

4. Convert the handle you received from the Resource Manager into a pointer and store it in Key1Trans (for a keyboard routine) or Key2Trans (for a keypad routine).

APPENDIX: STANDARD KEY AND CHARACTER CODES

The following tables show the key and character codes used by Macintosh and the characters assigned to keys on the keyboard and keypad under the standard Macintosh keyboard configuration. All key and character codes are given in hexadecimal; for the benefit of readers with only ten fingers, there's a hexadecimal/decimal conversion table at the end of this Appendix.

Table 2 shows the extended ASCII character set used by Macintosh and Lisa. The first digit of the hexadecimal character code is shown at the top of the table, the second down the left side. For example, character code $47 stands for the capital letter G, which appears in the table at the intersection of column 4 and row 7.

Character codes between $2Ø and $7E have their normal ASCII meanings. Codes between $8Ø and $CA denote special characters included in the extended character set for business, scientific, and international use;

codes from $CB to $FF are unassigned. ASCII control characters ($ØØ to $1F, as well as $2Ø and $7F) are identified in the table by their traditional ASCII abbreviations:

Code	Abbr.	Meaning	Code	Abbr.	Meaning
$ØØ	NUL	Null	$1Ø	DLE	Data Link Escape
$Ø1	SOH	Start of Header	$11	DC1	Device Control 1
$Ø2	STX	Start of Text	$12	DC2	Device Control 2
$Ø3	ETX	End of Text	$13	DC3	Device Control 3
$Ø4	EOT	End of Tape	$14	DC4	Device Control 4
$Ø5	ENQ	Enquiry	$15	NAK	Negative Acknowledge
$Ø6	ACK	Acknowledge	$16	SYN	Synchronous Idle
$Ø7	BEL	Bell	$17	ETB	End Transmission Block
$Ø8	BS	Backspace	$18	CAN	Cancel
$Ø9	HT	Horizontal Tab	$19	EM	End of Medium
$ØA	LF	Line Feed	$1A	SUB	Substitute
$ØB	VT	Vertical Tab	$1B	ESC	Escape
$ØC	FF	Form Feed	$1C	FS	Field Separator
$ØD	CR	Carriage Return	$1D	GS	Group Separator
$ØE	SO	Shift Out	$1E	RS	Record Separator
$ØF	SI	Shift In	$1F	US	Unit Separator
$2Ø	SP	Space	$7F	DEL	Delete

However, most of these characters have no special meaning on Macintosh and cannot be generated from the Macintosh keyboard under the standard keyboard configuration. The exceptions are the following:

Code	Character	Key
$Ø3	ETX	Enter (keyboard and keypad)
$Ø8	BS	Backspace
$Ø9	HT	Tab
$ØD	CR	Return
$1B	ESC	Clear (keypad)
$1C	FS	Left arrow (keypad)
$1D	GS	Right arrow (keypad)
$1E	RS	Up arrow (keypad)
$1F	US	Down arrow (keypad)
$2Ø	SP	Space bar

In addition, as shown in the table, codes from $11 to $15 denote special characters used on the Macintosh screen, such as the open and solid Apple characters. These characters are intended exclusively for use on the screen, and have no keyboard or keypad equivalents under the standard keyboard configuration.

The characters shaded in the table are accented letters used in various foreign languages. Under the standard keyboard configuration, these characters cannot be typed directly from the keyboard. Instead, they are generated by first typing the accent or diacritical mark alone, followed by the letter to be accented. For example, a lowercase letter e with a grave accent (è, character code $8F) is produced by typing a grave accent (`, code $6Ø) followed by a lowercase e (code $65). The Macintosh keyboard driver will *** (eventually) *** translate such two-

character sequences involving diacriticals into the corresponding single accented letters.

Tables 3 and 4 show the hexadecimal key codes corresponding to keys on the Macintosh keyboard and keypad, respectively. Modifier keys are not shown, since they never generate keyboard events of their own.

Table 5 shows the hexadecimal character codes generated by each key on the keyboard under the standard keyboard configuration. Table 5a gives the character generated when the key is pressed by itself, Table 5b when it is pressed with the Shift key held down, Table 5c the Caps Lock key, Table 5d the Option key, and Table 5e the Option and Shift or Option and Caps Lock keys. Again, the modifier keys themselves are not shown.

Table 6 shows the hexadecimal character codes for the keypad under the standard keyboard configuration. Table 6a gives the character generated when the key is pressed by itself, Table 6b when it is pressed with the Shift key held down.

Finally, Table 7 is a conversion table between hexadecimal and decimal. To convert a two-digit hexadecimal number to decimal, find its first digit at the top of the table and its second down the left side. The decimal equivalent is found at the intersection of that column and row. For example, hexadecimal $6C is equivalent to decimal 1Ø8, found at the intersection of column 6 and row C. To convert a decimal number to hexadecimal, find the number in the body of the table and read its first and second hexadecimal digits from the head of that column and row, respectively. For example, decimal 227 is in column E and row 3, so its hexadecimal equivalent is $E3.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	Ä	ê	†	∞	¿			
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡			
2	STX	DC2 ✓	"	2	B	R	b	r	Ç	í	¢	≤	¬			
3	ETX	DC3 ◆	#	3	C	S	c	s	É	ì	£	≥	√			
4	EOT	DC4 ●	$	4	D	T	d	t	Ñ	î	§	¥	ƒ			
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈			
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	∆			
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	∑	»			
8	BS ←	CAN	(	8	H	X	h	x	à	ò	®	∏	«			
9	HT →	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	␣			
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª				
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º				
D	CR	GS	-	=	M	]	m	}	ç	ù	≠	Ω				
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ				
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø				

Non-printing — Printing characters

Table 2. Macintosh and Lisa Extended ASCII Character Set

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$32	$12	$13	$14	$15	$17	$16	$1A	$1C	$19	$1D	$1B	$18	$33

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$3Ø	$ØC	$ØD	$ØE	$ØF	$11	$1Ø	$2Ø	$22	$1F	$23	$21	$1E	$2A

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$ØØ	$Ø1	$Ø2	$Ø3	$Ø5	$Ø4	$26	$28	$25	$29	$27	$24

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$Ø6	$Ø7	$Ø8	$Ø9	$ØB	$2D	$2E	$2B	$2F	$2C

Key:	[Space]	[Enter]
Code:	$31	$34

Table 3. Key Codes for Macintosh Keyboard

Key:	[Clear]	[-]	[+]	[*]
Code:	$47	$4E	$46	$42
Key:	[7]	[8]	[9]	[/]
Code:	$59	$5B	$5C	$4D
Key:	[4]	[5]	[6]	[,]
Code:	$56	$57	$58	$48
Key:	[1]	[2]	[3]	[Enter]
Code:	$53	$54	$55	
Key:	[Ø]		[.]	
Code:	$52		$41	$4C

Table 4. Key Codes for Macintosh Keypad

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3Ø	$2D	$3D	$Ø8
Char:	`	1	2	3	4	5	6	7	8	9	Ø	-	=	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$71	$77	$65	$72	$74	$79	$75	$69	$6F	$7Ø	$5B	$5D	$5C
Char:	HT	q	w	e	r	t	y	u	i	o	p	[	]	\

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[´]	[Return]
Code:	$61	$73	$64	$66	$67	$68	$6A	$6B	$6C	$3B	$27	$ØD
Char:	a	s	d	f	g	h	j	k	l	;	´	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$7A	$78	$63	$76	$62	$6E	$6D	$2C	$2E	$2F
Char:	z	x	c	v	b	n	m	,	.	/

Key:	[Space]	[Enter]
Code:	$2Ø	$Ø3
Char:	SP	ETX

(a) Unshifted

Table 5. Standard Character Codes for Macintosh Keyboard

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$7E	$21	$4Ø	$23	$24	$25	$5E	$26	$2A	$28	$29	$5F	$2B	$Ø8
Char:	~	!	@	#	$	%	^	&	*	(	)	_	+	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$51	$57	$45	$52	$54	$59	$55	$49	$4F	$5Ø	$7B	$7D	$7C
Char:	HT	Q	W	E	R	T	Y	U	I	O	P	{	}	|

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$41	$53	$44	$46	$47	$48	$4A	$4B	$4C	$3A	$22	$ØD
Char:	A	S	D	F	G	H	J	K	L	:	"	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$5A	$58	$43	$56	$42	$4E	$4D	$3C	$3E	$3F
Char:	Z	X	C	V	B	N	M	<	>	?

Key:	[Space]	[Enter]
Code:	$2Ø	$Ø3
Char:	SP	ETX

(b) Shift Key Down

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3Ø	$2D	$3D	$Ø8
Char:	`	1	2	3	4	5	6	7	8	9	Ø	-	=	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$51	$57	$45	$52	$54	$59	$55	$49	$4F	$5Ø	$5B	$5D	$5C
Char:	HT	Q	W	E	R	T	Y	U	I	O	P	[	]	\

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$41	$53	$44	$46	$47	$48	$4A	$4B	$4C	$3B	$27	$ØD
Char:	A	S	D	F	G	H	J	K	L	;	'	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$5A	$58	$43	$56	$42	$4E	$4D	$2C	$2E	$2F
Char:	Z	X	C	V	B	N	M	,	.	/

Key:	[Space]	[Enter]
Code:	$2Ø	$Ø3
Char:	SP	ETX

(c) Caps Lock Key Down

Table 5. Standard Character Codes for Macintosh Keyboard (continued)

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$C1	$AA	$A3	$A2	$BØ	$A4	$A6	$A5	$BB	$BC	$B1	$AD	$Ø8
Char:	`	¡	™	£	¢	∞	§	¶	•	ª	º	±	≠	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$A1	$B7	$AB	$A8	$AØ	$B4	$AC	$ØØ	$BF	$B9	$B5	$C8	$ØØ
Char:	HT	°	∑	´	®	†	¥	¨		ø	π	µ	«	

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$8C	$A7	$B6	$C4	$A9	$5E	$C6	$ØØ	$C2	$BD	$BE	$ØD
Char:	å	ß	∂	ƒ	©	ˆ	∆		¬	Ω	æ	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$ØØ	$C5	$8D	$C3	$BA	$7E	$ØØ	$B2	$B3	$CØ
Char:		≈	ç	√	∫	˜		≤	≥	¿

Key:	[Space]	[Enter]
Code:	$2Ø	$Ø3
Char:	SP	ETX

(d) Option Key Down

Key:	[`]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[Ø]	[-]	[=]	[Backspace]
Code:	$6Ø	$C1	$AA	$A3	$A2	$BØ	$A4	$A6	$A5	$BB	$BC	$B1	$AD	$Ø8
Char:	`	¡	™	£	¢	∞	§	¶	•	ª	º	±	≠	BS

Key:	[Tab]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[\]
Code:	$Ø9	$A1	$B7	$AB	$A8	$AØ	$B4	$AC	$ØØ	$AF	$B8	$B5	$C7	$ØØ
Char:	HT	°	∑	´	®	†	¥	¨		Ø	∏	µ	»	

Key:	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[Return]
Code:	$81	$A7	$B6	$C4	$A9	$5E	$C6	$ØØ	$C2	$BD	$AE	$ØD
Char:	Å	ß	∂	ƒ	©	ˆ	∆		¬	Ω	Æ	CR

Key:	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]
Code:	$ØØ	$C5	$82	$C3	$BA	$7E	$ØØ	$B2	$B3	$CØ
Char:		≈	Ç	√	∫	˜		≤	≥	¿

Key:	[Space]	[Enter]
Code:	$2Ø	$Ø3
Char:	SP	ETX

(e) Option and Shift or Option and Caps Lock Keys Down

Table 5. Standard Character Codes for Macintosh Keyboard (continued)

Key:	[Clear]	[-]	[+]	[*]
Code:	$1B	$2D	$1C	$1D
Char:	ESC	-	←	→
Key:	[7]	[8]	[9]	[/]
Code:	$37	$38	$39	$1E
Char:	7	8	9	↑
Key:	[4]	[5]	[6]	[,]
Code:	$34	$35	$36	$1F
Char:	4	5	6	↓
Key:	[1]	[2]	[3]	[E]
Code:	$31	$32	$33	[n]
Char:	1	2	3	[t]
				[e]
Key:	[0]		[.]	[r]
Code:	$30		$2E	$03
Char:	0		.	ETX

(a) Unshifted

Key:	[Clear]	[-]	[+]	[*]
Code:	$1B	$2D	$2B	$2A
Char:	ESC	-	+	*
Key:	[7]	[8]	[9]	[/]
Code:	$37	$38	$39	$2F
Char:	7	8	9	/
Key:	[4]	[5]	[6]	[,]
Code:	$34	$35	$36	$2C
Char:	4	5	6	,
Key:	[1]	[2]	[3]	[E]
Code:	$31	$32	$33	[n]
Char:	1	2	3	[t]
				[e]
Key:	[0]		[.]	[r]
Code:	$30		$2E	$03
Char:	0		.	ETX

(b) Shift Key Down

Table 6. Standard Character Codes for Macintosh Keypad

Second digit \ First digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Table 7. Hexadecimal/Decimal Conversion Table

SUMMARY OF THE EVENT MANAGER

```
CONST nullEvent    = 0;        {null}
      mouseDown    = 1;        {mouse down}
      mouseUp      = 2;        {mouse up}
      keyDown      = 3;        {key down}
      keyUp        = 4;        {key up}
      autoKey      = 5;        {auto-key}
      updateEvt    = 6;        {update}
      diskEvt      = 7;        {disk inserted}
      activateEvt  = 8;        {activate}
      abortEvt     = 9;        {abort}
      networkEvt   = 10;       {network}
      driverEvt    = 11;       {I/O driver}
      app1Evt      = 12;       {application-defined}
      app2Evt      = 13;       {application-defined}
      app3Evt      = 14;       {application-defined}
      app4Evt      = 15;       {application-defined}

      nullMask     = 1;        {null}
      mDownMask    = 2;        {mouse down}
      mUpMask      = 4;        {mouse up}
      keyDownMask  = 8;        {key down}
      keyUpMask    = 16;       {key up}
      autoKeyMask  = 32;       {auto-key}
      updateMask   = 64;       {update}
      diskMask     = 128;      {disk inserted}
      activMask    = 256;      {activate}
      abortMask    = 512;      {abort}
      networkMask  = 1024;     {network}
      driverMask   = 2048;     {I/O driver}
      app1Mask     = 4096;     {application-defined}
      app2Mask     = 8192;     {application-defined}
      app3Mask     = 16384;    {application-defined}
      app4Mask     = -32768;   {application-defined}

      everyEvent = -1;

TYPE EventRecord = RECORD
                     what:      INTEGER;
                     message:   LongInt;
                     when:      LongInt;
                     where:     Point;
                     modifiers: INTEGER
                   END;

     KeyMap = PACKED ARRAY [1..128] OF BOOLEAN;
```

Accessing Events

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) :
                       BOOLEAN;
FUNCTION EventAvail   (eventMask: INTEGER; VAR theEvent: EventRecord) :
                       BOOLEAN;
```

Posting and Removing Events

```
PROCEDURE PostEvent   (eventCode: INTEGER; eventMsg: LongInt);
PROCEDURE FlushEvents (eventMask,stopMask: INTEGER);
```

Reading the Mouse

```
PROCEDURE GetMouse      (VAR mouseLoc: Point);
FUNCTION  Button :      BOOLEAN;
FUNCTION  StillDown :   BOOLEAN;
FUNCTION  WaitMouseUp : BOOLEAN;
```

Reading the Keyboard and Keypad

```
PROCEDURE GetKeys (VAR theKeys: KeyMap);
```

Miscellaneous Utilities

```
PROCEDURE SetEventMask (theMask: INTEGER);
FUNCTION  TickCount :  LongInt;
```

GLOSSARY

abort event: An event generated when the user presses a special combination of keys *** (tentatively Command-.) ***, or when the Event Manager's journaling mechanism wants a program to prepare for replaying a journal.

activate event: An event generated by the Window Manager when a window changes from active to inactive or vice versa.

auto-key event: An event generated periodically when the user presses and holds down a key on the keyboard or keypad.

character code: An integer representing the character that a key or combination of keys on the keyboard or keypad stands for.

configuration routine: A machine-language routine that defines a particular keyboard configuration by translating a key code, together with the state of the modifier keys, into a corresponding character code.

disk inserted event: An event generated when the user inserts a disk in a disk drive.

event: A notification to an application program of some occurrence that the program must respond to.

event code: An integer representing a particular type of event.

event mask: A parameter passed to an Event Manager routine specifying which types of event the routine is to be applied to.

event message: A field of an event record containing information specific to the particular type of event.

event queue: The Event Manager's list of pending events waiting to be processed.

event record: The internal representation of an event, where the Event Manager stores all pertinent information about that event.

I/O driver event: An event generated by one of the Macintosh's input/output drivers. *** (Not yet implemented.) ***

journal: A record of all of a program's interactions with the Event Manager over a period of time, which can be played back in order to reproduce the original session.

keyboard configuration: A resource that defines a particular keyboard layout by associating a character code with each key or combination of keys on the keyboard or keypad.

keyboard event: An event generated when the user presses, releases, or holds down a key on the keyboard or keypad; any key down, key up, or auto-key event.

key code: An integer representing a key on the keyboard or keypad, without reference to the character that key stands for.

key down event: An event generated when the user presses a key on the keyboard or keypad.

key up event: An event generated when the user releases a key on the keyboard or keypad.

modifier key: A key (Shift, Caps Lock, Option, or Command) that generates no keyboard events of its own, but changes the meaning of those generated by other keys.

mouse down event: An event generated when the user presses the mouse button.

mouse up event: An event generated when the user releases the mouse button.

network event: An event generated by the Macintosh's network driver. *** (Not yet implemented.) ***

null event: An event returned by the Event Manager when it has no other events to report.

post: To place an event in the event queue for later processing.

system event mask: A global event mask that controls which types of event get posted into the event queue.

update event: An event generated by the Window Manager when a window's contents need to be redrawn.

MACINTOSH USER EDUCATION

The File Manager: A Programmer's Guide /OS/FS

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Macintosh Packages: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) B. Hacker 3/02/84

*** Review Draft. Not for distribution *** ABSTRACT

This manual describes the File Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and files.

TABLE OF CONTENTS

3 About This Manual
3 About the File Manager
4 Volumes
5 Accessing Volumes
5 Files
9 Accessing Files
10 File Information Used by the Finder
11 Using the File Manager
15 High-Level File Manager Routines
16 Accessing Volumes
18 Changing File Contents
22 Changing Information About Files
24 Low-Level File Manager Routines
25 Routine Parameters
27 I/O Parameters
29 File Information Parameters
29 Volume Information Parameters
30 Routine Descriptions
31 Initializing the File I/O Queue
31 Accessing Volumes
37 Changing File Contents
47 Changing Information About Files
53 Data Organization on Volumes
55 Volume Information
56 Volume Allocation Block Map
56 File Directory
57 File Tags on Volumes
58 Data Structures in Memory
59 The File I/O Queue
60 Volume Control Blocks
62 File Control Blocks
64 File Tags in Memory
64 The Drive Queue
65 Using an External File System
67 Summary of the File Manager
74 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the File Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and files. *** Eventually it will become part of a larger manual describing the entire Toolbox and Operating System. *** The File Manager allows you to create and access any number of files containing whatever information you choose.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager

- devices and device drivers, as described in the Inside Macintosh Road Map

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the File Manager and what you can do with it. It then discusses some basic concepts behind the File Manager: what files and volumes are and how they're accessed.

A section on using the File Manager introduces its routines and tells how they fit into the flow of your application. This is followed by sections explaining the File Manager's simplest, "high-level" Pascal routines and then its more complex, "low-level" Pascal and assembly-language routines. Both sections give detailed descriptions of all the procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that won't interest all readers. The data structures that the File Manager uses to store information in memory and on disks are described, and special information is provided for programmers who want to write their own file system.

Finally, there's a summary of the File Manager's data structures and routines, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE FILE MANAGER

The File Manager is the part of the Operating System that handles communication between applications and files on block devices such as disk drives. Files are a principal means by which data is stored and

transmitted on the Macintosh. A file is a named, ordered sequence of bytes. The File Manager contains routines used to read and write to files.

Volumes

A volume is a piece of storage medium, such as a disk, formatted to contain files. A volume can be an entire disk or only part of a disk. Currently, the 3 1/2-inch Macintosh disks are one volume.

(note)
Specialized memory devices other than disks can also contain volumes, but the information in this manual applies only to volumes on disks.

You identify a volume by its volume name, which consists of any sequence of 1 to 27 printing characters. Volume names must always be followed by a colon (:) to distinguish them from other names.

(note)
The colon (:) after a volume name should only be used when calling File Manager routines; it should never be seen by the user.

A volume contains descriptive information about itself, including its name and a file directory listing information about files contained on the volume; it also contains files. The files are contained in allocation blocks, which are areas of volume space occupying multiples of 512 bytes.

A volume can be mounted or unmounted. A volume becomes mounted when it's in a disk drive and the File Manager reads descriptive information about the volume into memory. Once mounted, a volume may remain in a drive, or be ejected. Only mounted volumes are known to the File Manager, and an application can only access information on mounted volumes. A volume becomes unmounted when the File Manager releases the memory used to store the descriptive information. Your application would unmount a volume when it's finished with the volume, or when it needs the memory occupied by the volume.

The File Manager assigns each mounted volume a volume reference number you can use instead of its volume name to refer to it. Every mounted volume is also assigned a volume buffer on the heap, which is temporary storage space used when reading and writing information on the volume. The number of volumes that may be mounted at any time is limited only by the number of drives attached and available memory.

A mounted volume can be on-line or off-line. A mounted volume is on-line as long as the volume buffer and all the descriptive information read from the volume when it was mounted remain in memory (about 1K to 1.5K bytes); it becomes off-line when all but 94 bytes of descriptive information are released. You can access information on on-line volumes immediately, but off-line volumes must be placed

on-line before their information can be accessed. An application would place a volume off-line whenever it needed most of the memory the volume occupies. When you eject a volume from a drive, the File Manager automatically places the volume off-line.

To prevent unauthorized writing to a volume, volumes can be locked. Locking a volume involves either setting a software flag on the volume, or changing some part of the volume physically (for example, sliding a tab from one position to another on a disk). Locking a volume ensures that none of the data on the volume can be changed.

Accessing Volumes

You can access a mounted volume via its volume name or volume reference number. On-line volumes in disk drives can also be accessed via the drive number of the drive on which the volume is mounted (the internal drive is number 1, and the external drive is number 2). You should always use the volume name or volume reference number, rather than a drive number, when accessing a mounted volume, because the volume may have been ejected or placed off-line. Whenever possible, use the volume reference number--to avoid confusion between volumes with the same name.

One volume is always the default volume. Whenever you call a routine to access a volume but don't specify which volume, the default volume is accessed. Initially, the volume used to start up the system is the default volume, but an application can designate any mounted volume as the default volume.

Whenever the File Manager needs to access a mounted volume that's been ejected from its drive, the dialog box shown in Figure 1 is displayed, and the File Manager waits until the user inserts the volume named volName into a drive.

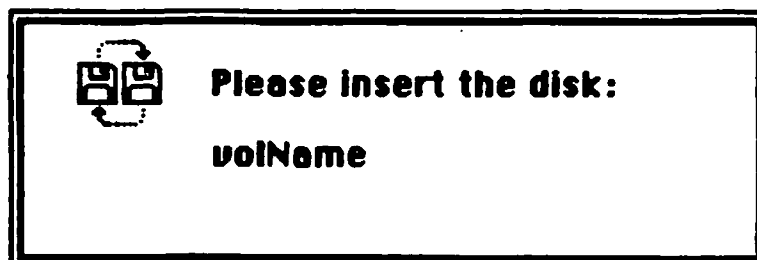


Figure 1. Disk-Switch Dialog

Files

A file is a finite sequence of numbered bytes. Any byte or group of bytes in the sequence can be accessed individually. A file is identified by its file name and version number. A file name consists of any sequence of 1 to 255 printing characters, excluding colons (:). The version number is any number from Ø to 255, and is used by the File Manager to distinguish between different files with the same name. A byte within a file is identified by its position within the ordered sequence.

(warning)
Your application should constrain file names to fewer than 64 characters, because the Finder will generate an error if given a longer name. You should always assign files a version number of Ø, because the Resource Manager and Segment Loader won't operate on files with nonzero version numbers, and the Finder ignores version numbers.

There are two parts or forks to a file: the data fork and the resource fork. Normally the resource fork of an application file contains the resources used by the application such as menus, fonts, and icons, and also the application code itself. The data fork can contain anything an application wants to store there. Information stored in resource forks should always be accessed via the Resource Manager. Information in data forks can only be accessed via the File Manager. For simplicity, we'll use "file" instead of "data fork" in this manual.

A file can contain anywhere from Ø to 16,772,216 bytes (16 megabytes). Each byte is numbered: the first byte is byte Ø. You can read bytes from and write bytes to a file either singly or in sequences of unlimited length. Each read or write operation can start anywhere in the file, regardless of where the last operation began or ended. Figure 2 shows the structure of a file.

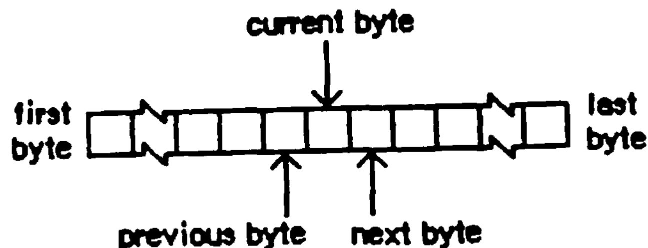


Figure 2. A File

A file's maximum size is defined by its physical end-of-file, which is one greater than the number of the last byte in its last allocation block (Figure 3). The physical end-of-file is equivalent to the maximum number of bytes the file can contain. A file's actual size is defined by its logical end-of-file, which is one greater than the number of the last byte in the file. The logical end-of-file is

equivalent to the actual number of bytes in the file, since the first byte is byte number Ø. The physical end-of-file is always greater than the logical end-of-file. For example, an empty file (one with Ø bytes) in a 1K-byte allocation block has a logical end-of-file of Ø and a physical end-of-file of 1Ø24. A file with 5Ø bytes has a logical end-of-file of 5Ø and a physical end-of-file of 1Ø24.

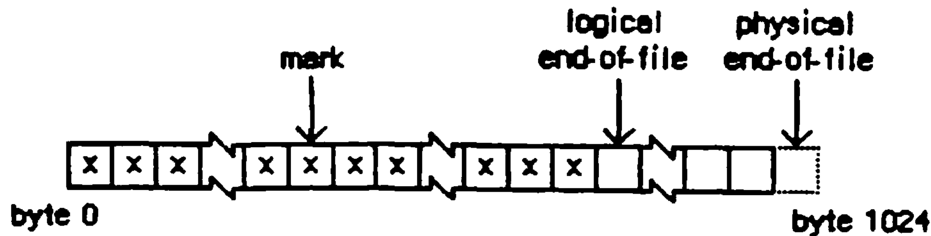


Figure 3. End-of-File and Mark

The current position marker, or <u>mark</u>, is the number of the next byte that will be read or written. The value of the mark can't exceed the value of the logical end-of-file. The mark automatically moves forward one byte for every byte read from or written to the file. If, during a write operation, the mark meets the logical end-of-file, both are moved forward one position for every additional byte written to the file. Figure 4 shows the movement of the mark and logical end-of-file.

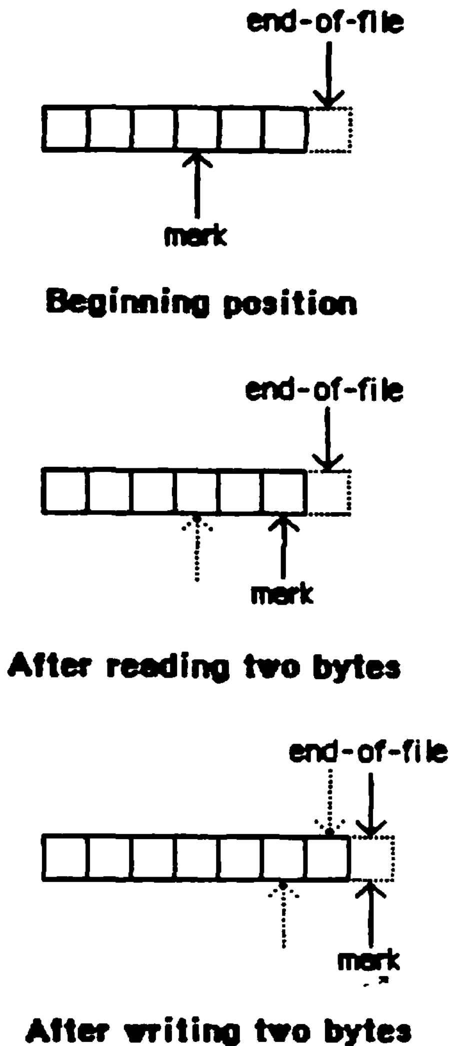


Figure 4. Movement of Logical End-of-File and Mark

If, during a write operation, the mark must move past the physical end-of-file, another allocation block is added to the file--the physical end-of-file is placed one byte beyond the end of the new allocation block, and the mark and logical end-of-file are placed at the first byte of the new allocation block.

An application can move the logical end-of-file to place it anywhere from the beginning of the file to the physical end-of-file (the mark is adjusted accordingly). If the logical end-of-file is moved to a position more than one allocation block short of the current physical end-of-file, the unneeded allocation block will be deleted from the file. The mark can be placed anywhere from the first byte in the file to the logical end-of-file.

Accessing Files

A file can be open or closed. An application can only perform certain operations, such as reading and writing, on open files; other operations, such as deleting, can only be performed on closed files.

To open a file, you must identify it by name and version number and specify the volume containing the file. When a file is opened, the File Manager creates an access path, a description of the route to be followed when accessing the file. The access path specifies the volume on which the file is located (by volume reference number, drive number, or volume name) and the location of the file on the volume. Every access path is assigned a unique path reference number used to refer to it. You should always refer to a file via its path reference number, so that files with the same name aren't confused with one another.

A file can have one access path open for writing or for both reading and writing, and one or more access paths for reading only; there cannot be more than one access path that writes to a file. Each access path is separate from all other access paths to the file. A maximum of 12 access paths can be open at one time. Each access path can move its own mark, and read at the position it indicates. All access paths to the same file share common logical and physical end-of-file markers.

The File Manager reads descriptive information about a newly opened file from its volume and stores it in memory. For example, each file has open permission information, which indicates whether data can only be read from it, or both read from and written to it. Each access path contains read/write permission information that specifies whether data is allowed to be read from the file, written to the file, both read and written, or whatever the file's open permission allows. If an application wants to write data to a file, both types of permission information must allow writing; if either type allows reading only, then no data can be written.

When an application requests that data be read from a file, the File Manager reads the data from the file and transfers it to the application's data buffer. Any part of the data that can be transferred in entire 512-byte blocks is transferred directly. Any part of the data composed of fewer than 512 bytes is also read from the file in one 512-byte block, but placed in temporary storage space in memory. Then, only the bytes containing the requested data are transferred to the application.

When an application writes data to a file, the File Manager transfers the data from the application's data buffer and writes it to the file. Any part of the data that can be transferred in entire 512-byte blocks is written directly. Any part of the data composed of fewer than 512 bytes is placed in temporary storage space in memory until 512 bytes have accumulated; then the entire block is written all at once.

Normally the temporary space in memory used for all reading and writing is the volume buffer, but an application can specify that an access path buffer be used instead for a particular access path (Figure 5).

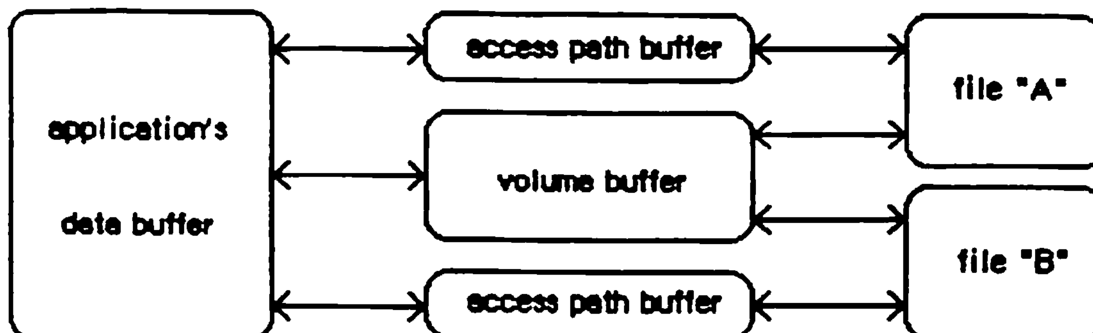


Figure 5. Buffers For Transferring Data

(w ning)

You must lock every access path buffer you use, so its location doesn't change while the file is open.

Your application can lock a file to prevent unauthorized writing to it. Locking a file ensures that none of the data in it can be changed *** The Finder doesn't treat locked and unlocked files differently ***.

(note)

Advanced programmers: The File Manager can also read a continuous stream of characters or a line of characters. In the first case, you ask the File Manager to read a specific number of bytes: when that many have been read or when the mark has reached the logical end-of-file, the read operation terminates. In the second case, called newline mode, the read will terminate when either of the above conditions is fulfilled or when a specified character, the newline character, is read. The newline character is usually Return (ASCII code $ØD), but can be any character whose ASCII code is between $ØØ and $FF, inclusive. Information about newline mode is associated with each access path to a file, and can differ from one access path to another.

FILE INFORMATION USED BY THE FINDER

A file directory lists information about all the files on a volume. The information used by the Finder is contained in a data structure of type FInfo:

```
TYPE FInfo = RECORD
               fdType:      OSType;  {type of file}
               fdCreator:   OSType;  {file's creator}
               fdFlags:     INTEGER; {flags}
               fdLocation:  Point;   {file's location}
               fdFldr:      INTEGER  {file's window}
             END;
```

Normally an application need only set the file type and creator when a file is created (see The Structure of a Macintosh Application), and the Finder will manipulate the other fields. Advanced programmers may be interested in changing the contents of the other fields as well.

FdFlags indicates whether the file's icon is invisible, whether the file has a bundle, and other characteristics used internally by the Finder:

Bit	Meaning if set
5	File has a bundle
6	File's icon is invisible

Masks for these two bits are built into the File Manager as predefined constants:

```
CONST fHasBundle = 32; {file has a bundle}
      fInvisible = 64; {file's icon is invisible}
```

You need only set the fHasBundle bit for documents with bundles. (The bundle bit for an application must have been set when the application was first installed.) FdFldr indicates the window in which the file's icon will appear:

FdFldr	Window
-3	Trash
-2	Desktop
Ø	Disk
>Ø	A folder

If fdFldr contains a positive number, the file's icon will appear in a folder; the numbers that identify folders are assigned by the Finder. Advanced programmers can get the folder number of an existing file, and place additional files in that same folder. FdLocation contains the location of the file's icon in its window, given in the local coordinate system of the window.

USING THE FILE MANAGER

This section discusses how the File Manager routines fit into the general flow of an application program and gives an idea of what routines you'll need to use. The routines themselves are described in detail in the next two sections.

You can call File Manager routines via three different methods: high-level Pascal calls, low-level Pascal calls, and assembly language. The high-level Pascal calls are designed for Pascal programmers interested in using the File Manager in a simple manner; they provide adequate file I/O and don't require much special knowledge to use. The low-level Pascal and assembly-language calls are designed for advanced Pascal programmers and assembly-language programmers interested in using the File Manager to its fullest capacity; they require some special knowledge to be used most effectively.

Information for all programmers follows here. The next two sections contain special information for high-level Pascal programmers and for low-level Pascal and assembly-language programmers.

(note)

> The names used to refer to routines here are actually the assembly-language macro names for the low-level routines, but the Pascal routine names are very similar.

The File Manager is automatically initialized each time the system is started up. Pascal programs must must include QuickDraw in their USES declaration, because the File Manager uses the QuickDraw data type Point.

To open an access path to a file, call Open. The File Manager creates an access path and returns a path reference number that you'll use every time you want to refer to it. Before you open a file, you may want to call the Standard File Package, which presents a standard interface through which the user can specify the file to be opened. If the user inserts an unmounted volume into a drive, the Standard File Package will automatically attempt to mount it. The Standard File Package will return the name of the file, the volume reference number of the volume containing the file, and additional information.

After a file has been opened, you can transfer data from it to an application's data buffer with Read, and send data from an application's data buffer to the file with Write. Read and Write allow you to specify a byte position within the data buffer, a number of bytes to transfer, and the location within the file. You can't use Write on a file whose open permission only allows reading, or on a file on a locked volume.

Once you've completed whatever reading and writing you want to do, call Close to close the file. Close writes the contents of the file's access path buffer to the volume and deletes the access path. You can remove a closed file (both forks) from a volume by calling Delete.

To protect against power loss or unexpected disk ejection, you should periodically call FlushVol (probably after each time you close a file), which writes the contents of the volume buffer and all access path buffers (if any) to the volume, and updates the descriptive information contained on the volume.

Whenever your application is finished with a disk, or the user chooses Eject from a menu, call Eject. Eject calls FlushVol, places the volume offline, and then physically ejects the volume from its drive.

To create a new, empty file, call Create. Create allows you to set some of the information about the file stored on the volume.

The preceding paragraphs covered the simplest File Manager routines: Open, Read, Write, Close, FlushVol, Eject, and Create. The remainder of this section describes the less commonly used routines, some of which are available only to advanced programmers. Skip the remainder of this section if the preceding paragraphs have provided you with all the information you want to know about using the File Manager.

Some applications may want to mount volumes themselves, bypassing the implicit mounting performed by the Standard File Package. In this case, the application must eject an unwanted disk from its drive (if necessary), and request that the user insert a different disk. The File Manager will automatically attempt to mount the volume on the disk that's inserted. If you call the Toolbox Event Manager function GetNextEvent, it will return the disk inserted event: the low-order word of the event message will contain the number of the drive, and the high-order word will contain the result code of the attempted mounting. If the result code indicates that an error occurred, you'll need to call the Disk Initialization Package to allow the user to initialize or eject the volume. Your application can then call GetVolInfo, which will return the name of the volume, the amount of unused space on the volume, and a volume reference number that you can use every time you refer to that volume.

To minimize the amount of memory used mounted volumes, an application can unmount or place off-line any volumes that aren't currently being used. To unmount a volume, call UnmountVol, which flushes a volume (by calling FlushVol) and deallocates all of the memory used for it (releasing about 1 to 1.5K bytes). To place a volume off-line, call OffLine, which flushes a volume (by calling FlushVol) and deallocates all of the memory used for it except for 94 bytes of descriptive information about the volume. Off-line volumes are placed on-line by the File Manager as needed, but your application must remount any unmounted volumes it wants to access. The File Manager itself may place volumes off-line during its normal operation.

If you would like all File Manager calls to apply to one volume, you can specify that volume as the default. You can use SetVol to set the default volume to any mounted volume, and GetVol to learn the name and volume reference number of the default volume.

Normally, volume initialization and naming is handled by the Standard File Package, which calls the Disk Initialization Package. If you want to explicitly initialize a volume or erase all files from a volume, you can call the Disk Initialization Package directly. When you want to change the name of a volume, call the File Manager function Rename.

Applications normally will use the Resource Manager to open resource forks and change the information contained within, but programmers writing unusual applications (such as a disk-copying utility) might want to use the File Manager to open resource forks. This is done by calling OpenRF. As with Open, the File Manager creates an access path and returns a path reference number that you'll use every time you want to refer to this resource fork.

As an alternative to specifying byte positions within a file with Read and Write, you can specify the byte position of the mark by calling SetFPos. GetFPos returns the byte position of the mark.

Whenever a disk has been reconstructed in an attempt to salvage lost files (because its directory or other file-access information has been destroyed), the logical end-of-file of each file will probably be equal to each physical end-of-file, regardless of where the actual logical end-of-file is. The first time an application attempts to read from a file on a reconstructed volume, it will blindly pass the correct logical end-of-file and read misinformation until it reaches the new, incorrect logical end-of-file. To prevent this from occurring, an application should always maintain an independent record of the logical end-of-file of each file it uses. To determine the File Manager's conception of the length of a file, or find out how many bytes have yet to be read from it, call GetEOF, which returns the logical end-of-file. You can change the length of a file by calling SetEOF.

Allocation blocks are automatically added to and deleted from a file as necessary. If this happens to a number of files alternately, each of the files will be contained in allocation blocks scattered throughout the volume, which increases the time required to access those files. To prevent such fragmentation of files, you can allocate a number of contiguous allocation blocks to an open file by calling Allocate.

Instead of calling FlushVol, an unusual application might call FlushFile. FlushFile forces the contents of a file's volume buffer and access path buffer (if any) to be written to its volume. FlushFile doesn't update the descriptive information contained on the volume, so the volume information won't be correct until you call FlushVol.

To get information about a file (such as its name and creation date) stored on a volume, call GetFileInfo. You can change this information by calling SetFileInfo. Changing the name or version number of a file is accomplished by calling Rename or SetFilType, respectively; they will have a similar effect, since both the file name and version number are needed to identify a file. You can lock or unlock a file by calling SetFilLock or RstFilLock, respectively.

You can't use Write, Allocate, or SetEOF on a locked file, a file whose open permission only allows reading, or a file on a locked volume. You can't use Rename or SetFilType on a file on a locked volume.

HIGH-LEVEL FILE MANAGER ROUTINES

This section describes all the high-level Pascal routines of the File Manager. Assembly-language programmers cannot call these routines. For information on calling the low-level Pascal and assembly-language routines, see the next section.

When accessing a volume, you must identify it by its volume name, its volume reference number, or the drive number of its drive--or allow the default volume to be accessed. The parameter names used in identifying a volume are volName, vRefNum, and drvNum. VRefNum and drvNum are both integers. VolName is a pointer, of type OSStrPtr, to a volume name.

The File Manager determines which volume to access by using one of the following:

1. VolName. (If volName points to a zero-length name, an error is returned.)

2. If volName is NIL or points to an improper volume name, then vRefNum or drvNum (only one is given per routine).

3. If vRefNum or drvNum is zero, the default volume. (If there isn't a default volume, an error is returned.)

(warning)

Before you pass a parameter of type OSStrPtr to a File Manager routine, be sure that memory has been allocated for the variable. For example, the following statements will ensure that memory is allocated:

```
VAR myStr: OSStr255;
BEGIN
    result := GetVol(@myStr,myRefNum);
    . . .
END;
```

When accessing a closed file on a volume, you must identify the volume by the method given above, and identify the file by its name in the fileName parameter. The high-level File Manager routines assume that the file's version number is 0. FileName can contain either the file name alone or the file name prefixed by a volume name.

(note)

Although fileName can include both the volume name and the file name, applications shouldn't encourage users to prefix a file name with a volume name.

You cannot specify an access path buffer when calling high-level Pascal routines. All access paths open on a volume will share the volume buffer, causing a slight increase in the amount of time required to access files.

All File Manager routines return a result code of type OSErr as their function result. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Accessing Volumes

```
FUNCTION GetVInfo (drvNum: INTEGER; VAR volName: OSStrPtr; VAR vRefNum:
            INTEGER; VAR freeBytes: LongInt) : OSErr;
```

GetVInfo returns the name, reference number, and available space (in bytes), in volName, vRefNum, and freeBytes, for the volume in the specified drive.

Result codes		
	noErr	No error
	nsvErr	No default volume
	paramErr	Bad drive number

```
FUNCTION GetVol (volName: OSStrPtr; VAR vRefNum: INTEGER) : OSErr;
```

GetVol returns the name of the default volume in volName and its volume reference number in vRefNum.

Result codes		
	noErr	No error
	nsvErr	No default volume

```
FUNCTION SetVol (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;
```

SetVol sets the default volume to the mounted volume specified by volName or vRefNum.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	nsvErr	No such volume
	paramErr	No default volume

FUNCTION FlushVol (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;

On the volume specified by volName or vRefNum, FlushVol writes the contents of the associated volume buffer and descriptive information about the volume (if they've changed since the last time FlushVol was called).

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

FUNCTION UnmountVol (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;

UnmountVol unmounts the volume specified by volName or vRefNum, by calling FlushVol to flush the volume buffer, closing all open files on the volume, and releasing the memory used for the volume.

(warning)
Don't unmount the startup volume.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

FUNCTION Eject (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;

Eject calls FlushVol to flush the volume specified by volName or vRefNum, places the volume offline, and then ejects the volume.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

Changing File Contents

```
FUNCTION Create (fileName: OSStr255; vRefNum: INTEGER; creator: OSType;
            fileType: OSType) : OSErr;
```

Create creates a new file with the specified name, file type, and creator, on the specified volume. The new file is unlocked and empty. Its modification and creation dates are set to the time of the system clock.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	dupFNErr	Duplicate file name
	dirFulErr	Directory full
	extFSErr	External file system
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION FSOpen (fileName: OSStr255; vRefNum: INTEGER; VAR refNum:
            INTEGER) : OSErr;
```

FSOpen creates an access path to the file having the name fileName on the specified volume. A path reference number is returned in refNum. The access path's read/write permission is set to whatever the file's open permission allows.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	mFulErr	Memory full
	nsvErr	No such volume
	opWrErr	File already open for writing
	tmfoErr	Too many files open

FUNCTION FSRead (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
OSErr;

FSRead attempts to read the number of bytes specified by the count parameter from the open file whose access path is specified by refNum, and transfer them to the data buffer pointed to by buffPtr. The read operation begins at the mark, so you might want to precede this with a call to SetFPos. If you try to read past the logical end-of-file, FSRead moves the mark to the end-of-file and returns eofErr as its function result. After the read is completed, the number of bytes actually read is returned in the count parameter.

Result codes		
	noErr	No error
	eofErr	End-of-file
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	paramErr	Negative count
	rfNumErr	Bad reference number

FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
OSErr;

FSWrite takes the number of bytes specified by the count parameter from the buffer pointed to by buffPtr and attempts to write them to the open file whose access path is specified by refNum. The write operation begins at the mark, so you might want to precede this with a call to SetFPos. After the write is completed, the number of bytes actually written is returned in the count parameter.

Result codes		
	noErr	No error
	dskFulErr	Disk full
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	paramErr	Negative count
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

FUNCTION GetFPos (refNum: INTEGER; VAR filePos: LongInt) : OSErr;

GetFPos returns, in filePos, the mark of the open file whose access path is specified by refNum.

Result codes		
	noErr	No error
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number

FUNCTION SetFPos (refNum: INTEGER; posMode: INTEGER; posOff: LongInt) : OSErr;

SetFPos sets the mark of the open file whose access path is specified by refNum, to the position specified by posMode and posOff. PosMode indicates whether the mark should be set relative to the beginning of the file, the logical end-of-file, or the mark:

PosMode	Position
0	Current position of mark (posOff is ignored)
1	Relative to beginning of file
2	Relative to logical end-of-file
3	Relative to mark

PosOff specifies the byte offset (either positive or negative) relative to posMode where the mark should actually be set. If you try to set the mark past the logical end-of-file, SetFPos moves the mark to the end-of-file and returns eofErr as its function result.

Result codes		
	noErr	No error
	eofErr	End-of-file
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	posErr	Tried to position before start of file
	rfNumErr	Bad reference number

FUNCTION GetEOF (refNum: INTEGER; VAR logEOF: LongInt) : OSErr;

GetEOF returns, in logEOF, the logical end-of-file of the open file whose access path is specified by refNum.

Result codes		
	noErr	No error
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number

FUNCTION SetEOF (refNum: INTEGER; logEOF: LongInt) : OSErr;

SetEOF sets the logical end-of-file of the open file whose access path is specified by refNum, to the position specified by logEOF. If you attempt to set the logical end-of-file beyond the physical end-of-file, the physical end-of-file is set to one byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and SetEOF returns dskFulErr as its function result. If logEOF is Ø, all space on the volume occupied by the file is released.

Result codes		
	noErr	No error
	dskFulErr	Disk full
	extFSErr	External file system
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

FUNCTION Allocate (refNum: INTEGER; VAR count: LongInt) : OSErr;

Allocate adds the number of bytes specified by the count parameter to the open file whose access path is specified by refNum, and sets the physical end-of-file to one byte beyond the last block allocated. The number of bytes allocated is always rounded up to the nearest multiple of the allocation block size, and returned in the count parameter. If there isn't enough empty space on the volume to satisfy the allocation request, the rest of the space on the volume is allocated, and Allocate returns dskFulErr as its function result.

Result codes		
	noErr	No error
	dskFulErr	Disk full
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

FUNCTION FSClose (refNum: INTEGER) : OSErr;

FSClose removes the access path specified by refNum, writes the contents of the volume buffer to the volume, and updates the file's entry in the file directory.

(note)
Some information stored on the volume won't be correct until FlushVol is called.

Result codes		
	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	fnOpnErr	File not open
	ioErr	Disk I/O error
	nsvErr	No such volume
	rfNumErr	Bad reference number

Changing Information About Files

All of the routines described in this section affect both forks of the file.

FUNCTION GetFInfo (fileName: OSStr255; vRefNum: INTEGER; VAR fndrInfo: FInfo) : OSErr;

GetFInfo returns information about the file having the name fileName on the specified volume. Information used by the Finder is returned in fndrInfo (see the "File Information Used by the Finder" section).

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	paramErr	Bad parameters and no default volume

FUNCTION SetFInfo (fileName: OSStr255; vRefNum: INTEGER; fndrInfo: FInfo) : OSErr;

For the file having the name fileName on the specified volume, SetFInfo sets information needed by the Finder to fndrInfo (see the "File Information Used by the Finder" section).

Result codes		
	noErr	No error
	extFSErr	External file system
	fLckdErr	File locked
	fnfErr	File not found
	ioErr	Disk I/O error

	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

FUNCTION SetFLock (fileName: OSStr255; vRefNum: INTEGER) : OSErr;

SetFLock locks the file having the name fileName on the specified volume. Access paths currently in use aren't affected.

Result codes	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

FUNCTION RstFilLock (fileName: OSStr255; vRefNum: INTEGER) : OSErr;

RstFilLock unlocks the file having the name fileName on the specified volume. Access paths currently in use aren't affected.

Result codes	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

FUNCTION Rename (oldName: OSStr255; vRefNum: INTEGER; newName:
OSStr255) : OSErr;

Given a file name in oldName, Rename changes the name of the file to newName. Access paths currently in use aren't affected. Given a volume name in oldName and a volume reference number in vRefNum, Rename changes the name of the specified volume to newName.

Result codes	noErr	No error
	bdNamErr	Bad file name
	dirFulErr	Directory full
	dupFNErr	Duplicate file name
	extFSErr	External file system
	fLckdErr	File locked
	fnfErr	File not found
	fsRnErr	Renaming difficulty
	ioErr	Disk I/O error
	nsvErr	No such volume
	paramErr	Bad parameters and no default volume
	vLckdErr	Software volume lock

	wPrErr	Hardware volume lock

FUNCTION FSDelete (fileName: OSStr255; vRefNum: INTEGER) : OSErr;

FSDelete removes the closed file having the name fileName from the specified volume.

(note)
This function will delete both forks of the file.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fBsyErr	File busy
	fLckdErr	File locked
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

LOW-LEVEL FILE MANAGER ROUTINES

This section contains special information for programmers using the low-level Pascal or assembly-language routines of the File Manager, and describes them in detail.

You can execute most File Manager routines either synchronously (meaning that the application must wait until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is executing). MountVol, UnmountVol, Eject, and OffLine cannot be executed asynchronously, because they use the Memory Manager to allocate and deallocate memory.

When an application calls a File Manager routine asynchronously, an I/O request is placed in the file I/O queue, and control returns to the calling application--even before the actual I/O is completed. Requests are taken from the queue one at a time (in the same order that they were entered), and processed. Only one request may be processed at any given time.

The calling application may specify a completion routine to be executed as soon as the I/O operation has been completed.

At any time, you can use the InitQueue procedure to clear all queued File Manager calls except the current one. InitQueue is especially useful when an error occurs and you no longer wish queued calls to be executed.

Routine parameters passed by an application to the File Manager and returned by the File Manager to an application are contained in a parameter block, which is memory space in the heap or stack. Most low-level Pascal calls to the File Manager are of the form

```
PBCallName (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

PBCallName is the name of the routine. ParamBlock points to the parameter block containing the parameters for the routine. If async is TRUE, the call will be executed asynchronously; if FALSE, it will be executed synchronously. Each call returns an integer result code of type OSErr if an error occurred during the call. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Assembly-language note: When you call a File Manager routine (except InitQueue), AØ must point to a parameter block containing the parameters for the routine. If you want the routine to be executed asynchronously, set bit 1Ø of the routine trap word. You can do this by supplying the word ASYNC as the second argument to the routine macro: for example

```
_Read  paramBlock,ASYNC
```

If you want the routine to be executed synchronously, set bit 9 of the routine trap word. This can be accomplished by supplying the word IMMED as the second argument to the routine macro: for example

```
_Write  paramBlock,IMMED
```

All routines except InitQueue return a result code in DØ.

Routine Parameters

There are three different kinds of parameter blocks you'll pass to File Manager routines. Each kind is used with a particular set of routine calls: I/O routines, file information routines, and volume information routines.

The lengthy, variable-length data structure of a parameter block is given below. The Device Manager and File Manager use this same data structure, but only the parts relevant to the File Manager are shown here. Each kind of parameter block contains eight fields of standard information and nine to 16 fields of additional information:

```
TYPE ParamBlkType = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
                       ioLink:       Ptr;      {next queue entry}
                       ioType:       INTEGER;  {always 5}
                       ioTrap:       INTEGER;  {routine trap}
                       ioCmdAddr:    Ptr;      {routine address}
                       ioCompletion: ProcPtr;  {completion routine}
                       ioResult:     OSErr;    {result code}
                       ioNamePtr:    OSStrPtr; {volume or file name}
                       ioVRefNum:    INTEGER;  {volume reference or
                                                drive number}
                       CASE ParamBlkType OF
                         ioParam:
                         . . .  {I/O routine parameters}
                         fileParam:
                         . . .  {file information routine parameters}
                         volumeParam:
                         . . .  {volume information routine parameters}
                         cntrlParam:
                         . . .  {Control and Status call parameters}
                     END;

     ParmBlkPtr = ^ParamBlockRec;
```

The first four fields in each parameter block are handled entirely by the File Manager, and most programmers needn't be concerned with them; programmers who are interested in them should see the section "Data Structures in Memory".

IOCompletion contains the address of a completion routine to be executed at the end of an asynchronous call; it should be NIL for asynchronous calls with no completion routine, and is automatically set to NIL for all synchronous calls. For asynchronous calls, ioResult is positive while the routine is executing, and returns the result code. Your application can poll ioResult during the asynchronous execution of a routine to determine when the routine has completed. Completion routines are executed after ioResult is returned.

IONamePtr points to either a volume name or a file name (which can be prefixed by a volume name).

(note)

> Although ioNamePtr can include both the volume name and the file name, applications shouldn't encourage users to prefix a file name with a volume name.

IOVRefNum contains either the reference number of a volume or the drive number of a drive containing a volume.

For routines that access volumes, the File Manager determines which volume to access by using one of the following:

1. IONamePtr, a pointer to the volume name.

2. If ioNamePtr is NIL, or points to an improper volume name, then ioVRefNum. (If ioVRefNum is negative, it's a volume reference number; if positive, it's a drive number.)

3. If ioVRefNum is Ø, the default volume. (If there isn't a default volume, an error is returned.)

For routines that access closed files, the File Manager determines which file to access by using ioNamePtr, a pointer to the name of the file (and possibly also of the volume).

- If the string pointed to by ioNamePtr doesn't include the volume name, the File Manager uses steps 2 and 3 above to determine the volume.

- If ioNamePtr is NIL or points to an improper file name, an error is returned.

The first eight fields are adequate for a few calls, but most of the File Manager routines require more fields, as described below. The parameters used with Control and Status calls are described in the Device Manager manual *** doesn't yet exist ***.

I/O Parameters

When you call one of the I/O routines, use these nine additional fields after the standard 8-field parameter block:

```
ioParam:
  ioRefNum:     INTEGER;    {path reference number}
  ioVersNum:    SignedByte; {version number}
  ioPermssn:    SignedByte; {read/write permission}
  ioMisc:       Ptr;        {miscellanous}
  ioBuffer:     Ptr;        {data buffer}
  ioReqCount:   LongInt;    {requested number of bytes}
  ioActCount:   LongInt;    {actual number of bytes}
  ioPosMode:    INTEGER;    {newline character and type of
                             positioning operation}
  ioPosOffset:  LongInt;    {size of positioning offset}
```

For routines that access open files, the File Manager determines which file to access by using the path reference number in ioRefNum. IOPermssn requests permission to read or write via an access path:

IOPermssn	I/O operation
Ø	Whatever is currently allowed
1	Reading only
2	Writing only
3	Reading and writing

This request is compared with the open permission of the file. If the open permission doesn't allow I/O as requested, an error will be returned.

The content of ioMisc depends on the routine called; it contains either a pointer to an access path buffer, a new logical end-of-file, a new version number, or a pointer to a new volume or file name. Since ioMisc is of type Ptr, while end-of-file is LongInt and version number is SignedByte, you'll need to use conversions like these:

```
VAR pBlock: ParmBlkPtr;
    myVers: SignedByte;
    myEOF: LongInt;

pBlock^.ioMisc := POINTER(ORD4(myVers));
myVers := ORD(pBlock^.ioMisc);

pBlock^.ioMisc := POINTER(ORD4(myEOF));
myEOF := ORD4(pBlock^.ioMisc);
```

IOBuffer points to a data buffer into which data is written by Read calls and from which data is read by Write calls. IOReqCount specifies the requested number of bytes to be read, written, or allocated. IOActCount contains the number of bytes actually read, written, or allocated.

IOPosMode and ioPosOffset contain positioning information used for Read, Write, and SetFPos calls. Bits Ø and 1 of ioPosMode indicate how to position the mark:

IOPosMode	Offset
Ø	Current position of mark (ioPosOffset ignored)
1	Relative to beginning of file
2	Relative to logical end-of-file
3	Relative to current mark

IOPosOffset specifies the byte offset (either positive or negative) relative to ioPosMode where the operation will be performed.

Assembly-language note: If bit 6 of ioPosMode is set, the File Manager will verify that all data read into memory by a Read call exactly matches the data on the volume (an error will be returned if any data don't match).

(note)

Advanced programmers: Bit 7 of ioPosMode is the newline flag--set if read operations should terminate at newline characters, and clear if reading should terminate at the end of the access path buffer or volume buffer. The

high-order byte of ioPosMode contains the ASCII code of the newline character.

File Information Parameters

When you call the GetFileInfo and SetFileInfo functions, use the following 16 additional fields after the standard 8-field parameter block:

```
fileParam:
 (ioFRefNum:     INTEGER;     {path reference number}
  ioFVersNum:    SignedByte;  {version number}
  filler1:       SignedByte;  {not used}
  ioFDirIndex:   INTEGER;     {file directory index}
  ioFlAttrib:    SignedByte;  {file attributes}
  ioFlVersNum:   SignedByte;  {version number}
  ioFlFndrInfo:  FInfo;       {information used by the Finder}
  ioFlNum:       LongInt;     {file number}
  ioFlStBlk:     INTEGER;     {first allocation block of data fork}
  ioFlLgLen:     LongInt;     {logical end-of-file of data fork}
  ioFlPyLen:     LongInt;     {physical end-of-file of data fork}
  ioFlRStBlk     INTEGER;     {first allocation block of resource fork}
  ioFlRLgLen     LongInt;     {logical end-of-file of resource fork}
  ioFlRPyLen     LongInt;     {physical end-of-file of resource fork}
  ioFlCrDat      LongInt;     {date and time of creation}
  ioFlMdDat      LongInt);    {date and time of last modification}
```

IOFDirIndex contains the file directory index, another method of referring to a file; most programmers needn't be concerned with information about file directories, but those interested can read the section "Data Organization on Volumes".

Assembly-language note: IOFlAttrib contains eight bits of file attributes: if bit 7 is set, the file is open; if bit Ø is set, the file is locked.

IOFlStBlk and ioFlRStBlk are zeroed if the file's data or resource fork is empty, respectively. The date and time in the ioFlCrDat and ioFlMdDat fields are specified in seconds since 12:ØØ AM, January 1, 19Ø4.

Volume Information Parameters

When you call GetVolInfo, use the following 14 additional fields:

```
volumeParam:
 (filler2:      LongInt;  {not used}
  ioVolIndex:   INTEGER;  {volume index}
  ioVCrDate:    LongInt;  {date and time of initialization}
  ioVLsBkUp:    LongInt;  {date and time of last volume backup}
  ioVAtrb:      INTEGER;  {bit 15=1 if volume locked}
  ioVNmFls:     INTEGER;  {number of files in file directory}
  ioVDirSt:     INTEGER;  {first block of file directory}
  ioVBlLn:      INTEGER;  {number of blocks in file directory}
  ioVNmAlBlks:  INTEGER;  {number of allocation blocks on volume}
  ioVAlBlkSiz:  LongInt;  {number of bytes per allocation block}
  ioVClpSiz:    LongInt;  {number of bytes to allocate}
  ioAlBlSt:     INTEGER;  {first block in volume block map}
  ioVNxtFNum:   LongInt;  {next free file number}
  ioVFrBlk:     INTEGER); {number of free allocation blocks}
```

IOVolIndex contains the volume index, another method of referring to a volume; the first volume mounted has an index of 1, and so on. Most programmers needn't be concerned with the parameters providing information about file directories and block maps (such as ioVNmFls), but interested programmers can read the section "Data Organization on Volumes".

Routine Descriptions

This section describes the procedures and functions. Each routine description includes the low-level Pascal form of the call and the routine's assembly-language macro. A list of the fields in the parameter block affected by the call is also given.

Assembly-language note: The field names given in these descriptions are those of the ParamBlockRec data type; see the "Summary of the File Manager" for the equivalent assembly-language equates.

The number next to each parameter name indicates the byte offset of the parameter from the start of the parameter block pointed to by AØ; only assembly-language programmers need be concerned with it. An arrow drawn next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
<--	Parameter must be passed to the routine
-->	Parameter will be returned by the routine
<->	Parameter must be passed to and will be returned by the routine

Initializing the File I/O Queue

PROCEDURE InitQueue;

Trap macro _InitQueue

InitQueue clears all queued File Manager calls except the current one. There are no parameters or result codes associated with InitQueue.

Accessing Volumes

FUNCTION PBMountVol (paramBlock: ParmBlkPtr) : OSErr;

Trap macro _MountVol

Parameter block

<--	16	ioResult	word
<->	22	ioVRefNum	word

Result codes

noErr	No error
badMDBErr	Master directory block is bad
extFSErr	External file system
ioErr	Disk I/O error
mFulErr	Memory full
noMacDskErr	Not a Macintosh volume
nsDrvErr	No such drive
paramErr	Bad drive number
volOnLinErr	Volume already on-line

PBMountVol mounts the volume in the drive whose number is ioVRefNum, and returns a volume reference number in ioVRefNum. If there are no volumes already mounted, this volume becomes the default volume. PBMountVol is always executed synchronously.

FUNCTION PBGetVolInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetVolInfo

Parameter block

→	12	ioCompletion	pointer
←	16	ioResult	word
↔	18	ioNamePtr	pointer
↔	22	ioVRefNum	word
→	28	ioVolIndex	word
←	30	ioVCrDate	long word
←	34	ioVLsBkUp	long word
←	38	ioVAtrb	word
←	40	ioVNmFls	word
←	42	ioVDirSt	word
←	44	ioVBlLn	word
←	46	ioVNmAlBlks	word
←	48	ioVAlBlkSiz	long word
←	52	ioVClpSiz	long word
←	56	ioAlBlSt	word
←	58	ioVNxtFNum	long word
←	62	ioVFrBlk	word

Result codes

noErr	No error
nsvErr	No such volume
paramErr	No default volume

PBGetVolInfo returns information about the specified volume. If ioVolIndex is positive, the File Manager attempts to use it to find the volume. If ioVolIndex is negative, the File Manager uses ioNamePtr and ioVRefNum in the standard way to determine which volume. If ioVolIndex is 0, the File Manager attempts to access the volume by using ioVRefNum only. The volume reference number is returned in ioVRefNum, and the volume name is returned in ioNamePtr, unless ioNamePtr is NIL.

FUNCTION PBGetVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _GetVol

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
<--	18	ioNamePtr	pointer
<--	22	ioVRefNum	word

<u>Result codes</u>

noErr	No error
nsvErr	No default volume

PBGetVol returns the name of the default volume in ioNamePtr and its volume reference number in ioVRefNum, unless ioNamePtr is NIL.

FUNCTION PBSetVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _SetVol

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

<u>Result codes</u>

noErr	No error
bdNamErr	Bad volume name
nsvErr	No such volume
paramErr	No default volume

PBSetVol sets the default volume to the mounted volume specified by ioNamePtr or ioVRefNum.

FUNCTION PBFlshVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _FlushVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBFlshVol writes descriptive information, the contents of the associated volume buffer, and all access path buffers to the volume specified by ioNamePtr or ioVRefNum, to the volume (if they've changed since the last time PBFlshVol was called). The volume modification date is set to the current time.

FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr) : OSErr;

Trap macro _UnmountVol

Parameter block

←--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBUnmountVol unmounts the volume specified by ioNamePtr or ioVRefNum, by calling PBFlushVol to flush the volume, closing all open files on the volume, and releasing all the memory used for the volume. PBUnmountVol is always executed synchronously.

(eye)
Don't unmount the startup volume.

FUNCTION PBOffLine (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _OffLine

Parameter block

-->	12	ioCompletion	pointer
←--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBOffLine places off-line the volume specified by ioNamePtr or ioVRefNum, by calling PBFlshVol to flush the volume, and releasing all the memory used for the volume except for 94 bytes of descriptive information.

FUNCTION PBEject (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Eject

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBEject calls PBOffLine to place the volume specified by ioNamePtr or ioVRefNum offline, and then ejects the volume.

You may call PBEject asynchronously; the first part of the call is executed synchronously, and the actual ejection is executed asynchronously.

Changing File Contents

FUNCTION PBCreate (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Create

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
bdNamErr	Bad file name
dupFNErr	Duplicate file name
dirFulErr	Directory full
extFSErr	External file system
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBCreate creates a new file having the name ioNamePtr and the version number ioVersNum, on the specified volume. The new file is unlocked and empty. Its modification and creation dates are set to the time of the system clock. The application should call PBSetFInfo to fill in the information needed by the Finder.

FUNCTION PBOpen (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Open

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	27	ioPermssn	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
mFulErr	Memory full
nsvErr	No such volume
opWrErr	File already open for writing
tmfoErr	Too many files open

PBOpen creates an access path to the file having the name ioNamePtr and the version number ioVersNum, on the specified volume. A path reference number is returned in ioRefNum.

IOMisc either points to a 522-byte portion of memory to be used as the access path's buffer, or is NIL if you want the volume buffer to be used instead.

(eye)

You should ensure that all access paths to a single file share the same buffer so that they will read and write the same data.

IOPermssn specifies the path's read/write permission. A path can be opened for writing even if it accesses a file on a locked volume, and an error won't be returned until a PBWrite, PBSetEOF, or PBAllocate call is made.

If you attempt to open a locked file for writing, PBOpen will return opWrErr as its function result. If you attempt to open a file for writing and it already has an access path that allows writing, PBOpen will return the reference number of the existing access path in ioRefNum and opWrErr as its function result.

FUNCTION PBOpenRF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _OpenRF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	27	ioPermssn	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
mFulErr	Memory full
nsvErr	No such volume
opWrErr	File already open for writing
permErr	Open permission doesn't allow reading
tmfoErr	Too many files open

PBOpenRF is identical to PBOpen, except that it opens the file's resource fork instead of its data fork.

FUNCTION PBRead (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Read

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
<->	46	ioPosOffset	long word

Result codes

noErr	No error
eofErr	End-of-file
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
paramErr	Negative ioReqCount
rfNumErr	Bad reference number

PBRead attempts to read ioReqCount bytes from the open file whose access path is specified by ioRefNum, and transfer them to the data buffer pointed to by ioBuffer. If you try to read past the logical end-of-file, PBRead moves the mark to the end-of-file and returns eofErr as its function result. After the read operation is completed, the mark is returned in ioPosOffset and the number of bytes actually read is returned in ioActCount.

(note)

Advanced programmers: IOPosMode contains the newline character (if any), and indicates whether the read should begin relative to the beginning of the file, the mark, or the end-of-file. The byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset. If a newline character is not specified, the data will be read one byte at a time until ioReqCount bytes have been read or the end-of-file is reached. If a newline character is specified, the data will be read one byte at a time until the newline character is encountered, the end-of-file is reached, or ioReqCount bytes have been read.

FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Write

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
dskFulErr	Disk full
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
paramErr	Negative ioReqCount
posErr	Position is beyond end-of-file
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBWrite takes ioReqCount bytes from the buffer pointed to by ioBuffer and attempts to write them to the open file whose access path is specified by ioRefNum. After the write operation is completed, the mark is returned in ioPosOffset, and the number of bytes actually written is returned in ioActCount.

IOPosMode indicates whether the write should begin relative to the beginning of the file, the mark, or the end-of-file. The byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset.

FUNCTION PBGetFPos (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetFPos

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
<--	36	ioReqCount	long word
<--	40	ioActCount	long word
<--	44	ioPosMode	word
<--	46	ioPosOffset	long word

Result codes

noErr	No error
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number

PBGetFPos returns, in ioPosOffset, the mark of the open file whose access path is specified by ioRefNum. GetFPos sets ioReqCount, ioActCount, and ioPosMode to 0.

FUNCTION PBSetFPos (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetFPos

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
eofErr	End-of-file
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
posErr	Tried to position before start of file
rfNumErr	Bad reference number

PBSetFPos sets the mark of the open file whose access path is specified by ioRefNum, to the position specified by ioPosMode and ioPosOffset. IoPosMode indicates whether the mark should be set relative to the beginning of the file, the mark, or the logical end-of-file. The byte offset from the position given by ioPosMode, where the mark should actually be set, is given by ioPosOffset. If you try to set the mark past the logical end-of-file, PBSetFPos moves the mark to the end-of-file and returns eofErr as its function result.

FUNCTION PBGetEOF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetEOF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
<--	28	ioMisc	long word

Result codes

noErr	No error
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number

PBGetEOF returns, in ioMisc, the logical end-of-file of the open file whose access path is specified by ioRefNum.

FUNCTION PBSetEOF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetEOF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	28	ioMisc	long word

Result codes

noErr	No error
dskFulErr	Disk full
extFSErr	External file system
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBSetEOF sets the logical end-of-file of the open file whose access path is specified by ioRefNum, to ioMisc. If the logical end-of-file is set beyond the physical end-of-file, the physical end-of-file is set to one byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and PBSetEOF returns dskFulErr as its function result. If ioMisc is Ø, all space on the volume occupied by the file is released.

FUNCTION PBAllocate (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Allocate

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	36	ioReqCount	long word
<--	40	ioActCount	long word

Result codes

noErr	No error
dskFulErr	Disk full
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBAllocate adds ioReqCount bytes to the open file whose access path is specified by ioRefNum, and sets the physical end-of-file to one byte beyond the last block allocated. The number of bytes allocated is always rounded up to the nearest multiple of the allocation block size, and returned in ioActCount. If there isn't enough empty space on the volume to satisfy the allocation request, PBAllocate allocates the rest of the space on the volume and returns dskFulErr as its function result.

FUNCTION PBFlshFile (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _FlushFile

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
fnOpnErr	File not open
ioErr	Disk I/O error
nsvErr	No such volume
rfNumErr	Bad reference number

PBFlshFile writes the contents of the access path buffer indicated by ioRefNum to the volume, and updates the file's entry in the file directory.

(eye)

Some information stored on the volume won't be correct until PBFlshVol is called.

FUNCTION PBClose (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Close

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
fnOpnErr	File not open
ioErr	Disk I/O error
nsvErr	No such volume
rfNumErr	Bad reference number

PBClose removes the access path specified by ioRefNum and writes the contents of the access path buffer to the volume.

(eye)
Some information stored on the volume won't be correct until PBFlshVol is called.

Changing Information About Files

All of the routines described in this section affect both forks of a file.

FUNCTION PBGetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _GetFileInfo

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	28	ioFDirIndex	word
<--	30	ioFlAttrib	byte
<--	31	ioFlVersNum	byte
<--	32	ioFndrInfo	16 bytes
<--	48	ioFlNum	long word
<--	52	ioFlStBlk	word
<--	54	ioFlLgLen	long word
<--	58	ioFlPyLen	long word
<--	62	ioFlRStBlk	word
<--	64	ioFlRLgLen	long word
<--	68	ioFlRPyLen	long word
<--	72	ioFlCrDat	long word
<--	76	ioFlMdDat	long word

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
paramErr	No default volume

PBGetFInfo returns information about the specified file. If ioFDirIndex is positive, the File Manager returns information about the file whose file number is ioFDirIndex on the specified volume (see the section "Data Organization on Volumes" if you're interested in using this method). If ioFDirIndex is negative or zero, the File Manager returns information about the file having the name ioNamePtr and the version number ioVersNum, on the specified volume. Unless ioNamePtr is NIL, ioNamePtr returns a pointer to the name of the file. If the file is open, the reference number of the first access path found is returned in ioRefNum.

FUNCTION PBSetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetFileInfo

Parameter block

→	12	ioCompletion	pointer
←	16	ioResult	word
→	18	ioNamePtr	pointer
→	22	ioVRefNum	word
→	26	ioVersNum	byte
→	32	ioFndrInfo	16 bytes
→	72	ioFlCrDat	long word
→	76	ioFlMdDat	long word

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFInfo sets information about the file (including its creation and modification dates, and information needed by the Finder) having the name ioNamePtr and the version number ioVersNum on the specified volume. You should call PBGetFInfo just before PBSetFInfo, so the current information is present in the parameter block.

FUNCTION PBSetFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _SetFilLock

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

<u>Result codes</u>

noErr	No error
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFLock locks the file having the name ioNamePtr and the version number ioVersNum on the specified volume. Access paths currently in use aren't affected.

FUNCTION PBRstFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

<u>Trap macro</u> _RstFilLock

<u>Parameter block</u>

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

<u>Result codes</u>

noErr	No error
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBRstFLock unlocks the file having the name ioNamePtr and the version number ioVersNum on the specified volume. Access paths currently in use aren't affected.

FUNCTION PBSetFVers (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _SetFilType

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	28	ioMisc	byte

Result codes

noErr	No error
bdNamErr	Bad file name
dupFNErr	Duplicate file name and version
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
nsvErr	No such volume
ioErr	Disk I/O error
paramErr	No default volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFVers changes the version number of the file having the name ioNamePtr and version number ioVersNum on the specified volume, to ioMisc. Access paths currently in use aren't affected.

(warning)
The Resource Manager and Segment Loader operate only on files with version number Ø; changing the version number of a file to a nonzero number will prevent them from operating on it.

FUNCTION PBRename (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Rename

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
dirFulErr	Directory full
dupFNErr	Duplicate file name and version
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
fsRnErr	Renaming difficulty
ioErr	Disk I/O error
nsvErr	No such volume
paramErr	No default volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

Given a file name in ioNamePtr and a version number in ioVersNum, Rename changes the name of the specified file to ioMisc; given a volume name in ioNamePtr or a volume reference number in ioVRefNum, it changes the name of the specified volume to ioMisc. Access paths currently in use aren't affected.

FUNCTION PBDelete (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _Delete

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fBsyErr	File busy
fLckdErr	File locked
fnfErr	File not found
nsvErr	No such volume
ioErr	Disk I/O error
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBDelete removes the closed file having the name ioNamePtr and the version number ioVersNum, from the specified volume. You can't issue PBDelete to remove an open file.

(note)
This function will delete both forks of the file.

DATA ORGANIZATION ON VOLUMES

This section explains how information is organized on volumes. Most of the information is accessible only through assembly language, but some advanced Pascal programmers may be interested.

The File Manager communicates with device drivers that read and write data via block-level requests to devices containing Macintosh-initialized volumes. (Macintosh-initialized volumes are volumes initialized by the Disk Initialization Package.) The actual type of volume and device is unimportant to the File Manager; the only requirements are that the volume was initialized by the Disk Initialization Package and that the device driver be able to communicate via block-level requests.

The 3 1/2-inch built-in and optional external drives are accessed via the Disk Driver. If you want to use the File Manager to access files on Macintosh-initialized volumes on other types of devices, you must write a device driver that can read and write data via block-level requests to the device on which the volume will be mounted. If you want to access files on nonMacintosh-initialized volumes, you must write your own external file system (see the section "Using an External File System").

The information on all block-initialized volumes is organized in logical blocks and allocation blocks. Logical blocks contain a number of bytes of standard information (512 bytes on Macintosh-initialized volumes), and an additional number of bytes of information specific to the disk driver (12 bytes on Macintosh-initialized volumes). Allocation blocks are composed of any integral number of logical blocks, and are simply a means of grouping logical blocks together in more convenient parcels.

The remainder of this section applies only to Macintosh-initialized volumes. NonMacintosh-initialized volumes must be accessed via an external file system, and the information on them must be organized by an external initializing program.

A Macintosh-initialized volume contains information needed to start up the system in logical blocks Ø and 1 (Figure 6). Logical block 2 of the volume begins the master directory block. The master directory block contains volume information and the volume allocation block map, which records whether each block on the volume is unused or what part of a file it contains data from.

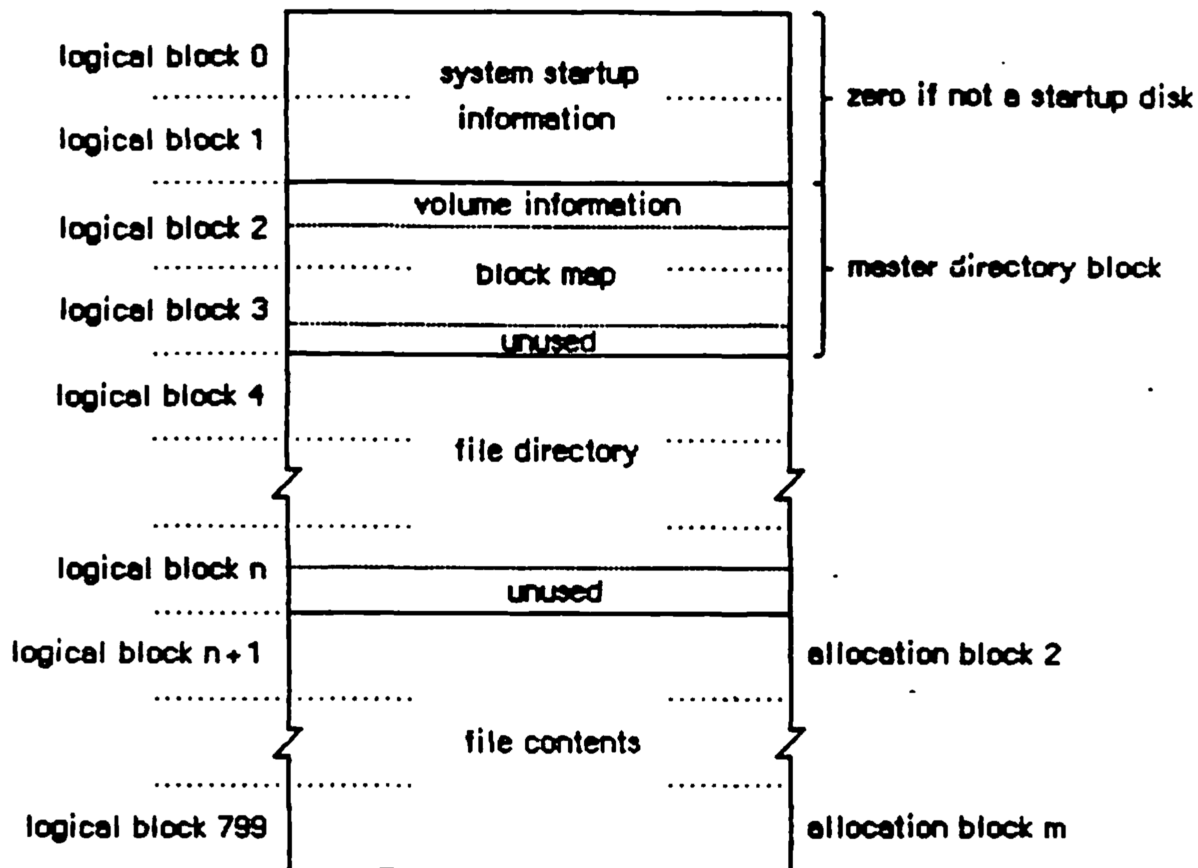


Figure 6. A 400K-Byte Volume With 1K-Byte Allocation Blocks

The master directory "block" always occupies two blocks--the Disk Initialization Package varies the allocation block size as necessary to achieve this constraint.

In the next logical block following the block map begins the file directory, which contains descriptions and locations of all the files on the volume. The rest of the logical blocks on the volume contain files or garbage (such as parts of deleted files). The precise format of the volume information, volume allocation block map, file directory, and files is explained in the following sections.

Volume Information

The volume information is contained in the first 64 bytes of the master directory block (Figure 7). This information is written on the volume when it's initialized, and modified thereafter by the File Manager.

byte	field	description
byte 0	drSigWord (word)	always $D2D7
2	drCrDate (long word)	date and time of intialization
6	drLsBkUp (long word)	date and time of last backup
10	drAtrb (word)	volume attributes
12	drNmFls (word)	number of files in file directory
14	drDirSt (word)	first logical block of file directory
16	drBlLen (word)	number of logical blocks in file directory
18	drNmAlBlks (word)	number of allocation blocks on volume
20	drAlBlkSiz (long word)	size of allocation blocks
24	drClpSiz (long word)	number of bytes to allocate
28	drAlBlSt (word)	logical block number of first allocation block
30	drNxtFNum (long word)	next unused file number
34	drFreeBks (word)	number of unused allocation blocks
36	drVN (byte)	length of volume name
37	drVN+1 (bytes)	characters of volume name

Figure 7. Volume Information

DrAtrb contains the volume attributes. Its bits, if set, indicate the following:

Bit	Meaning
7	Volume is locked by hardware
15	Volume is locked by software

DrClpSiz contains the minimum number of bytes to allocate each time the Allocate function is called, to minimize fragmentation of files; it's always a multiple of the allocation block size. DrNxtFNum contains the next unused file number (see the "File Directory" section below for an explanation of file numbers).

Volume Allocation Block Map

The volume allocation block map represents every allocation block on the volume with a 12-bit entry indicating whether the block is unused or allocated to a file. It begins in the master directory block at the byte following the volume information, and continues for as many logical blocks as needed. For example, a 400K-byte volume with a 10-block file directory and 1K-byte allocation blocks would have a 591-byte block map.

The first entry in the block map is for block number 2; the block map doesn't contain entries for the startup blocks. Each entry specifies whether the block is unused, whether it's the last block in the file, or which allocation block is next in the file:

Entry	Meaning
0	Block is unused
1	Block is the last block of the file
2..4095	Number of next block in the file

For instance, assume that there's one file on the volume, stored in allocation blocks 8, 11, 12, and 17; the first 16 entries of the block map would read

0 0 0 0 0 0 11 0 0 12 17 0 0 0 0 1

The first allocation block on a volume typically follows the file directory. The first allocation block is number 2 because of the special meaning of numbers 0 and 1.

(note)

> As explained below, it's possible to begin the allocation blocks immediately following the master directory block and place the file directory somewhere within the allocation blocks. In this case, the allocation blocks occupied by the file directory must be marked with $FFF's in the allocation block map.

File Directory

The file directory contains an entry for each file. Each entry lists information about one file on the volume, including its name and location. Each file is listed by its own unique file number, which the File Manager uses to distinguish it from other files on the volume.

A file directory entry contains 51 bytes plus one byte for each character in the file name (Figure 8); if the file names average 20 characters, a directory can hold seven file entries per logical block. Entries are always an integral number of words and don't cross logical block boundaries. The length of a file directory depends on the maximum number of files the volume can contain; for example, on a 400K-byte volume the file directory occupies 12 logical blocks.

The file directory conventionally follows the block map and precedes the allocation blocks, but a volume-initializing program could actually place the file directory anywhere within the allocation blocks as long as the blocks occupied by the file directory are marked with $FFF's in the block map.

Byte	Field	Description
byte 0	flFlags (byte)	bit 7=1 if entry used; bit 0=1 if file locked
1	flTyp (byte)	version number
2	flUsrWds (16 bytes)	information used by the Finder
18	flFlNum (long word)	file number
22	flStBlk (word)	first allocation block of data fork
24	flLgLen (long word)	data fork's logical end-of-file
28	flPyLen (long word)	data fork's physical end-of-file
32	flRStBlk (word)	first allocation block of resource fork
34	flRLgLen (long word)	resource fork's logical end-of-file
38	flRPyLen (long word)	resource fork's physical end-of-file
42	flCrDat (long word)	date and time file was created
46	flMdDat (long word)	date and time file was last modified
50	flName (byte)	length of file name
51	flNam+1 (bytes)	characters of file name

Figure 8. A File Directory Entry

FlStBlk and flRStBlk are 0 if the data or resource fork doesn't exist. FlCrDat and flMdDat are given in seconds since 12:00 AM, January 1, 1904.

Each time a new file is created, an entry for the new file is placed in the file directory. Each time a file is deleted, its entry in the file directory is zeroed, and all blocks used by that file on the volume are released as free space.

File Tags on Volumes

As mentioned previously, logical blocks contain 512 bytes of standard information preceded by 12 bytes of file tags (Figure 9). The file tags are designed to allow easy reconstruction of files from a volume whose directory or other file-access information has been destroyed.

Offset	Field	Description
byte 0	file number (long word)	file number
4	fork type (byte)	bit 1=1 if resource fork
5	file attributes (byte)	bit 7=1 if open; bit 0=1 if locked
6	file sequence (word)	logical block sequence number
8	mod date (long word)	date and time last modified

Figure 9. File Tags on Volumes

The file sequence indicates which relative portion of a file the block contains--the first logical block of a file has a sequence number of Ø, the second a sequence number of 1, and so on.

DATA STRUCTURES IN MEMORY

This section describes the memory data structures used by the File Manager and any external file system that accesses files on Macintosh-initialized volumes. Most of this information is accessible only through assembly language, but some advanced Pascal programmers may be interested.

The data structures in memory used by the File Manager and all external file systems include:

- the file I/O queue, listing the currently executing routine (if any), and any asynchronous routines awaiting execution
- the volume-control-block queue, listing information about each mounted volume
- copies of volume allocation block maps; one for each on-line volume
- the file-control-block buffer, listing information about each access path
- volume buffers; one for each on-line volume
- optional access path buffers; one for each access path
- the drive queue, listing information about each drive connected to the Macintosh

The File I/O Queue

The file I/O queue contains a list of all asynchronous routines awaiting execution. Each time a routine is called, an entry is placed in the queue; each time a routine is completed, its entry is removed from the queue. Entries in the queue are processed in a first-in, first-out order.

The file I/O queue is shown in Figure 10. Bit 7 of fsBusy is set if there are any entries in the queue. FSQHead points to first entry in the queue, and fsQTail points to the last entry in the queue.

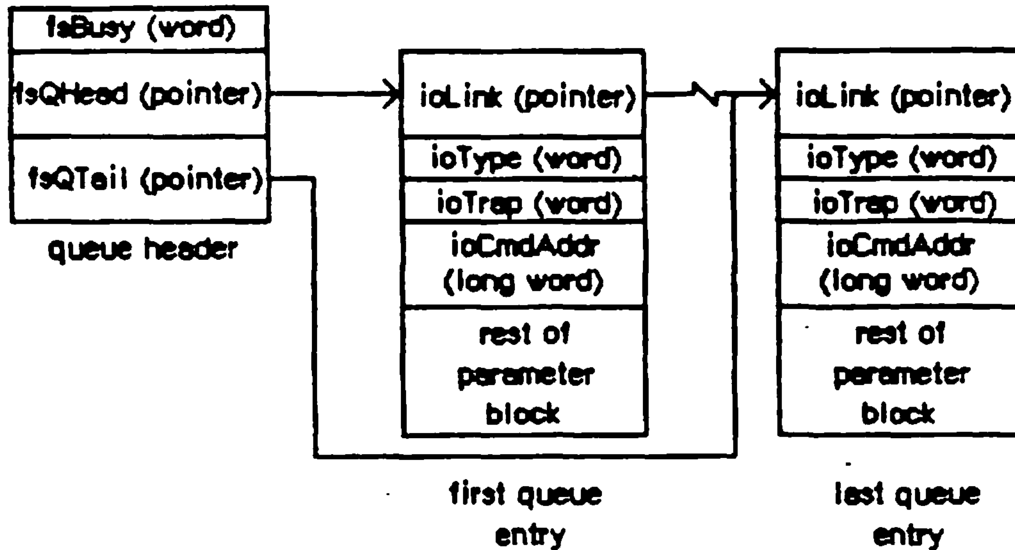


Figure 10. The File I/O Queue

Each queue entry consists of a parameter block for the routine that was called. The structure of this block is shown in part below:

```
TYPE ParamBlockRec = RECORD
                     ioLink:      Ptr;      {next entry}
                     ioType:      INTEGER;  {always fsQType}
                     ioTrap:      INTEGER;  {routine trap}
                     ioCmdAddr:   Ptr;      {routine address}
                       . . .                {rest of block}
                     END;
```

IOLink points to the next entry in the queue, and ioType indicates the queue type, which should always be the value of the predefined constant fsQType. IOTrap and ioCmdAddr contain the trap word and address of the File Manager routine that was called.

You can refer to the file I/O queue by using the system global fsQHdr, which points to the fsBusy word.

Volume Control Blocks

Each time a volume is mounted, its volume information is read from the volume and used to build a new volume control block in the volume-control-block queue (unless an ejected or off-line volume is being remounted). A copy of the volume block map is also read from the system heap, and a volume buffer is created on the system heap.

The volume-control-block queue is a list of the volume control blocks for all mounted volumes, maintained on the system heap. Its data structure is shown in Figure 11. Bit 7 of qFlags is set if there are any entries in the queue. QHead points to first entry in the queue, and qTail points to the last entry in the queue.

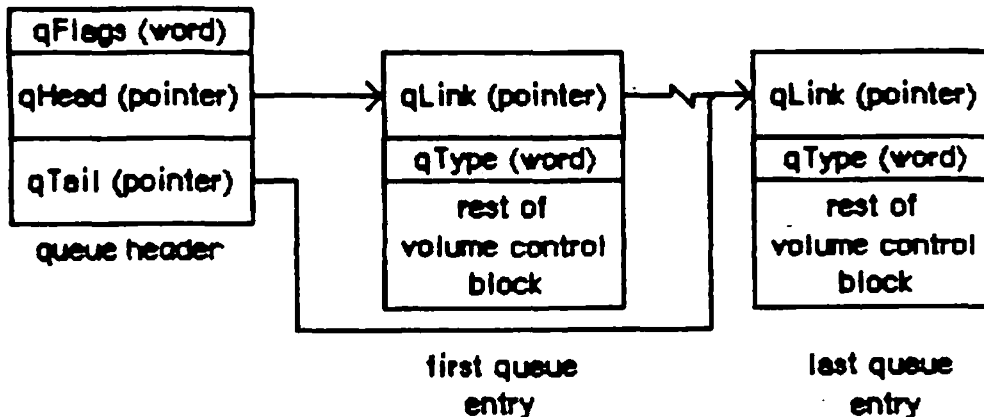


Figure 11. Volume-Control-Block Queue

Each queue entry consists of six bytes followed by a volume control block (Figure 12). A volume control block is a 94-byte nonrelocatable block that contains volume-specific information, including the first 64 bytes of the master directory block (bytes 8 to 72 of the volume control block match bytes Ø to 64 of the volume information).

Byte	Field	Description
byte 0	qLink (pointer)	pointer to next queue entry
4	qType (word)	not used
6	vcbFlags (word)	bit 15=1 if volume control block is dirty
8	vcbSigWord (word)	always $D2D7
10	vcbCrDate (long word)	date and time volume was initialized
14	vcbLsBkUp (long word)	date and time last backup copy was made
18	vcbAtrb (word)	volume attributes
20	vcbNmFls (word)	number of files in file directory
22	vcbDirSt (word)	first logical block of file directory
24	vcbBlLn (word)	length of file directory
26	vcbNmBlks (word)	number of allocation blocks on volume
28	vcbAlBlkSiz (long word)	size of allocation blocks
32	vcbClpSiz (long word)	number of bytes to allocate
36	vcbAlBlSt (word)	first logical block in block map
38	vcbNxtFNum (long word)	next unused file number
42	vcbFreeBks (word)	number of unused allocation blocks
44	vcbVN (byte)	length of volume name
45	vcbVN+1 (bytes)	characters of volume name
72	vcbDrvNum (word)	drive number of drive in which volume is mounted
74	vcbDRefNum (word)	driver reference number of driver for drive in which volume is mounted
76	vcbFSID (word)	ID for file system handling volume
78	vcbVRefNum (word)	volume reference number
80	vcbMAdr (pointer)	memory location of volume block map
84	vcbBufAdr (pointer)	memory location of volume buffer
88	vcbMLen (word)	number of bytes in volume block map
90	vcbDirIndex (word)	for internal File Manager use
92	vcbDirBlk (word)	for internal File Manager use

Figure 12. A Volume Control Block

QLink points to the next entry in the queue.

Bit 15 of vcbFlags is set if the volume information has been changed by a routine call since the volume was last affected by a FlushVol call. VCBAtr contains the volume attributes. Each bit, if set, indicates the following:

Bit	Meaning
Ø-2	Inconsistencies were found between the volume information and the file directory when the volume was mounted
6	Volume is busy (one or more files are open)
7	Volume is locked by hardware
15	Volume is locked by software

VCBDrvNum contains the drive number of the drive on which the volume is mounted; vcbDRefNum contains the driver reference number of the driver used to access on volume is mounted. When a mounted volume is placed off-line, vcbDrvNum is zeroed. When ejected, vcbDrvNum is zeroed and vcbDRefNum is set to the negative of vcbDrvNum (becoming a positive number). VCBFSID identifies the file system handling the volume; it's Ø for volumes handled by the File Manager, and nonzero for volumes handled by other file systems.

When a volume is placed off-line, its buffer and block map are deallocated. When a volume is unmounted, its volume control block is removed from the volume-control-block queue.

You can refer to the volume-control-block queue by using the system global vcbQHdr, which points to the qFlags word. The default volume's volume control block is pointed to by the system global defVCBPtr.

File Control Blocks

Each time a file is opened, the file's directory entry is used to build a 3Ø-byte file control block in the file-control-block buffer, which contains information about all access paths. The file-control-block buffer can contain up to 12 file control blocks (since up to 12 paths can be open at once), and is a 362-byte (2 + 3Ø bytes*12 paths) nonrelocatable block on the system heap (see Figure 13).

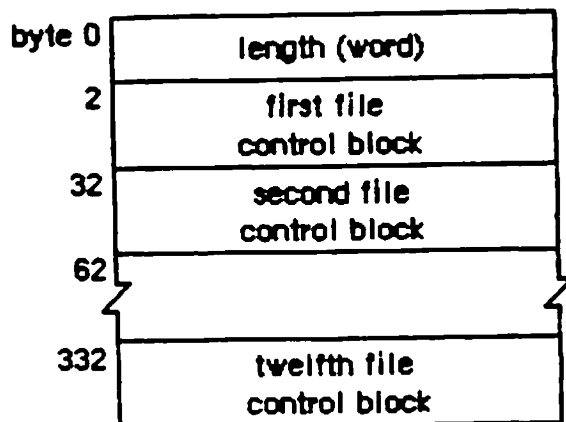


Figure 13. The File-Control-Block Buffer

You can refer to the file-control-block buffer by using the system global fcbSPtr, which points to the length word. Each file control block contains 30 bytes of information about an access path (Figure 14).

byte	field	description
0	fcbFlNum (long word)	file number
4	fcbMdRByt (byte)	flags
5	fcbTypByt (byte)	version number
6	fcbSBlk (word)	first allocation block of file
8	fcbEOF (long word)	logical end-of-file
12	fcbPLen (long word)	physical end-of-file
16	fcbCrPs (long word)	mark
20	fcbVPtr (pointer)	location of volume control block
24	fcbBfAdr (pointer)	location of access path buffer
28	fcbFlPos (word)	for internal use of File Manager

Figure 14. A File Control Block

Bit 7 of fcbMdRByt is set if the file has been changed since it was last flushed; bit 1 is set if the entry describes a resource fork; bit 0 is set if data can be written to the file.

Files Tags in Memory

As mentioned previously, logical blocks on Macintosh-initialized volumes contain 12 bytes of file tags. Normally, you'll never need to know about file tags, and the File Manager will let you read and write only the 512 bytes of standard information in each logical block. The File Manager automatically removes the file tags from each logical block it reads into memory (Figure 15) and places them at the location referred to by the system global tagData + 2. It replaces the last four bytes of the file tags with the number of the logical block from which the file was read (leaving a total of 10 bytes).

byte 0	file number (long word)	file number
4	fork type (byte)	bit 1=1 if resource fork
5	file attributes (byte)	bit 0=1 if locked
6	file sequence (word)	logical block sequence number
8	logical block number (word)	logical block

Figure 15. File Tags in Memory

(note)

> Access path buffers and volume buffers are 522 bytes long in order to contain the ten bytes of file tags and 512 bytes of standard information.

The Drive Queue

Disk drives connected to the Macintosh are opened when the system starts up, and information describing each is placed in the drive queue. The data structure of the drive queue is shown in Figure 16. QHead points to the first entry in the queue, and qTail points to the last entry in the queue.

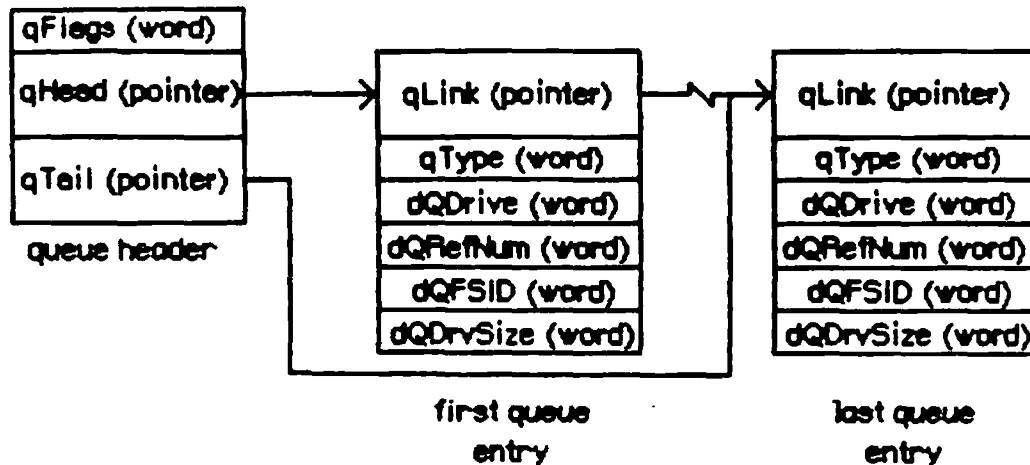


Figure 16. Drive Queue

Each queue entry contains 12 bytes of information about each drive. QLink points to the next entry in the queue; qType is ignored. QDrvNum contains the drive number of the drive on which the volume is mounted; qDRefNum contains the driver reference number of the driver controlling the device on which the volume is mounted. QFSID identifies the file system handling the volume in the drive; it's Ø for volumes handled by the File Manager, and nonzero for volumes handled by other file systems. DQDrvSize contains the number of 512-byte blocks the volumes mounted in this drive contain.

You can refer to the drive queue by using the system global drvQHdr, which points to the qFlags word. The drive queue can support any number of drives, limited only by memory space.

USING AN EXTERNAL FILE SYSTEM

The File Manager is used to access files on Macintosh-initialized volumes. If you want to access files on nonMacintosh-initialized volumes, you must write your own external file system and volume-initializing program. After the external file system has been written, it must be used in conjunction with the File Manager as described in this section.

Before any File Manager routines are called, you must place the memory location of the external file system in the system global toExtFS, and link the drive(s) accessed by your file system into the drive queue. As each nonMacintosh-initialized volume is mounted, you must create your own volume control block for each mounted volume and link each one into the volume-control-block queue. As each access path is opened, you must create your own file control block and add it to the file-control-block buffer.

All SetVol, GetVol, and GetVolInfo calls then can be handled by the File Manager via the volume-control-block queue and drive queue; external file systems needn't support these calls.

When an application calls any other File Manager routine accessing a nonMacintosh-initialized volume, the File Manager passes control to the address contained in toExtFS (if toExtFS is Ø, the File Manager returns directly to the application with an extFSErr). The external file system must then use the information in the file I/O queue to handle the call as it wishes, clear the extFSErr condition, and return control to the File Manager. Control is passed to an external file system for the following specific routine calls:

- For MountVol if the drive queue entry for the requested drive has a nonzero file-system identifier.
- For Create, Open, OpenRF, GetFileInfo, SetFileInfo, SetFilLock, RstFilLock, SetFilType, Rename, Delete, FlushVol, Eject, OffLine, and UnmountVol, if the volume control block for the requested file or volume has a nonzero file-system identifier.
- For Close, Read, Write, Allocate, GetEOF, SetEOF, GetFPos, SetFPos, and FlushFile, if the file control block for the requested file points to a volume control block with a nonzero file-system identifier.

SUMMARY OF THE FILE MANAGER

Constants

```
CONST fHasBundle = 32;
      fInvisible = 64;
```

Data Structures

```
TYPE FInfo = RECORD
               fdType:     OSType;
               fdCreator:  OSType;
               fdFlags:    INTEGER;
               fdLocation: Point;
               fdFldr:     INTEGER
             END;

     ParmBlkPtr     = ^ParamBlockRec;

     ParamBlkType   = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
                       ioLink:       QElemPtr;
                       ioType:       INTEGER;
                       ioTrap:       INTEGER;
                       ioCmdAddr:    Ptr;
                       ioCompletion: ProcPtr;
                       ioResult:     INTEGER;
                       ioNamePtr:    OSStrPtr;
                       ioVRefNum:    INTEGER;
                     CASE ParamBlkType OF
                       ioParam:
                         (ioRefNum:     INTEGER;
                          ioVersNum:    SignedByte;
                          ioPermssn:    SignedByte;
                          ioMisc:       Ptr;
                          ioBuffer:     Ptr;
                          ioReqCount:   LongInt;
                          ioActCount:   LongInt;
                          ioPosMode:    INTEGER;
                          ioPosOffset   LongInt);
                       fileParam:
                         (ioFRefNum:    INTEGER;
                          ioCVersNum:   SignedByte;
                          filler1:      SignedByte;
                          ioFDirIndex:  INTEGER;
                          ioFlAttrib:   SignedByte;
                          ioFlVersNum:  SignedByte;
                          ioFlFndrInfo: FInfo;
                          ioFlNum:      LongInt;
```

```
                ioFlStBlk:    INTEGER;
                ioFlLgLen:    LongInt;
                ioFlPyLen:    LongInt;
                ioFlRStBlk    INTEGER;
                ioFlRLgLen    LongInt;
                ioFlRPyLen    LongInt;
                ioFlCrDat     LongInt;
                ioFlMdDat     LongInt);
            volumeParam:
               (filler2:     LongInt;
                ioVolIndex:  INTEGER;
                ioVCrDate:   LongInt;
                ioVLsBkUp:   LongInt;
                ioVAtrb:     INTEGER;
                ioVNmFls:    INTEGER;
                ioVDirSt:    INTEGER;
                ioVBlLn:     INTEGER;
                ioVNmAlBlks: INTEGER;
                ioVAlBlkSiz: LongInt;
                ioVClpSiz:   LongInt;
                ioAlBlSt:    INTEGER;
                ioVNxtFNum:  LongInt;
                ioVFrBlk:    INTEGER);
            cntrlParam:
               {used by Device Manager}
         END;
```

High-Level Routines

Accessing Volumes

```
FUNCTION GetVInfo    (drvNum: INTEGER; VAR volName: OSStrPtr; VAR
                      vRefNum: INTEGER; VAR freeBytes: LongInt) :
                      OSErr;
FUNCTION GetVol      (volName: OSStrPtr; VAR vRefNum: INTEGER) :
                      OSErr;
FUNCTION SetVol      (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;
FUNCTION FlushVol    (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;
FUNCTION UnmountVol  (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;
FUNCTION Eject       (volName: OSStrPtr; vRefNum: INTEGER) : OSErr;
```

Changing File Contents

```
FUNCTION Create    (fileName: OSStr255; versNum: SignedByte; vRefNum:
                    INTEGER; creator: OSType; fileType: OSType) :
                    OSErr;
FUNCTION FSOpen    (fileName: OSStr255; versNum: SignedByte; vRefNum:
                    INTEGER; VAR refNum: INTEGER) :  OSErr;
FUNCTION FSRead    (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                    : OSErr;
```

```
FUNCTION FSWrite  (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                   : OSErr;
FUNCTION GetFPos  (refNum: INTEGER; VAR filePos: LongInt) : OSErr;
FUNCTION SetFPos  (refNum: INTEGER; posMode: INTEGER; posOff: LongInt)
                   : OSErr;
FUNCTION GetEOF   (refNum: INTEGER; VAR logEOF: LongInt) : OSErr;
FUNCTION SetEOF   (refNum: INTEGER; logEOF: LongInt) : OSErr;
FUNCTION Allocate (refNum: INTEGER; VAR count: LongInt) : OSErr;
FUNCTION FSClose  (refNum: INTEGER) : OSErr;
```

Changing Information About Files

```
FUNCTION GetFInfo (fileName: OSStr255; vRefNum: INTEGER; VAR
                   fndrInfo: FInfo) : OSErr;
FUNCTION SetFInfo (fileName: OSStr255; vRefNum: INTEGER; fndrInfo:
                   FInfo) : OSErr;
FUNCTION SetFLock (fileName: OSStr255; vRefNum: INTEGER) : OSErr;
FUNCTION RstFLock (fileName: OSStr255; vRefNum: INTEGER) : OSErr;
FUNCTION Rename   (oldName: OSStr255; vRefNum: INTEGER; newName:
                   OSStr255) : OSErr;
FUNCTION FSDelete (fileName: OSStr255; vRefNum: INTEGER) : OSErr;
```

Low-Level Routines

Initialization

```
PROCEDURE InitQueue;
```

Accessing Volumes

```
FUNCTION PBMountVol   (paramBlock: ParmBlkPtr) : OSErr;
FUNCTION PBGetVolInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetVol     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetVol     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBFlushVol   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr) : OSErr;
FUNCTION PBOffLine    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBEject      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Changing File Contents

```
FUNCTION PBCreate    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBOpen      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBOpenRF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRead      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBWrite     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetFPos   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFPos   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

```
FUNCTION PBGetEOF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetEOF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBAllocate  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBFlushFile (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBClose     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Changing Information About Files

```
FUNCTION PBGetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRstFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFVers (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRename   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBDelete   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Assembly-Language Information

Constants

```
ioQElSize      .EQU   50   ;I/O parameter block size
ioFQElSize     .EQU   80   ;file information parameter block size
ioVQElSize     .EQU   64   ;volume information parameter block size
fsQType        .EQU    5   ;I/O request queue entry type
fHasBundle     .EQU    5   ;file has a bundle
fInvisible     .EQU    6   ;file is invisible
```

Structure of Information Used by the Finder

```
fdType         Type of file
fdCreator      Creating program
fdFlags        Flags
fdLocation     File's location in folder
fdFldr         Window containing the file
```

Standard Parameter Block Data Structure

```
ioLink         Next queue entry
ioType         Always fsQType
ioTrap         Routine trap
ioCmdAddr      Routine address
ioCompletion   Completion routine
ioResult       Result code
ioFileName     File name (and possibly volume name too)
ioVNPtr        Volume name
ioVRefNum      Volume reference number
ioDrvNum       Drive number
```

I/O Parameter Block Data Structure

ioRefNum	Path reference number
ioFileType	Version number
ioPermssn	Read/write permission
ioNewName	New file or volume name for Rename
ioLEOF	Logical end-of-file for SetEOF
ioOwnBuf	Access path buffer
ioNewType	New version number for SetFilType
ioBuffer	Data buffer
ioReqCount	Requested number of bytes
ioActCount	Actual number of bytes
ioPosMode	Newline character and type of positioning operation
ioPosOffset	Size of positioning offset

File Information Parameter Block Data Structure

ioRefNum	Path reference number
ioFileType	Version number
ioFDirIndex	File directory index
ioFlAttrib	File attributes
ioFFlType	Version number
ioFlUsrWds	Information used by the Finder
ioFFlNum	File number
ioFlStBlk	First allocation block of data fork
ioFlLgLen	Logical end-of-fork of data fork
ioFlPyLen	Physical end-of-fork of data fork
ioFlRStBlk	First allocation block of resource fork
ioFlRLgLen	Logical end-of-fork of resource fork
ioFlRPyLen	Physical end-of-fork of resource fork
ioFlCrDat	Date and time file was created
ioFlMdDat	Date and time file was last modified

Volume Information Parameter Block Data Structure

ioVolIndex	Volume index number
ioVCrDate	Date and time volume was initialized
ioVLsBkUp	Date and time of last volume backup
ioVAtrb	Bit 15=1 if volume is locked
ioVNmFls	Number of files in file directory
ioVDirSt	First block of file directory
ioVBlLn	Number of blocks in file directory
ioVNmAlBlks	Number of allocation blocks on volume
ioVAlBlkSiz	Number of bytes per allocation block
ioVClpSiz	Number of bytes to allocate
ioAlBlSt	First block in volume block map
ioVNxtFNum	Next free file number
ioVFrBlk	Number of free allocation blocks

Macro Names

Routine name	Macro name
InitQueue	_InitQueue
PBMountVol	_MountVol
PBGetVolInfo	_GetVolInfo
PBGetVol	_GetVol
PBSetVol	_SetVol
PBFlshVol	_FlushVol
PBUnmountVol	_UnmountVol
PBOffLine	_OffLine
PBEject	_Eject
PBCreate	_Create
PBOpen	_Open
PBOpenRF	_OpenRF
PBRead	_Read
PBWrite	_Write
PBGetFPos	_GetFPos
PBSetFPos	_SetFPos
PBGetEOF	_GetEOF
PBSetEOF	_SetEOF
PBAllocate	_Allocate
PBFlshFile	_FlushFile
PBClose	_Close
PBGetFInfo	_GetFileInfo
PBSetFInfo	_SetFileInfo
PBSetFLock	_SetFilLock
PBRstFLock	_RstFilLock
PBSetFVers	_SetFilType
PBRename	_Rename
PBDelete	_Delete

System Globals

Name	Size	Contents
fsQHdr	4 bytes	Pointer to I/O request queue
vcbQHdr	4 bytes	Pointer to volume-control-block queue
defVCBPtr	4 bytes	Pointer to default volume control block
fcbSPtr	4 bytes	Pointer to file-control-block buffer
tagData + 2	4 bytes	Location of file tags
drvQHdr	4 bytes	Pointer to drive queue
toExtFS	4 bytes	Pointer to external file system

Result Codes

Name	Value	Meaning
badMDBErr	-60	Master directory block is bad; must reinitialize volume
bdNamErr	-37	Bad file name or volume name (perhaps zero-length)
dirFulErr	-33	File directory full

dskFulErr	-34	All allocation blocks on the volume are full
dupFNErr	-48	A file with the specified name already exists
eofErr	-39	Logical end-of-file reached during read operation
extFSErr	-58	External file system; file-system identifier is nonzero, or path reference number is greater than 1Ø24
fBsyErr	-47	One or more files are open
fLckdErr	-45	File locked
fnfErr	-43	File not found
fnOpnErr	-38	File not open
fsRnErr	-59	Problem during Rename
ioErr	-36	Disk I/O error
mFulErr	-41	System heap is full
noErr	Ø	No error
nsDrvErr	-56	Specified drive number doesn't match any number in the drive queue
noMacDskErr	-57	Volume lacks Macintosh-format directory
nsvErr	-35	Specified volume doesn't exist
opWrErr	-49	The read/write permission of only one access path to a file can allow writing
paramErr	-5Ø	Parameters don't specify an existing volume, and there's no default volume
permErr	-54	Read/write permission doesn't allow writing
posErr	-4Ø	Attempted to position before start of file
rfNumErr	-51	Reference number specifies nonexistent access path
tmfoErr	-42	Only 12 files can be open simultaneously
volOffLinErr	-53	Volume not on-line
volOnLinErr	-55	Volume specified is already mounted and on-line
vLckdErr	-46	Volume is locked by a software flag
wrPermErr	-61	Read/write permission or open permission doesn't allow writing
wPrErr	-44	Volume is locked by a hardware setting

GLOSSARY

access path: A description of the route that the File Manager follows to access a file; created when a file is opened.

access path buffer: Memory used by the File Manager to transfer data between an application and a file.

allocation block: Volume space composed of an integral number of logical blocks.

asynchronous execution: During asynchronous execution of a File Manager routine, the calling application is free to perform other tasks.

block map: See volume allocation block map.

closed file: A file without an access path. Closed files cannot be read from or written to.

completion routine: Any application-defined code to be executed when an asynchronous call to a File Manager routine is completed.

data buffer: Heap space containing information to be written to a file from an application, or read from a file to an application.

data fork: The part of a file that contains data accessed via the File Manager.

default volume: A volume that will receive I/O during a File Manager routine call, whenever no other volume is specified.

drive number: A number used to identify a drive. The internal drive is number 1, and the external drive is number 2.

drive queue: A list of disk drives connected to the Macintosh.

end-of-file: See logical end-of-file or physical end-of-file.

file: A named, ordered sequence of bytes; a principal means by which data is stored and transmitted on the Macintosh.

file directory: The part of a volume that contains descriptions and locations of all the files on the volume.

file I/O queue: A queue containing parameter blocks for all I/O requests.

file name: A sequence of up to 255 characters that identifies a file.

file number: A unique number assigned to a file, which the File Manager uses to distinguish it from other files on the volume. A file number specifies the entry of the file in a file directory.

file tags: Information associated with each logical block, designed to allow reconstruction of files on a volume whose directory or other file-access information has been destroyed.

fork: One of the two parts of a file; see data fork and resource fork.

file control block: 3Ø bytes of system heap space in a file-control-block buffer containing information about an access path.

file-control-block buffer: A 362-byte nonrelocatable block containing one file control block for each access path.

format a volume: To write information on the volume that will be read by the Disk Driver.

I/O request: A request for input from or output to a file; caused by calling a File Manager routine asynchronously.

locked file: A file that cannot be written to or deleted.

locked volume: A volume that cannot be written to or renamed. Volumes can be locked by either a software flag or a hardware setting.

logical block: 512 consecutive bytes on a volume or in memory.

logical end-of-file: The position of the last byte in a file; equal to the actual number of bytes in the file.

mark: The position of the next byte in a file that will be read or written.

master directory block: Part of the data structure of a volume; contains the volume information and the first 448 bytes of the block map.

mounted volume: A volume that previously was inserted into a disk drive and had descriptive information read from it by the File Manager.

newline character: Any ASCII character, but usually Return (ASCII code $ØD), that indicates the end of a sequence of bytes.

newline mode: A mode of reading data where the end of the data is indicated by a newline character (and not by a specific byte count).

off-line volume: A mounted volume with all but 94 bytes of its descriptive information deallocated.

on-line volume: A mounted volume with its volume buffer and descriptive information contained in memory.

open file: A file with an access path. Open files can be read from and written to.

open permission: Information about a file that indicates whether the file can be read from, written to, or both.

parameter block: Memory space used to transfer information between applications and the File Manager.

path reference number: A number that uniquely identifies an individual access path; assigned when the access path is created.

physical end-of-file: The position of one byte past the last allocation block of a file; equal to one more than the maximum number of bytes the file can contain.

read/write permission: Information associated with an access path that indicates whether the file can be read from, written to, both read from and written to, or whatever the file's open permission allows.

resource fork: The part of a file that contains the resources used by an application (such as menus, fonts, and icons) and also the application code itself; usually accessed via the Resource Manager.

synchronous execution: During synchronous execution of a File Manager routine, the calling application must wait until the routine is completed, and isn't free to perform any other task.

unmounted volume: A volume that hasn't been inserted into a disk drive and had descriptive information read from it, or a volume that previously was mounted and has since had its memory space released..

version number: One byte used to distinguish between files with the same name.

volume: A piece of storage medium formatted to contain files; usually a disk or part of a disk. The 3 1/2-inch Macintosh disks are one volume.

volume allocation block map: A list of 12-bit entries, one for each allocation block, that indicate whether the block is currently allocated to a file, whether it's free for use, or which block is next in the file. Block maps exist both on volumes and in memory.

volume attributes: Information contained on volumes and in memory indicating whether the volume is locked, has one or more files open (in memory only), and whether the volume control block matches the volume information (in memory only).

volume buffer: Memory used initially to load the master directory block; used thereafter for reading from files that are opened without their own access path buffer.

volume control block: A 90-byte nonrelocatable block that contains volume-specific information, including the first 64 bytes of the master directory block.

volume-control-block queue: A list of the volume control blocks for all mounted volumes.

volume index: A number identifying a mounted volume listed in the volume-control-block queue. The first volume in the queue has an index number of 1, and so on.

volume information: Volume-specific information contained on a volume; includes the volume name, number of files on the volume, and so on.

volume name: A sequence of up to 27 printing characters that identifies a volume; always followed by a colon (:) to distinguish it from a file name.

volume reference number: A unique number assigned to a volume as it's mounted, used to refer to the volume.

MSG#:B41194
IN#: 111
TO: MAC
FROM: SUPT MAC
SENT: 30 NOV 83 16:17:25
READ: 05 DEC 83 09:59:19

FILE MENU AND FILING COMMANDS

The File menus should read:

Application	Finder
New	Open
Open...	Duplicate
Close	Get Info
Save	Put Back
Save As...	
Revert to Saved	Close
Page Setup...	Close All
Print...	Print
< your items here>	
Quit	Eject

New... in a one-document application, is disabled while a document is open. When chosen, opens a new document window with the title "Untitled". Get the word "Untitled" from the System Resource file.

Open... brings up the GetFile dialog showing the contents of the disk (default to first on-line volume in the volume queue, usually the boot volume). User can use the Drive and Eject buttons to switch disks or drives. The disk directory shows only documents that the current application can open (the application passes a type mask, or can install itself in a filterProc to select which names to display). The user can select one and only one document from the list. Pressing "Open" attempts to load that document (the GetFile dialog will check to see that it's there, readable, the right kind, etc.) If your application has trouble loading the selected file, it should alert the user and cancel the command. It should not automatically return to the GetFile dialog.

Place the name of the opened file in the title bar of the document window. Record the volume number and version byte to be used in saving the document.

Open: is disabled if no more documents can be opened at the moment. The user must manually close an (or the) open document before opening a new one.

Save: attempts to save the current document with the same name, volume number, and version number as given when it was opened. If the document name is "Untitled", Save drops into Save As.

Save As... calls the PutFile dialog, prompting "Save document as:" with the current name as the default (unless the current name is "Untitled", in which case the default is a null string). The default volume is the first on-line volume in the volume queue (usually the boot volume), not the volume the file came from (because it may be off-line). When the user clicks Save, PutFile verifies that the file is writable, and does overwrite warnings. If you get an error while writing, or the disk is full, etc., loop back to the PutFile dialog until a save is successful, or cancelled.

Once the file is written, change the document's name in the title bar to that of the file written. Remember the volume number and version byte for the next Save command.

Close: If the frontmost window is a desk accessory, a modeless dialog, or an accessory window, Close closes that window. If the frontmost window is a document window, Close does an implicit Save before closing the window. Most applications should be able to function without an open document window (so the user can open another document, use desk accessories, etc. without leaving the application); those that can't should either force an Open or Quit after closing the document window.

Page Setup: brings up the first printing dialog, with document-specific information that should be saved on disk with the documents. This includes page size and orientation, and any other information desired by the application.

Print... brings up the printing dialog for the configured printer. Settings stick to the printer. See Owen for details of implementation.

Revert to Saved: confirms that the user really wants to revert to the saved copy. If OK, it then confirms that the old copy exists and is OK before dumping the current document and loading the old copy. The name remains the same.

Quit... confirms that the user really wants to Quit. If OK, then does an implicit Close (and Save, if dirty) of all open documents, followed by closing all other windows. It then returns to the Finder.

An implicit Save begins with checking to see if the document is dirty; the Save is skipped if it isn't. If the document is dirty, the user is asked to confirm whether to save it or not. Pressing Don't Save skips the save; pressing OK drops into the Save code.

OTHER FILING ISSUES

Changing Volumes: A Drive button on the GetFile and PutFile dialogs allows the user to cycle through the mounted volumes, showing the disk names (and, in Get, the contents). Any resulting filename is prefixed with the volume name (unless the user typed one in). This does not set the working volume. Drive does not appear on one-drive systems, and is disabled on two-drive systems with only one volume on-line.

Ejecting Disks: An Eject button on the GetFile and PutFile dialogs allows the user to eject the current volume. In the GetFile dialog, Eject cycles to the next volume, if any; if there's no next volume, the volume name and directory go blank and remain blank until another disk is inserted.

Remounting Disks: When your application gets a Disk Remount event (event bit 7) with an I/O error code, trap to package 3, which will allow the user to initialize the disk, or eject it if it's a mistake. NOTE that you should do this on every remount event with an I/O error, even during modey dialogs; be sure to check for remount errors in your modey dialog filterProcs.

Save: is an optimization of Save As to reduce button clicks, and also to work around the volume name ambiguity in saving to an off-line disk. Although we strongly recommend you support it, it is optional.

Revert to Saved: is an optimization for a "global Undo"; the user can just do a close and open. We recommend it for its clarity and speed, but it is optional.

To: All Developers
From: Cary Clark Re: Above Note

Hard Copy of the above information will be included in the next mailing.

Let me know if the info is sufficiently clear.

END/CRC

MACINTOSH USER EDUCATION

INSIDE MACINTOSH: A ROAD MAP

/ROAD.MAP/ROAD

See Also:
Pascal Reference Manual for the Lisa
Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Menu Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
CoreEdit: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Scrap Manager: A Programer's Guide
The Toolbox Utilities: A Programmer's Guide
The Memory Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application
Index to Technical Documentation

Modification History:

	First Draft (ROM 4.4)	C. Rose	8/8/83
	Second Draft (ROM 7)	C. Rose	12/22/83

ABSTRACT

This manual introduces you to the "inside" of Macintosh: the Operating System and other routines that your Macintosh application program will call. It will help you figure out which software you need to learn more about and how to proceed with the rest of the technical documentation.

Summary of significant changes and additions since last version:

- The Toolbox overview has been rewritten, and the Operating System overview has been added.

- "About Using Assembly Language" has been removed; it will be replaced by other documentation.

- "Where to Go From Here" has been updated.

TABLE OF CONTENTS

3 About This Manual
3 General Overview
5 About the User Interface Toolbox
7 About the Operating System
9 Where to Go From Here
11 Glossary

Copyright (c) 1983 Apple Computer, Inc. All rights reserved.
Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual introduces you to the "inside" of Macintosh: the Operating System, the User Interface Toolbox, and other routines that your application program may call. It will help you figure out which software you need to learn more about and how to proceed with the rest of the technical documentation. *** Eventually it will be an introductory chapter in a comprehensive manual that describes everything in detail. ***

You should already be familiar with the Macintosh User Interface Guidelines. All Macintosh programmers should follow these guidelines to ensure that the end user is presented with a consistent interface. It would also be helpful for you to be familiar with an existing Macintosh application.

This manual begins with a general overview of the software your application program will use, followed by individual overviews of the User Interface Toolbox and the Operating System. Following these overviews is a section that tells you how to proceed with reading the rest of the Toolbox and Operating System documentation. Finally, there's a glossary of terms used in this manual.

GENERAL OVERVIEW

The routines available for use in Macintosh application programs are divided into functional units, many of which are called "managers" of the application feature that they support. As shown in Figure 1 on the following page, most units are part of either the Operating System or the User Interface Toolbox and are in the Macintosh ROM.

The Operating System is at the lowest level; it does basic tasks such as interrupt handling, memory management, and I/O. The User Interface Toolbox is a level above the Operating System; it exists to help you implement the standard Macintosh user interface in your application. The Toolbox calls the Operating System when necessary to do low-level operations, and you'll also call the Operating System directly yourself.

Other software is available for performing specialized operations that aren't integral to the user interface but may be useful to some applications. This includes routines for doing printing and floating-point arithmetic. Such software isn't located in the Macintosh ROM, nor are certain special-purpose Toolbox units (such as CoreEdit, for doing sophisticated text editing). The entire Operating System and all the commonly used Toolbox units are in ROM.

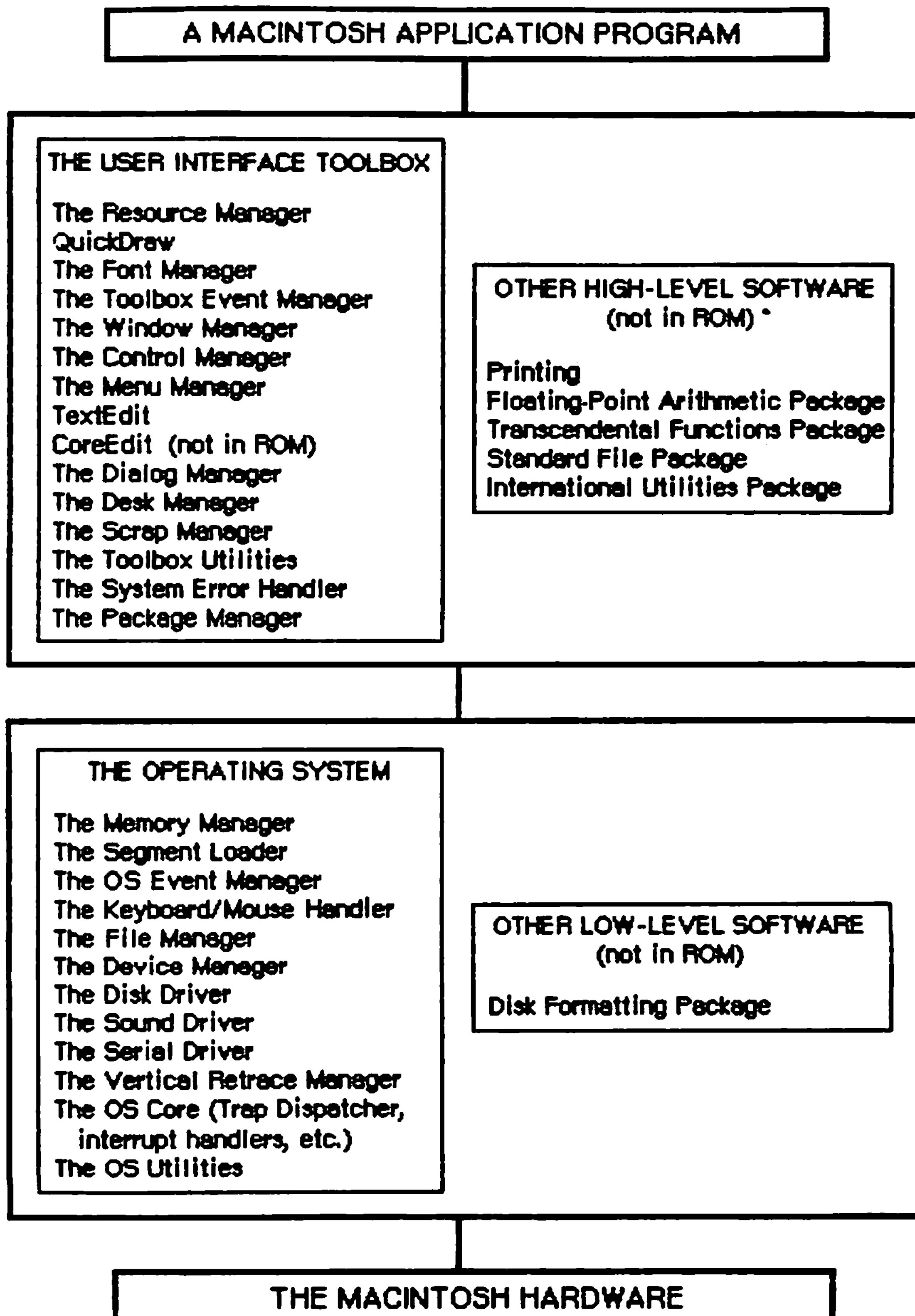


Figure 1. Overview

Macintosh applications can be written most easily in Pascal, since all units have a Pascal interface *** or will eventually ***. For greater efficiency, however, you may want to use assembly language or a combination of Pascal and assembly language. *** Currently you must develop your application on a Lisa computer and convert it to a Macintosh disk before trying it out. Eventually development will be possible on the Macintosh itself. ***

ABOUT THE USER INTERFACE TOOLBOX

The Macintosh User Interface Toolbox provides a simple means of constructing application programs that conform to the Macintosh User Interface Guidelines. By offering a common set of routines that every application calls to implement the user interface, the Toolbox not only ensures consistency but also helps reduce the application's code size and development time. At the same time, it allows a great deal of flexibility: an application can use its own code instead of a Toolbox call wherever appropriate, and can define its own types of windows, menus, controls, and desk accessories.

Figure 2 shows the Toolbox units in rough order of their level, from the lowest level at the bottom to the highest level at the top. There are many interconnections between these units; the lower-level ones are in many cases called by those at the higher levels.

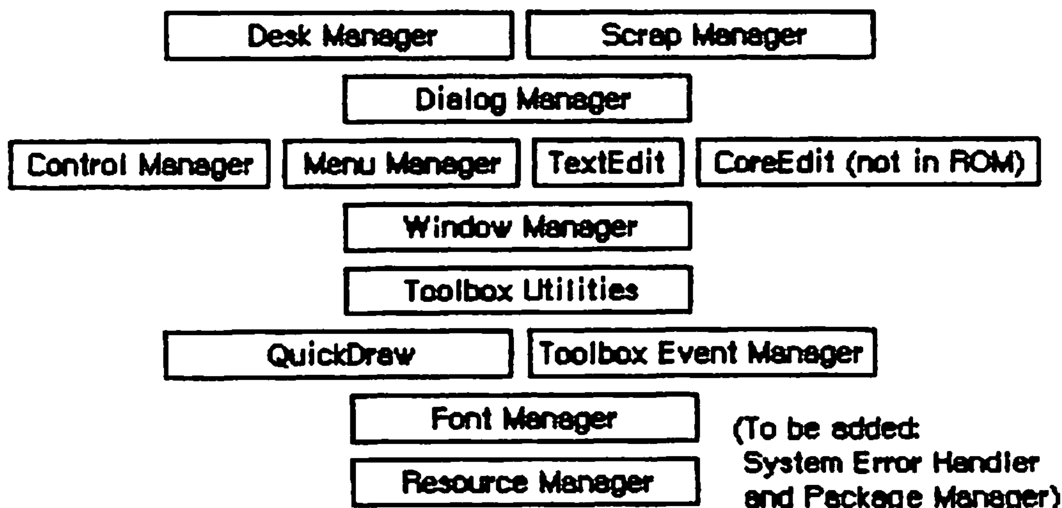


Figure 2. Toolbox Units

To keep the data of an application separate from its code, making the data easier to modify and easier to share among applications, the Toolbox includes the <u>Resource Manager</u>. The Resource Manager lets you, for example, store menus separately from your code so that they can be edited or translated without requiring recompilation of the code. It also allows you to get standard data, such as the wristwatch graphic that means "wait", from a shared system file. When you call other Toolbox units that need access to the data, they call the Resource Manager. Although most applications never need to call the Resource

Manager directly, an understanding of the concepts behind it is essential.

Graphics are an important part of every Macintosh application. All graphic operations on the Macintosh are performed by the QuickDraw unit. To draw something on the screen, you'll often call one of the other Toolbox units, but that unit will in turn call QuickDraw. You'll also call QuickDraw directly, usually to draw inside a window. QuickDraw's underlying concepts, like those of the Resource Manager, are important for you to understand.

Graphics include text as well as pictures. To draw text, QuickDraw calls the Font Manager, which does the background work necessary to make a variety of character fonts available in various sizes and styles. Unless an application includes a font menu, it usually need not be concerned with the Font Manager.

An application decides what to do from moment to moment by examining input from the user, in the form of mouse and keyboard actions. It learns of such actions by repeatedly calling the Toolbox Event Manager (which in turn calls another, lower-level Event Manager in the Operating System). The Toolbox Event Manager also reports occurrences within the application that may require a response, such as when a window that was overlapped becomes exposed and needs to be redrawn.

All information presented by a standard Macintosh application appears in windows. To create windows, activate them, move them, resize them, or close them, you'll call the Window Manager. It keeps track of overlapping windows, so you can manipulate windows without concern for how they overlap. The Window Manager, for example, tells the Toolbox Event Manager when to inform your application that a window has to be redrawn. Also, when the user presses the mouse button, you call the Window Manager to learn which part of which window it was pressed in, if any, or whether it was pressed in the menu bar or a desk accessory.

Any window may contain controls, such as buttons, check boxes, and scroll bars. You create and manipulate controls with the Control Manager. When you learn from the Window Manager that the user pressed the mouse button inside a window containing controls, you call the Control Manager to find out which control it was pressed in, if any.

A common place for the user to press the mouse button is, of course, in the menu bar. You set up menus in the menu bar by calling the Menu Manager. When the user gives a command, either from a menu with the mouse or from the keyboard with the Command key, you call the Menu Manager to find out which command was given.

To accept text typed by the user and allow the standard editing capabilities, such as cutting and pasting within a document via the Clipboard, your application can call either TextEdit or CoreEdit. TextEdit is especially easy to use but doesn't support advanced editing and formatting features such as fully justified text, tabbing, or recognition of word boundaries during cutting and pasting; for these, you'll have to use CoreEdit. Bear in mind, however, that CoreEdit is

not in the Macintosh ROM; instead, it occupies over 6K of your application's available memory.

When an application needs more information from the user about a command, it presents a dialog box. In case of errors or potentially dangerous situations, it gives the user an alert, in the form of an alert box or sound from the Macintosh's speaker (or both). To create and present dialogs and alerts, and find out the user's responses to them, you call the Dialog Manager.

Every Macintosh application should support the use of desk accessories. The user opens desk accessories through the Apple menu, which you set up by calling the Menu Manager. When you learn that the user has pressed the mouse button in a desk accessory, you pass that information on to the accessory by calling the Desk Manager. The Desk Manager also includes routines that you must call to ensure that desk accessories behave properly.

As mentioned above, you can use TextEdit or CoreEdit to implement the standard text editing capability of cutting and pasting via the Clipboard in your application. However, to extend the use of the Clipboard to allow cutting and pasting between your application and another application or a desk accessory, you need to call the Scrap Manager.

Finally, some generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits may be performed with the Toolbox Utilities.

*** To be added: System Error Handler, Package Manager, and other high-level software ***

ABOUT THE OPERATING SYSTEM

The Macintosh Operating System provides the low-level support that applications need in order to use the Macintosh hardware. As the Toolbox is your program's interface to the user, the Operating System is its interface to the Macintosh.

The Memory Manager dynamically allocates and releases memory for use by applications and by the other parts of the Operating System. Most of the memory that your program uses is in an area called the heap; the code of the program itself occupies space in the heap. Memory space in the heap must be obtained from the Memory Manager.

The Segment Loader is the part of the Operating System that loads the program code into memory to be executed. Your program can be loaded all at once as a single unit, or you can divide it up into dynamically loaded segments to economize on memory usage.

Low-level, hardware-related events such as mouse-button presses and keystrokes are reported by the Operating System Event Manager. (The

Toolbox Event Manager then passes them along to the application, along with higher-level, software-generated events added at the Toolbox level.) The Operating System Event Manager learns of mouse and keyboard actions in particular from the Keyboard/Mouse Handler. Your program will ordinarily deal only with the Toolbox Event Manager and rarely call the Operating System Event Manager or the Keyboard/Mouse Handler directly.

File I/O is supported by the File Manager, and device I/O by the Device Manager. The task of making the various types of devices present the same interface to the application is performed by specialized device drivers. The Operating System includes three built-in drivers:

- The Disk Driver controls data storage and retrieval on 400K-byte 3 1/2-inch disks.

- The Sound Driver controls sound generation, including music composed of four simultaneous tones.

- The Serial Driver reads and writes asynchronous data through the two serial ports, providing communication between applications and serial peripheral devices such as a modem or printer.

Other drivers can be added independently or built on the existing drivers. For example, a printer driver can be built on the Serial Driver or a music driver built on the Sound Driver.

The Macintosh video circuitry generates a vertical retrace interrupt (also known as the vertical blanking or VBL interrupt) sixty times a second while the beam of the display tube returns from the bottom of the screen to the top to display the next frame. The system uses this interrupt as a convenient time to perform recurrent tasks such as checking the state of the mouse button. An application can also schedule routines to be executed at regular intervals based on this "heartbeat" of the system. The Vertical Retrace Manager handles the scheduling and execution of tasks during the vertical retrace interrupt.

At the very lowest level is the Operating System Core, which does the actual interrupt handling, initialization, and other important background work necessary to keep the Macintosh functioning. Via the Trap Dispatcher, it provides the connection between your request for a Toolbox or Operating System service and the physical code that performs that service.

Finally, there are miscellaneous Operating System Utilities for doing such things as setting the date and time or finding out the user's preferred speaker volume.

*** To be added: other low-level software (Disk Formatting Package) ***

WHERE TO GO FROM HERE

*** This section will be considerably rewritten for the final comprehensive manual. ***

The technical documentation will eventually be ordered in such a way that you can follow it if you read it sequentially. The proposed order for the documentation that's already written is given below. Before you begin, you should be familiar with Lisa Pascal, as described in the Pascal Reference Manual for the Lisa. You should also know a little bit about the Macintosh's memory management--heaps, handles, and the like. For now you can read about these in the Memory Manager manual, from "About the Memory Manager" through "Utility Data Types"; eventually there will be a separate overview of memory management.

The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Menu Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
CoreEdit: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
The Toolbox Utilities: A Programmer's Guide
The Memory Manager: A Programmer's Guide
Macintosh Operating System Reference Manual
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

(hand)
The Macintosh Operating System Reference Manual is very out-of-date, incomplete, and in a different format from the other manuals. It will eventually be completely replaced by up-to-date documentation in the usual format.

(hand)
Anything not listed above hasn't been documented yet by Macintosh User Education, although programmer's notes or other preliminary documentation may be available. Check with Macintosh Technical Support.

The individual manuals identify any special-purpose information that can possibly be skipped. Most likely you won't need to read everything in each manual and can even skip entire manuals. You should at least read the manuals on the Toolbox units that deal with the fundamental aspects of the user interface: the Resource Manager, QuickDraw, the Toolbox Event Manager, the Window Manager, and the Menu Manager. Read the other manuals if you're interested in what they discuss, which you should be able to tell from the above overviews and from the

introductions to the manuals themselves. Each manual's introduction will also tell you what you should already know before reading that manual.

The documentation is oriented toward Pascal programmers. If you want to program in assembly language, read the "Using QuickDraw from Assembly Language" section of the QuickDraw manual. (Eventually that section will be removed and there will be a separate, more detailed discussion of using assembly language.) There are also notes for assembly-language programmers throughout every manual.

Read the manual "Putting Together a Macintosh Application" when you're ready to try something out. Currently the documentation doesn't include any sample programs, but you can get some through Macintosh Technical Support in the meantime.

GLOSSARY

Control Manager: A Toolbox unit that provides routines for creating and manipulating controls (such as buttons, check boxes, and scroll bars).

CoreEdit: A Toolbox unit that handles sophisticated text editing and formatting, including fully justified text, tabbing, and recognition of word boundaries during cutting and pasting.

Desk Manager: A Toolbox unit that supports the use of desk accessories from an application.

device driver: A piece of Operating System software that controls a peripheral device and makes it present the standard interface to the application.

Device Manager: The part of the Operating System that supports device I/O.

Dialog Manager: A Toolbox unit that provides routines for implementing dialogs and alerts.

Disk Driver: The device driver that controls data storage and retrieval on 4ØØK-byte 3 1/2-inch disks.

Event Manager: See Toolbox Event Manager or Operating System Event Manager.

File Manager: The part of the Operating System that supports file I/O.

Font Manager: A Toolbox unit that supports the use of various character fonts for QuickDraw when it draws text.

heap: An area of memory in which space can be allocated and released on demand, using the Memory Manager.

Keyboard/Mouse Handler: The part of the Operating System that controls communication with the keyboard and the mouse.

Memory Manager: The part of the Operating System that dynamically allocates and releases memory space in the heap.

Menu Manager: A Toolbox unit that deals with setting up menus and letting the user choose from them.

Operating System: The lowest-level software in the Macintosh. It does basic tasks such as interrupt handling, memory management, and I/O.

Operating System Core: The part of the Operating System that does the actual interrupt handling, initialization, and other important background work necessary to keep the Macintosh functioning.

Operating System Event Manager: The part of the Operating System that reports hardware-related events such as mouse-button presses and keystrokes.

Operating System Utilities: Operating System routines that perform miscellaneous tasks such as setting the date and time or finding out the user's preferred speaker volume.

QuickDraw: The Toolbox unit that performs all graphic operations on the Macintosh screen.

resource: Data used by an application (such as menus, fonts, and icons), and also the application code itself.

Resource Manager: The Toolbox unit that reads and writes resources.

Scrap Manager: The Toolbox unit that enables cutting and pasting between applications, desk accessories, or an application and a desk accessory.

Segment Loader: The part of the Operating System that loads the code of an application into memory, either as a single unit or divided into dynamically loaded segments.

Serial Driver: The device driver that controls communication, via serial ports, between applications and serial peripheral devices.

Sound Driver: The device driver that controls sound generation in an application.

TextEdit: A Toolbox unit that supports the basic text entry and editing capabilities of a standard Macintosh application.

Toolbox: Same as User Interface Toolbox.

Toolbox Event Manager: A Toolbox unit that allows your application program to monitor the user's actions with the mouse, keyboard, and keypad.

Toolbox Utilities: A Toolbox unit that performs generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits.

Trap Dispatcher: The part of the Operating System Core that provides the connection between your request for a Toolbox or Operating System service and the physical code that performs that service.

User Interface Toolbox: A set of routines and data types that help you implement the standard Macintosh user interface in your application.

vertical retrace interrupt: An interrupt generated sixty times a second by the Macintosh video circuitry while the beam of the display tube returns from the bottom of the screen to the top; also known as the vertical blanking or VBL interrupt.

Vertical Retrace Manager: The part of the Operating System that schedules and executes tasks during the vertical retrace interrupt.

Window Manager: A Toolbox unit that provides routines for creating and manipulating windows.

MACINTOSH USER EDUCATION

The Font Manager: A Programmer's Guide /FMGR/FONT

See Also: Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide

Modification History:			
	Preliminary Draft	C. Rose	4/20/83
	First Draft (ROM 3.0)	C. Rose	4/22/83
	Second Draft (ROM 7)	B. Hacker	2/7/84

ABSTRACT

The Font Manager is the part of the Macintosh User Interface Toolbox that supports the use of various character fonts when you draw text with QuickDraw. This manual introduces you to the Font Manager and describes the routines your application can call to get font information. It also describes the data structures of fonts and discusses how the Font Manager communicates with QuickDraw.

Summary of significant changes and additions since last version:

- A new routine, SwapFont, has been documented (page 11).

- A description of the method of communication between the Font Manager, QuickDraw, and device drivers has been added (page 11).

- A section describing the format of a font, including font records, has been added (page 15).

TABLE OF CONTENTS

3 About This Manual
4 About the Font Manager
6 Font Numbers
7 Characters in a Font
7 Font Scaling
9 Using the Font Manager
9 Font Manager Routines
9 Initializing the Font Manager
10 Getting Font Information
10 Keeping Fonts in Memory
11 Advanced Routine
11 Communication Between QuickDraw and the Font Manager
15 Format of a Font
19 Font Records
22 Font Widths
22 How QuickDraw Draws Text
23 Fonts in a Resource File
25 Summary of the Font Manager
29 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

The Font Manager is the part of the Macintosh User Interface Toolbox that supports the use of various character fonts when you draw text with QuickDraw. This manual introduces you to the Font Manager and describes the routines your application can call to get font information. It also describes the data structures of fonts and discusses how the Font Manager communicates with QuickDraw. *** Eventually this will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

(hand)
This manual describes version 7 of the ROM. If you're using a different version, the Font Manager may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with:

- Resources, as described in the Resource Manager manual

- The basic concepts and structures behind QuickDraw, particularly bit images and how to draw text

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an overview of the Font Manager and what you can do with it. It then discusses the font numbers by which fonts are identified, the characters in a font, and the scaling of fonts to different sizes. Next, a section on using the Font Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of Font Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. There's a discussion of how QuickDraw and the Font Manager communicate, followed by a section that describes the format of the data structures used to define fonts, and how QuickDraw uses the data to draw characters. Next is a section that gives the exact format of fonts in a resource file.

Finally, there's a summary of the Font Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE FONT MANAGER

The main function of the Font Manager is to provide font support for QuickDraw. To the Macintosh user, font means the complete set of characters of one typeface; it doesn't include the size of the characters, and usually doesn't include any stylistic variations (such as bold and italic).

(hand)

> Usually fonts are defined in the normal style and stylistic variations are applied to them; for example, the italic style simply slants the normal characters. However, fonts may be designed to include stylistic variations in the first place.

The way you identify a font to QuickDraw or the Font Manager is with a font number. Every font also has a name (such as "New York") that's appropriate to include in a menu of available fonts.

The size of the characters, called the font size, is given in points. Here this term doesn't have the same meaning as the "point" that's an intersection of lines on the QuickDraw coordinate plane, but instead is a typographical term that stands for approximately 1/72 inch. The font size measures the distance between the ascent line of one line of text and the ascent line of the next line of single-spaced text (see Figure 1). It assumes 8Ø pixels per inch, the approximate resolution of the Macintosh screen. For example, since an Imagewriter printer has twice the resolution of the screen, high-resolution 9-point output to the printer is actually accomplished with an 18-point font.

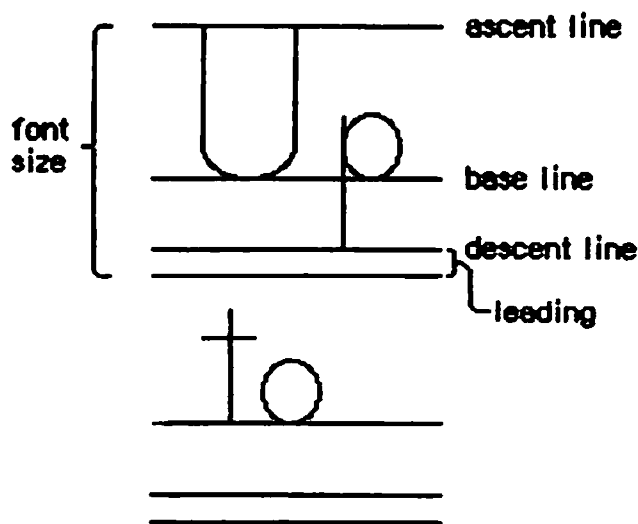


Figure 1. Font Size

(hand)

Because measurements cannot be exact on a bit-mapped output device, the actual font size may be slightly different from what it would be in normal typography.

Whenever you call a QuickDraw routine that does anything with text, QuickDraw passes the following information to the Font Manager:

- The font number.
- The character style, a set of stylistic variations. The empty set indicates normal text. (See the QuickDraw manual for details.)
- The font size. The size may range from 1 point to 127 points, but for readability should be at least 6 points.
- The horizontal and vertical scaling factors, each of which is represented by a numerator and a denominator (for example, a numerator of 2 and a denominator of 1 indicates 2-to-1 scaling in that direction).
- A Boolean value indicating whether the characters will actually be drawn or not. They will not be drawn, for example, when the QuickDraw function CharWidth is called (since it only measures characters) or when text is drawn after the pen has been hidden (such as by the HidePen procedure or the OpenPicture function, which calls HidePen).
- A number specifying the device on which the characters will be drawn or printed. The number Ø represents the Macintosh screen. The Font Manager can adapt fonts to other devices.

Given this information, the Font Manager provides QuickDraw with information describing the font and--if the characters will actually be drawn--the bits comprising the characters.

Fonts are stored as resources in resource files; the Font Manager calls the Resource Manager to read them into memory. System-defined fonts are stored in the system resource file. You may define your own fonts with the aid of the Resource Editor and include them in the system resource file so they can be shared among applications. *** (The Resource Editor doesn't yet exist, but an interim Font Editor is available from Macintosh Technical Support.) *** In special cases, you may want to store a font in an application's resource file or even in the resource file for a document. It's also possible to store only the character widths and general font information, and not the bits comprising the characters, for those cases where the characters won't actually be drawn.

A font may be stored in any number of sizes in a resource file. If a size is needed that's not available as a resource, the Font Manager scales an available size.

Fonts occupy a large amount of storage: a 12-point font typically occupies about 3K bytes, and a 24-point font, about 10K bytes; fonts for use on a high-resolution output device can take up four times as much space as that (up to a maximum of 32K bytes). Fonts normally are purgeable, which means they may be removed from the heap when space is required by the Memory Manager. If you wish, you can call a Font Manager routine to make a font temporarily unpurgeable.

There are also routines that provide information about a font. You can find out the name of a font having a particular font number, or the font number for a font having a particular name. You can also learn whether a font is available in a certain size or will have to be scaled to that size.

FONT NUMBERS

The Font Manager includes the following font numbers for identifying system-defined fonts:

```
CONST systemFont = 0;  {system font}
      applFont   = 1;  {application font}
      newYork    = 2;
      geneva     = 3;
      monaco     = 4;
      venice     = 5;
      london     = 6;
      athens     = 7;
      sanFran    = 8;
      toronto    = 9;
```

Font number 0 refers to the _system font_, so called because it's the font used by the system (for drawing menu titles and commands in menus, for example). The name of the system font is Chicago. The size of text drawn by the system in this font is fixed at 12 points (called the _system font size_).

Font number 1 represents the _application font_, which is a suitable font for general use by the application. Unlike the system font, the application font isn't a separate font with its own typeface, but is essentially a reference to another font--New York, by default. *** In the future, there will be a way for the user to change the default, perhaps through the Control Panel desk accessory. ***

Assembly-language note: The font number of the default application font is stored in parameter RAM, in the system global spFont. You can change the application font via the system global apFontID, which contains the font number of the current application font.

CHARACTERS IN A FONT

A font can consist of up to 255 distinct characters; not all characters need be defined in a single font. Figure 2 on the following page shows the standard printing characters on the Macintosh and their ASCII codes (for example, the ASCII code for "A" is 41 hexadecimal, or 65 decimal).

In addition to its maximum of 255 characters, every font contains a missing symbol that's drawn in case of a request to draw a character that's missing from the font.

FONT SCALING

The information QuickDraw passes to the Font Manager includes the font size and the scaling factors QuickDraw wants to use. The Font Manager determines the font information to return to QuickDraw by looking for the exact size needed among the sizes stored for the font. If the exact size requested isn't available, it then looks for a nearby size that it can scale.

1. It looks first for a font that's twice the size, and scales down that size if there is one.
2. If there's no font that's twice the size, it looks for a font that's half the size, and scales up that size if there is one.
3. If there's no font that's half the size, it looks for a larger size of the font, and scales down the next larger size if there is one.
4. If there's no larger size, it looks for a smaller size of the font, and scales up the closest smaller size if there is one.
5. If the font isn't available in any size at all, it uses the application font instead, scaling the font to the proper size.
6. If the application font isn't available in any size at all, it uses the system font instead, scaling the font to the proper size.

Scaling looks best when the scaled size is an even multiple of an available size.

Assembly-language note: You can use the system global fScaleDisable to defeat scaling, if desired. Normally, fScaleDisable is Ø. If you set it to a nonzero value, the Font Manager will look for the size as described above but will return the font unscaled.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1		⌘	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2		✓	"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3		◆	#	3	C	S	c	s	É	ì	£	≥	√	”		
4			$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5			%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6			&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷		
7			'	7	G	W	g	w	á	ó	ß	∑	«	◇		
8			(	8	H	X	h	x	à	ò	®	∏	»	ÿ		
9			)	9	I	Y	i	y	â	ô	©	π	…			
A			*	:	J	Z	j	z	ä	ö	™	∫	␣			
B			+	;	K	[	k	{	ã	õ	´	ª	À			
C			,	<	L	\	l	|	å	ú	¨	º	Ã			
D			–	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E			.	>	N	^	n	~	é	û	Æ	æ	Œ			
F			/	?	O	_	o		è	ü	Ø	ø	œ			

SP stands for a space.
␣ stands for a nonbreaking space, same width as numbers.

The first four characters are only in the system font (Chicago).
The shaded characters are only in the Geneva, Monaco, and system fonts.

Figure 2. Font Characters

USING THE FONT MANAGER

This section introduces you to the Font Manager routines and how they fit into the general flow of an application program. The routines themselves are described in detail in the next section.

The InitFonts procedure initializes the Font Manager; you should call it after initializing QuickDraw but before initializing the Window Manager.

You can set up a menu of fonts in your application by using the Menu Manager procedure AddResMenu (see the Menu Manager manual for details). When the user chooses a menu item from the font menu, you call the Menu Manager procedure GetItem to get the name of the corresponding font, and then the Font Manager function GetFNum to get the font number. The GetFontName function does the reverse of GetFNum: given a font ID, it returns the font name.

In a menu of font sizes in your application, you may want to let the user know which sizes the current font is available in and therefore will not require scaling. You can call the RealFont function to find out whether a font is available in a given size.

If you know you'll be using a font a lot and don't want it to be purged, you can use the SetFontLock procedure to make the font unpurgeable during that time.

Advanced programmers who want to write their own font editors or otherwise manipulate fonts can access fonts directly with the SwapFont function.

FONT MANAGER ROUTINES

This section describes all the Font Manager procedures and functions. The routines are presented in their Pascal form; for information on using them from assembly language, see the *** forthcoming *** manual _Programming Macintosh Applications in Assembly Language_.

Initializing the Font Manager

```
PROCEDURE InitFonts;
```

InitFonts initializes the Font Manager. If the system font isn't already in memory, InitFonts reads it into memory. Call this procedure once before all other Font Manager routines or any Toolbox routine that will call the Font Manager.

Assembly-language note: InitFonts also sets the current application font to the default application font (by setting the system global apFontID to the font number stored in parameter RAM in the system global spFont).

Getting Font Information

PROCEDURE GetFontName (fontNum: INTEGER; VAR theName: Str255);

GetFontName returns in theName the name of the font having the font number fontNum. If there's no such font, GetFontName returns the empty string.

Assembly-language note: The macro you invoke to call GetFontName from assembly language is named _GetFName.

PROCEDURE GetFNum (fontName: Str255; VAR theNum: INTEGER);

GetFNum returns in theNum the font number for the font having the given fontName. If there's no such font, GetFNum returns Ø.

FUNCTION RealFont (fontNum: INTEGER; size: INTEGER) : BOOLEAN;

RealFont returns TRUE if the font having the font number fontNum is available in the given size in a resource file, or FALSE if the font has to be scaled to that size.

Keeping Fonts in Memory

PROCEDURE SetFontLock (lockFlag: BOOLEAN);

SetFontLock applies to the font in which text was most recently drawn; it makes the font unpurgeable if lockFlag is TRUE or purgeable if lockFlag is FALSE. Since fonts are normally purgeable, this procedure is useful for making a font temporarily unpurgeable.

Advanced Routine

The following low-level routine will not normally be used by an application directly, but may be of interest to advanced programmers who want to bypass the QuickDraw routines that deal with text.

FUNCTION SwapFont (inRec: FMInput) : FMOutPtr;

SwapFont returns a pointer to an FMOutput record containing the size, style, and other information about an adapted version of the font requested in the given FMInput record. (FMInput and FMOutput records are explained in the following section.) SwapFont is called by QuickDraw every time a QuickDraw routine that does anything with text is used. If you want to call SwapFont yourself, you must build an FMInput record and then use the returned pointer to access the resulting FMOutput record.

COMMUNICATION BETWEEN QUICKDRAW AND THE FONT MANAGER

This section describes the data structures that allow QuickDraw and the Font Manager to exchange information. It also discusses the communication that may occur between the Font Manager and the driver of the device on which the characters are being drawn or printed. You can skip this section if you want to change fonts, character style, and font sizes by calling QuickDraw and aren't interested in the lower-level data structures and routines of the Font Manager.

Whenever you call a QuickDraw routine that does anything with text, QuickDraw requests information from the Font Manager about the characters. The Font Manager performs any necessary calculations and returns the requested information to QuickDraw. As illustrated in Figure 3, this information exchange occurs via two data structures, a font input record (type FMInput) and a font output record (type FMOutput).

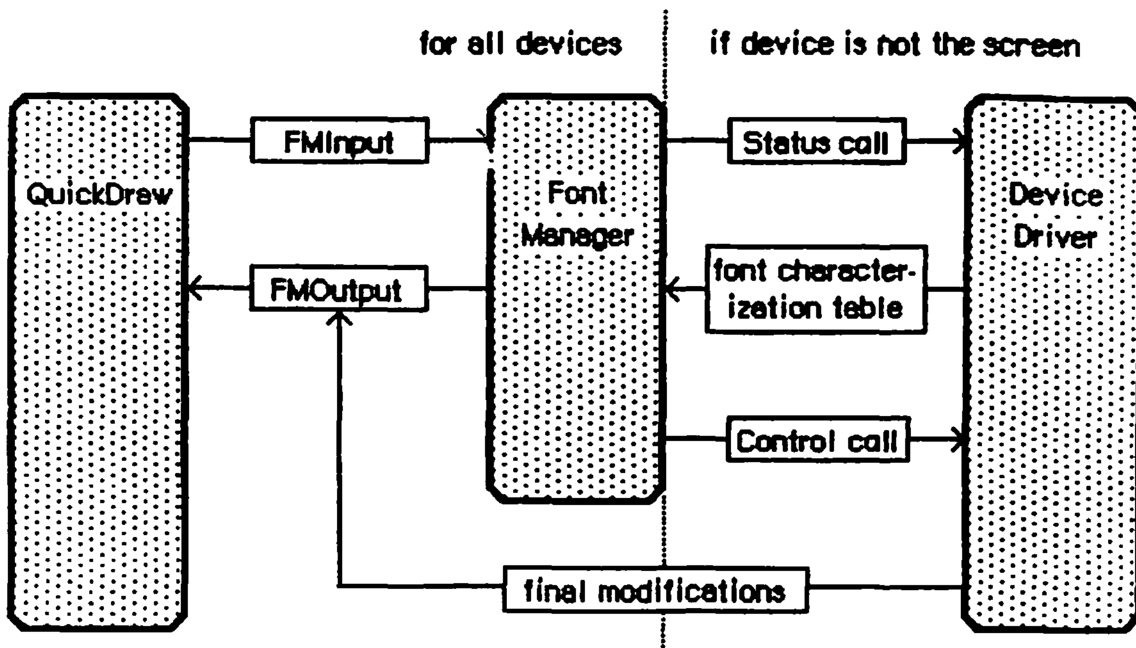


Figure 3. Communication About Fonts

First, QuickDraw passes the Font Manager a font input record:

```
TYPE FMInput = PACKED RECORD
                 family:   INTEGER;  {font number}
                 size:     INTEGER;  {font size}
                 face:     Style;    {character style}
                 needBits: BOOLEAN;  {TRUE if drawing}
                 device:   INTEGER;  {output device}
                 numer:    Point;    {numerators of scaling factors}
                 denom:    Point     {denominators of scaling factors}
               END;
```

The first three fields contain the font number, size, and character style that QuickDraw wants to use. The needBits field indicates whether the characters actually will be drawn or not. If the characters are being drawn, all of the information describing the font, including the bit image comprising the characters, will be read into memory. If the characters aren't being drawn and there's a resource consisting of only the character widths and general font information, that resource will be read instead.

As shown in Figure 4, the high-order byte of the device field contains the low-order byte of the reference number for the device driver (the high-order byte of the reference number is $FF). The low-order byte of the device field contains a device subclass, which is passed on to the device driver and may be used as a means of distinguishing different kinds of output on the same device (for example, high resolution vs. low resolution). The value Ø in the device field represents the Macintosh screen.

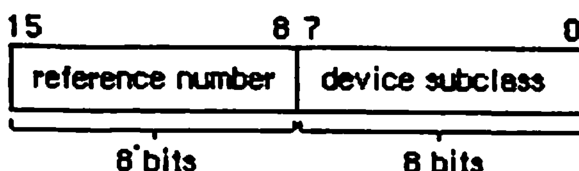


Figure 4. Device Field

The numer and denom fields contain the scaling factors to be used; numer.v divided by denom.v gives the vertical scaling, and numer.h divided by denom.h gives the horizontal scaling.

The Font Manager takes the FMInput record and asks the Resource Manager for the font. If the requested size isn't available, the Font Manager scales another size to match (as described previously).

Then, if the device field is nonzero, the Font Manager calls the device driver's status routine to get the device's font characterization table *** which will be documented in a future version of this manual ***. The Font Manager takes the information in the font characterization table and determines the optimum ascent, descent, leading, and ways of doing stylistic variations on that device, so that the highest quality printing or drawing available will be produced. It then stores this information in a font output record:

```
TYPE FMOutput = PACKED RECORD
          errNum:      INTEGER;    {not used}
          fontHandle:  Handle;     {handle to font record}
          bold:        Byte;       {bold factor}
          italic:      Byte;       {italic factor}
          ulOffset:    Byte;       {underline offset}
          ulShadow:    Byte;       {underline shadow}
          ulThick:     Byte;       {underline thickness}
          shadow:      Byte;       {shadow factor}
          extra:       SignedByte; {not used}
          ascent:      Byte;       {ascent}
          descent:     Byte;       {descent}
          widMax:      Byte;       {maximum character width}
          leading:     SignedByte; {leading}
          unused:      Byte;       {not used}
          numer:       Point;      {numerators of scaling factors}
          denom:       Point       {denominators of scaling factors}
        END;
```

ErrNum is reserved for future implementation, and is currently zero. FontHandle is a handle to the font record of the font, as described in the next section. Bold, italic, ulOffset, ulShadow, ulThick, and shadow are all fields that modify the way stylistic variations are done. Ascent, descent, widMax, and leading are the same as the fields of the FontInfo record returned by the QuickDraw procedure GetFontInfo. Numer and denom contain the scaling factors.

Just before returning this record to QuickDraw, the Font Manager calls the device driver's control routine to allow the driver to make any final modifications to the record. Finally, the font information is returned to QuickDraw via a pointer to the record, defined as follows:

```
TYPE FMOutPtr = ^FMOutput;
```

Assembly-language note: If you want to make your own assembly-language calls to the device driver's status routine, the parameter block pointed to by AØ must contain 8 in the csCode field (a word located at 26(AØ)), and the parameters (starting at 28(AØ)) must be a pointer to where the device driver should put the font characterization table, and a word containing the value of the font input record's device field. If you call the device driver's control routine, 8 is again passed in the csCode field, and the parameters must be a pointer to the font output record and a word containing the value of the device field.

FORMAT OF A FONT

This section describes the data structure that defines a font; read it if you're going to define your own fonts with the Resource Editor or write your own font editor.

Each character in a font is defined by pixels arranged in rows and columns. This pixel arrangement is called a character image; it's the image inside each of the character rectangles shown in Figure 5.

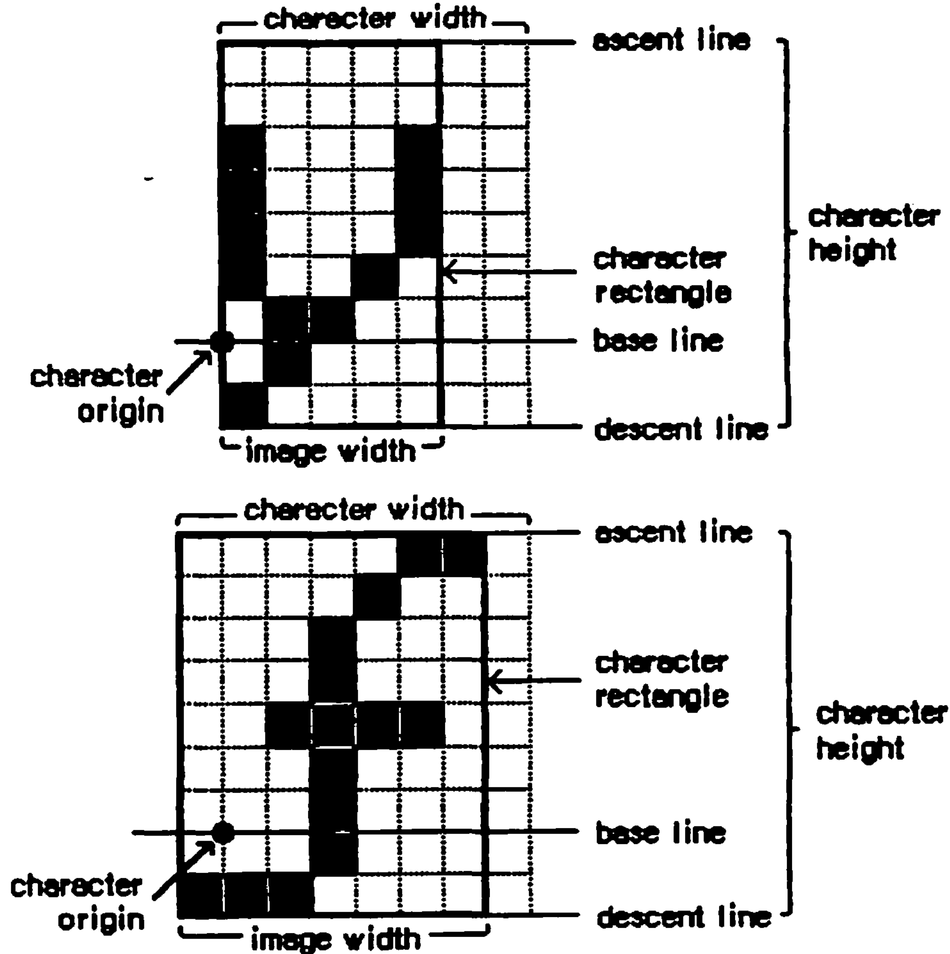


Figure 5. Character Images

The base line is a horizontal line coincident with the bottom of each character, excluding descenders. The character origin is a point on the base line used as a reference location for drawing the character. Conceptually the base line is the line that the pen is on when it starts drawing a character, and the characer origin is the point where the pen starts drawing.

The horizontal extent of a character image is called the image width. The image width may or may not include space on either side of the character; to minimize the amount of memory required to store the font, the image width should not include space. The vertical extent of the

character rectangle, the character height, is the number of pixels from the ascent line to the descent line. The character rectangle is the rectangle enclosing the character image; its sides are defined by the image width and character height.

The image width is different from the character width, which is the distance to move the pen from this character's origin to the next while drawing—in other words, the image width plus the amount of blank space to leave before the next character.

(hand)
If the character width is 0, the character that follows will be superimposed on this character (useful for accents, underscores, and so on).

(hand)
Characters whose image width is 0 (such as a space) can have a nonzero character width.

Characters in a proportional font all have character widths proportional to their image width, whereas characters in a fixed-width font all have the same character width.

Characters can kern; that is, they can overlap adjacent characters. The first character in Figure 5 above doesn't kern, but the second one kerns left.

Every font has a bit image that contains a complete sequence of all its character images (see Figure 6 on the following page). The number of rows in the bit image is equivalent to the character height. The character images in the font are stored in the bit image as though the characters were laid out horizontally (in ASCII order, by convention) along a common base line.

The bit image doesn't have to contain a character image for every character in the font. Instead, any characters missing from the font are omitted from the bit image, and a missing symbol is drawn instead. The missing symbol is stored in the bit image after all the other character images.

(eye)
Every font must have a missing symbol.

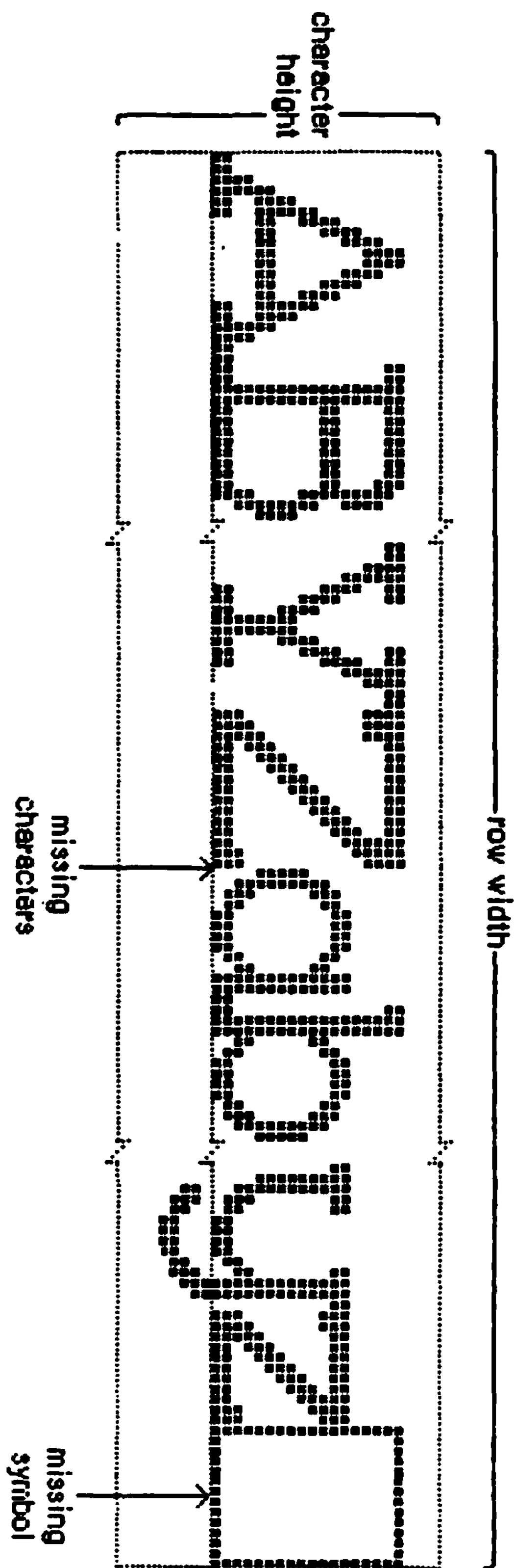


Figure 6. Partial Bit Image for a Font

In addition to the terms used to describe individual characters, there are terms describing features of the font as a whole (see Figure 7).

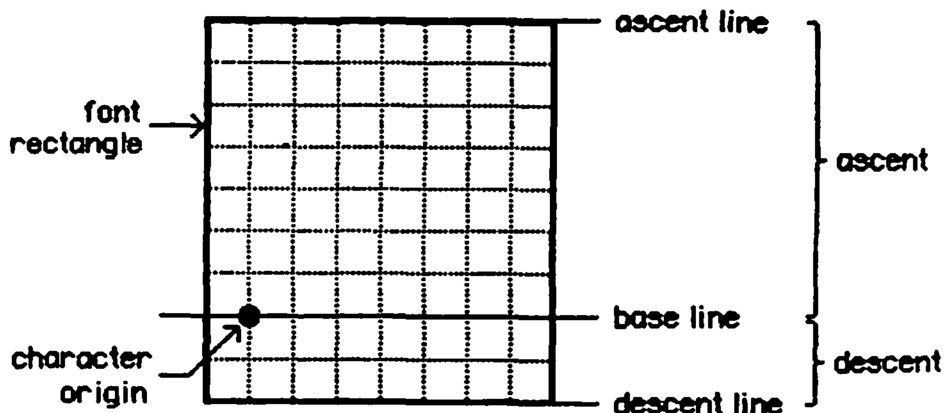


Figure 7. Features of Fonts

The font rectangle is somewhat analogous to a character rectangle. Imagine that all the character images in the font are superimposed with their origins coinciding. The smallest rectangle enclosing all the superimposed images is the font rectangle.

The ascent is the distance from the base line to the top of the font rectangle, and the descent is the distance from the base line to the bottom of the font rectangle.

The character height is the vertical extent of the font rectangle. The maximum character height is 127 pixels. The maximum width of the font rectangle is 254 pixels.

The leading is the amount of blank space to draw between lines of single-spaced text--the number of pixels between the descent line of one line of text and the ascent line of the next line of text.

Finally, for each character in a font there's a character offset. As illustrated in Figure 8, the character offset is simply the difference in position of the character rectangle for a given character and the font rectangle.

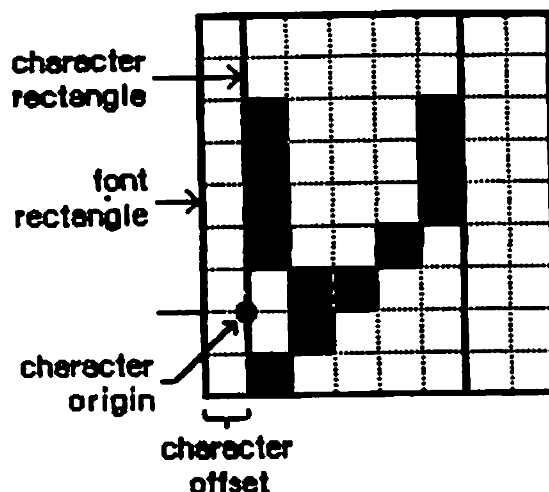


Figure 8. Character Offset

Font Records

The information describing a font is contained in a data structure called a font record, which contains the following:

- the font type (fixed-width or proportional)
- the ASCII code of the first character and the last character in the font
- the maximum character width and maximum amount any character kerns
- the character height, ascent, descent, and leading
- the bit image of the font
- a location table, which is an array of words specifying the location of each character image within the bit image
- an offset/width table, which is an array of words specifying the character offset and character width for each character in the font.

For every character, the location table contains a word that specifies the bit offset to the location of that character's image in the bit image. The entry for a character missing from the font contains the same value as the entry for the next character. The last word of the table contains the offset to one bit beyond the end of the bit image (that is, beyond the character image for the missing symbol). The image width of each character is determined from the location table by subtracting the bit offset to that character from the bit offset to the next character in the table.

There's also one word in the offset/width table for every character: the high-order byte specifies the character offset and the low-order byte specifies the character width. Missing characters are flagged in this table by a word value of -1. The last word is also -1, indicating the end of the table.

Figure 9 illustrates a sample location table and offset/width table corresponding to the bit image in Figure 6.

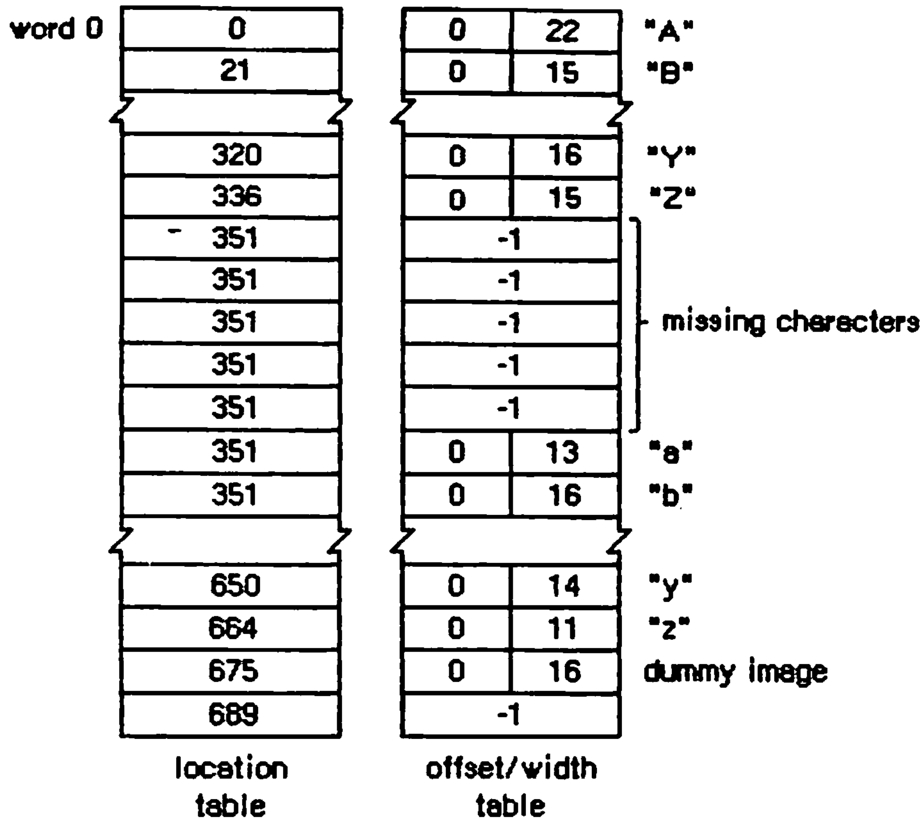


Figure 9. Sample Location Table and Offset/Width Table

A font record is a dynamic structure and is referred to by a handle that you can get by calling the SwapFont function or the Resource Manager function GetResource. The data type for a font record is as follows:

```
TYPE FontRec = RECORD
                  fontType:  INTEGER;   {font type}
                  firstChar: INTEGER;   {ASCII code of first character}
                  lastChar:  INTEGER;   {ASCII code of last character}
                  widMax:    INTEGER;   {maximum character width}
                  kernMax:   INTEGER;   {maximum character kern}
                  nDescent;  INTEGER;   {negative of descent}
                  fRectWid:  INTEGER;   {width of font rectangle}
                  chHeight:  INTEGER;   {character height}
                  owTLoc:    INTEGER;   {offset to offset/width table}
                  ascent:    INTEGER;   {ascent}
                  descent:   INTEGER;   {descent}
                  leading:   INTEGER;   {leading}
                  rowWords:  INTEGER;   {row width of bit image / 2}
               {  bitImage:  ARRAY [1..rowWords, 1..chHeight] OF INTEGER; }
                                        {bit image}
               {  locTable:  ARRAY [firstChar..lastChar+2] OF INTEGER;     }
                                        {location table}
               {  owTable:   ARRAY [firstChar..lastChar+2] OF INTEGER;     }
                                        {offset/width table}
                END;
```

(hand)

The variable-length arrays appear as comments because they're not valid Pascal syntax; they're used only as conceptual aids.

The fontType field must contain one of the following predefined constants:

```
CONST propFont  = $9000;  {proportional font}
      fixedFont = $B000;  {fixed-width font}
```

The values in the widMax, kernMax, nDescent, fRectWid, chHeight, ascent, descent, and leading fields all specify a number of pixels. KernMax indicates the largest number of pixels any character kerns, and should always be negative or zero, because kerning is specified by negative numbers (the kerned pixels are to the left of the character origin). NDescent must be set to the negative of the descent.

The owTLoc field contains a word offset from itself to the offset/width table; it's equivalent to

```
4 + (rowWords * chHeight) + (lastChar - firstChar + 3)
```

(eye)

Remember, the offset and row width in a font record are given in **words**, not bytes.

Normally, the Resource Editor will change the fields in a font record for you. You shouldn't have to change any fields unless you edit the font without the aid of the Resource Editor.

Assembly-language note: The system global romFontØ contains a handle to the font record for the system font.

Font Widths

A resource can be defined that consists of only the character widths and general font information--everything but the font's bit image and location table. If there is such a resource, it will be read in whenever QuickDraw doesn't need to draw the text, such as when you call one of the routines CharWidth, HidePen, or OpenPicture (which calls HidePen). The FontRec data type described above, minus the rowWords, bitImage, and locTable fields, reflects the structure of this type of resource. The owTLoc field will contain 4, and the fontType field will contain the following predefined constant:

```
CONST fontWid = $ACBØ  {font width data}
```

How QuickDraw Draws Text

This section provides a conceptual discussion of the steps QuickDraw takes to draw characters (without scaling or stylistic variations such as bold and outline). Basically, QuickDraw simply copies the character image onto the drawing area at a specific location.

1. Take the initial pen location as the character origin for the first character.

2. Check the word in the offset/width table for the character to see if it's -1. The word to check is entry (charCode - firstChar), where charCode is the ASCII code of the character to be drawn.

   2a. The character exists if the entry in the offset/width table isn't -1. Determine the character offset and character width from the bytes of this same word. Find the character image at the location in the bit image specified by the location table. Calculate the image width by subtracting this word from the succeeding word in the location table. Determine the number of pixels the character kerns by subtracting kernMax from the character offset.

   2b. The character is missing if the entry in the offset/width table is -1; information about the missing symbol is needed. Determine the missing symbol's character offset and character width from the next-to-last word in the offset/width table. Find the missing symbol at the location in the bit image specified by the next-to-last word in the location table (lastChar - firstChar + 1). Calculate the image width by

subtracting the next-to-last word in the location table from the last word (lastChar - firstChar + 2). Determine the number of pixels the missing symbol kerns by subtracting kernMax from the character offset.

3. Move the pen to the left the number of pixels that the character kerns. Move the pen.up the number of pixels specified by the ascent.

4. If the fontType field is fontWid, skip to step 5; otherwise, copy each row of the character image onto the screen or paper, one row at a time. The number of bits to copy from each word is given by the image width, and the number of words is given by the chHeight field.

5. If the fontType field is fontWid, move the pen to the right the number of pixels specified by the character width. If fontType is fixedFont, move the pen to the right the number of pixels specified by the widMax field.

6. If it's time to move down to the next line, return the pen to the left margin and move it down the number of bits specified by the leading.

7. Return to step 2.

FONTS IN A RESOURCE FILE

This section contains details about fonts in resource files that most programmers need not be concerned about, since they can use the Resource Editor *** eventually *** to define fonts. It's included here to give background information to those who are interested. Every size of a font is stored as a separate resource.

The resource type for a font is 'FONT'. The resource data for a font is simply a font record:

Number of bytes	Contents
2 bytes	FontType field of font record
2 bytes	FirstChar field of font record
2 bytes	LastChar field of font record
2 bytes	WidMax field of font record
2 bytes	KernMax field of font record
2 bytes	NDescent field of font record
2 bytes	FRectWid field of font record
2 bytes	ChHeight field of font record
2 bytes	OWTLoc field of font record
2 bytes	Ascent field of font record
2 bytes	Descent field of font record
2 bytes	Leading field of font record
2 bytes	RowWords field of font record
n bytes	Bit image of font n = 2 * rowWords * chHeight
m bytes	Location table of font m = 2 * (lastChar - firstChar + 3)
m bytes	Offset/width table of font m = 2 * (lastChar - firstChar + 3)

The resource type 'FWID' is used to store only the character widths and general information for a font; its resource data is a font record without the rowWords field, bit image, and location table.

As shown in Figure 10, the resource ID of a font is composed of two parts: bits 15 to 7 are the font number, and bits 0 to 6 are the font size. Thus the resource ID corresponding to a given font number and size is

(128 * font number) + font size

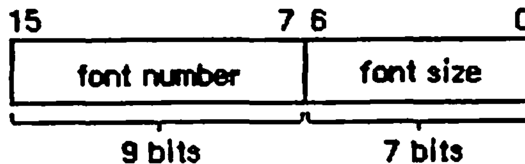


Figure 10. Resource ID for a Font

Since 0 is not a valid font size, the resource ID having 0 in the size field is used to provide only the name of the font: the name of the resource is the font name. For example, for a font named Griffin and numbered 400, the resource naming the font would have a resource ID of 51200 and the resource name 'Griffin'. Size 10 of that font would be stored in a resource numbered 51210.

Font numbers 0 through 127 are reserved for fonts provided by Apple, and font numbers 128 through 383 are reserved for assignment, by Apple, to software vendors. Each font will be assigned a unique number, and that font number should be used to identify only that font (for example, font number 9 will always be Toronto). Font numbers 384 through 511 are available for your use in whatever way you wish.

SUMMARY OF THE FONT MANAGER

Constants

```
CONST systemFont = 0;  {system font}
      applFont   = 1;  {application font}
      newYork    = 2;
      geneva     = 3;
      monaco     = 4;
      venice     = 5;
      london     = 6;
      athens     = 7;
      sanFran    = 8;
      toronto    = 9;

      propFont  = $9000;  {proportional font}
      fixedFont = $B000;  {fixed-width font}
      fontWid   = $ACB0;  {font width data}

TYPE FMInput = PACKED RECORD
                 family:   INTEGER; {font number}
                 size:     INTEGER; {font size}
                 face:     Style;   {character style}
                 needBits: BOOLEAN; {TRUE if drawing}
                 device:   INTEGER; {output device}
                 numer:    Point;   {numerators of scaling factors}
                 denom:    Point    {denominators of scaling factors }
               END;

     FMOutPtr = ^FMOutput;
     FMOutput = PACKED RECORD
                  errNum:     INTEGER;    {not used}
                  fontHandle: Handle;     {handle to font record}
                  bold:       Byte;       {bold factor}
                  italic:     Byte;       {italic factor}
                  ulOffset:   Byte;       {underline offset}
                  ulShadow:   Byte;       {underline shadow}
                  ulThick:    Byte;       {underline thickness}
                  shadow:     Byte;       {shadow factor}
                  extra:      SignedByte; {not used}
                  ascent:     Byte;       {ascent}
                  descent:    Byte;       {descent}
                  widMax:     Byte;       {maximum character width}
                  leading:    SignedByte; {leading}
                  unused:     Byte;       {not used}
                  numer:      Point;      {numerators of scaling factors}
                  denom:      Point       {denominators of scaling factors}
                END;
```

```
FontRec = RECORD
            fontType:  INTEGER;   {font type}
            firstChar: INTEGER;   {ASCII code of first character}
            lastChar:  INTEGER;   {ASCII code of last character}
            widMax:    INTEGER;   {maximum character width}
            kernMax:   INTEGER;   {maximum character kern}
            nDescent:  INTEGER;   {negative of descent}
            fRectMax:  INTEGER;   {width of font rectangle}
            chHeight:  INTEGER;   {character height}
            owTLoc:    INTEGER;   {offset to offset/width table}
            ascent:    INTEGER;   {ascent}
            descent:   INTEGER;   {descent}
            leading:   INTEGER;   {leading}
            rowWords:  INTEGER;   {row width of bit image / 2}
         {  bitImage:  ARRAY [1..rowWords, 1..chHeight] OF INTEGER; }
                                  {bit image}
         {  locTable:  ARRAY [firstChar..lastChar+2] OF INTEGER;    }
                                  {location table}
         {  owTable:   ARRAY [firstChar..lastChar+2] OF INTEGER     }
                                  {offset/width table}
          END;
```

Routines

Initializing the Font Manager

```
PROCEDURE InitFonts;
```

Getting Font Information

```
PROCEDURE GetFontName (fontNum: INTEGER; VAR theName: Str255);
PROCEDURE GetFNum     (fontName: Str255; VAR theNum: INTEGER);
FUNCTION  RealFont    (fontNum: INTEGER; size: INTEGER) : BOOLEAN;
```

Keeping Fonts in Memory

```
PROCEDURE SetFontLock (lockFlag: BOOLEAN);
```

Advanced Routine

```
FUNCTION SwapFont (inRec: FMInput) : FMOutPtr;
```

Assembly-Language Information

Font Input Record Data Structure

fmInFamily	Font number
fmInSize	Font size
fmInFace	Character style
fmInNeedBits	TRUE if drawing
fmInDevice	Output device
fmInNumer	Numerators of scaling factors
fmInDenom	Denominators of scaling factors

Font OutPut Record Data Structure

fOutError	Not used
fOutFontHandle	Handle to font record
fOutBold	Bold factor
fOutItalic	Italic factor
fOutUlOffset	Underline offset
fOutUlShadow	Underline shadow
fOutUlThick	Underline thickness
fOutShadow	Shadow factor
fOutExtra	Not used
fOutAscent	Ascent
fOutDescent	Descent
fOutWidMax	Maximum character width
fOutLeading	Leading
fOutUnused	Not used
fOutNumer	Numerators of scaling factors
fOutDenom	Denominators of scaling factors

Font Record Data Structure

fFormat	Font type
fMinChar	ASCII code of first character
fMaxChar	ASCII code of last character
fMaxWd	Maximum character width
fBBOX	Maximum character kern
fBBOY	Negative of descent
fBBDX	Width of font rectangle
fBBDY	Character height
fLength	Offset to offset/width table
fAscent	Ascent
fDescent	Descent
fLeading	Leading
fRaster	Row width of bit image / 2

Special Macro Name

Routine name	Macro name
GetFontName	_GetFName

System Globals

Name	Size	Contents
spFont	2 bytes	Font number of default application font
apFontID	2 bytes	Font number of current application font
fScaleDisable	1 byte	Nonzero to disable scaling
romFontØ	4 bytes	Handle to font record for system font

GLOSSARY

application font: A font, referred to by font number 1, that's suitable for general use by an application--New York, by default.

ascent: The vertical distance from a font's base line to its ascent line.

ascent line: A horizontal line coincident with the top of the tallest characters in a font.

base line: A horizontal line coincident with the bottom of each character in a font, excluding descenders.

character height: The vertical extent of a character rectangle.

character image: The bit image that defines a character.

character offset: The horizontal separation between a character rectangle and a font rectangle.

character origin: The point on a base line used as a reference location for drawing a character.

character rectangle: The smallest rectangle enclosing an entire character image.

character style: A set of stylistic variations, such as bold, italic, and underline. The empty set indicates normal text (no stylistic variations).

character width: The distance to move the pen from one character's origin to the next; equivalent to the image width plus the amount of blank space to leave before the next character.

descent: The vertical distance from a font's base line to its descent line.

descent line: A horizontal line coincident with the bottom of the lowest-reaching characters in a font, including descenders.

fixed-width font: A font whose characters all have the same width.

font: The complete set of characters of one typeface.

font characterization table: A table of parameters in a device driver that specifies how best to adapt fonts to that device.

font number: The number by which you identify a font to QuickDraw or the Font Manager.

font record: A data structure that contains all the information describing a font.

font rectangle: The smallest rectangle enclosing all the character images in a font, if the images were all superimposed over the same character origin.

font size: The size of a font in points; equivalent to the distance between the ascent line of one line of text and the ascent line of the next of line of single-spaced text.

image width: The horizontal extent of a character image.

kern: To draw part of a character so that it overlaps an adjacent character.

leading: The amount of blank vertical space between the descent line of one line of text and the ascent line of the next line of single-spaced text.

location table: An array of words (one for each character in a font) that specifies the location of each character's image in the font's bit image.

missing symbol: A character to be drawn in case of a request to draw a character that's missing from a particular font.

offset/width table: An array of words that specifies the character offsets and character widths of all characters in a font.

point: The intersection of a horizontal grid line and a vertical grid line on the coordinate plane, defined by a horizontal and a vertical coordinate; also, a typographical term meaning approximately 1/72 inch.

proportional font: A font whose characters all have character widths that are proportional to their image width.

row width: The number of bytes in each row of a bit image.

scaling factor: A value, given as a fraction, that specifies the amount a character should be stretched or shrunk before it's drawn.

style: See character style.

system font: The font, identified by font number Ø, that the system uses (in menus, for example).

system font size: The size of text drawn by the system in the system font; 12 points.

SUMMARY OF FP68K DOCUMENTS

User's Guide -- an overview of FP68K ELEMS68K and their design philosophy.

Programmer's Guide -- hints on how to build the packages, and how to modify them, if necessary; details about system dependencies involving the state area. Includes register map templates.

System Interface -- how FP68K and ELEMS68K affect their execution environment.

High-Level Interface -- the SANE Pascal unit and assembly macros.

Integer Conversion Tests

Binary-Decimal Conversion Tests

IEEE Tests -- a set of test vectors designed for this style of arithmetic and distributed through the standards subcommittee

Binary-Decimal Conversions -- what is available through the SANE interface, and what FP68K provides at the low level. A sample parser and formatter from the SANE interface is shown.

P754 stuff -- papers related to the arithmetic standard.

FPxxx.TEXT -- source files for FP68K, except for binary-decimal conversions

FBxxx.TEXT -- source files for binary-decimal part of FP68K

SAxxx.TEXT -- SANE68.TEXT -- SANE interface section
SAIMP68.TEXT -- SANE implementation section
SAASM68.TEXT -- SANE assembly procedures
SAMAC68.TEXT -- EQU's and MACRO's for assembly interface

DOxxx.TEXT -- documentation using SCRIPT formatter, with macros in DODRIVER.TEXT

TVxxx.TEXT -- IEEE test vector files, required operations

TWxxx.TEXT -- IEEE test vector files, appendix funtions

TDxxx.TEXT -- Test vector driver program files

ITxxx.TEXT -- integer <--> extended conversion tests

IOxxx.TEXT -- binary <--> decimal conversions tests

Zyyyy.OBJ -- executable test programs

ELxxx.TEXT -- elementary transcendental and financial functions

Introduction

The 68000 software floating-point packages, FP68K and ELEMS68K, appear much like simple subroutines but their interaction with the host system is somewhat more subtle. This document indicates possible trouble spots. It is intended for system implementors, rather than users of FP68K and ELEMS68K.

The following sections describe the various issues in turn.

Registers and stack used

FP68K and ELEMS68K receive all of their parameters on the stack. They save and restore all of the CPU registers across calls, except that D0 is modified by the REMAINDER operation. FP68K modifies the CPU Condition Code Register as described later.

As detailed in the "Program Notes" document, FP68K typically uses up to 41 words of stack beyond the input parameters. The only exceptions are the binary-decimal conversion and nextafter routines, which may use up to 120 words beyond the input parameters. ELEMS68K uses at most 30 words of stack for temporary storage.

Single entry point

FP68K has just one entry point — with the label 'FP68K'. When invoked, FP68K expects the return address on the stack, followed by a one-word opcode described in the user's guide. Beyond the opcode are up to three operand addresses (depending on the operation). Note that because the operands are passed by address, they must be in memory, NOT IN THE REGISTER FILE.

If FP68K is to be invoked by a mechanism like the A-line trap, care must be taken that stack is set up properly. Depending on the system, it should be possible to execute FP68K either as a subprogram linked to an application program, or as system-provided utility.

Because of the varying number of input parameters, it is impossible to call FP68K directly from Pascal, since the number of parameters is fixed when the EXTERNAL procedure is defined. In any case programmers should use the provided Pascal interface, called SANE (Standard Apple Numeric Environment).

ELEMS68K has a similar design, but is configured as a separate package for modularity.

Exit points

Typically, FP68K exits by clearing all input operands from the stack and jumping to the return address.

However, a 'halting' mechanism is provided whereby control is transferred from FP68K to an address saved in the floating-point state area (see below). This address should refer to a subprogram in the user's code space. When the halt routine is invoked, the top of the stack is a word containing the number of bytes of parameters (including the return address) on the stack when FP68K was originally called. Beyond that word is the exact stack frame from when FP68K was originally called.

ELEMS68K has no built-in halt mechanism, though a subsidiary FP68K operation may halt.

State area

ELEMS68K maintains no static state. FP68K maintains 3 words of static state across invocations. The first word contains mode and flag bits, much like the CPU Status Register. The next long word is the user trap address. There are two important issues: where is the state area and how is it initialized?

The state area may be a fixed area in memory, as in MAC, or at a fixed offset from a register like A6, as in LISA, or in some user area if FP68K is linked as a subroutine. The state area may even be kept within FP68K itself, though this makes the code self-modifying and thus NON-REENTRANT.

In multi-process environments, care must be taken to see that different state areas are kept for the different processes (again, think of the CPU Status Register). For example, if the state area is kept in a fixed location in memory, it must be swapped each time a new process is swapped in.

The location of the state area must be known at ASSEMBLY TIME. As indicated in the programmer's guide document, the code must be set up for the particular host environment.

When a new process is started up, the state area must be initialized. Fortunately, this is easy. Just clear to 0 the first word of the state area (i.e. the mode and flag word).

CPU Condition Code Register

The Comparison operation leaves the CCR in a well-defined state. After Comparison, the CCR is set for a conditional branch, although the flags are used in a way different from the integer CPU comparisons; see the "User's Guide" for details.

CPU Register D0

The Remainder operation leaves the low-order integer quotient (between -127 and +127) in D0.W. The high half of D0.L is undefined. This intrusion into the register file is extremely valuable in argument reduction -- the principal use of Remainder. The state of D0 after an invalid remainder is

undefined.

SANE

There is a SANE (Standard Apple Numeric Environment) library of utility functions based on FP68K, as well as a corresponding Elems library based on ELEMS68K. These libraries are supported on Apple III Pascal systems as well. The library provides access to the package from (Lisa) Pascal. Aside from support of basic arithmetic and elementary functions, the utilities manipulate the modes and flags and provide ASCII <--> floating-point conversions. All applications software should use this package because of its high degree of portability.

Assembly language programmers will invoke FP68K and ELEMS68K directly but will depend on some library for routines to convert between ASCII strings and the canonical decimal format which FP68K recognizes. A set of mnemonic MACROS has been provide to expedite assembly coding.

Compiling Pascal programs

A Pascal program which exploits the SANE and Elems interfaces must include lines such as

```
uses {$U <some volume>:SANE.OBJ} SANE;
uses {$U <some volume>:ELEMS.OBJ} Elems;
```

in order to gain access to the types and procedures defined there. Then the program must be linked with SANE.OBJ and ELEMS.OBJ (the Pascal parts of the interface), as well as SANEASM.OBJ and ELEMSASM.OBJ (the assembly language parts of the interface).

Pascal procedures

Programmers should consult the INTERFACE section of the SANE and Elems interfaces (files SANE.TEXT and ELEMS.TEXT) in the following pages. This interface reflects the architecture discussed in the "User's Guide". It is two-address, with the destination operand in the extended format except for format conversions conversions.

Macros

A set of macros provides direct contact with the arithmetic package, using the interface described in the "User's Guide". The macros take care of the opcode and the JSR, but the programmer must explicity push the required argument addresses. The macros do not take effective address arguments and push them itself because of the problems that arise if the destination operand is given as an offset from SP (which changes when the first operand address is pushed). The macros are listed after the Pascal interface.

Sample program

The test programs ITxxx.TEXT, IOxxx.TEXT, and TDFP.TEXT provide a nontrivial view of how to use the Pascal interface to FP68K and ELEMS68K.

```
{^he ''SANE Interface'' }
{^fo '28 December 1982'Page %'Apple Confidential' }
{$C Copyright Apple Computer, 1982 }
{MacIntosh version.}

UNIT Sane;

    INTERFACE

        CONST

            SIGDIGLEN = 20;      { Maximum length of SigDig. }

            DECSTRLEN = 80;      { Maximum length of DecStr. }

        TYPE

            {-----------------------------------------------------------------
            ** Numeric types.
            -----------------------------------------------------------------}

                Single   = array [0..1] of integer;
                Double   = array [0..3] of integer;
                Comp     = array [0..3] of integer;
                Extended = array [0..4] of integer;

            {-----------------------------------------------------------------
            ** Decimal string type and intermediate decimal type,
            ** representing the value:
            **      (-1)^sgn * 10^exp * dig
            -----------------------------------------------------------------}

                SigDig   = string [SIGDIGLEN];
                DecStr   = string [DECSTRLEN];
                Decimal  = record
                               sgn : 0..1;     { Sign (0 for pos, 1 for neg). }
                               exp : integer; { Exponent. }
                               sig : SigDig    { String of significant digits. }
                           end;

{^ne 16 }
            {-----------------------------------------------------------------
            ** Modes, flags, and selections.
            ** NOTE: the values of the style element of the DecForm record
            ** have different names from the PCS version to avoid name
            ** conflicts.
            -----------------------------------------------------------------}

                Environ   = integer;
                RoundDir  = (TONEAREST, UPWARD, DOWNWARD, TOWARDZERO);
                RelOp     = (GT, LT, GL, EQ, GE, LE, GEL, UNORD);
                            { >   <  <>   =  >=  <=  <=> }
                Exception = (INVALID, UNDERFLOW, OVERFLOW, DIVBYZERO,
                                                                INEXACT);
```

```
            NumClass  = (SNAN, QNAN, INFINITE, ZERO, NORMAL, DENORMAL);
            DecForm   = record
                            style  : (FloatDecimal, FixedDecimal);
                            digits : integer
                        end;

{^ne 35 }
        {-------------------------------------------------------------
        ** Two address, extended-based arithmetic operations.
        -------------------------------------------------------------}

        procedure AddS (x : Single;   var y : Extended);
        procedure AddD (x : Double;   var y : Extended);
        procedure AddC (x : Comp;     var y : Extended);
        procedure AddX (x : Extended; var y : Extended);
            { y := y + x }

        procedure SubS (x : Single;   var y : Extended);
        procedure SubD (x : Double;   var y : Extended);
        procedure SubC (x : Comp;     var y : Extended);
        procedure SubX (x : Extended; var y : Extended);
            { y := y - x }

        procedure MulS (x : Single;   var y : Extended);
        procedure MulD (x : Double;   var y : Extended);
        procedure MulC (x : Comp;     var y : Extended);
        procedure MulX (x : Extended; var y : Extended);
            { y := y * x }

        procedure DivS (x : Single;   var y : Extended);
        procedure DivD (x : Double;   var y : Extended);
        procedure DivC (x : Comp;     var y : Extended);
        procedure DivX (x : Extended; var y : Extended);
            { y := y / x }

        function  CmpX (x : Extended; r : RelOp;
                                        y : Extended) : boolean;
            { x r y }

        function  RelX (x, y : Extended) : RelOp;
            { x RelX y, where RelX in [GT, LT, EQ, UNORD] }

{^ne 18 }
        {-------------------------------------------------------------
        ** Conversions between Extended and the other numeric types,
        ** including the types integer and Longint.
        -------------------------------------------------------------}

        procedure I2X (x : integer;  var y : Extended);
        procedure S2X (x : Single;   var y : Extended);
        procedure D2X (x : Double;   var y : Extended);
        procedure C2X (x : Comp;     var y : Extended);
        procedure X2X (x : Extended; var y : Extended);
            { y := x (arithmetic assignment) }
```

```
        procedure X2I (x : Extended; var y : integer);
        procedure X2S (x : Extended; var y : Single);
        procedure X2D (x : Extended; var y : Double);
        procedure X2C (x : Extended; var y : Comp);
            { y := x (arithmetic assignment) }

{^ne 9 }
        {------------------------------------------------------------
        ** These conversions apply to 68K systems only.  Longint is
        ** a 32-bit two's complement integer.
        ------------------------------------------------------------}

        procedure L2X (x : Longint;  var y : Extended);
        procedure X2L (x : Extended; var y : Longint);
            { y := x (arithmetic assignment) }

{^ne 17 }
        {------------------------------------------------------------
        ** Conversions between the numeric types and the intermediate
        ** decimal type.
        ------------------------------------------------------------}

        procedure S2Dec (f : DecForm; x : Single;   var y : Decimal);
        procedure D2Dec (f : DecForm; x : Double;   var y : Decimal);
        procedure C2Dec (f : DecForm; x : Comp;     var y : Decimal);
        procedure X2Dec (f : DecForm; x : Extended; var y : Decimal);
            { y := x (according to the format f) }

        procedure Dec2S (x : Decimal; var y : Single);
        procedure Dec2D (x : Decimal; var y : Double);
        procedure Dec2C (x : Decimal; var y : Comp);
        procedure Dec2X (x : Decimal; var y : Extended);
            { y := x }

{^ne 18 }
        {------------------------------------------------------------
        ** Conversions between the numeric types and strings.
        ** (These conversions have a built-in scanner/parser to convert
        ** between the intermediate decimal type and a string.)
        ------------------------------------------------------------}

        procedure S2Str (f : DecForm; x : Single;   var y : DecStr);
        procedure D2Str (f : DecForm; x : Double;   var y : DecStr);
        procedure C2Str (f : DecForm; x : Comp;     var y : DecStr);
        procedure X2Str (f : DecForm; x : Extended; var y : DecStr);
            { y := x (according to the format f) }

        procedure Str2S (x : DecStr; var y : Single);
        procedure Str2D (x : DecStr; var y : Double);
        procedure Str2C (x : DecStr; var y : Comp);
        procedure Str2X (x : DecStr; var y : Extended);
            { y := x }
```

```
{^ne 31 }
       {------------------------------------------------------------
       ** Numerical 'library' procedures and functions.
       ------------------------------------------------------------}

       procedure RemX    (x : Extended; var y : Extended;
                                                   var quo : integer);
           { new y := remainder of ((old y) / x), such that
                      |new y| <= |x| / 2;
             quo := low order seven bits of integer quotient y / x,
                    so that -127 <= quo <= 127.                    }
       procedure SqrtX   (var x : Extended);
           { x := sqrt (x) }
       procedure RintX   (var x : Extended);
           { x := rounded value of x }
       procedure NegX    (var x : Extended);
           { x := -x }
       procedure AbsX    (var x : Extended);
           { x := |x| }
       procedure CpySgnX (var x : Extended; y : Extended);
           { x := x with the sign of y }

       procedure NextS   (var x : Single;   y : Single);
       procedure NextD   (var x : Double;   y : Double);
       procedure NextX   (var x : Extended; y : Extended);
           { x := next representable value from x toward y }

       function  ClassS  (x : Single;   var sgn : integer) : NumClass;
       function  ClassD  (x : Double;   var sgn : integer) : NumClass;
       function  ClassC  (x : Comp;     var sgn : integer) : NumClass;
       function  ClassX  (x : Extended; var sgn : integer) : NumClass;
           { sgn := sign of x (0 for pos, 1 for neg) }

       procedure ScalbX (n : integer; var y : Extended);
           { y := y * 2^n }
       procedure LogbX  (var x : Extended);
           { returns unbiased exponent of x }
{^ne 16 }
       {------------------------------------------------------------
       ** Manipulations of the static numeric state.
       ------------------------------------------------------------}

       procedure SetRnd  (r : RoundDir);
       procedure SetEnv  (e : Environ);
       procedure ProcExit(e : Environ);

       function  GetRnd  : RoundDir;
       procedure GetEnv  (var e : Environ);
       procedure ProcEntry (var e : Environ);

       function  TestXcp (x : Exception) : boolean;
       procedure SetXcp  (x : Exception; OnOff : boolean);
       function  TestHlt (x : Exception) : boolean;
       procedure SetHlt  (x : Exception; OnOff : boolean);
```

```
    {-----------------------------------------------------------------------}
{^sp 32767 }
    {-----------------------------------------------------------------------}

    IMPLEMENTATION

        {$I SANEIMP.TEXT}

END

{=========================================================================} .
```

```
{$C Copyright Apple Computer Inc., 1983 }

UNIT Elems;

    { Macintosh version. }

    {-------------------------------------------------------------------}

    INTERFACE

        USES

            {$U OBJ:SANE.OBJ }

                SANE { Standard Apple Numeric Environment } ;

        procedure Log2X (var x : Extended);
            { x := log2 (x) }

        procedure LnX (var x : Extended);
            { x := ln (x) }

        procedure Ln1X (var x : Extended);
            { x := ln (1 + x) }

        procedure Exp2X (var x : Extended);
            { x := 2^x }

        procedure ExpX (var x : Extended);
            { x := e^x }

        procedure Exp1X (var x : Extended);
            { x := e^x - 1 }

        procedure XpwrI (i : integer; var x : Extended);
            { x := x^i }

        procedure XpwrY (y : Extended; var x : Extended);
            { x := x^y }

        procedure Compound (r, n : Extended; var x : Extended);
            { x := (1 + r)^n }

        procedure Annuity (r, n : Extended; var x : Extended);
            { x := (1 - (1 + r)^-n) / r }

        procedure SinX (var x : Extended);
            { x := sin(x) }

        procedure CosX (var x : Extended);
            { x := cos(x) }

        procedure TanX (var x : Extended);
            { x := tan(x) }
```

```
    procedure AtanX (var x : Extended);
        { x := atan(x) }

    procedure NextRandom (var x : Extended);
        { x := next random (x) }

{$p---------------------------------------------------------------}
IMPLEMENTATION

    procedure Log2X { (var x : Extended) } ;                  EXTERNAL;
    procedure LnX { (var x : Extended) } ;                    EXTERNAL;
    procedure Ln1X { (var x : Extended) } ;                   EXTERNAL;
    procedure Exp2X { (var x : Extended) } ;                  EXTERNAL;
    procedure ExpX { (var x : Extended) } ;                   EXTERNAL;
    procedure Exp1X { (var x : Extended) } ;                  EXTERNAL;

    {
        Since Elems implementation expects pointer to integer argument,
        use this extra level of interface.
    }
    procedure XpwrIxxx(var i : integer; var x : Extended);   EXTERNAL;
    procedure XpwrI { (i : integer; var x : Extended) } ;
    begin
            XpwrIxxx(i, x);
    end;

    procedure XpwrY { (y : Extended; var x : Extended) } ;   EXTERNAL;
    procedure Compound { (r, n : Extended; var x : Extended) } ;  EXTERNAL;
    procedure Annuity { (r, n : Extended; var x : Extended) } ;  EXTERNAL;
    procedure SinX { (var x : Extended) } ;                   EXTERNAL;
    procedure CosX { (var x : Extended) } ;                   EXTERNAL;
    procedure TanX { (var x : Extended) } ;                   EXTERNAL;
    procedure AtanX { (var x : Extended) } ;                  EXTERNAL;
    procedure NextRandom { (var x : Extended) } ;             EXTERNAL;

END
{=================================================================}
{=================================================================}
{=================================================================}.
```

```
;-----------------------------------------------------------
;
;  These macros give assembly language access to the Mac
;  floating-point arithmetic routines.  The arithmetic has
;  just one entry point.  It is typically accessed through
;  the tooltrap _FP68K, although a custom version of the
;  package may be linked as an object file, in which case
;  the entry point is the label %FP68K.
;
;  All calls to the arithmetic take the form:
;       PEA     <source address>
;       PEA     <destination address>
;       MOVE.W  <opcode>,-(SP)
;       _FP68K
;
;  All operands are passed by address.  The <opcode> word
;  specifies the instruction analogously to a 68000 machine
;  instruction.  Depending on the instruction, there may be
;  from one to three operand addresses passed.
;
;  This definition file specifies details of the <opcode>
;  word and the floating point state word, and defines
;  some handy macros.
;
;  Modification history:
;       29AUG82: WRITTEN BY JEROME COONEN
;       13OCT82: FB___ CONSTRANTS ADDED (JTC)
;       28DEC82: LOGB, SCALB ADDED, INF MODES OUT (JTC).
;       29APR83: ABS, NEG, CPYSGN, CLASS ADDED (JTC).
;       03MAY83: NEXT, SETXCP ADDED (JTC).
;       28MAY83: ELEMENTARY FUNCTIONS ADDED (JTC).
;       04JUL83: SHORT BRANCHES, TRIG AND RAND ADDED (JTC).
;       01NOV83: PRECISION CONTROL MADE A MODE (JTC).
;
;-----------------------------------------------------------


;-----------------------------------------------------------
; This constant determines whether the floating point unit
; is accessed via the system dispatcher after an A-line
; trap, or through a direct subroutine call to a  custom
; version of the package linked directly to the application.
;-----------------------------------------------------------
ATRAP           .EQU    0       ;0 for JSR and 1 for A-line
BTRAP           .EQU    0       ;0 for JSR and 1 for A-line

        .MACRO  JSRFP
        .IF     ATRAP
        _FP68K
        .ELSE
        .REF    FP68K
        JSR     FP68K
        .ENDC
        .ENDM
```

```
        .MACRO  JSRELEMS
        .IF     BTRAP
        _ELEMS68K
        .ELSE
        .REF    ELEMS68K
        JSR     ELEMS68K
        .ENDC
        .ENDM


;-----------------------------------------------------------
; OPERATION MASKS: bits $001F of the operation word
; determine the operation.  There are two rough classes of
; operations:  even numbered opcodes are the usual
; arithmetic operations and odd numbered opcodes are non-
; arithmetic or utility operations.
;-----------------------------------------------------------
FOADD           .EQU    $0000
FOSUB           .EQU    $0002
FOMUL           .EQU    $0004
FODIV           .EQU    $0006
FOCMP           .EQU    $0008
FOCPX           .EQU    $000A
FOREM           .EQU    $000C
FOZ2X           .EQU    $000E
FOX2Z           .EQU    $0010
FOSQRT          .EQU    $0012
FORTI           .EQU    $0014
FOTTI           .EQU    $0016
FOSCALB         .EQU    $0018
FOLOGB          .EQU    $001A
FOCLASS         .EQU    $001C
; UNDEFINED     .EQU    $001E

FOSETENV        .EQU    $0001
FOGETENV        .EQU    $0003
FOSETTV         .EQU    $0005
FOGETTV         .EQU    $0007
FOD2B           .EQU    $0009
FOB2D           .EQU    $000B
FONEG           .EQU    $000D
FOABS           .EQU    $000F
FOCPYSGNX       .EQU    $0011
FONEXT          .EQU    $0013
FOSETXCP        .EQU    $0015
FOPROCENTRY     .EQU    $0017
FOPROCEXIT      .EQU    $0019
FOTESTXCP       .EQU    $001B
; UNDEFINED     .EQU    $001D
; UNDEFINED     .EQU    $001F


;-----------------------------------------------------------
```

```
; OPERAND FORMAT MASKS: bits $3800 determine the format of
; any non-extended operand.
;-----------------------------------------------------------
FFEXT           .EQU    $0000    ; extended -- 80-bit float
FFDBL           .EQU    $0800    ; double   -- 64-bit float
FFSGL           .EQU    $1000    ; single   -- 32-bit float
FFINT           .EQU    $2000    ; integer  -- 16-bit integer
FFLNG           .EQU    $2800    ; long int -- 32-bit integer
FFCOMP          .EQU    $3000    ; accounting -- 64-bit int


;-----------------------------------------------------------
; Bit indexes for error and halt bits and rounding modes in
; the state word.  The word is broken down as:
;
;       $8000 -- unused
;
;       $6000 -- rounding modes
;                $0000 -- to nearest
;                $2000 -- toward +infinity
;                $4000 -- toward -infinity
;                $6000 -- toward zero
;
;       $1F00 -- error flags
;                $1000 -- inexact
;                $0800 -- division by zero
;                $0400 -- overflow
;                $0200 -- underflow
;                $0100 -- invalid operation
;
;       $0080 -- result of last rounding
;                $0000 -- rounded down in magnitude
;                $0080 -- rounded up in magnitude
;
;       $0060 -- precision control
                 $0000 -- extended
                 $0020 -- double
                 $0040 -- single
                 $0060 -- ILLEGAL
;
;       $001F -- halt enables, corresponding to error flags
;
; The bit indexes are based on the byte halves of the state
; word.
;-----------------------------------------------------------
FBINVALID       .EQU    0        ; invalid operation
FBUFLOW         .EQU    1        ; underflow
FBOFLOW         .EQU    2        ; overflow
FBDIVZER        .EQU    3        ; division by zero
FBINEXACT       .EQU    4        ; inexact
FBRNDLO         .EQU    5        ; low bit of rounding mode
FBRNDHI         .EQU    6        ; high bit of rounding mode
FBLSTRND        .EQU    7        ; last round result bit
FBDBL           .EQU    5        ; double precision control
```

```
FBSGL           .EQU    6       ; single precision control


;--------------------------------------------------------
; FLOATING CONDITIONAL BRANCHES: floating point comparisons
; set the CPU condition code register (the CCR) as follows:
;       relation        X N Z V C
;       ---------------------
;         equal         0 0 1 0 0
;       less than       1 1 0 0 1
;       greater than    0 0 0 0 0
;        unordered      0 0 0 1 0
; The macros below define a set of so-called floating
; branches to spare the programmer repeated refernces to the
; the table above.
;--------------------------------------------------------
        .MACRO  FBEQ
        BEQ     %1
        .ENDM

        .MACRO  FBLT
        BCS     %1
        .ENDM

        .MACRO  FBLE
        BLS     %1
        .ENDM

        .MACRO  FBGT
        BGT     %1
        .ENDM

        .MACRO  FBGE
        BGE     %1
        .ENDM

        .MACRO  FBULT
        BLT     %1
        .ENDM

        .MACRO  FBULE
        BLE     %1
        .ENDM

        .MACRO  FBUGT
        BHI     %1
        .ENDM

        .MACRO  FBUGE
        BCC     %1
        .ENDM

        .MACRO  FBU
        BVS     %1
```

```
        .ENDM

        .MACRO  FBO
        BVC     %1
        .ENDM

        .MACRO  FBNE
        BNE     %1
        .ENDM

        .MACRO  FBUE
        BEQ     %1
        BVS     %1
        .ENDM

        .MACRO  FBLG
        BNE     %1
        BVC     %1
        .ENDM

; Short branch versions.

        .MACRO  FBEQS
        BEQ.S   %1
        .ENDM

        .MACRO  FBLTS
        BCS.S     %1
        .ENDM

        .MACRO  FBLES
        BLS.S     %1
        .ENDM

        .MACRO  FBGTS
        BGT.S     %1
        .ENDM

        .MACRO  FBGES
        BGE.S     %1
        .ENDM

        .MACRO  FBULTS
        BLT.S     %1
        .ENDM

        .MACRO  FBULES
        BLE.S     %1
        .ENDM

        .MACRO  FBUGTS
        BHI.S     %1
        .ENDM
```

```
        .MACRO  FBUGES
        BCC.S     %1
        .ENDM

        .MACRO  FBUS
        BVS.S     %1
        .ENDM

        .MACRO  FBOS
        BVC.S     %1
        .ENDM

        .MACRO  FBNES
        BNE.S     %1
        .ENDM

        .MACRO  FBUES
        BEQ.S     %1
        BVS.S     %1
        .ENDM

        .MACRO  FBLGS
        BNE.S     %1
        BVC.S     %1
        .ENDM


;-----------------------------------------------------------
; OPERATION MACROS:
;       THESE MACROS REQUIRE THAT THE OPERANDS' ADDRESSES
;       FIRST BE PUSHED ON THE STACK.  THE MACROS CANNOT
;       THEMSELVES PUSH THE ADDRESSES SINCE THE ADDRESSES
;       MAY BE SP-RELATIVE, IN WHICH CASE THEY REQUIRE
;       PROGRAMMER CARE.
; OPERATION MACROS: operand addresses should already be on
; the stack, with the destination address on top.  The
; suffix X, D, S, or C determines the format of the source
; operand — extended, double, single, or computational
; respectively; the destination operand is always extended.
;-----------------------------------------------------------


;-----------------------------------------------------------
; Addition.
;-----------------------------------------------------------
        .MACRO  FADDX
        MOVE.W  #FFEXT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDD
        MOVE.W  #FFDBL+FOADD,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO   FADDS
        MOVE.W   #FFSGL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO   FADDC
        MOVE.W   #FFCOMP+FOADD,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Subtraction.
;-----------------------------------------------------------
        .MACRO   FSUBX
        MOVE.W   #FFEXT+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBD
        MOVE.W   #FFDBL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBS
        MOVE.W   #FFSGL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO   FSUBC
        MOVE.W   #FFCOMP+FOSUB,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Multiplication.
;-----------------------------------------------------------
        .MACRO   FMULX
        MOVE.W   #FFEXT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULD
        MOVE.W   #FFDBL+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO   FMULS
        MOVE.W   #FFSGL+FOMUL,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FMULC
        MOVE.W  #FFCOMP+FOMUL,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Division.
;-------------------------------------------------------------
        .MACRO  FDIVX
        MOVE.W  #FFEXT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVD
        MOVE.W  #FFDBL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVS
        MOVE.W  #FFSGL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVC
        MOVE.W  #FFCOMP+FODIV,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Compare, signaling no exceptions.
;-------------------------------------------------------------
        .MACRO  FCMPX
        MOVE.W  #FFEXT+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPD
        MOVE.W  #FFDBL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPS
        MOVE.W  #FFSGL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPC
        MOVE.W  #FFCOMP+FOCMP,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------
; Compare, signaling invalid operation if the two operands
; are unordered.
;-----------------------------------------------------------
        .MACRO  FCPXX
        MOVE.W  #FFEXT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXD
        MOVE.W  #FFDBL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXS
        MOVE.W  #FFSGL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXC
        MOVE.W  #FFCOMP+FOCPX,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Remainder.  The remainder is placed in the destination,
; and the low bits of the integer quotient are placed in
; the low word of register D0.
;-----------------------------------------------------------
        .MACRO  FREMX
        MOVE.W  #FFEXT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMD
        MOVE.W  #FFDBL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMS
        MOVE.W  #FFSGL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMC
        MOVE.W  #FFCOMP+FOREM,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Compare the source operand to the extended format and
; place in the destination.
```

```
;-----------------------------------------------------------
        .MACRO  FX2X
        MOVE.W  #FFEXT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2X
        MOVE.W  #FFDBL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2X
        MOVE.W  #FFSGL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FI2X                    ; 16-bit integer
        MOVE.W  #FFINT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2X                    ; 32-bit integer
        MOVE.W  #FFLNG+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2X
        MOVE.W  #FFCOMP+FOZ2X,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Convert the extended source operand to the specified
; format and place in the destination.
;-----------------------------------------------------------
        .MACRO  FX2D
        MOVE.W  #FFDBL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2S
        MOVE.W  #FFSGL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2I                    ; 16-bit integer
        MOVE.W  #FFINT+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2L                    ; 32-bit integer
        MOVE.W  #FFLNG+FOX2Z,-(SP)
        JSRFP
```

```
        .ENDM

        .MACRO  FX2C
        MOVE.W  #FFCOMP+FOX2Z,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Miscellaneous operations applying only to extended
; operands.  The input operand is overwritten with the
; computed result.
;-------------------------------------------------------------

; Square root.
        .MACRO  FSQRTX
        MOVE.W  #FOSQRT,-(SP)
        JSRFP
        .ENDM

; Round to integer, according to the current rounding mode.
        .MACRO  FRINTX
        MOVE.W  #FORTI,-(SP)
        JSRFP
        .ENDM

; Round to integer, forcing rounding toward zero.
        .MACRO  FTINTX
        MOVE.W  #FOTTI,-(SP)
        JSRFP
        .ENDM

; Set the destination to the product:
;  (destination) * 2^(source)
; where the source operand is a 16-bit integer.
        .MACRO  FSCALBX
        MOVE.W  #FFINT+FOSCALB,-(SP)
        JSRFP
        .ENDM

; Replace the destination with its exponent, converted to
; the extended format.
        .MACRO  FLOGBX
        MOVE.W  #FOLOGB,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Non-arithmetic sign operations on extended operands.
;-------------------------------------------------------------

; Negate.
        .MACRO  FNEGX
```

```
        MOVE.W  #FONEG,-(SP)
        JSRFP
        .ENDM

; Absolute value.
        .MACRO  FABSX
        MOVE.W  #FOABS,-(SP)
        JSRFP
        .ENDM

; Copy the sign of the destination operand onto the sign of
; the source operand.  Note that the source operand is
; modified.
        .MACRO  FCPYSGNX
        MOVE.W  #FOCPYSGN,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; The nextafter operation replaces the source operand with
; its nearest representable neighbor in the direction of the
; destination operand.  Note that both operands are of the
; the same format, as specified by the usual suffix.
;-----------------------------------------------------------
        .MACRO  FNEXTS
        MOVE.W  #FFSGL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTD
        MOVE.W  #FFDBL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTX
        MOVE.W  #FFEXT+FONEXT,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; The classify operation places an integer in the
; destination.  The sign of the integer is the sign of the
; source.  The magnitude is determined by the value of the
; source, as indicated by the equates.
;-----------------------------------------------------------
FCSNAN          .EQU    1       ; signaling NAN
FCQNAN          .EQU    2       ; quiet NAN
FCINF           .EQU    3       ; infinity
FCZERO          .EQU    4       ; zero
FCNORM          .EQU    5       ; normal number
FCDENORM        .EQU    6       ; denormal number
```

```
        .MACRO  FCLASSS
        MOVE.W  #FFSGL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSD
        MOVE.W  #FFDBL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSX
        MOVE.W  #FFEXT+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSC
        MOVE.W  #FFCOMP+FOCLASS,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; These four operations give access to the floating point
; state (or environment) word and the halt vector address.
; The sole input operand is a pointer to the word or address
; to be placed into the arithmetic state area or read from
; it.
;-----------------------------------------------------------
        .MACRO  FGETENV
        MOVE.W  #FOGETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETENV
        MOVE.W  #FOSETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FGETTV
        MOVE.W  #FOGETTV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETTV
        MOVE.W  #FOSETTV,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Both FPROCENTRY and FPROCEXIT have one operand -- a
; pointer to a word.  The entry procedure saves the current
; floating point state in that word and resets the state
; to 0, that is all modes to default, flags and halts to
; OFF.  The exit procedure performs the sequence:
```

```
;       1. Save current error flags in a temporary.
;       2. Restore the state saved at the address given by
;               the parameter.
;       3. Signal the exceptions flagged in the temporary,
;               halting if so specified by the newly
;               restored state word.
; These routines serve to handle the state word dynamically
; across subroutine calls.
;-----------------------------------------------------------
        .MACRO  FPROCENTRY
        MOVE.W  #FOPROCENTRY,-(SP)
        JSRFP
        .ENDM

        .MACRO  FPROCEXIT
        MOVE.W  #FOPROCEXIT,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; FSETXCP is a null arithmetic operation which stimulates
; the indicated exception.  It may be used by library
; routines intended to behave like elementary operations.
; The operand is a pointer to an integer taking any value
; between FBINVALID and FBINEXACT.
; FTESTXCP tests the flag indicated by the integer pointed
; to by the input address.  The integer is replaced by a
; Pascal boolean (word $0000=false, $0100=true)
;-----------------------------------------------------------
        .MACRO  FSETXCP
        MOVE.W  #FOSETXCP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FTESTXCP
        MOVE.W  #FOTESTXCP,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; WARNING: PASCAL ENUMERATED TYPES, LIKE THOSE OF THE
; DECIMAL RECORD, ARE STORED IN THE HIGH-ORDER BYTE OF THE
; ALLOCATED WORD, IF POSSIBLE.  THUS THE SIGN HAS THE
; INTEGER VALUE 0 FOR PLUS AND 256 (RATHER THAN 1)
; FOR MINUS.
; BINARY-DECIMAL CONVERSION:  The next routines convert
; between a canonical decimal format and the binary format
; specified.  The decimal format is defined in Pascal as
;
;   CONST
;       SIGDIGLEN = 20;
;
```

```
;   TYPE
;       SigDig  = string [SIGDIGLEN];
;       Decimal = record
;                     sgn : 0..1;
;                     exp : integer;
;                     sig : SigDig
;                 end;
;
; Note that Lisa Pascal stores the sgn in the high-order
; byte of the allotted word, so the two legal word values
; of sgn are 0 and 256.
;-----------------------------------------------------------


;-----------------------------------------------------------
; Decimal to binary conversion is governed by a format
; record defined in Pascal as:
;
;   TYPE
;       DecForm = record
;                     style  : (FloatDecimal, FixedDecimal);
;                     digits : integer
;                 end;
;
; Note again that the style field is stored in the high-
; order byte of the allotted word.
;
; These are the only operations with three operands.  The
; pointer to the format record is deepest in the stack,
; then the source pointer, and finally the destination
; pointer.
;-----------------------------------------------------------
        .MACRO  FDEC2X
        MOVE.W  #FFEXT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2D
        MOVE.W  #FFDBL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2S
        MOVE.W  #FFSGL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2C
        MOVE.W  #FFCOMP+FOD2B,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
```

```
; Binary to decimal conversion.
;-----------------------------------------------------------
        .MACRO  FX2DEC
        MOVE.W  #FFEXT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2DEC
        MOVE.W  #FFDBL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2DEC
        MOVE.W  #FFSGL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2DEC
        MOVE.W  #FFCOMP+FOB2D,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
; Equates and macros for elementary functions.
;-----------------------------------------------------------
FOLNX           .EQU    $0000
FOLOG2X         .EQU    $0002
FOLN1X          .EQU    $0004
FOLOG21X        .EQU    $0006

FOEXPX          .EQU    $0008
FOEXP2X         .EQU    $000A
FOEXP1X         .EQU    $000C
FOEXP21X        .EQU    $000E

FOXPWRI         .EQU    $8010
FOXPWRY         .EQU    $8012
FOCOMPOUNDX     .EQU    $C014
FOANNUITYX      .EQU    $C016

FOSINX          .EQU    $0018
FOCOSX          .EQU    $001A
FOTANX          .EQU    $001C
FOATANX         .EQU    $001E
FORANDOMX       .EQU    $0020

        .MACRO  FLNX
        MOVE.W  #FOLNX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLOG2X
        MOVE.W  #FOLOG2X,-(SP)
```

```
JSRELEMS
.ENDM

.MACRO   FLN1X
MOVE.W   #FOLN1X,-(SP)
JSRELEMS
.ENDM

.MACRO   FLOG21X
MOVE.W   #FOLOG21X,-(SP)
JSRELEMS
.ENDM

.MACRO   FEXPX
MOVE.W   #FOEXPX,-(SP)
JSRELEMS
.ENDM

.MACRO   FEXP2X
MOVE.W   #FOEXP2X,-(SP)
JSRELEMS
.ENDM

.MACRO   FEXP1X
MOVE.W   #FOEXP1X,-(SP)
JSRELEMS
.ENDM

.MACRO   FEXP21X
MOVE.W   #FOEXP21X,-(SP)
JSRELEMS
.ENDM

.MACRO   FXPWRI
MOVE.W   #FOXPWRI,-(SP)
JSRELEMS
.ENDM

.MACRO   FXPWRY
MOVE.W   #FOXPWRY,-(SP)
JSRELEMS
.ENDM

.MACRO   FCOMPOUNDX
MOVE.W   #FOCOMPOUNDX,-(SP)
JSRELEMS
.ENDM

.MACRO   FANNUITYX
MOVE.W   #FOANNUITYX,-(SP)
JSRELEMS
.ENDM

.MACRO   FSINX
```

```
        MOVE.W  #FOSINX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FCOSX
        MOVE.W  #FOCOSX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FTANX
        MOVE.W  #FOTANX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FATANX
        MOVE.W  #FOATANX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FRANDOMX
        MOVE.W  #FORANDOMX,-(SP)
        JSRELEMS
        .ENDM
;-----------------------------------------------------------
;-----------------------------------------------------------
;-----------------------------------------------------------
;-----------------------------------------------------------
```

Introduction

FP68K provides conversions between the extended floating-point format and three integer formats:

```
int16  -- 16-bit two's complement
int32  -- 32-bit two's complement
comp64 -- 64-bit two's complement with the reserved value
          hexadecimal 8000000000000000.
```

One Pascal program, ITBATTERY.TEXT, tests all three conversions. This document describes how to use and, if necessary, modify the tests.

Compiling and running

ITBATTERY.TEXT uses the SANE interface (see the "High Level Interface" document, so it must be linked with the SANE object files, as well as with the usual nonarithmetic Pascal run-time libraries (e.g. *MPASLIB on Lisa). The program will simply run to completion, with a Pascal HALT if an error is found; execution time may run to 15 minutes on a Lisa system.

What is tested

Each of the integer formats is tested in two phases. First, a collection of specific extended numbers is converted to the integer format, with tests for correct rounding and signaling of the invalid exception when appropriate. Then a set of

```
integer --> extended --> integer
```

conversions is run, with the input and output integers compared for equality. In the case of int16, all 2^16 cases are run. However exhaustive testing of int32 and comp64 is infeasible so a loop is set up to do 2^16 tests from several starting points.

Introduction

The most important and rigorous set of tests of FP68K is the set of so-called IEEE test vectors. These tests, developed by the author while at Zilog, are used to test implementations of proposed standard P754. They were donated to the IEEE subcommittee 754 by Zilog Inc., and are now distributed by that subcommittee. The tests have undergone major revision within Apple, thanks especially to Jim Thomas of PCS.

Form of the tests

Each vector is an ascii string describing an operation, operands, and the result. For example, "lincl" is the floating-point number (of the format under consideration) next larger than 1. When "1" is subtracted from "lincl", the result is "lulpl", just one unit in the last place of 1. Written this way, the vectors may be applied to any floating-point format. The tests carefully inspect the nuances of rounding and exception handling. A document is under development to explain in detail the next release of the test vectors, scheduled for early 1983, after some last details of the standard are cleared up.

Files

The test vectors are contained in a family of files by the name of TVxxxx.2.TEXT and TWxxx.2.TEXT. The "2" refers to version 2 of the tests. (Version 1 was based on Draft 8.0 of the standard.) The file TLIST.TEXT is a list of the test file names to be used in any given run of the test. Pascal file TD68.TEXT with unit TD68FP.TEXT actually run the tests. These interface with FP68K exclusively through the SANE interface.

Introduction

Since the binary <--> conversions within FP68K approximate the mathematical identity operation, they lend themselves to certain types of self-testing. For example, if enough decimal digits are kept, then the conversion

binary --> decimal --> binary

is the identity mapping when results are rounded to nearest. The number of digits required turns out to be 9 for single and 17 for double. A similar test performs the first conversion rounding toward plus infinity and the second rounding toward minus infinity. In this case the final result may differ from the starting value by one unit in the direction of the latter rounding, so the program allows this discrepancy.

This document describes the test files and how they can be run. For details of the underlying error analysis (which is quite subtle) see the paper "Accurate Yet Economical Binary-Decimal Conversions" by J. Coonen.

Test programs

The test programs are:

IOS.TEXT
IOSF.TEXT
IOD.TEXT
IODF.TEXT
IONAN.TEXT
IOPSCAN.TEXT

The letter "S" and "D" distinguishes single and double tests. The IOS.TEXT and IOD.TEXT tests run with both rounding to nearest and the directed roundings. The "F" tests use fixed-format output rather than floating-format output for the intermediate decimal string. The IOPSCAN test is used to check the preformance of the printer and scanner used by SANE68, and included from file SAPSCAN.TEXT. The IONAN test checks the input and output conversion of some 20 stock NANs, and then allows the user to enter any decimal string to be converted to the three formats in three rounding modes. Neither IOPSCAN nor IONAN are self-checking; rather, the user must monitor their output.

The tests cover extreme intervals where the decimal numbers are sparsest and densest with respect to the binary numbers. Sparse intervals have the form $[10^N, 2^n]$ where the endpoints are nearly equal. Dense intervals have the corresponding form $[2^m, 10^M]$.

Running the tests

Each of the programs is compiled and run separately. The programs use the SANE interface. A test will HALT with a suitable diagnostic if the test

fails.

The single format cases are few enough that their tests can be run overnight. However, the double format cases will run essentially forever since the number of interesting cases is so great. A few overnight tests should be sufficient.

Background

The so-called I/O routines for scanning and printing floating-point numbers in decimal form are complicated by subtle numerical issues and nettlesome design decisions. For example, even the simplest, stripped-down conversion routines require over one-third the code space (about 1.3K) of the rest of the FP68K binary floating-point package. With a full parser and formatter, the conversion routines are much larger. And it is unclear whether full routines would be flexible enough for use in different language systems and I/O-intensive applications like Visi-Calc.

Where does the responsibility lie? This note argues that the core conversion routines, which are part of the arithmetic package, should be kept very simple. Above them -- somewhere in the system -- should be a full scanner and formatter available to languages and applications, but not forced upon them. This would lead to the most efficient use of code space and execution time.

The Sad Truth

Numerical I/O can be monstrous. Since each computer language has its own grammar for floating-point numbers and its own conventions for output format, it almost necessary for each language system on a computer to provide significant I/O support. Unfortunately, this may be layered upon the host system's I/O system. And it is not unusual (Apple III, for example) for a language compiler to use different conversion routines than the I/O system the compiled code utilizes.

In another case, designers of the UNIX operating system attempted to route all conversion through the routines atof(), ascii to floating, and the pair ecvt(), fcvt() for floating and fixed conversion to ascii. But even this fairly clean design has led to VERY complicated software shells around atof, ecvt, and fcvt. Numerical accuracy aside, the complexity of just the character hacking is forbidding.

One problem with the UNIX design lies in its failure to properly divide responsibility for the distinct processes involved in conversion, namely:

1. Recognize floating-point strings (in compilers, ...)
2. Translate strings to numerical values.
3. Determine which output format (fixed or floating) is appropriate for a given value.
4. Translate a numerical value to a string.

The utilities atof() and ecvt() provide items 2 and 4. Item 3, printing a number in its "nicest" form is provided in rough form through ecvt(). But recognizing strings is left to each language compiler's lexical scanner. Unfortunately, after a scanner has parsed a floating decimal string, it passes it along to atof() where it is parsed once more.

A Proposal for Change

(1) Support, at the arithmetic level, conversions between each of the available binary floating-point types and one decimal structure describable in Pascal as:

```
{*
** Low-level format of the floating decimal value:
**   (-1)^sgn * 10^exp * dig
** The constant DECSTRLEN is 20 for MAC and 28 for III, since
** the latter uses very high precision for intermediates.
*}
type
    DecStr  = string[DECSTRLEN]
    Decimal = record
        sgn: 0..1;       {0 for +, 1 for -}
        exp: integer;
        sig: DecStr
    end;
```

(2) Rigidly specify the format of Decimal.sig for decimal to binary conversions, relying upon a lexical scanner to perform the first parse. The decimal value would depend upon the first character of decrec.dig:

```
'I'          --> infinity
'Nxxx...x'   --> NAN, with optional ascii hex digits 0-9, A-F, a-f
'0'          --> zero
'ddd...d'    --> string of digits stripped of leading and trailing zeros
```

The digit string would never be more than 20 digits long. If present, the 20-th digit would indicate the absence of nonzero trailing digits beyond the 20-th (to aid in correct rounding).

(3) Specify decimal output format through a structure like the Pascal:

```
{*
** Output format specifier.
*}
type
    DecForm = record
        style: (float, fixed);
        digits: integer
    end;
```

For "float" conversions, digits is the number of significant digits to be delivered in Decimal.sig. For "fixed" conversions, count is the number of fraction digits to be converted (a negative count suppresses conversion of low-order integer digits).

Sometimes it is desired to print a number in the nicest form possible for a given field width. For example, the string "1.23456789" conveys much more information in 10 characters than does "1.2345e+04". Such conversions are

discussed in the next section.

(4) Provide a scanner and formatter which, if not of most general use, provide models that can be tailored to a particular application. Samples are built into the implementation section of the SANE Pascal interface; they are contained in the file SAPSCAN.TEXT.

Binary --> Decimal

The family of routines:

S2Dec
D2Dec
X2Dec
C2Dec

provide conversions to the Decimal record format described above. Special cases are keyed by the first character of Decimal.sig:

'0' : zero
'I' : infinity
'N' : not-a-number, followed by optional ascii hex digits; if there are fewer than four, they are padded on the left with 0's.
'?' : overflow of fixed-style format

These must be used with a formatter to produce output strings.

The family:

S2Str
D2Str
X2Str
C2Str
uses the built-in formatter, Dec2Str, to generate ascii string output.

Decimal --> Binary

These conversions are povided by the complementary set of procedures: Dec2S, Dec2D, Dec2X, Dec2C, and Str2S, Str2D, Str2X, Str2C. In the case of the Dec2* conversions, the first character of Decimal.sig indicates special cases as noted above for *2Dec conversions.

Infinity and NAN conversions

Infinity is printed and read as a string of sign characters, "+++++" or "-----".

On input, NANs have the general form NAN'xxxx:yyy...y'. The x's and y's should be ascii hex digits: 0-9, A-F, a-f. The string portion following NAN may be omitted. The x's are padded on the LEFT with 0's to width 4. The y's are padded on the RIGHT with 0's to the width of the NAN's significant bit

field.

On output, NANs will be printed in the same format. Leading x=0 and trailing y=0 are omitted, but at least one x is printed. If all y=0, then the colon and the y field is dropped.

Any unrecognizable string is converted to a NAN.

Background

Applications like accounting spreadsheets typically need to display floating-point values in decimal form within a field of fixed width. For maximum readability, the output should be in integer or fixed-point format if possible, with floating-point format as a last resort. The idea is to avoid listing small integers in the abominable form 0.100000000000E1 reminiscent of computing in the McCarthy era.

The problem

Given a binary floating-point number X and an ascii field F, display X in the "nicest", most informative way within F.

A proposal

1. If X may be displayed in a subfield of F, pad X on the left with blanks.

2. Display the sign of X only if it is '-'.

3. If X is an integer and F is wide enough to accommodate X, then display X as an integer, without a trailing '.'.

4. Else if X has nonzero integer and fraction parts and F is wide enough to accommodate at least the integer part of F and its trailing '.', then display X in the fixed-point form ZZZZ.YYYY with as many fraction digits as F will accommodate, up to a maximum of 17 significant digits.

5. Else if |X| < 1 and F is wide enough that X may be displayed in the form 0.00000ZZZZZ with no more 0s just to the right of the decimal point than digits following those 0s, then display X in that fixed-point form with up to 17 significant digits.

6. Finally, if all the above fail, then display X in the floating-point form Z.ZZZZZEYYY with as many significant digits up to 17 as F will accommodate, taking into account the width of the exponent field, including its possible sign. Display the sign of the exponent field only if it is '-'.

An implementation

The above choices depend on detailed knowledge of the magnitude of X. For example, in producing floating-point output, it is necessary to know the number of spaces that will be occupied by the decimal exponent (with sign, it could be 1 to 5) in order to know how many significant digits to which to round X. In the worst case, this could mean several calls upon the low-level conversion routine until the proper output is finally obtained.

One easy way to bypass these problems, and keep the fundamental conversion routine simple, is perform the binary -> decimal conversion in two stages. First convert the binary value X to the SANE decimal form:

```
type
    DecStr  = string[DECSTRLEN];   { length is 20 for MAC }
    Decimal = record
        sgn: integer;   {0 for +, nonzero for -}
        exp: integer;   {as though decimal is at the right of...}
        sig: DecStr
    end;
```

If the conversion is performed with rounding toward 0, conversion style = float, and digit count = 19, and if the inexact exception flag is cleared before the conversion, then the 19-digit result may be correctly rounded to the desired width after the ultimate output format is determined. Since no more than 17 digits will ever be displayed (recall that 17 digits suffice to distinguish double format binary numbers), the 19 digits together with the inexact exception flag permit correct rounding.

The second step of the conversion decides, on the basis of the intermediate decimal form, which format is appropriate. Then the decimal value is rounded (in decimal!) and displayed as desired. Note that this scheme has as a happy byproduct the ability to round in the (time-honored?) "add half and chop" manner that is unavailable within Apple arithmetic itself.

In the interest of compatibility of the floating-point arithmetic on Apples II/III and Mac/(Lisa?), the following GRITTY DETAILS were discussed on June 29. This is an update on the decisions made then.

1. Distinguishing signaling and quiet NANs: use the leading fraction bit, 0-quiet and 1-signaling.

2. Explicit leading bit of extended NANs and INFs: ignore it, that is decide whether NAN or INF on the basis of the fraction bits only.

3. Quiet NANs have an 8-bit "indicator field" marked by stars in the following extended format hex mask: XXXX XX** XXXX XXXX XXXX. This byte is the low half of the leading word of significant bits. The interpretation of the field is as given page 70 of Apple III Pascal, volume 2, subject to enhancements.

4. When two quiet NANs are operands to the operations +, -, *, /, and REM, one or the other of the NANs is output. When the indicator fields differ, the NAN with the larger indicator field prevails; ties are broken arbitrarily.

5. True to the standard, the sign of an output NAN is unspecified.

6. Signaling NANs precipitate the invalid operation exception when they appear as operands.

7. Underflow is tested before rounding. CHANGE: this may change depending on P754 deliberations in the late summer of '82

8. Projective INF follows the same rules of signs as affine INF. The ABSOLUTELY ONLY differences between affine and projective modes are: the UNORDERED-ness of projective INF in comparisons with finite numbers, and the invalid operation exception that arises from the sum of two projective INFs with the same sign. CHANGE: projective mode may be removed from P754 in late summer '82.

9. Treatment of unnormalized extended numbers may differ between systems. 68K implementations will normalize all such, as is expected of the Motorola and Zilog chips. 6502 implementations may support the ANTIQUE warning mode in preliminary releases, though it may never be documented for general consumption.

10. The bottom of the extended exponent range is as in the Motorola and Zilog implementations (as opposed to Intel). That is, there is no redundancy between the bottom two exponent values.

11. The exponent bias in extended is hex 3FFF, which is used by Intel, Zilog, and Motorola. Motorola may insert a word of garbage between the sign/exp fields and the significant bits in order to have a 96-bit data type.

12. Comparisons return results according to local system convenience. 68K: return from the floating-point software with the CPU condition codes set appropriately for a conditional branch. 6502: for lack of a rich set of conditional branches, let the comparison operation be a family of boolean tests like "Is X <= Y?" The difference between the two systems should be

hidden well below the high-level language interfaces.

13. Auxiliary functions: relegate functions like nextafter() to the system numerical library rather than putting them in the arithmetic engine.

14. The data types specified by SANE are int16, comp32, comp64, f32, f64, x80. 68K systems will require int32 as well.

15. Is the Pascal assignment: X := Y; an arithmetic operation when both X and Y are variables of the same floating-point format? Or is a straight byte copy sufficient? This is really a language issue -- one left dangling by the standard. The arithmetic units, if asked to perform a floating move between two floating entities of the same format, will perform a full-blown arithmetic operation. This will cause side effects if the floating value is a signaling NAN (invalid operation) or a denormalized number (underflow).

16. Precision control is supported by 6502 and 68K packages, but it is available only through assembly language -- it is intended only for SPECIAL applications anyway. Precision control implies range control, too.

17. There is no "integer overflow" exception.

18. Traps? These are so system-dependent there is no hope for perfect consistency. So the issue is left as a local matter for each system. The question relevant to each floating-point engine is: "What information will I be required to spew out in case of a trap?"

2. Data Types

The arithmetic supports the following data types. All are specified in SANE except for int32 and decimal. Int32 is included for convenience in 68K environments, where 32-bit integers are common. Through the decimal type the package provides the basis for the binary<->decimal conversions required by languages and the I/O system.

int16	--	16-bit two's-complement integer
int32	--	32-bit two's-complement integer
comp64	--	64-bit integer, with one reserved operand value
f32	--	32-bit single floating-point
f64	--	64-bit double floating-point
x80	--	80-bit extended floating-point
decimal	--	ascii digit string with integer sign and exponent

3. Arithmetic Operations

These operations apply to floating-point operands:

+, -, *, /, SQRT, REMAINDER, COMPARE,
ROUND TO INTEGER, TRUNCATE TO INTEGER, LOGB, SCALB,
ABSOLUTE VALUE, NEGATE, COPYSIGN, NEXAFTER, CLASS

Except for COMPARE, each produces a floating-point result. COMPARE sets the CPU flag bits according to the two operands. Besides its floating-point result, REMAINDER returns the sign and four least significant bits of its integer quotient in the CPU flags (a very useful trick for argument reduction in the transcendental functions). LOGB replaces a number by is unbiased exponent, in floating form; SCALB scales a number by an integer power of 2.

4. Format Conversions

intXX	<-->	extended	
comp64	<-->	extended	
floating	<-->	floating	(one operand must be extended)
decimal	<-->	extended	

5. Internal Architecture

The package provides 2-address memory to memory arithmetic operations of the form

<op> DST --> DST and
SRC <op> DST --> DST

where DST and SRC are the destination and source operands, respectively. The DST operand is always in the extended format. The conversions have the form:

SRC --> DST

where at least one of SRC and DST is a floating-point format. The package also provides a few support functions in connection with the floating-point error flags and modes.

Extended format results may be coerced to the PRECISION and RANGE of the single or double formats, on an instruction by instruction basis. Then subsequent operations are able to take advantage of the trailing zeros to improve performance. This feature is provided to expedite special-purpose applications such as graphics and is not intended for general use. Only under certain circumstances will it actually obtain a speed advantage, rather than a DISADVANTAGE, since the package is built to do extended arithmetic.

6. External Access

The package is re-entrant, position-independent code, which may be shared in multi-process environments. It is accessed through one entry point, labeled FP68K. Each user process has a static state area consisting of one word of mode bits and error flags, and a two-word halt vector. The package allows for different access to the state word in one-process (Mac) and multi-process (Lisa) environments.

The package preserves all CPU registers across invokations, except that REMAINDER modifies D0. It modifies the CPU condition flags. Except for binary-decimal conversions, it uses little more stack area than is required to save the sixteen 32-bit CPU registers. Since the binary-decimal conversions themselves call the package (to perform multiplies and divides), they use about twice the space of the regular operations.

7. Calling Sequence

A typical invokation of the package will consist of a sequence of four 68K assembly instructions:

```
PEA      <source address>        ;"Push Effective Address"
PEA      <destination address>   ;"Push Effective Address"
MOVE.W   <opword>, -(SP)         ;"Push" operation word
JSR      FP68K                   ;"Call" the package
```

(If FP68K resides in system memory, the JSR may be replaced by an A-line trap opcode.) Other calls will have more or fewer operand addresses to push onto the stack. The opword is the logical OR of two fields, given here in hexadecimal:

```
"non-extended" operand format, bits 3800:
    0000 -- x80
    0800 -- f64
    1000 -- f32
    1800 -- ILLEGAL
    2000 -- int16
```

2800 -- int32
3000 -- comp64
3800 -- ILLEGAL

arithmetic operation code, bits 001F:
0000 -- add
0002 -- subtract
0004 -- multiply
0006 -- divide
0008 -- compare
000A -- compare and signal invalid if UNORDERED
000C -- remainder
000E -- floating, intxx, comp64 --> extended convert
0010 -- extended --> intXX, comp64, floating convert
0012 -- square root
0014 -- round to integer in floating format
0016 -- truncate to integer in floating format
0018 -- scale by integer power of 2
001A -- replace by unbiased exponent
001C -- classify the floating input
001E -- ILLEGAL

0001 -- put state word
0003 -- get state word
0005 -- put halt vector
0007 -- get halt vector
0009 -- decimal --> floating convert
000B -- floating --> decimal convert
000D -- negate
000F -- absolute value
0011 -- copy sign
0013 -- nextafter
0015 -- set exception
0017 -- procedure entry protocol
0019 -- procedure exit protocol
001B -- test exception
001D and 001F are ILLEGAL

8. Comparisons

In this arithmetic, comparisons require some extra thought. The trichotomy rule of the real number system -- that two numbers are related as LESS, EQUAL, or GREATER -- is violated by the NANs, which compare UNORDERED with everything, even themselves. So it is necessary for floating-point comparisons to use the CPU condition codes in a way that seems surprising at first blush:

RELATION	FLAGS: X N Z V C
LESS	1 1 0 0 1
EQUAL	0 0 1 0 0
GREATER	0 0 0 0 0
UNORDERED	0 0 0 1 0

This encoding leads to a very convenient mapping between the "floating-point conditional branches" and the CPU conditional branches. In the following table, the '?' refers to UNORDERED. The second column gives the name of the branch macro that provides the "floating branch" (see the "Assembler Support" document).

BRANCH CONDITION		MACRO NOTATION	CPU BRANCH
=		FBEQ	BEQ
<		FBLT	BCS
<, =		FBLE	BLS
>		FBGT	BGT
>, =		FBGE	BGE
?, <		FBULT	BLT
?, <, =		FBULE	BLE
?, >		FBUGT	BHI
?, >, =		FBUGE	BCC
?	(unordered)	FBU	BVS
<, =, >	(ordered)	FBO	BVC
?, <, >	(not equal)	FBNE	BNE
?, =		FBUE	BEQ / BVS
<, >		FBLG	BNE / BVC

Only in the last two instances, are two branches required.

The variant comparison instruction, that signals the invalid operation exception if its operands are UNORDERED, is useful in high-level languages since P754 (and SANE) require that certain UNORDERED comparisons be marked invalid.

Further discussion of the language issues of comparisons may be found in "Comparisons and Branching" by Jerome Coonen.

9. Binary-Decimal Conversions

The package provides conversion functions intended to be used in conjunction with scanners and formatters peculiar to the user environment. For decimal to binary conversions, the input parameters are:

```
address of Pascal decimal structure:
    record
        sgn : 0..1;
        exp : integer;
        sig : string[20]
    end;

address of target floating variable
```

The format (f32, f64, x80) of the target is given in the opword. For binary to decimal conversions, the input paramaters are:

```
address of format structure:
    record
        style : (FloatDecimal, FixedDecimal);
        digits: integer
    end;

address of source floating variable

address of decimal structure:
        sign
        exponent
        ascii string of significant digits
```

The interpretation of the latter format element depends on the style of the conversion. For fixed conversions, the digit count gives the number of fraction digits desired (which may be negative). For float conversions, the digit count gives the number of significant digits desired.

Free format binary --> decimal conversions, which display numbers in the "nicest" format possible within given field width constraints, are supported in software, using the float style of conversion. Nice conversions are handy in applications like accounting spreadsheets where tables of numbers are displayed. See the "Binary-Decimal Conversion" document for details. The SANE interface gives details about the decimal format.

10. The State Area

Each user of the package has three words of static floating-point state information. All accesses to the state should be made through the four state operations. The state consists of:

```
modes and flags word:
        8000 -- unused

        6000 -- rounding direction:
                0000 -- to nearest
                2000 -- toward +INF
                4000 -- toward -INF
                6000 -- toward zero (chop)

        1F00 -- error flags, from high to low order:
                1000 -- inexact result
                0800 -- division by zero
                0400 -- floating overflow
                0200 -- floating underflow
                0100 -- invalid operation


        0080 -- rounding of last result
```

```
                    0000 -- not rounded up in magnitude
                    0080 -- rounded up in magnitude

            0060 -- precision control:
                    0000 -- extended
                    0020 -- double
                    0040 -- single
                    0060 -- ILLEGAL

            001F -- halt enables, correspond to error flags

    halt vector:
            32-bit address of alternate exit from package
```

11. Halts

When an error arises for which the corresponding halt is enabled, a trap is taken through the vector in the floating-point state area. The halt routine is called as a Pascal procedure of the form

```
PROCEDURE MyHalt(VAR r: fpRegs; op3, op2, op1: fpPtr; opcode: integer);
where
TYPE
  fpRegs = RECORD BEGIN
             FPRCCR,              { 68000 CCR register }
             FPRDOHI,             { high word of register DO }
             FPRDOLO              { low word of register DO }
           END;
  fpPtr = ^Extended;              { but may be pointer to any type }
```

The only way to return to the package from a halt is to initiate a new floating-point operation. There is no way to resume execution of the halted operation.

The state-related operations never halt. The binary-decimal conversions do not halt, though the individual operations they employ (such as multiplication to form 10^N for some integer N) might halt.

12. Other Pseudo-Machines

The package is simple and general enough to be the basis for pseudo-machines with register architectures like the 68881 or the Z8070 or with an evaluation stack like the Intel 8087. What is needed is simply the mechanism to compute addresses in the register file or stack (and check for internal consistency), and the set of functions required to manipulate that isolated data file (e.g. duplicate the top stack element, negate a register).

13. Arithmetic Abuse

The package is designed to be as robust as possible but it is not bullet-proof, since it is specified to modify the stack. If the user passes

illegal addresses, a memory fault may arise when the package attempts to access the operands. And if the user passes the wrong number of address operands, then in general the stack will be irreparably damaged. Operation is undefined if ILLEGAL values are used in the opword parameter.

14. Size and Performance

FP68K is about 4000 bytes long. On a 4mhz system it executes the simplest arithmetic operations in about 0.4ms and requires just over 1.0ms for a full extended multiply. Divide and square root are longer yet.

Comparative timings show that, for double format operations, FP68K is just faster than the AMD 9512 on Lisa and is about twice twice as fast as the Motorola 68341 code. For single format operations, FP68K is about half as fast as the Lisa single-only package, which is just slower than the 9512.

15. Floating-Point at a Glance

Figure 1 at the end of this document illustrates the basic control of flow in the execution of the floating-point package. The figure is followed by a list of observations on the behavior of the package, and of IEEE arithmetic in general.

1. The package has a single entry point.

2. The package has two exit points, one for normal subroutine returns and one for halts through a vector.

3. Three classes of operations are distinguished: arithmetic operations, binary-decimal conversions, and accesses to the state word and halt vector.

4. The not-a-number symbols, NANs, are detected at the start of each operation. Of them, signaling NANs are the most virulent; they always trigger the invalid operation exception. Quiet NANs propagate through operations; a precedence rule determines which is output if two are input.

5. Invalid operations always result in a quiet NAN output. In the case of the discrete types INT16, INT32, COMP64, the output value is all zero bits except for a leading one bit (that is, 100000...). Floating-point NANs contain an error code to indicate their origin (such as 01 for square root of a negative number).

6. When the input operands are unpacked, the special cases 0, FNZ (finite nonzero number), and INF (infinity) are detected. This expedites special cases such as

+INF + FNZ --> +INF

7. When 0 or INF results from a trivial operation like the example above, no further processing is required before the value is packed. All nontrivial floating-point results are subject to precision and range coercion to assure that they fit in the intended destination.

8. Integer results are subject to coercion to detect overflow.

9. Floating-point NAN results are coerced by chopping them to the precision of the destination, and checking that a legitimate value results.

10. Comparisons require special care, since they produce no results but rather modify the CPU condition-code register. Comparisons, even when NANs are involved, must bypass the coercion steps.

Introduction

This is a brief guide to the program FP68K, a software implementation of proposed IEEE standard P754 (Draft 10.0) for binary floating-point arithmetic. This guide is intended to aid a programmer wishing to understand the workings of FP68K.

The code

The software is in the assembly language of the Motorola MC68000, following the Apple "TLA" syntax of the Lisa assembler. FP68K is non-self-modifying, position-independent code. It has no local data area, that is it uses dynamically allocated stack area for all of its temporaries. FP68K is one large subroutine whose single entry point has the name FP68K.

The code is separated into the functionally distinct files:

FPDRIVER.TEXT	-- "includes" the other files...
FPEQUS.TEXT	-- defines set of named constants
FPCONTROL.TEXT	-- organizes the flow of control
FPUNPACK.TEXT	-- unpack input operands to intermediate format
FPADD.TEXT	-- add and subtract
FPMUL.TEXT	-- multiply
FPDIV.TEXT	-- divide
FPREM.TEXT	-- remainder
FPCMP.TEXT	-- compare
FPSQRT.TEXT	-- square root
FPCVT.TEXT	-- floating <--> floating,integer conversions
FPSLOG.TEXT	-- logb, scalb, and class appendix functions
FPNANS.TEXT	-- handle "Not A Number" symbols
FPCOERCE.TEXT	-- post-normalize, round, check over/underflow...
FPPACK.TEXT	-- pack result to storage format
FPODDS.TEXT	-- non-arithmetic operations
FBD2B.TEXT	-- decimal --> binary conversion
FBB2D.TEXT	-- binary --> decimal conversion
FBPTEN.TEXT	-- computes 10^N for nonnegative integer N

As noted, FPDRIVER.TEXT is a short file which simply includes the other files between the ".PROC" header and ".END" trailer.

Assembling FP68K

Assemble the file FPDRIVER.OBJ to produce the FP68K object file.

The one system dependency of FP68K is its access of the floating-point state area, as discussed in the "System Implementor's Guide". Near the top of FPCONTROL.TEXT is the code which pulls the address of the the 3-word state area into register A0. This code will typically require modification when FP68K is moved to a new system. The well-marked comment within FPCONTROL.TEXT indicates the different access schemes systems might use. If the state area

is to be located using a constant defined in a public "include" file, then that file should be included within FPDRIVER.TEXT. See the comment there for details.

Other than its access to the state area, FP68K is intended to system-independent and should not be tailored recklessly.

Control flow

There are three fundamentally distinct classes of operations performed by FP68K: basic arithmetic, binary-decimal conversions, and manipulations of the floating-point state area. The last of these, namely reading and writing the state word and the halt vector, is trivial and needs no explanation beyond the simple code contained in FPODDS.TEXT.

The basic arithmetic operations are illustrated in the flow chart at the end of this note. The chart is marked to distinguish the function of the various files listed above.

The binary-decimal conversions are quite different from the basic operations, and are not described by the basic flow chart. The conversions might better be thought of as subroutines which have been implemented within FP68K as a matter of architectural convenience. The conversions invoke FP68K itself to perform various basic operations like multiply and divide. The binary-decimal algorithms are described in considerable detail in the attached paper "Accurate, Yet Economical Binary-Decimal Conversions" by J. Coonen.

Exponent calculations

FP68K manipulates exponents in a way that might seem surprising at first glance. The P754 extended format, on which all FP68K arithmetic is based, has a 1-bit sign, 15-bit exponent, and a 64-bit significand. However, the actual exponent range is not 0 to 32767 (biased by 16383) as the 15-bit exponent field would suggest. Rather, it is -63 to 32767 because of the presence of tiny denormalized numbers; this is "just a little bit" beyond the stated 15-bit range. (See the attached paper "Underflow and the Denormalized Numbers" by J. Coonen for a discussion of tiny values in P754 arithmetic.)

Because the operations multiply and divide require the addition and subtraction, respectively, of operand exponents in forming their intermediate results, the implementor typically expects to have one extra exponent bit for intermediate calculations. Thus for P754 extended format calculations, there is need for "just a little bit" beyond 16 exponent bits. This elusive 17-th bit is discussed in yet another attached paper, "Are 17 Exponent Bits Too Many?" It is shown there that 16 bits suffice, if care is taken to perform some extra tests in the right places.

On the 68000 it turns out to be convenient to perform exponent calculations in the ADDRESS REGISTERS — with a full 32 bits. The address registers provide just the right functionality: add, subtract, and compare. And since floating-point arithmetic is computation-intensive on a small data set, only a few of the address registers are actually needed for addresses.

Finally, 16-bit constants like the exponent bias may be added into the 32-bit exponents with a 2-word instruction, since for "address" calculations the constant is first sign-extended out to a full 32 bits.

Bit field encodings

This section describes the various bit fields used by FP68K. Some of them, like the opcode and the state word, are visible to programs invoking FP68K. Others, like the rounding and sign bits, are local to FP68K.

The OPCODE is the last word pushed on the stack before calling FP68K. It is composed of the fields:

```
3800 -- "non-extended" operand format:
     0000 -- x80
     0800 -- f64
     1000 -- f32
     1800 -- ILLEGAL
     2000 -- int16
     2800 -- int32
     3000 -- comp64
     3800 -- ILLEGAL

07E0 -- must be zero

001F -- operation code:
     0000 -- add
     0002 -- subtract
     0004 -- multiply
     0006 -- divide
     0008 -- compare
     000A -- compare (invalid if UNORDERED)
     000C -- remainder
     000E -- x80, f64, f32, int16, int32, comp64 --> x80
     0010 -- x80 --> x80, f64, f32, int16, int32, comp64
     0012 -- square root (in x80)
     0014 -- round to integer (in x80)
     0016 -- truncate to integer (in x80)
     0018 -- scale by unbiased power of 2
     001A -- replace by unbiased exponent
     001C -- classify the floating input
     001E -- ILLEGAL

     0001 -- put state word
     0003 -- get state word
     0005 -- put halt vector
     0007 -- get halt vector
     0009 -- decimal --> floating convert
     000B -- floating --> decimal convert
     000D -- negate
     000F -- absolute value
     0011 -- copy sign
     0013 -- nextafter
```

0015 — set exception
0017 -- procedure entry protocol
0019 — procedure exit protocol
001B — test exception
001D and 001F are ILLEGAL

The STATE word is static data that perseveres across calls to FP68K. As such, it must live in an area outside FP68K, defined by the host system. Typically the state word (and the halt vector, which is a 32-bit address) will live in the system's "per-process data area", perhaps a fixed location in memory or a fixed offset from some reserved address register. Although the STATE word is directly available to the programmer, typical access will be through an intermediate layer of software (available, say, in a Pascal unit) that insulates the programmer from the details of the actual bit encodings. The STATE word is composed of the fields:

8000 — unused

6000 — rounding mode:
- 0000 — to nearest
- 2000 — toward +INF
- 4000 — toward -INF
- 6000 — toward 0 (chop)

1F00 — error flags:
- 1000 — inexact result
- 0800 — division by zero
- 0400 — floating overflow
- 0200 — floating underflow
- 0100 — invalid operation

0080 — rounding of last result
- 0000 — not rounded up in magnitude
- 0080 — rounded up in magnitude

0060 — precision control:
- 0000 — extended
- 0020 — double
- 0040 — single
- 0060 — ILLEGAL

001F — exception halt enables:
(correspond to error flags above)

After preliminary decoding in FPCONTROL.TEXT, the OPCODE is expanded out into the following 16-bit form:

8000 — nonzero iff result has single precision and range

4000 — nonzero iff result has double precision and range

3800 — source operand format:

(same encoding as in OPCODE)

0700 -- destination operand format:
(same encoding as in OPCODE)

0080 -- nonzero iff destination operand is input

0040 -- nonzero iff source operand is input

0020 -- nonzero iff destination operand is output

001E -- operation code:
(same encoding as in OPCODE but with low bit 0)

0001 -- nonzero iff two-address operation

The ROUND BITS, known as "guard", "round", and "sticky" in documentation about P754, are kept in a 16-bit word. Roughly speaking, the guard and round bits are the two bits beyond the least significant bit of the intermediate result, and the sticky bit is the logical Or of all bits thereafter. The sticky bit is necessary to implement the rounding modes of P754. The ROUND BITS are kept as:

8000 -- guard bit
4000 -- round bit
3F00 -- 6 extra round bits
00FF -- sticky bits

The reason for keeping an entire byte of sticky bits lies in the 68000 instruction set. The archetype operation involving the sticky bit is the right-shift. Any time a bit is shifted off the low end of the sticky "byte", it must be logically Or-ed back into sticky. This is done with the 68000 "SCS" instruction, which sets a given byte to all 1s if the carry bit is set, and clears the byte to 0 otherwise. Typically, a bit is shifted off to the right, it is SCS-ed into an auxiliary byte, and that byte is Or-ed into the sticky byte. Although this is the typical use of the sticky byte, the programmer should not assume that the sticky byte is always either all 0s or all 1s. Sometimes, such as in the right shift after a carry-out in ADD/SUB, the logical Or will be omitted since it is known that if a 1 was shifted out of the sticky byte there will necessarily be another 1 left in sticky.

The operands' SIGNS are kept together in a byte as follows:

80 -- source operand sign
40 -- destination operand sign
20 -- Exclusive Or of the two operands' signs
1F -- unused, but not necessarily zero

If there is just one input operand, its sign is in the high order bit. The Exclusive Or is computed just once, at the start of every arithmetic operation. Not only is it required for many common operations (+, -, *, /, REM, CMP), but it is costly in time and space because of the inefficacy of the

68000 bit instructions, so it is worthwhile to implement the code sequence just once.

The CCR (condition code register) bits of the 68000 are modified by every arithmetic operation, though only the compare instructions leave them in a well defined state. A CCR word is maintained by FP68K:

```
FFE0 -- unused, forced to 0
0010 -- X = Extend
0008 -- N = Negative
0004 -- Z = Zero
0002 -- V = Overflow
0001 -- C = Carry
```

The compare operations encode their results as follows:

RELATION	FLAGS: X N Z V C
LESS	1 1 0 0 1
EQUAL	0 0 1 0 0
GREATER	0 0 0 0 0
UNORDERED	0 0 0 1 0

See the FP68K programmer's manual for the software applications of the CCR field.

Register usage

The key to the speed (such as it is) and compactness of FP68K is that its entire working data set may be held in the 68000 register file. Immediately upon entry, FP68K saves registers D0-D7, A0-A4 on the stack. Then the registers are loaded up as the operation proceeds. Several of the registers have a meaning that perseveres across nearly the entire instruction. The following list gives a rough idea of register usage:

```
D7 hi -- CCR word
D7 lo -- round bits
D6 hi -- opcode word
D6 lo -- error byte (hi) and sign byte (lo)
D5    -- low 32 source (later result) significant bits
D4    -- high 32 source (later result) significant bits
D3-D0 -- scratch area
```

A

```
 7     -- SP = stack pointer
A6     -- stack link pointer
A5     -- Mac globals poionter
A4     -- source (later result) exponent
A3     -- destination exponent
A2     -- low 32 destination significant bits
A1     -- high 32 destination significant bits
A0     -- pointer to 3-word state area
```


Of course, the arithmetic operations may be viewed as transformations of the register file. Following this view, a set of register maps are included at the end of this note. They are keyed to MILESTONES marked in the source code. The maps indicate register dependencies, and as such should aid in any modification of FP68K. Some maps simply indicate the state of the register file at a given point, and some indicate register use in a routine, such as the widely used right-shift procedure RTSHIFT.

For convenience the maps are printed on onion skin paper; a reference sheet slips under the map to fill in the register mask.

Register D0 is modified by the REMAINDER operation, in which case a partial integer quotient is returned in D0.W.

Stack usage

When called, FP68K assumes that the stack has the form:

```
ADDRESS 3 -- used for decimal format code only
ADDRESS 2 -- source pointer, if any
ADDRESS 1 -- destination pointer
OPCODE    -- one word
RETURN ADDRESS
```

The number of address operands depends on the operation. FP68K then allocates 3 more stack words:

```
COUNT     -- number of bytes in original call frame
HALT ADDRESS
```

This frame is used if a halt is taken. The COUNT field allows the halt handler to simply pop the original operands and return, if desired.

Above this frame, FP68K pushes registers D0-7, A0-6. In the progress of an operation, up to 6 more words of stack may be used. The total stack usage, after the call, is then up to 3 + 32 + 6 = 41 words. The binary-decimal conversions may use twice this much since they invoke FP68K to perform basic arithmetic operations.

Conditional assembly

There are two instances of conditional assembly in FP68K. The pointer to the floating-point state area is loaded into register A0 at the start of FPCONTROL.TEXT. Since the location of this area is system-dependent, conditional assembly is used to locate the field. Of course, this means that the effective address of the state area must be known at assembly time.

Conditional assembly is also used to resolve syntactic inconsistencies between various 68000 assembly language formats. The program counter (PC) relative addressing modes are heavily used in the implementation of jump offset tables within FP68K. A typical use is the instruction sequence:


```
MOVE.W    JMPTAB(D0),D0
JMP       JMPTOP(D0)
```

Here JMPTAB is a table of address offsets from the label JMPTOP, and register D0 contains a word index into JMPTAB. Some assemblers force the programmer to write:

```
MOVE.W    JMPTAB(PC,D0),D0
JMP       JMPTOP(PC,D0)
```

in order to assure PC-relative addressing. However, the Lisa assembler PROHIBITS this syntax, although it produces the desired code. An assembly flag is used to generate whichever of the two formats is suitable for a given compiler.

Pascal enumerated types

Lisa Pascal attempts to encode enumerated types in byte fields, which are then stored as the high byte of the target word. This affects structures like DecForm and Decimal, defined in the Pascal interface (see that document for details). Although the most seriously affected programs are the test drivers, the affected files in the basic package are FBB2D.TEXT and FBD2B.TEXT. Those files contain explicit comments when a byte test is used where an Apple III programmer (for example) might expect a word test.

MACINTOSH USER EDUCATION

Index to Technical Documentation

/TOOLBOX/INDEX

See Also: Inside Macintosh: A Road Map
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
TextEdit: A Programmer's Guide
CoreEdit: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
Toolbox Utilities: A Programmer's Guide
The Memory Manager: A Programer's Guide
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:			
	First Draft	C. Rose	8/5/83
	Second Draft	C. Rose	10/5/83
	Third Draft	C. Rose	1/9/84

ABSTRACT

This is an index to all the documentation listed under "See Also:" above, as of 1/9/84. It will be expanded and updated periodically.

Note the following change since the last draft:

"MM" now stands for the Memory Manager manual, not the Menu Manager manual; the latter is designated by "MN".

INDEX

The page numbers are preceded by a two-letter designation of which manual the information is in:

CE	CoreEdit: A Programmer's Guide	8/15/83
CM	Macintosh Control Manager Programmer's Guide	3/16/83
DL	The Dialog Manager: A Programmer's Guide	11/16/83
DS	The Desk Manager: A Programmer's Guide	9/26/83
EM	The Event Manager: A Programmer's Guide	6/20/83
FM	The Font Manager: A Programmer's Guide	4/22/83
MM	The Memory Manager: A Programmer's Guide	10/10/83
MN	The Menu Manager: A Programmer's Guide	11/1/83
PT	Putting Together a Macintosh Application	7/14/83
QD	QuickDraw: A Programmer's Guide	3/2/83
RD	Inside Macintosh: A Road Map	12/22/83
RM	The Resource Manager: A Programmer's Guide	10/3/83
SL	The Segment Loader: A Programmer's Guide	6/24/83
SM	The Scrap Manager: A Programmer's Guide	10/21/83
TE	TextEdit: A Programmer's Guide	9/28/83
TU	The Toolbox Utilities: A Programmer's Guide	1/4/84
WM	The Window Manager: A Programmer's Guide	8/25/83

abort event EM-5
action procedure CM-21, CM-22
activate event WM-15, EM-6
active
  control CM-8
  window WM-5, WM-23
AddPt procedure QD-65
AddReference procedure RM-26
AddResMenu procedure MN-17
AddResource procedure RM-25
alert box DL-5
Alert function DL-23
alert stages DL-15
alert template DL-8, DL-29, DL-31
alert window DL-7
AlertTemplate data type DL-29
AlertTHndl data type DL-29
AlertTPtr data type DL-29
allocated block MM-5
AppendMenu procedure MN-17
application font FM-6
application heap MM-4
  limit MM-12, MM-28
  subdividing MM-50
application parameters SL-4
application window WM-4
ApplicZone function MM-30
auto-key event EM-5
BackColor procedure QD-46
BackPat procedure QD-39
backspace buffer CE-19
BeginUpdate procedure WM-29
bit image QD-12
BitAnd function TU-8
BitClr procedure TU-7
bitMap QD-13
BitMap data type QD-13
BitNot function TU-8
BitOr function TU-8
BitSet procedure TU-7
BitShift function TU-8
BitTst function TU-7
BitXor function TU-8
block MM-5
block contents MM-5
block header MM-5
  structure MM-19
BlockMove procedure MM-47
BringToFront procedure WM-22
button CM-5, DL-10
Button function EM-19
Byte data type MM-13

CalcMenuSize procedure MN-26
CalcVis procedure WM-32
CalcVisBehind procedure WM-32
caret CE-10, TE-7
CautionAlert function DL-24
CEBackSpace procedure CE-22
CEBlinkCaret procedure CE-27
CEBtnDown procedure CE-26
CEBtnUp procedure CE-27
CEChngEdit procedure CE-20
CEChngFont procedure CE-25
CEChngSize procedure CE-25
CEChngStyle procedure CE-25
CECopy procedure CE-23
CECut procedure CE-22
CEDispPar procedure CE-28
CEDoneEdit procedure CE-20
CEForwardSpace procedure CE-22
CEGetRangeInfo procedure CE-26
CEGetSelRng procedure CE-25
CEInitEdit procedure CE-19
CEInsertChar procedure CE-21
CEKillEdit procedure CE-21
CEMouseMoved procedure CE-27
CENewPar function CE-19
CEParCutOrCopy procedure CE-23
CEParPaste procedure CE-24
CEPaste procedure CE-24
CEPrepEdit procedure CE-20
CERedraw procedure CE-28
CESetCaret procedure CE-27
CESetFldRect procedure CE-28
CESetSelRng procedure CE-25
CEStrtEdit procedure CE-19
Chain routine SL-6
ChangedResource procedure RM-24
character code EM-8
  table EM-25
character position CE-6, TE-6
character style QD-23
  of menu items MN-12
Chars data type TE-14
CharsHandle data type TE-14
CharsPtr data type TE-14
CharWidth function QD-44
check box CM-5, DL-10
check mark in a menu MN-6, MN-11
CheckItem procedure MN-23
CheckUpdate function WM-31
ClearMenuBar procedure MN-19
ClipAbove procedure WM-31
ClipRect procedure QD-38
clipRgn of a grafPort QD-19
CloseDeskAcc procedure DS-7
CloseDialog procedure DL-20
ClosePicture procedure QD-62
ClosePoly procedure QD-63
ClosePort procedure QD-36
CloseResFile procedure RM-16
CloseRgn procedure QD-56
CloseWindow procedure WM-19
color drawing QD-30
ColorBit procedure QD-46
compaction, heap MM-9, MM-39
CompactMem function MM-39
configuration routine EM-23
content region of a window WM-6
control CM-4
  defining your own CM-25
  in a dialog/alert DL-10
control definition function CM-9, CM-26
control definition ID CM-9, CM-26
Control Manager RD-6, CM-4
control record CM-11
control template CM-10, CM-25
ControlHandle data type CM-12
ControlMessage data type CM-26
ControlPtr data type CM-12
ControlRecord data type CM-13
coordinate plane QD-6
CopyBits procedure QD-60
CopyRgn procedure QD-55
CoreEdit RD-6, CE-4
CouldAlert procedure DL-25
CountMItems function MN-26
CountResources function RM-19
CountTypes function RM-18
CreateResFile procedure RM-16
current heap zone MM-5
current resource file RM-7, RM-18
CurResFile function RM-18
CursHandle data type TU-10
cursor QD-15
Cursor data type QD-16
CursPtr data type TU-10

data fork RM-6
default button DL-5
DeleteMenu procedure MN-18
dereferencing a handle MM-23, MM-48
desk accessory DS-3
  defining your own DS-10
Desk Manager RD-7, DS-3
desk scrap SM-3, SM-13
  data types SM-7

desktop WM-4
destination rectangle TE-5
DetachResource procedure RM-22
device driver RD-8
Device Manager RD-8
dial CM-6
dialog box DL-4
Dialog Manager RD-7, DL-4
dialog record DL-13
dialog template DL-8, DL-28, DL-30
dialog window DL-6
DialogPeek data type DL-13
DialogPtr data type DL-13
DialogRecord data type DL-14
DialogSelect function DL-21
DialogTemplate data type DL-28
DialogTHndl data type DL-28
DialogTPtr data type DL-28
DiffRgn procedure QD-57
dimmed
  menu item MN-5, MN-6
  menu title MN-5
disabled
  dialog/alert item DL-10
  menu MN-5
  menu item MN-6, MN-13
DisableItem procedure MN-22
Disk Driver RD-8
disk inserted event EM-5
display rectangle DL-12
DisposDialog procedure DL-20
DisposeControl procedure CM-18
DisposeMenu procedure MN-16
DisposeRgn procedure QD-54
DisposeWindow procedure WM-20
DisposHandle procedure MM-31
DisposPtr procedure MM-35
document window WM-4
drag region of a window WM-7
DragControl procedure CM-22
DragGrayRgn function WM-30
DragWindow procedure WM-25
DrawChar procedure QD-44
DrawControls procedure CM-19
DrawDialog procedure DL-23
DrawGrowIcon procedure WM-23
drawing QD-27
  color QD-30
DrawMenuBar procedure MN-18
DrawNew procedure WM-32
DrawPicture procedure QD-62
DrawString procedure QD-44
DrawText procedure QD-44
edit record CE-7, TE-4
edit rectangle CE-7
empty handle MM-10, MM-41
EmptyHandle procedure MM-41
EmptyRect function QD-48
EmptyRgn function QD-58
EnableItem procedure MN-23
EndUpdate procedure WM-29
EqualPt function QD-65
EqualRect function QD-48
EqualRgn function QD-58
EraseArc procedure QD-53
EraseOval procedure QD-50
ErasePoly procedure QD-65
EraseRect procedure QD-49
EraseRgn procedure QD-59
EraseRoundRect procedure QD-51
ErrorSound procedure DL-18
event EM-4
event code EM-9
Event Manager
  Operating System RD-7
  Toolbox RD-6, EM-4
event mask EM-12
event message EM-11
event queue EM-6
event record EM-9
EventAvail function EM-18
EventRecord data type EM-9
ExitToShell procedure SL-7

File Manager RD-8
FillArc procedure QD-54
FillOval procedure QD-50
FillPoly procedure QD-65
FillRect procedure QD-49
FillRgn procedure QD-59
FillRoundRect procedure QD-52
filterProc DL-22
FindControl function CM-20
FindWindow function WM-23
Fixed data type TU-3
fixed-point
  arithmetic TU-4
  numbers TU-3
FixMul function TU-4
FixRatio function TU-4
FixRound function TU-4
FlashMenuBar procedure MN-26
FlushEvents procedure EM-19
FmtRun data type CE-5
font FM-3
  scaling FM-6

Font Manager RD-6, FM-3
font number FM-3
FontInfo data type QD-45
ForeColor procedure QD-45
format, paragraph CE-5
Formats data type CE-6
FrameArc procedure QD-52
FrameOval procedure QD-50
FramePoly procedure QD-64
FrameRect procedure QD-49
FrameRgn procedure QD-58
FrameRoundRect procedure QD-51
free block MM-5
FreeAlert procedure DL-25
FreeMem function MM-38
FrontWindow function WM-23

GetAppParms procedure SL-6
GetClip procedure QD-38
GetCRefCon function CM-25
GetCTitle procedure CM-19
GetCtlAction function CM-25
GetCtlMax function CM-24
GetCtlMin function CM-24
GetCtlValue function CM-24
GetCursor function TU-9
GetDItem procedure DL-26
GetFNum procedure FM-9
GetFontInfo procedure QD-45
GetFontName procedure FM-8
GetHandleSize function MM-31
GetIcon function TU-9
GetIndResource function RM-19
GetIndType function RM-18
GetItem procedure MN-22
GetItemIcon procedure MN-24
GetItemMark procedure MN-25
GetItemStyle procedure MN-24
GetIText procedure DL-27
GetKeys procedure EM-20
GetMenu function MN-16
GetMenuBar function MN-19
GetMHandle function MN-26
GetMouse procedure EM-19
GetNamedResource function RM-20
GetNewControl function CM-18
GetNewDialog function DL-19
GetNewMBar function MN-19
GetNewWindow function WM-19
GetNextEvent function EM-17
GetPattern function TU-9
GetPen procedure QD-40
GetPenState procedure QD-41
GetPicture function TU-10
GetPixel function QD-68
GetPort procedure QD-36
GetPtrSize function MM-36
GetResAttrs function RM-22
GetResFileAttrs function RM-29
GetResInfo procedure RM-22
GetResource function RM-20
GetScrap function SM-12
GetString function TU-5
GetWindowPic function WM-29
GetWMgrPort procedure WM-18
GetWRefCon function WM-29
GetWTitle procedure WM-20
GetZone function MM-29
global coordinates QD-27
GlobalToLocal procedure QD-66
go-away region of a window WM-7
GrafDevice procedure QD-36
grafPort QD-17
GrafPort data type QD-18
GrafPtr data type QD-18
GrafVerb data type QD-71
grow image of a window WM-25
grow region of a window WM-7
grow zone function MM-12, MM-44
GrowWindow function WM-25
GZCritical function MM-45
GZSaveHnd function MM-46

handle MM-7, QD-10
  dereferencing MM-23, MM-48
  empty MM-10
Handle data type MM-13
HandleZone function MM-33
heap RD-7, MM-4
  compaction MM-9, MM-39
  creating on the stack MM-53
HideControl procedure CM-19
HideCursor procedure QD-39
HidePen procedure QD-40
HideWindow procedure WM-21
HiliteControl procedure CM-19
HiliteMenu procedure MN-21
HiliteWindow procedure WM-22
HiWord function TU-8
HLock procedure MM-42
HNoPurge procedure MM-43
HomeResFile function RM-18
HPurge procedure MM-43
HUnlock procedure MM-42

I/O driver DS-10
  event EM-6
icon number MN-11
inactive
  control CM-8
  window WM-5
InfoScrap function SM-10
InitApplZone procedure MM-25
InitCursor procedure QD-39
InitDialogs procedure DL-17
InitFonts procedure FM-8
InitGraf procedure QD-34
InitMenus procedure MN-15
InitPort procedure QD-35
InitResources function RM-15
InitWindows procedure WM-18
InitZone procedure MM-27
insertion point CE-10, TE-7
InsertMenu procedure MN-18
InsertResMenu procedure MN-18
InsetRect procedure QD-47
InsetRgn procedure QD-57
Int64Bit data type TU-9
InvalRect procedure WM-27
InvalRgn procedure WM-28
InvertArc procedure QD-54
InvertOval procedure QD-50
InvertPoly procedure QD-65
InvertRect procedure QD-49
InvertRgn procedure QD-59
InvertRoundRect procedure QD-52
IsDialogEvent function DL-20
item
  dialog/alert DL-8
  menu MN-4
item list DL-8, DL-9, DL-32
item number
  dialog/alert DL-12
  menu MN-14

journal EM-22
jump table SL-8
justification CE-11, TE-8
Justification data type CE-11

kerning QD-23
key code EM-8
  table EM-25
key down event EM-5
key up event EM-5
keyboard configuration EM-8
keyboard equivalent MN-6, MN-12
keyboard event EM-5
Keyboard/Mouse Handler RD-8
KeyMap data type EM-20
KillControls procedure CM-18
KillPicture procedure QD-62
KillPoly procedure QD-63

Launch routine SL-7
limit pointer MM-16
line height TE-9
Line procedure QD-42
LineTo procedure QD-42
LoadResource procedure RM-20
LoadScrap function SM-11
LoadSeg procedure SL-8
local coordinates QD-25
local reference RM-10
LocalToGlobal procedure QD-66
lock bit MM-20
locked block MM-6
locked resource RM-12
locking a block MM-6, MM-42
logical operations TU-8
logical size of a block MM-18
LongMul procedure TU-9
LoWord function TU-8

MapPoly procedure QD-69
MapPt procedure QD-69
MapRect procedure QD-69
MapRgn procedure QD-69
margins CE-11
master pointer MM-7
  structure MM-20
MaxMem function MM-38
MemErr data type MM-21
MemError function MM-48
Memory Manager RD-7, MM-4
menu MN-4, MN-29
  defining your own MN-26
menu bar MN-4, MN-30
menu definition procedure MN-7, MN-27
menu ID MN-8
menu item MN-4
menu item number MN-14
menu list MN-9
Menu Manager RD-6, MN-4
menu record MN-8
menu title MN-4
MenuHandle data type MN-8
MenuInfo data type MN-8
MenuKey function MN-21

Welcome to Macintosh Technical Support!

Your ID: SUPT.

Your KEY:

Welcome to Macintosh Technical Support. We provide developer support through Apple's electronic mail system on TYMNET. Using this system increases our efficiency, which translates into better support for development.

The attached document is the second draft of the user's guide for the electronic mail system. The last page is a list of TYMNET phone numbers, sorted by state. To use the system, you'll need a computer (we suggest an Apple), a modem, and a phone. You pay for the telephone call, and Apple pays for the electronic mail system charges, with the understanding that the system is to be used for technical support matters.

The first time you log into the system, you should change your key (password). Change it to something that you'll remember, and something not obvious. Please keep it secret. If you have problems, or suspect unauthorized use of the service, send me a message or give me a call. For instructions on changing your key, type

:READ ** CHANGE.KEY

When you have a question or problem, send a message to me at the MAC mail station. The mailbox is usually checked 3 or 4 times a day (I check it from home on weekends), and we will try for 24-hour turnaround at worst.

I'd appreciate comments on the document (this is a draft), and on the system. Let me know what kind of equipment you're using to access the system, and how it works for you.

OLD USERS OF THE SYSTEM:

If you've been using the system for a while, you'll notice a couple of changes. The account name is now SUPT (it used to be APPLE). You send mail to MAC rather than MACTECH. Phone numbers are the same.

1200 BAUD USERS:

If you're experiencing trouble at 1200 baud, and your software uses XON/XOFF protocols, be sure and send CONTROL-X CONTROL-R before "EMSAPP" in talking to the system. This lets their software know you're using XON/XOFF for flow control.

Bob Martin

Macintosh Technical Support

Macintosh Technical Support
Electronic Mail System
User's Guide

Second Draft -- 12/5/83

Apple Computer, Inc.
2Ø525 Mariani Avenue
Cupertino, CA 95Ø14

Table of Contents

Section	Page
Introduction	1
Preparing and Sending Messages	2
Mail Management	3
The IN List	3
The OUT List	4
Receiving Messages	5
Retrieving a Message	6
Cancelling a Message	6
Control Characters	6
Accessing EMS	7
Leaving EMS	8
Error Messages and Trouble Reporting	9
Command Reference	11
Using Access /// on EMS	12
Using Micro-Courier on EMS	13

Introduction

This document is a user's guide for Apple Computer third-party developers using Apple's electronic mail system (EMS). Apple uses TYMNET's "ONTYME II" message-switching network with Apple][and Apple /// computers to create an effective and efficient computer- based message network.

The ONTYME II message-switching system operates on a store-and-forward basis. It accepts messages for transmission, and then either stores them while waiting for users to make contact with the service, or delivers the messages to on-line designated hardcopy printers. The network is supported by Macintosh Technical Support.

Messages (mail) are created off-line using either an Apple][or an Apple ///, and then entered into the electronic mail system.

This service is provided to third-party developers to improve communication and productivity between Apple and outside developers.

Each station or user in the Apple account is identified with a unique call directing code (CDC) and password (key).

All commands entered into the system are preceded by a colon (:), to indicate that they are commands and not text. Commands must begin at the left margin (column Ø).

The following conventions are used in the examples given in this guide. Refer to page 11 for a summary of command definitions.

Convention Used	Definition
EMS	Apple's electronic mail system
CDC	A call directing code (a user)
Underlined text	Commands typed by the user
Regular text	Text or messages printed by EMS
(cr)	A carriage return typed by the user
(sp)	A space that is required

Questions or suggestions regarding EMS should be forwarded to EMS station "MAC".

The :SEND command takes the text in your workspace and sends it to the CDC (or CDCs) that you specify.

Managing the Workspace - :TYPE and :ERASE Commands

Anything that is not recognized as a command goes into your workspace as part of the message you want to send. So if you type ";IN" rather than ":IN", the erroneous command goes into your workspace. If you are unsure of what text is in your workspace, use the :TYPE command to display it. The :ERASE command clears the workspace.

Standard Message Format

All messages should be created and edited off-line, using the text editor of your choice. This saves valuable connect time on EMS.

When preparing a message, do not exceed a 68-character line length. By keeping your lines under 68 characters, you ensure that international locations will be able to read each line in its entirety. Although there have been recent improvements, most international locations still use the older TELEX-type equipment, which has a 68-character line limitation.

Text editors such as Applewriter and the Pascal editor let you set up a "boilerplate" file that includes a right margin (or line length) selection of 68 characters. Using a boilerplate file is a good way of ensuring a consistent format.

All messages should be prepared in the following format:

Format	Example
Date	12/5/83
Addressee	To: MAC Tech Support
Sender	Fr: Bob Martin "MAC1Ø23"
Text	This is the standard message format for a message sent through the ONTYME or TELEX network.
End of message and operator's initials	END/RTM
Send command	:SEND(sp)MAC(cr)

The :SEND command packages up the text in your workspace and sends it to the destination CDC. If you want to send the message to more than one CDC, separate the CDCs by at least one space.

Mail Management

The Apple electronic mail system provides tools that allow you to manage your mailbox effectively and efficiently.

These tools are described below as follows:

I. The IN List

II. The OUT List

III. Receiving Messages

IV. Retrieving a Message

V. Cancelling a Message

VI. Control Characters

I. The IN List

Each time someone sends you a message, an entry is made in a list of all messages waiting to be received by your CDC. This list is called your "IN list". When you actually ask to read a message (with the :READ command), the entry in the IN list is removed and entered into the "IN OLD list".

To see if any messages are waiting for you in your mailbox, you examine the IN list by issuing this command:

:IN(cr)

If no messages are waiting, EMS will respond with "NONE". Otherwise, the following information is transmitted to you:

SENDER	SENT	MSG#	LENGTH
aaa	ddmmyy hh:mm	bbbb	ccc

aaa = Sending CDC
ddmmyy = Day, month, and year the message was sent
hh:mm = Time the message was sent (your local time)
bbbb = Message number assigned by EMS
ccc = Number of characters in the message

To examine the list of messages you have received and read in the last five days, give the command:

:IN(sp)OLD(cr)

The following information is transmitted to you:

SENDER	SENT	MSG#	READ
aaa	ddmmyy hh:mm	bbbb	dddd

This is similar to the IN list, except that "dddd" indicates the date and time the message was read, and the message length is omitted.

Now, if you want to read the message again, type:

:READ(sp)bbbb(cr)

II. The OUT list

Each time you send a message, it is entered in a list of all messages you have sent, called your "OUT list". When the recipient of a single-addressed message has read it, the entry is removed from your OUT list and placed in your "OUT OLD list", where you have confirmation of its receipt, plus the date and time it was read. For messages sent to multiple CDCs, an asterisk (*) is added before each location that has received it. When all locations have received the message, the entry is moved to the OUT OLD list.

To see if there are messages waiting to be delivered, examine the OUT list by issuing the command:

:OUT(cr)

If all your messages have been received, EMS responds with "NONE". Otherwise, the following information is transmitted:

NAME	SENT	MSG#
aaaa	ddmmyy hh:mm	bbbb

aaaa	=	Receiving CDC
ddmmyy	=	Day, month, and year the message was sent
hh:mm	=	Time the message was sent (your local time)
bbbb	=	Message number assigned by EMS

Although EMS keeps lists of your unread outgoing messages for up to 14 days after transmission, it is a good habit to check your OUT list at least once a day.

III. Receiving Messages

If you are a dial-up user, you must check your mailbox for incoming mail (as you do at home). It is recommended that this be done daily.

To receive your mail from EMS, you must use one of the :READ commands listed below. But before entering :READ commands, make sure you are ready to record the mail you will be reading, either in memory or on a disk.

Command	Purpose
:READ(cr)	Transmits the oldest message in your IN list, then removes it from your IN list, and removes your CDC from the sender's OUT list.
:READ(sp)ALL(cr)	Transmits all the messages in your IN list, removes them from your IN list, and removes your CDC from the sender's OUT list.
:READ(sp)message number(cr)	Transmits a specific message recently received to your CDC. This works only if the message is still in either your IN list or your OUT list.

IV. Retrieving a Message

Any message still in your IN or OUT list can be retrieved with the following command:

:GET(sp)message number(cr)

This command does not automatically display the retrieved text; the following command should be used to display it:

:TYPE(cr)

V. Cancelling a Message

You can cancel any message sent via your CDC that has not been read by all recipients (if some recipients have already read the message, EMS will notify you). "Message number" below is the message number that EMS assigned to the message at transmission.

:CANCEL(sp)message number(cr)

VI. Control Characters

EMS uses these control characters:

CTRL-H Deletes the last character typed.

CTRL-S Stops the data being received.

CTRL-Q Starts the data being received.

CTRL-H is the left arrow on most keyboards. Typing CTRL-S suspends output, but it may take 2Ø characters or so before EMS recognizes it. To start things going again after CTRL-S, type CTRL-Q.

Accessing EMS

Accessing EMS is done by placing a telephone call to the local TYMNET office (see attachment to determine local number) and then "logging into" the EMS computer using the instructions listed below.

TYMNET/ONTYME II Log-In Instructions

Proper connection with TYMNET has been made when the following appears on your screen:

    xxxxxxxxxxxxxØØØØØØØØØØØxxxxxxØØØØØØxxxxxØØØØ(garbage)

Type:

    A

This lets TYMNET know what terminal speed you are using. The system will respond with:

    please log in:

Type:

    EMSAPP(cr)

The system will respond with:

    >ONTYME II date time GMT
    ID?

Type:

    SUPT.xxx(cr)   (xxx = user's CDC)

The system will respond with:

    KEY?

Type your key (password), which should not echo, followed by a carriage return. The system will respond with:

    ACCEPTED (Refer to page 9 if this does not appear.)

A sample log-in sequence is shown below.

```
xxxxxxxØØØØØØxxxxxxØØØØØ(garbage)
A
-1326-Ø23-
please log in: EMSAPP(cr)
ID? SUPT.MAC(cr)
KEY? non-printing password(cr)
ACCEPTED
```

Leaving EMS

You normally end your EMS session with the command ":QUIT". If there are any messages still waiting in your IN list, the following message will be displayed:

MESSAGES WAITING: (if your IN list is not empty)

Typing a carriage return in response to this message ends the session. Any other response will ignore the :QUIT command, leaving you connected to EMS.

For example:

```
:QUIT(cr)
MESSAGES WAITING: (cr)
DROPPED BY ONTYME II
Ø1 MAR 83 11:47:35

please log in:   (message will appear when you have successfully
                 terminated from ONTYME II)
```

Note: When using Access ///, you must type open-apple Q to exit after you have left ONTYME II.

To leave the system immediately, use the command:

:LOGOUT(cr)

Error Messages and Trouble Reporting

NOTE: Error messages in quotes will appear on the screen when you are logged into EMS.

Symptom or Message	Action Required
Ring, no answer	Call message network supervisor.
Busy signal when call placed	Wait 5 minutes and try again.
"ERROR TYPE USER NAME"	Retype user name.
Terminal prints DDOOUUBBLLEE	Ensure that the Apple/software is set for full duplex.
"HOST DOWN"	Wait 15 minutes and try again.
"please log in" printed during session	Communications failure: log in again and restart from point of last accepted message.
"all ports busy"	Wait 5 minutes and try again.
"all circuits busy"	Wait 5 minutes and try again.
"no such recipient"	Printed station name (CDC) is not a valid station.
"invalid command"	Command was misspelled.
"message not on in list"	Message requested is not on IN list or IN OLD list.
"all messages read"	IN list is empty.
"group code file not found"	Group code was either mistyped or nonexistent.
"invalid message number"	Erroneous message number was entered.
"invalid user"	Specified user name is not valid in the system.

Error Messages and Trouble Reporting (continued)

Symptom or Message	Action Required
"message #??"	Invalid message number was entered in a :GET or :READ command.
"message not on out list"	User attempted to cancel a message that has already been read.

In general, if you are having trouble logging into ONTYME II, or having trouble with the command formats, you should contact Macintosh Technical Support at (408) 973-2282.

Telephone number of ONTYME: ______________________

All commands are preceded by a colon (:).

:IN	Checks the mailbox for any messages. Messages are listed in order of the oldest message first.
:IN OLD	Lists the messages for the last 14 days that have already been read.
:OUT	Lists any messages sent that have not been read by the recipient.
:OUT OLD	Lists the messages sent for the last 14 days that have already been read by the recipient.
:READ	Transmits the message in the mailbox. If there is more than one message, the oldest is transmitted first.
:READ ALL	Transmits all messages in the mailbox.
:CANCEL	Followed by a message number, cancels the message.
:SEND	Followed by a CDC or CDCs, sends a message.
:SEND CC	Followed by a CDC or CDCs, sends a message with a "carbon copy" list attached.
:SEND RUSH	Followed by a CDC or CDCs, sends a message immediately to recipient(s) who have available dial-out stations.
:GET	Retrieves a message recently received or sent via your ID, if the message is still on your IN or OUT list.
:TYPE	Displays the text in your workspace.
:ERASE	Deletes the text in your workspace.
:KEY	Lets you change your key (password).

The above information is available from EMS via the command:

<u>:READ(sp)**(sp)COMMANDS(cr)</u>

For more information on EMS, try:

<u>:READ(sp)**(sp)HELP(cr)</u>

Using Access /// on EMS

To use Access /// (Revision 2) on EMS:

1) Before you log into the electronic mail system, you should have your message(s) created and stored on a disk, ready for sending. At this time, boot Access ///. When you see the first menu, select terminal mode, then press RETURN, and you will see only a cursor on the screen. Now place a call to the local TYMNET office and follow the instructions on page 7.

2) You must set up a recording file for your message to be stored. To do this, type open-apple S. Select "Change the recording file" with the up/down arrows and press RETURN. Access /// will ask for the file name, or the new file name if you are changing the recording file. If you are operating at 300 or 1200 baud and want to use a Silentype as the file, it would be ".SILENTYPE". If you want to record to a disk, name the file with the path first then the name (e.g. .D2/EMSLOG) and the message will be recorded to the disk. When saving messages to a disk, you may want to change the recording file name to avoid writing over a previous message.

3) To record to a disk or Silentype, you must turn on the recording file by typing open-apple R. The cursor starts flashing. The message will be sent to the recording file. After the message is recorded, you must turn off the recording file so that the message buffer in the Apple is cleared. To close the recording file, press open-apple R again, just as you did to turn it on. You now see that the cursor is not flashing. Follow the "Mail Management" instructions on checking your mailbox for messages and reading messages.

4) To send a message or messages, return to the Access /// set up mode by pressing open-apple S. Use the up/down arrow to select "Exit terminal mode" and press RETURN. Select "Transmit a file" and press RETURN. Enter the pathname (e.g. ".D2/APPLE") for the file to be sent. Access /// now asks for delay parameters. The following normally work satisfactorily:

   line delay 0(cr)

   character delay 8(cr)

5) After transmitting the message, Access /// responds "File transmission complete". To return to terminal mode, press up arrow twice and then RETURN. If you put the :SEND command at the end of the message, you should see a message number. It is a good practice to write these down, in case you need to refer to a specific message again. Enter the :SEND command now, followed by RETURN if you did not embed it into your text.

6) After sending and receiving all mail, leave ONTYME II (refer to page 8).

Using Micro-Courier on EMS

1) Load Micro-Courier.

2) When the main menu appears, type "1" and press RETURN.

3) When the editing menu appears, type "1" or "2" and press RETURN.

4) Prepare or edit your messages using the correct format (see page 2).

NOTE: More than one message can be prepared per file using Micro-Courier. The important thing is not to forget the :SEND command after preparing each message.

5) After message(s) have been prepared and edited, press the ESC key.

6) Micro-Courier will ask for a file name.

   a) Type month day/msg# (use the first message number).

      Example: MARØ1/ØØ1

   b) After typing the file name, press RETURN.

7) Micro-Courier will ask which drive to store the file on.

   a) Type either "1" or "2" (in most cases it is "2").

   b) Press RETURN.

8) Press the ESC key to leave the storage area.

9) Press the ESC key again to go back to the main menu.

1Ø) Type "6" and press RETURN.

11) Type "1" and press RETURN. Micro-Courier will ask for phone number. Type the local TYMNET number.

12) Type "3" and press RETURN. Type "6" and press RETURN.

13) Type "7" and press RETURN.

14) Type "5" and press RETURN. Micro-Courier will ask for an out file name. Type the name created in step 6. After typing the file name, press RETURN. Micro-Courier will ask which drive; type the same number as in step 7, then press RETURN.

15) Type "6" and press RETURN. Micro-Courier will ask which file name will store incoming traffic. Type month day/IN msg#.

      Example: MARØ1/INØØ1

    Press RETURN. Micro-Courier will ask which drive will store messages. Indicate either 1 or 2 and press RETURN.

Using Micro-Courier on EMS (continued)

16) Connecting to EMS

    a) Type "1" and press RETURN.

    b) Log into TYMNET/ONTYME (see page 7).

    c) When you have logged into ONTYME and have received the "ACCEPTED" acknowledgment, type ":LOAD ON" and press RETURN.

17) How to send

    a) When you have received the accepted acknowledgment, press the ESC key and "T" key. Micro-Courier is now sending the out file to ONTYME.

    b) As messages are accepted, ONTYME will send message numbers back to you which will show on screen.

18) How to receive

    a) Once sending has been completed, you can receive incoming mail. Press the ESC key and "R" key. This tells Micro-Courier to store all messages received in the file that was named in step 15.

    b) Type ":READ ALL" and press RETURN. ONTYME is now sending your incoming mail.

19) After sending and receiving mail, leave ONTYME (refer to page 8).

20) Press ESC and "E". Type "4" and press RETURN.

21) Type "8" and press RETURN. Micro-Courier will go back to the main menu.

22) How to print incoming messages

    a) Type "1" and press RETURN.

    b) Type "2" and press RETURN.

    c) Type the file indicated in step 15. Type the drive number.

    d) Press the ESC key.

    e) Type "3" and press RETURN.

    f) When printing has been completed, press ESC. Micro-Courier will go back to the main menu.

23) Type "8" and press RETURN.

Listing of TYMNET access numbers sorted by state, 5/1/83

H, M, L, I = High, Medium, Low, International capacity
3 = 300 baud only, 300/1200 baud otherwise

State	Phone	Type												
AL	205-432-3302	M	CA	805-682-9641	M	KAN	913-233-1682	L	NH	603-623-0055	L	RI	401-273-0200	H
AL	205-834-3410	L	CA	916-440-0151	H	KAN	913-384-1544	H	NH	603-882-0435	M	SC	803-252-0040	M3
AL	205-882-3003	M	CO	303-475-2121	M	KY	502-499-7110	H	NH	603-893-6200	L	SC	803-271-2410	M3
AL	205-942-4141	H	CO	303-830-9210	H	KY	606-253-3463	H3	NJ	201-432-0792	L3	SC	803-271-9967	M3
ALASK	907-279-3511	I	CT	203-227-7109	M	LA	318-237-9500	M	NJ	201-440-0100	H3	SC	803-577-2179	L
ALASK	907-584-6611	I	CT	203-242-7140	H3	LA	318-688-4444	L	NJ	201-483-3937	H3	SC	803-585-2637	L
ARK	501-372-5700	M3	CT	203-367-6021	M	LA	504-291-2650	H	NJ	201-785-4400	H3	TN	615-367-9302	H
ARK	501-782-3210	L	CT	203-755-1153	L3	LA	504-524-4371	H	NJ	201-894-0250	M	TN	615-637-3110	M
AZ	602-254-5011	H	CT	203-789-0579	H3	MA	413-781-6830	H	NJ	201-901-1900	H	TN	615-756-5056	M3
AZ	602-790-0764	H	CT	203-965-0000	H	MA	616-482-5605	H3	NJ	609-235-3761	M3	TN	901-529-0170	H3
CA	209-260-1211	L	DC	703-442-3900	H3	MA	617-482-4677	H3	NJ	609-452-1010	H	TX	214-263-4501	M
CA	209-577-5602	L	DC	703-442-3960	H	MA	617-482-7035	H	NJ	609-452-1018	H	TX	214-630-0000	H
CA	213-203-9065	M	DC	703-691-0200	H	MA	617-755-0916	L	NM	505-843-6301	M	TX	214-750-1756	L
CA	213-280-1103	M	DC	703-734-3900	H3	MD	301-547-0100	H	NY	212-269-9642	H	TX	512-225-0002	H
CA	213-331-354	L3	DEL	302-429-0112	L	MD	301-770-1600	M3	NY	212-532-0437	H3	TX	512-444-3200	H
CA	213-365-2013	L3	DEL	302-670-0449	L	ME	207-774-2654	H	NY	212-685-4414	H3	TX	512-803-0050	M
CA	213-435-7000	L	FLA	305-467-3007	M	MI	517-487-2040	M	NY	212-785-5400	H	TX	713-427-5856	L3
CA	213-518-3773	M	FLA	305-624-7900	H	MI	616-385-3150	M	NY	315-437-7111	M	TX	713-832-2589	L3
CA	213-572-0999	H3	FLA	305-627-5410	M	MI	616-429-2560	L	NY	516-549-2700	M	TX	713-975-0500	H
CA	213-574-7636	L3	FLA	305-725-0011	L3	MI	616-459-5069	M	NY	516-872-6500	M	TX	713-977-4000	H3
CA	213-574-8834	L	FLA	305-851-3530	H3	MI	616-723-0373	L	NY	518-463-3111	M3	TX	806-762-0136	L3
CA	213-577-0696	L	FLA	813-535-6441	H3	MI	616-775-1261	L	NY	607-257-6601	M	TX	915-533-1453	H
CA	213-865-0555	L	FLA	904-252-4401	L3	MI	616-946-0002	L	NY	607-962-4481	M	TX	915-563-3745	L
CA	213-906-0450	M	FLA	904-434-0134	L	MICH	313-459-0900	H3	NY	716-240-0000	H	TX	915-683-5645	M
CA	213-906-9503	L3	FLA	904-721-0100	H3	MICH	313-569-0350	H3	NY	716-285-6691	L3	UTAH	801-364-0700	H
CA	213-990-0461	M	GA	912-236-1904	L	MICH	313-665-2626	H3	NY	716-845-6610	H3	VA	703-345-4730	L
CA	213-990-3331	L	GA	912-352-7259	3	MICH	313-963-3300	H3	NY	914-320-9500	H	VA	703-691-0200	H
CA	408-426-0400	M	ID	208-343-0404	M	MICH	517-787-9461	M3	NY	914-471-6100	L	VA	804-528-1903	L
CA	408-443-4333	L	ILL	217-753-7905	H	MINN	612-339-5200	H3	NY	914-684-6075	H	VA	804-596-7689	M
CA	408-900-8100	H	ILL	309-673-2154	L3	MISS	601-769-6502	M3	OH	216-535-1061	M	VA	804-744-4040	H
CA	415-442-0900	M	ILL	312-346-4961	H3	MISS	601-944-0040	M	OH	216-744-5326	L	VA	804-744-4060	H
CA	415-490-7366	M	ILL	312-360-4607	H3	MO	314-421-5110	H3	OH	513-223-3047	H	VA	804-872-9592	M
CA	415-770-3420	L	ILL	312-360-4700	H3	MO	314-731-2304	L3	OH	513-489-2100	H	VT	802-650-2123	L
CA	415-785-3431	L	ILL	312-430-5003	L3	MO	314-875-1290	M	OHIO	419-243-3144	M3	WA	206-285-0109	H
CA	415-790-2093	L	ILL	312-790-4400	H	MO	417-782-3037	L	OHIO	614-421-7270	H3	WA	206-473-7010	L
CA	415-836-0700	H	ILL	815-233-5505	L3	MO	417-831-5044	L	OK	405-947-6307	H	WA	206-754-3900	L
CA	415-932-0116	L	ILL	815-390-6090	H3	MO	816-232-1097	L	OK	918-582-4433	H	WA	206-825-6576	L
CA	415-966-0550	M	IND	219-233-4163	H3	MO	913-384-1544	H	OR	503-226-0627	H	WA	509-375-3367	M
CA	415-986-0200	H3	IND	219-424-5162	L3	MONT	406-494-4437	L	OR	503-399-1453	M	WA	509-747-4105	M
CA	619-296-3370	H	IND	219-769-7254	L3	NC	704-376-2545	H3	PA	215-269-9861	L	WI	414-235-1082	M
CA	619-320-0772	L	IND	219-836-5452	L3	NC	919-323-4202	M	PA	215-337-9900	M	WI	414-437-9897	L
CA	619-485-1990	L	IND	317-662-0091	L3	NC	919-379-0470	M	PA	215-432-1500	M	WI	414-632-3006	L
CA	619-727-6011	L	IND	812-425-5211	L	NC	919-549-0952	H	PA	215-666-9190	H	WI	414-722-5500	H
CA	707-575-0160	L	IOWA	319-233-9227	L	NC	919-725-9252	M	PA	412-765-1320	H	WI	414-735-9390	L
CA	714-370-1200	H	IOWA	319-324-7197	L	NC	919-832-1551	L	PA	717-233-0531	M	WI	414-785-1614	L
CA	714-490-3130	L	IOWA	319-354-7371	M	NC	919-885-0171	L	PA	717-846-3900	M	WI	608-221-4211	H3
CA	714-662-0490	H	IOWA	319-363-2402	L	NEB	402-397-0414	H	PA	814-946-0000	L	WVA	304-522-6261	L3
CA	714-662-0490	H	IOWA	515-277-7752	M	NEB	402-475-0459	L	PUERT	809-792-5900	I			
CA	805-324-2653	L	IOWA	515-753-0667	L	NEV	702-293-0300	M	PUERT	809-833-4535	I			
CA	805-486-4811	H	KAN	316-265-1241	H	NEV	702-882-7810	M	PUERT	809-840-9110	I			

December 8, 1983

TO MACINTOSH SOFTWARE DEVELOPERS:

We hope that this letter finds all of you busy at work on your application for Macintosh. We at Apple are very excited to have you as a software developer and look forward to seeing your product on Macintosh.

The purpose of this letter is to inform you that with no incremental effort, your application will also run on. the Lisa system. We will provide a Macintosh environment for the Lisa which allows Macintosh software to run standalone on the Lisa without any modification. Specifically, we will be marketing a single diskette which will boot the Lisa into a Macintosh environment and allow the Lisa to use the extentsive software base we expect to be offered for Macintosh.

From a user's perspective, using Macintosh software on the Lisa would work as follows:

- The user would boot the Lisa from a 3-1/2" microfloppy diskette using a microfloppy drive supplied for the Lisa.
- By then inserting their Macintosh application diskette, they are ready to work.

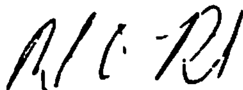
In addition to just being able to run Macintosh software, the user will also be able to take advantage of the additional memory in the Lisa as well as the larger screen. At some point in the future we also plan to have this environment support Lisa's hard disk.

So by following some simple rules in writing your Macintosh software (see attached), you will be able to leverage your efforts over both machines. We already have a substantial installed base of Lisa's which is growing daily. These Lisa users are anxious for the types of applications which you are developing for Macintosh and thus represent a sizable market to you.

I would strongly urge you to following the attached directions in writing your Macintosh applications. Not only will they insure your current ability to sell your software on the Lisa, but will also make it much more likely that your software will run on future Macintosh products. Additionally, I can provide you with information on copy protection implementations which will insure that your software is viable on both the Lisa as well as Macintosh.

If you have any questions, please either give me a call or don't hesitate to call Burt Cummings in the Lisa group. We are both here to help you succeed and are very excited about the prospects for the software you have underway.

Sincerely,



Mike Boich
Apple Computer, Inc.

Notes for applications concerned with LisaMac compatibility

Date: December 5,1983

The following is a compendium of suggestions intended to help guide anyone who wants to write software that will run on both Macintosh, and Lisa 2.

(1) The size of the screen, or rowbytes, should never be assumed. An application can always determine the size of the screen by looking at the "bounds" field of the QuickDraw global variable "ScreenBits".

In Pascal this might look like:

```
thisScreenSize := ScreenBits.bounds
```

where thisScreenSize is of TYPE Rect.

In Assembly this might look like:

```
MOVE.L bounds(A0),(A0)      ; get start of screenbits.bounds
MOVE.L(A0)+,(A1)+           ; copy topLeft
MOVE.L(A0)+,(A1)+           ; copy bottomRight
```

where A0 is the address of the QuickDraw global "ScreenBits", and A1 points to our screen size.

(2) Use of sound should be limited to only the routine "SysBeep". Later on it may be possible to loosen this constraint to include access to the square wave generating capabilities of the sound driver.

(3) The size of memory should not be assumed. Memory size can be determined by using system routines such as "FreeMem".

(4) Most, if not all, attempts to access hardware directly (e.g. BTST #3,#$EXXXXX to see if the mouse button is up or down) will result in fatal system errors.

(5) In general, access to system globals should be through system routines. Perhaps later on there will be time to compile a list of those few global variables which, in fact, are not valid.

(6) Do not use the TAS (test and set) instruction of the 68000. A BSET instruction is not that much slower.

(7) The screen memory should not be accessed directly. All screen access should be through QuickDraw.

(8) The ROM code should not be accessed (i.e. jumped into) directly.

(9) The address of the dispatch table (used in replacing traps) should not be assumed. The address of individual traps can be determined by using the system routine

"GetTrapAddress".

(10) A program should not count on parameter memory being saved across system boots; "Time" will be saved however. In the case of a power loss, all parameter memory including Time will be initialized.

(11) Serial port "B" (i.e. one of the two serial ports) will not support 19.2k baud.

(12) Timing sensitive parts of programs should not be implemented with timing loops, or other application internal timing methods. Instead, use "ReadDateTime" for 1 second values or look at the "Ticks" global for ≈ 1/60 second values.

(13) Software protection????????

The MacPaint Document Format

by Bill Atkinson

MacPaint documents are easy to read and write, and have become a standard interchange format for full-page bitmap images on Macintosh. Their internal format is described here to aid program developers in generating and reading MacPaint documents.

MacPaint documents use only the data fork of the file system; the resource fork is not used and may be ignored. The data fork contains a 512 byte header and then the compressed data representing a single bitmap of 576 pixels wide by 720 pixels tall. At 72 pixels per inch, this bitmap occupies the full 8 by 10 inch printable area of the Imagewriter printer page.

HEADER:

The first 512 bytes of the document form a header with a 4 byte version number (default = 2), then 38*8 = 304 bytes of patterns, then 204 unused bytes reserved for future expansion. If the version number is zero, the rest of the header block is ignored and default patterns are used, so programs generating MacPaint documents can simply write out 512 bytes of zero as the document header. Most programs which read MacPaint documents can simply skip over the header when reading.

BITMAP:

Following the header are 720 compressed scanlines of data which form the 576 wide by 720 tall bitmap. Without compression, this bitmap would occupy 51840 bytes and chew up disk space pretty fast; typical MacPaint documents compress to about 10 Kbytes using the PackBits procedure in the Macintosh ROM to compress runs of equal bytes within each scanline. The bitmap part of a MacPaint document is simply 720 times the output of PackBits with 72 bytes input.

READING SAMPLE:

```
CONST srcBlocks = 2;  { at least 2, bigger makes it faster }
      srcSize = 1024;  { 512 * srcBlocks }
TYPE  diskBlock = PACKED ARRAY[1..512] OF QDByte;
VAR   srcBuf: ARRAY[1..srcBlocks] OF diskBlock;
      srcPtr,dstPtr: QDPtr;

{ skip the header }
ReadData(srcFile,@srcBuf,512);

{ prime srcBuf }
ReadData(srcFile,@srcBuf,srcSize);

{ unpack each scanline into dstBits, reading more source as needed }
srcPtr := @srcBuf;
dstPtr := dstBits.baseAddr;
FOR scanLine := 1 to 720 DO
  BEGIN
    UnPackBits(srcPtr,dstPtr,72);                { bumps both ptrs }
    { time to read next chunk of packed source ? }
    IF ORD(srcPtr) > ORD(@srcBuf) + srcSize - 512 THEN
      BEGIN
        srcBuf[1] := srcBuf[srcBlocks];          { move up last block }
        ReadData(srcFile,@srcBuf[2],srcSize-512);
        srcPtr := Pointer(ORD(srcPtr) - srcSize + 512);
      END;
  END;
```

WRITING SAMPLE:

To write out a 576 by 720 bitmap which is contained in memory, the following fragment of code could be used:

```
TYPE  diskBlock =        PACKED ARRAY[1..512] OF QDByte;
VAR   srcPtr,dstPtr:     QDPtr;
      dstBuf:            diskBlock;
      dstBytes:          INTEGER;

{ write the header, all zeros }
FOR i := 1 to 512 DO dstBuf[i] := 0;
WriteData(dstFile,@dstBuf,512);

{ Compress each scanline and write it }
srcPtr := srcBits.baseAddr;
FOR scanLine := 1 to 720 DO
   BEGIN
      dstPtr := @dstBuf;
      PackBits(srcPtr,dstPtr,72);                { bumps both ptrs }
      dstBytes := ORD(dstPtr) - ORD(@dstBuf);    { calc packed size }
      WriteData(dstFile,@dstBuf,dstBytes);       { write packed data }
   END;
```

TO: MacPrint Users **DATE:** December 6, 1983

SUBJECT: The MacPrint Interface **FROM:** Owen Densmore

Introduction

Because MacPrint is is not part of the ROM code it must be included with the Application's code. Typically this would be done by including the MacPrint code in your Linker job. There are two reasons we don't do this. One is that we want to be able to configure new printers without re-linking the new print code into the Applications. The other is that we cannot assume that all OEM's are using the Lisa Workshop's Linker! MicroSoft, for example, runs an interpreted "C" environment with a VAX development system!

Our solution to this packaging problem is to provide four "PDEF" definition procs in a special printing resource file. These four def procs break the printing code into four disjoint code segments: Dialogs, Spooling, Draft printing, and Picture printing. In addition, there is a driver, ".Print", installed in System.rsrc which is accessed like any other driver with Open, Close, Status and Control calls.

A new printer is configured by supplying a new printer resource file, and installing its file name [ImageWriter, for example], its ".Print" driver [id=2], and the driver's configuration record [PREC, id=2] in System.rsrc.

Access to these "PDEF" procs is via a very small (374 byte) piece of Print Glue called "PrLink" which must be Linked (or some how included!) into your application. In addition, the .Print driver can be accessed directly. The driver and its use is discussed in detail in a seperate document.

The Pascal interface is "MacPrint.obj" and the Assembly interface is "PrEqu.text". There are four main groups of procedures: Init/Termination, Dialogs, Spooling/Draft, and Picture Printing. This release is our "Beta" release. This means that the interface is as stable as we can make it. No further procedural interface changes will be made. Only changes internal to the code will be allowed.

Changes

This is a list of recent changes to printing:

- Added rPaper rectangle to Print record.
- Print record size increased from 80 to 120 bytes.
- Added the PrValidate procedure.
- Made the dialogs configurable by the Application for adding their own buttons.
- Square Pixel correction options have been provided for both screen and document printing via dialog buttons.
- The default print file name is now included in the Print resource file as a string.
- The .Print Driver (screen printing) now uses a parameter record for configurability.
- We now look for Cmd "." aborts if a Nil idle proc is used.
- The spooler now provides breaking text into words for screen-HiRes alignment. This corrects for minor scaling differences between low and high res.
- Spooling now uses page alligned file buffering. This gives up to 3x speed improvement when word breaking!
- Added two new segments to the PrLink interface, both private to printing:
  - PrCfgDialog: used to configure the printer from the PrApp program.
  - PrHack: a general addition to the interface for stuff we've forgotten!
- Added the procedure PrJobMerge to install the results of a PrJobDialog into several documents. This is used as part of Finder printing.
- Added iPrFileVol & bPrFileType to the Print record and the Printer IO unit.
- And of course lots of bug fixes.

This release also contains Don's many improvements to the PrApp:
Installation of new printers via StdFile.
Print file selection using StdFile.
Icons via bundles.
Changing of the default print record and screen parameters.
..and making PrApp a "Real" application.

Initialization

The Init/Termination code consists of:

```
PROCEDURE PrOpen;
PROCEDURE PrClose;
```

These Open/Close the printing code by opening/closing two files: the current print resource file [ImageWriter.rsrc, for example] and the Print driver [.Print] which is part of the System.rsrc file.

```
PROCEDURE PrintDefault ( hPrint: THPrint );
```

This fills a handle to the defaulted Print record from the current print resource file. This does not actually dialog. It is used to initialize new "stationary" and to let "listing" style applications to get a valid Print record for printing without dialogs. The handle must be pre-allocated as 120 bytes.

```
FUNCTION PrValidate ( hPrint: THPrint ): Boolean;
```

Performs a validity check on the Print record, correcting it if invalid.
Returns True if the record was changed due to being invalid, False otherwise [ie: fChanged].
The current validity check is for software version number and for printer type. If either are not current, the hPrint record is simply set to the current printer's default values.
Note: This also updates the information sub-records: PrInfo, PrXInfo etc according to the current values in the PrStl and PrJob sub-records of the Print record.
This insures that these "dependent variables" are in synch with their "independent variables", the PrStl and PrJob. The returned boolean is not affected if there was a change in these values.

Dialogs

The Dialogs maintain the primary printing data structure, the "Print" record. Note that one of these should be stored in each of your documents so that client use of printing can be "remembered". We'll discuss this protocol more later. The dialog procs are:

```
FUNCTION PrStlDialog ( hPrint: THPrint ): Boolean;
```

The Mac Applications provide "visual fidelity", i.e. "What you see is what you get". This means that you must know something about the printing request *before* it is actually made! The PrStlDialog asks for the part of the print request that causes one print job to vary from another. For most printers, this is simply the page size and orientation for the document. No guarantee is made that this will always be so, however! The guarantee is only that enough information is obtained to fill out the part of the Print record called the PrInfo data structure, which will be described further below.

```
FUNCTION PrJobDialog ( hPrint: THPrint ): Boolean;
```

The rest of the Print record is filled out by this dialog. It mainly consists of the page range and number of copies. For the Image Writer it also has the HiRes/LoRes/Draft choice and the type of paper feed.

The boolean result for both dialogs is the DoIt button: if true, the Client has clicked "OK". The Print record should be saved and, for the PrJob, the document spooled and/or printed. Note that the initial button settings are derived from the existing Print record values, and should be either an old, valid one, or a new, defaulted one. Each dialog calls PrValidate for you.

```
PROCEDURE PrJobMerge (hPrintSrc, hPrintDst: THPrint);
```
"Merges" hPrintSrc's PrJob into hPrintDst [Source/Destination]. Validates both records before using them. Updates the associated "Info" records.

This procedure allows one print job dialog to be applied to several documents. The main use is for printing from the Finder. See "Usage" discusson below for details.

Applications may add their own buttons to the printing dialogs to customize them for their own purpose. For example, you could add margins to the PrStlDialog and row/column selection to the PrJobDialog. See me for details and samples of how MicroSoft is doing it.

Spooling and Draft Printing

The Spooling/Draft procs do one of two things: Spool a print file to disk, or provide for an Ascii like style of printing. Both are provided by setting up a graf port and intercepting QuickDraw calls by using our own versions of the QuickDraw bottleneck procs. Thus these embody the minimum use of printing by an App: you can either do "Cheap" ascii printing or spool the file to be printed later, "offline", by the Printer Application that will read and print the file. The interface is via four procs that "bracket" calls to QuickDraw:

```
FUNCTION PrOpenDoc ( hPrint:    THPrint;
                     pPrPort:   TPPrPort;
                     pIOBuf:    Ptr ): TPPrPort;
```
Initialize the printing code for this document. The hPrint parameter is a handle to a valid Print record. The pPrPort is similar to the Window Manager's Storage parameter: if Non-NIL, I use it, rather than calling NewPtr. The IO Buffer pointer is passed allong to the OS: if NIL, it uses the volume buffer, otherwise it uses yours. The returned pointer is to the initialized Print "Port" which is simply a GrafPtr, its associated bottleneck procs, and a few extra longs for me. The code will initialize for Draft printing or for Pic file spooling by looking at the fDraft flag in the hPrint data. If the hPrint has a non-NIL IdleProc, it will be called by the draft printing proc.

```
PROCEDURE PrCloseDoc ( pPrPort: TPPrPort );
```
Puts the above stuff to bed: Flushes the Pic file directories or closes the print driver for draft printing.

```
PROCEDURE PrOpenPage ( pPrPort: TPPrPort; pPageFrame: TPRect );
```
Initializes the next page. The page frame rectangle is for wizards: set it to NIL.

```
PROCEDURE PrClosePage( pPrPort: TPPrPort );
```
Cleans up the Pic file data structures or ejects the current page.

Picture & Bitmap Printing

The Pic printing procs are the standard way to print. A third proc is provided to do simple bitmap printing.

```
PROCEDURE PrPicFile( hPrint:        THPrint;
                     pPrPort:       TPPrPort;
                     pIOBuf:        Ptr;
                     pDevBuf:       Ptr;
                     VAR PrStatus:  TPrStatus );
```
This reads and prints the spooled print file. If an IdleProc is included in the Print record, it is run both during imaging and writing to the serial port. The first three parameters are identical to the PrOpenDoc parameters. The device buffer is the "band" buffer and associated data. If NIL, I allocate it. Its size is Print.PrXInfo.iDevBytes. The status record simply records the progress of printing and may be used by the IdleProc. See PrApp for its use.

```
PROCEDURE PrPic ( hPic:          PicHandle;
                  hPrint:        THPrint;
                  pPrPort:       TPPrPort;
                  pDevBuf:       Ptr;
                  VAR PrStatus:  TPrStatus );
```

Simply prints from your picture rather than the spooled file.

```
PROCEDURE PrCtlCall (iWhichCtl: Integer; lParam1, lParam2, lParam3: LongInt);
```

..is a general control call to the Print driver. In particular:

```
PROCEDURE PrCtlCall ( iPrBitsCtl{= 4}          --The bitmap printing control
                      pBitMap:   Ptr;          --QuickDraw bitmap
                      pPortRect: TPRect;       --a portrect. use bounds for whole bitma
                      lControl:  LongInt );    --0=>Screen resolution/Portrait
```

..dumps a bitmap to the printer. lControl is a device dep param; use 0 for screen printing. Thus PrCtlCall (iPrBitsCtl, @MyPort^.ScreenBits, @MyPort^.PortRect.Bounds,0) performs a screen dump of just my port's data. See the Print Driver memo for more details.

Usage

The Printer is initialized by calling the PrOpen procedure. You may either keep the printer open all the time, or bracket every print call with a PrOpen/PrClose pair.

Each document has its own Print record and must store it in the document file. This allows the client to configure their documents once rather than each time they open the document. To get a vanilla Print record, simply call the PrintDefault procedure. [Note: Non-document printing, listings, for example, may always use defaulting.]

When an existing document is reopened, the PrValidate procedure must be called. This allows the client to change printers with the Printer application and automatically update their Print records.

The two printer dialogs are accessed by menu items. Each returns the DoIt button. The Print record should be updated in the document file whenever fDoIt is True. After a PrJobDialog is called returning True, the following print loop is generally used:

```
pMyPort := PrOpenDoc  ( hPrint, pMyPort, pIOBuf );

FOR iPage := 1 TO iPages DO BEGIN    { ..or WHILE NOT EOF(myDoc) DO BEGIN }
   PrOpenPage ( pMyPort, NIL );

{ Here you image the current page by calling QuickDraw! The drawing proc
  will need the page size and printer resolution stored in the Print.PrInfo. }

   PrMyPage (iPage, hPrint^^.PrInfo);

   PrClosePage (pMyPort);
END;

PrCloseDoc (pMyPort);
```

This will either spool or draft print the document. If you are printing from your Application rather than the Printer Application, you will do the following:

```
IF hPrint^^.PrJob.bJDocLoop=bSpoolLoop THEN BEGIN
   SwapMeOut;
   PrPicFile (....)
   SwapMeIn;
END;
```

The "SwapMeOut" procedure swaps as much of you out as reasonable. You may also set flags

for your GrowZone procedure to let it try to do a last ditch swapout, or at least Alert the client that the Printer Application can be used to print the document. PrPicFile then takes over the machine to print the spooled file. Your "SwapMeIn" brings your world back in.

Printing from the Finder may be done however the App sees fit. The two most obvious approaches are:

-Simply use the doc's current Print record. You should first set the page range and number of copies to [1, 999] and 1. This will print the doc exactly as it last printed. This allows the job mix to include draft, low res and high res. You should call PrValidate before calling the PrPicFile procedure.

-Put up just one Print Job dialog and apply it to each doc in the queue. This is slightly more involved than the above because you want each doc's formatting information to be preserved. Here's how to do it:

```
     Get the first doc's Print record and perform a PrJobDialog with it.
     {Note that you don't have to call PrValidate here because the dialog does.]
     Print it.
     For each additional doc do:
        Get the Print record for this doc.
        Call PrJobMerge.
        Print it.
```

The advantage of the first style is that no dialogs occur, thus making is easy for someone to submit several documents from different Application for printing. The advantage of the second is that it allows one PrJob dialog to be spread over several documents. A third possibility is to have the Finder call PrJobDialog with a defaulted Print record and pass it as a parameter in the AppParam record. The Apps would then use the second method, possibly with a Print Shop proc to move in the job and validate (update) the Print record.

It is important to stress the use of the PrInfo sub-record when imaging your document. The PrInfo record contains the device dependent parameters for the current printer. If carefully used, it provides the Application with "parametric device independence".

```
  TPrInfo = RECORD
     iDev:       Integer;    {Font mgr/QuickDraw device code}
     iVRes:      Integer;    {Resolution of device, in device coordinates}
     iHRes:      Integer;    {    ..note: V before H => compatable with Point.}
     rPage:      Rect;       {The page (printable) rectangle in device coordinates.}
  END;
```

The most important field is the page rectangle. This gives the current paper size in bits. The next is the h/v resolution, in spots per inch. Finally, there is the QuickDraw - FontMgr device number. This lets you get the metrics for the printing fonts, so that you can adjust for screen-printer differences. Correct use of these will result in a very surprising degree of printer independence.

In addition PrInfo, there is another field in the Print record: rPaper: Rect which gives the paper rectangle in which the PrInfo.rPage is embedded, in device coordinates. This is "Outset" from the rPage rectangle, whch always has 0,0 top left coordinates. Its use is for margin calculations.

The Print record contains an IdleProc pointer that is set to Nil by the PrJobDialog. By changin this to your own procedure, you may simulate multi-processing! For example, you can look at either PrTest or PrApp to see how you may run the ornaments while printing. If you don't provid an IdleProc, we provide a simple Comand Period abort procedure; you should post an alert/dialog informing the client that this is available.

Spooling may be to files other than the default spool file (whose name is configurable by the client). The file name is provided as a string pointer in the Print record. It is set to Nil by the PrJobDialog. This causes spooling to the default file name. Simply provide

your own file name if you'd like to. This is especially useful for Applications that cannot print from within themselves. The volume/version parameters are similarly changable.

Notes

Errors: The Alert (ha) reader will have noticed a lack of error returns from printing. Wher we can issue our own Alerts, we do so. Other errors are handled by posting the error in the printer globals. The first integer is the current error number. In case the user's disk does not have a printing resource file, or it is incorrectly named in System.rsrc, or there is no .Print driver; we simply post an error and No-op in the PrLink code. We do not alert from PrLink.

Size: Printing is really a Mini-Application rather than a library. The code is currently roughly 8K. But this is really a small part of the cost of printing; even if the code were "free", the data used by printing can be huge. Current sizes [10/6]:

Code:	.Print Driver	= 780
	PrLink	= 374
	Dialogs	= 2,226
	Spooling	= 1,294
	Draft	= 2,134
	Pic Printing	= 4,630
Data:	Bands	= 6K for HiRes
	Picture	= 5K to 15K, typically. Max = 32K
	Fonts	= 3K to 10K, typically. Max = 32K
	QuickDraw Buf	= 2K to 6K, typically; up to 12K for 24 Pt shadow HiRe

This is why a separate Printer Application is provided: you may simply spool and let the client use it. For spread sheets, for example, Draft printing may be adequate for most uses.

Bands: The size of a HiRes page bitmap is in the order of 1/4 Megabyte! Printing handles this by breaking the page into smaller "bands" and alternates imaging and printing t to print a page. For example, a HiRes US Letter size page has 47 bands of 5120 byte for a total of 240,640 bytes. LoRes is 24 bands of 2560 bytes for a total of 61,440 [Note: as resolution doubles, data volume quadruples!] Please note that even though may appear gargantuan, the printer uses far less than the screen's 20K!

Spooling: Even if you plan to try to print from within the App, spooling is useful! It allows you to have a very clean swapping stratagy: First you spool, with only 3K of printi code and no more than 1K data. This may require much of your code and data to be re But when the data is spooled, you can swap all of your code and data out and call PrPicFile from its own micro segment! The banding stratagy requires very fast imagi of the data if it is to be drawn 50 times per page. This is another reason for spoo Pic drawing is optimal use of QuickDraw! Generally spooling is done to the default Print File whose name is stored the Print.rsrc file for the translators to change. You may over-ride this by setting three fields in the Print record:

```
pFileName:  TPStr80;       {Spool File Name: NIL for default.}
iFileVol:   Integer;       {Spool File vol, set to 0 initially}
bFileVers:  SignedByte;    {Spool File version, set to 0 initially}
```

These are set to NIL, 0, 0 by the Print Job dialog. Change them if you'd like.

Draft: Draft printing is a compromise between Ascii/WriteLn/Fast printing and QuickDraw. One of the strongest Mac attitudes is the full use of the QuickDraw style of imaging Note that there are NO WriteLn's available for Mac programmers, and NO programs use command line interface! I decided that the best compromise was Draft printing. It called "Draft" mainly because it "simulates" the output you will get when you print with standard printing. It is done by simply installing QuickDraw capture procs and translating them to the printer's command codes. It thus can provide full use of th printer's native capabilities, such as bold, underline, fonts, line plotting, etc. It also requires no special interface such as WriteLn; thus the standard Apps get it without even being aware its happening! ..Its completely under client control!

"But what do I do if all I want to do is make a Listing?!" Well, its really not so bad: simply get a default Print record, and make your own WriteLn! The only real headache is having to be aware of the Page boundaries. But the advantage is that the results will nicely fit European paper sizes, and you'll probably find that "Pretty Printing" will be so easy that you'll provide it. Note that it also lets you provide Spooled standard printing by simply changing one flag!

Idle: The Print record has a IdleProc: Print.PrJob.pIdleProc. It is always returned NIL b the default and dialog procs. To use it, simply stuff it with your own proc after t PrJobDialog and before calling the Draft or Pic Printing procs. A word of caution, however! The "concurrancy" problems caused by the Idle proc are subtle and many! Look at the PrTest and PrApp samples for how they do it. The major problem is makin sure the GrafPort is reset to mine when returning from your idle. Also, DON'T allow calling Printing procs while idling. They are accessed through the PrLink code whic is not re-entrant. The suggested idle proc is one polling for Cmd "." aborts. To a printing, simply put iPrAbort into iPrErr in the PrintVars in low memory.

Release: Billions of files are released on the MacPrint disk, only four of which you need: PrLink to Link with, either MacPrint.obj or PrEqu.text to compile/assemble with, the current print resource to run with, and PrApp to let your client print spooled files with. New releases of printing simply use a new print resource, even if addin a new printer! Note however, that you must have the newest System.rsrc file which contains two vital printing resources: the .Print driver and a string containing the file name of the current printing resource [=ImageWriter]. If you have an older System.rsrc, it can be updated with these, using RMover, by pasting the small file PrSys.rsrc included in the printing release into System.rsrc.

TO: MacPrint Users **DATE:** December 7, 1983

SUBJECT: The .Print Driver **FROM:** Owen Densmore

Introduction

The MacPrint system is packaged as three separate items:
- -Three entries in System.rsrc, the .Print driver being the major one,
- -The Printer resource file, currently "ImageWriter",
- -The Printer Application program, "Printer".

The three entries in the System.rsrc file are:
- -The .Print driver [ResType=DRVR, ResID=2, ResName=".Print", RefNum=$FFFD],
- -A parameter record used to configure the .Print driver[ResType=PREC, ResID=2].
- -A string naming the current printer [ResType=STRG, ResID=$E000(-8192)],

The System.rsrc string is used to locate the "current" printer, and allows the translators to name the printer appropriately for the target country. This allows several printers to be available at once, but with only one being active. The installation of a new printer (or renaming of the existing one) is done by installation dialogs in the Printer application program. Thus I may change my ImageWriter's name to "StarChild" by simply editing its name in the Finder and using the Printer application to install it.

The .Print driver and its associated parameter record reside ***both*** in System.rsrc ***and*** in the printer resource file. This redundancy is necessary for the installation scheme discussed above. We require a place in System.rsrc so that the system itself can use the current printer for screen printing. It also allows applications not using the complete printing system to do bitmap-only printing easily. The copies in the printer resource are never executed; they are simply used as a storage area for installing into System.rsrc.

This document discusses the use of the Printer driver. The printer resource is documented in the "MacPrint Interface" specification memo.

Driver Calls

The driver contains the following general calls:

```
Status: Returns the Font Manager's device information record.
Controls:
      Control 4 = Bitmap Printing,
      Control 5 = Block IO to printer,
      Control 6 = Keyboard event controls,
      Control 8 = Font Manager's font selection over-ride option.
```

The Bitmap Printing control (4) uses three long parameters for printing a portion of a bitmap:

```
CSParam     = pointer to QuickDraw bitmap,
CSParam+4   = pointer to rectangle within bitmap, in local coordinates,
CSParam+8   = a device dependent parameter; use 0 for screen printing.
```

Thus to print the entire screen: --or-- the contents of a window:

```
CSParam    = screenBits,                CSParam    = window.portBits,
CSParam+4  = screenBits.bounds,         CSParam+4  = window.portRect,
CSParam+8  = 0.                         CSParam+8  = 0.
```

The Block IO control (5) uses three long parameters for writing a block of raw data to the printer. It's primarily useful for ascii printing, form feeds, cr's etc and for use by the higher level printing code for sending escape commands to the printer. The parameters are:

```
CSParam     = pointer to block,
CSParam+4   = a long count of bytes,
CSParam+8   = a pointer to an Idle procedure; use NIL for none.
```

The Print Event control (6) uses one long parameter for handling two special cases of Bitmap printing. The parameter's format is:

```
CSParam     = Event "Message"; which is formatted as follows:
      Byte 3=0,
      Byte 2=$FF for printing screen, $FE for printing just the top folder,
      Bytes 1 & 2=$FFFD,.the driver's RefNum.
```

Thus $00FFFFFD is the screen dump message, while $00FEFFFD is the top folder message.

The Font Manager control (8) is the "tail hook" that is called by the Font Manager after it responds to a QuickDraw request for a new font. It allows a device to over-ride the Font Manager's selection heuristic.

The following constants are in our "Printing Equates" file, "PrEqu.text":

```
;-----------------------------------------------------------------------
; These are the PrDrvr constants.
;-----------------------------------------------------------------------
iPrDrvrID      .EQU   2           ;Driver's ResID
iPrDrvrRef     .EQU   $FFFD       ;Driver's RefNum = NOT ResID
iPrBitsCtl     .EQU   4           ;The Bitmap Print Proc's ctl number
iPrIOCtl       .EQU   5           ;The Raw Byte IO Proc's ctl number
iPrEvtCtl      .EQU   6           ;The PrEvent Proc's ctl number
lPrEvtAll      .EQU   $00FFFFFD   ;The PrEvent Proc's CParam for the screen
lPrEvtTop      .EQU   $00FEFFFD   ;The PrEvent Proc's CParam for the top folder
iFMgrCtl       .EQU   8           ;The FMgr's Tail-hook
```

..there are similar definitions in the MacPrint Pascal interface.

Unit PrScreen, a simple Driver interface

The unit PrScreen [<100 bytes!] contains a very simple Pascal interface to .Print. The interface definition is included in MacPrint. You may also simply declare these as External references if you'd rather not {$Use MacPrint}.

```
PROCEDURE PrDrvrOpen;
PROCEDURE PrDrvrClose;
PROCEDURE PrCtlCall (iWhichCtl: Integer; lParam1, lParam2, lParam3: LongInt);
```

The first two simply open & close the driver in System.rsrc. Be careful not to close the driver if someone else has opened it before you got there!

The third is simply a generalized control call to the driver. It takes a control call number, and up to three parameters. Thus the above controls are accessed by:

```
PrCtlCall (iPrBitsCtl, pBitMap, pPortRect, lControl);
PrCtlCall (iPrEvtCtl, lPrEvtAll, 0, 0);
PrCtlCall (iPrEvtCtl, lPrEvtTop, 0, 0);
PrCtlCall (iPrIOCtl, pBuf, lBufCount, pIdleProc);
```

These constants are also declared in MacPrint:

```
iPrBitsCtl = 4;          {The Bitmap Print Proc's ctl number}
iPrIOCtl   = 5;          {The Raw Byte IO Proc's ctl number}
iPrEvtCtl  = 6;          {The PrEvent Proc's ctl number}
lPrEvtAll  = $0002FFFD;  {The PrEvent Proc's CParam for the entire screen}
lPrEvtTop  = $0001FFFD;  {The PrEvent Proc's CParam for the top folder}
```

```
iFMgrCtl   = 8;          {The FMgr's Tail-hook Proc's ctl number}
```

MACINTOSH USER EDUCATION

The Memory Manager: A Programmer's Guide /MEM.MGR/MEMORY

See Also: The Resource Manager: A Programmer's Guide

Modification History: First Draft (ROM 7) S. Chernicoff 10/10/83

ABSTRACT

This manual describes the Memory Manager, the part of the Macintosh Operating System that controls the dynamic allocation of memory space on the heap.

TABLE OF CONTENTS

3 About This Manual
4 About the Memory Manager
7 Pointers and Handles
8 How Heap Space Is Allocated
12 The Stack and the Heap
13 Utility Data Types
15 Memory Manager Data Structures
15 Structure of Heap Zones
18 Structure of Blocks
20 Structure of Master Pointers
21 Result Codes
22 Using the Memory Manager
24 Memory Manager Routines
25 Initialization and Allocation
29 Heap Zone Access
30 Allocating and Releasing Relocatable Blocks
35 Allocating and Releasing Nonrelocatable Blocks
38 Freeing Space on the Heap
42 Properties of Relocatable Blocks
44 Grow Zone Functions
47 Utility Routines
48 Special Techniques
48 Dereferencing a Handle
50 Subdividing the Application Heap Zone
53 Creating a Heap Zone on the Stack
54 Notes for Assembly-Language Programmers
54 Constants
55 Global Variables
55 Trap Macros
56 Result Codes
56 Offsets and Masks
58 Handy Tricks
59 Summary of the Memory Manager
62 Glossary

Copyright (c) 1983 Apple Computer, Inc. All rights reserved.
Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Memory Manager, the part of the Macintosh Operating System that controls the dynamic allocation of memory space on the heap. *** Eventually it will become part of a larger manual describing the entire Operating System. ***

(eye)

This manual describes version 7, the final, "frozen" version of the Macintosh ROM. Earlier versions may not work exactly as described here. *** There may someday be one or more special, RAM-based versions of the Memory Manager for software development purposes, doing more extensive error checking or gathering statistics on a program's memory usage. This manual describes the ROM-based version only. ***

Like all Operating System documentation, this manual is intended for both Pascal and assembly-language programmers. All readers are assumed to be familiar with Lisa Pascal; information of interest only to assembly-language programmers is isolated and labeled so that Pascal programmers can conveniently skip it. Whichever is your preferred language, please bear with occasional remarks addressed solely to the other group.

The manual begins with an introduction to the Memory Manager and what it's used for. It then discusses some basic concepts behind the Memory Manager's operation: how blocks of memory are allocated within the heap and how the allocated blocks are referred to by programs that use them. Following this is a discussion of the internal data structures that the Memory Manager uses to find its way around in the heap.

A section on using the Memory Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Memory Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not be of interest to all readers. Special information is given on unusual techniques that you may find useful in working with the Memory Manager and on how to use it from assembly-language programs.

Finally, there is a quick-reference summary of the Memory Manager's data structures and routines, along with a glossary of terms used in this manual.

ABOUT THE MEMORY MANAGER

Using the Memory Manager, your program can maintain one or more independent areas of heap memory (called heap zones) and use them to allocate blocks of memory of any desired size. Unlike stack space, which is always allocated and released in strict LIFO (last-in-first-out) order, blocks on the heap can be allocated and released in any order, according to your program's needs. So instead of growing and shrinking in an orderly way like the stack, the heap tends to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 1. The Memory Manager does all the necessary "housekeeping" to keep track of the blocks as it allocates and releases them.

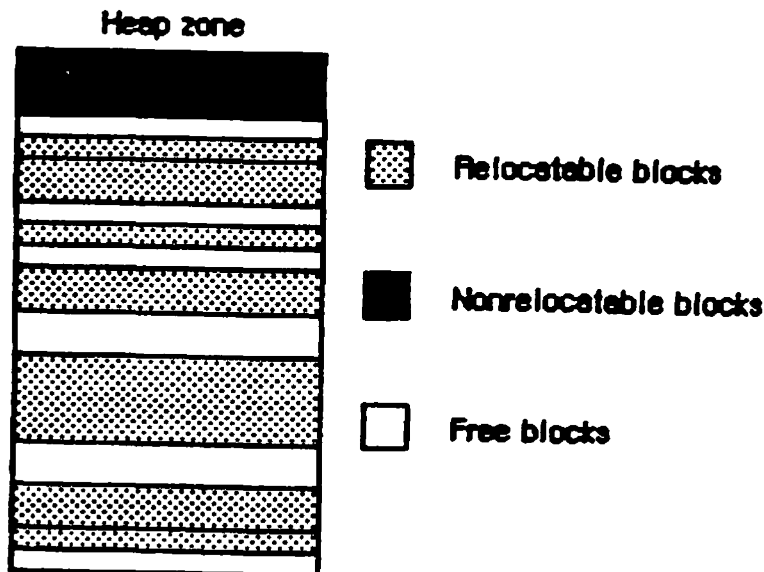


Figure 1. A Fragmented Heap

All memory allocation is performed within a particular heap zone. The Memory Manager always maintains at least two heap zones: a system heap zone, reserved for the system's own use, and an application heap zone for use by your program. The system heap zone is initialized to 16K bytes when the system is started up. Objects in this zone remain allocated even when one application terminates and another is launched. The application heap zone is automatically reinitialized at the start of each new application program, and the contents of any previous application zone are lost. The initial size of the application zone is 6K bytes, but it can grow as needed to create more heap space while the program is running. Your program can create additional heap zones if it chooses, either by subdividing this original application zone or by allocating space on the stack for more heap zones.

(hand)

In this manual, unless otherwise stated, the term "application heap zone" (or just "application zone")

always refers to the original application heap zone provided by the system, before any subdivision.

Various parts of the Macintosh Operating System and Toolbox also use space in the application heap zone. For instance, the actual machine-language code of your program resides in the application zone, in space reserved for it at the request of the Segment Loader. Similarly, the Resource Manager requests space in the application zone to hold resources it has read into memory from a resource file. Toolbox routines that create new entities of various kinds, such as NewWindow, NewControl, and NewMenu, implicitly call the Memory Manager to allocate the space they need.

At any given time, there is exactly one current heap zone, to which most Memory Manager operations implicitly apply. You can control which heap zone is current by calling a Memory Manager procedure. Whenever the system needs to access its own (system) heap zone, it saves the setting of the current heap zone and restores it later, so that the operation is transparent to your program.

Space within a heap zone is divided up into contiguous pieces called blocks. The blocks in a zone fill it completely: every byte in the zone is part of exactly one block, which may be either allocated (reserved for use by your program or by the system) or free (available for allocation). Each block has a block header containing information for the Memory Manager's own use, followed by the block's contents, the area available for use (see Figure 2). There may also be some unused bytes at the end of the block, beyond the end of the contents.

Assembly-language note: Blocks are always aligned on even word boundaries, so you can access them with word (.W) and long-word (.L) instructions.

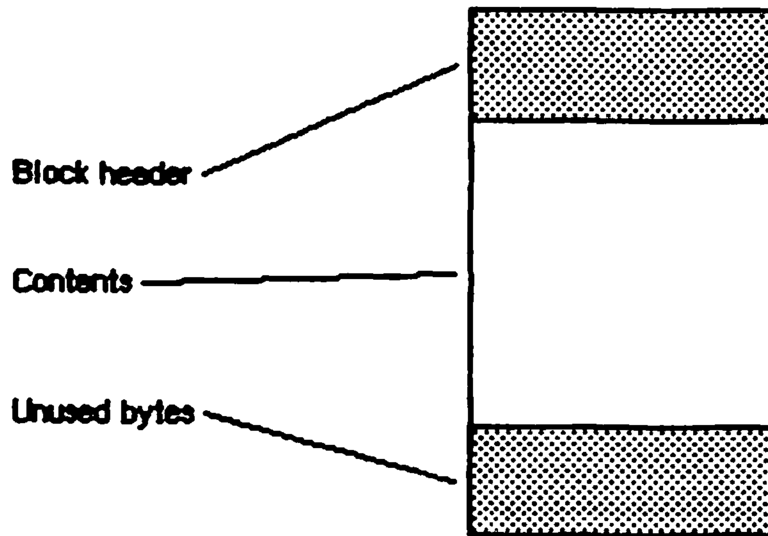


Figure 2. A Block

A block can be of any size, limited only by the size of the heap zone itself. What's inside the block is of no concern to the Memory Manager: it may contain data being used by your program, executable code forming part of the program itself, resource information read from a resource file, or anything else that may be appropriate. To the Memory Manager, it's just a block of a certain size.

(hand)
Don't confuse the blocks manipulated by the Memory Manager with disk blocks, which are always 512 bytes long.

An allocated block may be relocatable or nonrelocatable; if relocatable, it may be locked or unlocked; if unlocked, it may be purgeable or unpurgeable. Relocatable blocks can be moved around within the heap zone to create space for other blocks; nonrelocatable blocks can never be moved. These are permanent properties of a block that can never be changed once the block is allocated. The remaining attributes (locked and unlocked, purgeable and unpurgeable) can be set and changed as necessary. Locking a relocatable block prevents it from being moved, but only temporarily: you can unlock the block at any time, again allowing the Memory Manager to move it. Making a block purgeable allows the Memory Manager to remove it from the heap zone, if necessary, to make room for another block. (Purging of blocks is discussed further below under "How Heap Space Is Allocated".) A newly allocated block is initially unlocked and unpurgeable.

POINTERS AND HANDLES

Relocatable and nonrelocatable blocks are referred to in different ways: nonrelocatable blocks by pointers, relocatable blocks by handles (discussed below). When the Memory Manager allocates a new nonrelocatable block, it returns a pointer to the block. Thereafter, whenever you need to refer to the block, you use this pointer. Like any other pointer, it's simply a memory address: that of the first byte in the block's contents (see Figure 3). You can make as many copies of this pointer as you like. Since the block they point to can never be moved within its heap zone, you can rely on all copies of the pointer to remain correct. They will continue to point to the block for as long as the block remains allocated.

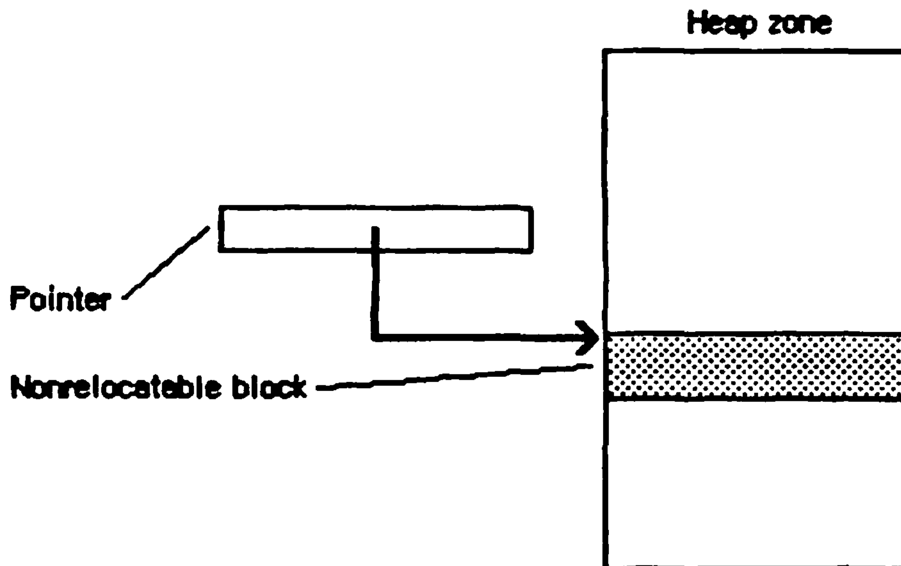


Figure 3. A Pointer to a Nonrelocatable Block

Relocatable blocks don't share this property, however. If necessary to make room for some other block, the Memory Manager can move a relocatable block at any time to a new location in its heap zone. This would leave any pointers you might have to the block pointing to the wrong place in memory, or "dangling". Dangling pointers can be very difficult to diagnose and correct, since their effects typically aren't discovered until long after the pointer is left dangling.

To help avoid dangling pointers, the Memory Manager maintains a single master pointer to each relocatable block, allocated from within the same heap zone as the block itself. The master pointer is created at the same time as the block and set to point to it. What you get back from the Memory Manager when you allocate a relocatable block is a pointer to the master pointer, called a handle to the block (see Figure 4). From then on, you always use this handle to refer to the block. If the Memory Manager later has to move the block, it has only to

update the master pointer to point to the block's new location; the master pointer itself is never moved. Since all copies of the handle point to the block by double indirection through this same master pointer, they can be relied on not to dangle, even after the block has been moved.

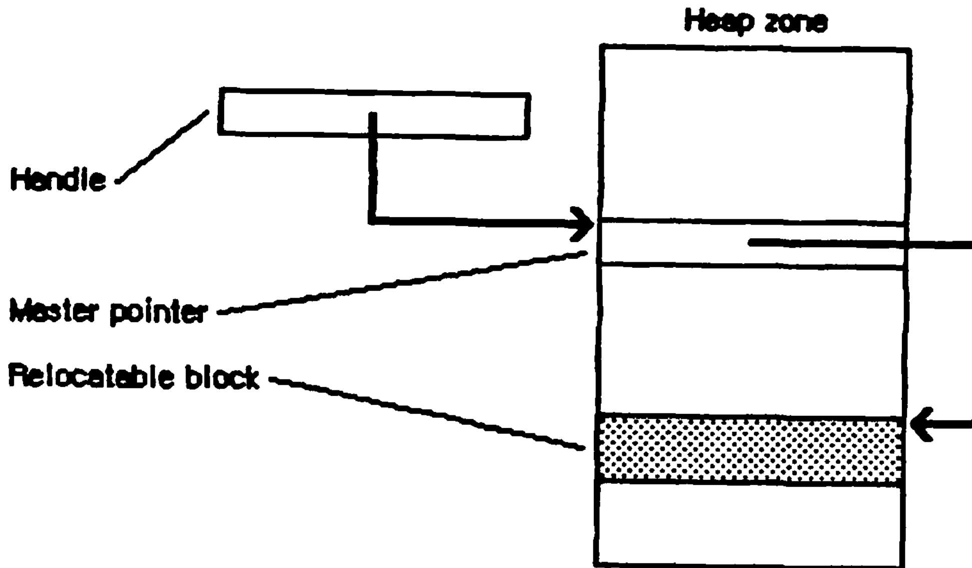


Figure 4. A Handle to a Relocatable Block

(eye)

To maintain the integrity of the memory allocation system, always use the Memory Manager routines provided (or other Operating System or Toolbox routines that call them) to allocate and release space on the heap. Don't use the Pascal standard procedures NEW and DISPOSE. *** Eventually the versions of these routines in the Pascal Library will be changed to work through the Memory Manager. ***

HOW HEAP SPACE IS ALLOCATED

The Memory Manager allocates space in a heap zone according to a "first fit" strategy. When you ask to allocate a block of a certain size, the Memory Manager scans the current heap zone looking for a place to put the new block. For relocatable blocks, it looks for a free block of at least the requested size, scanning forward from the end of the last block allocated and "wrapping around" if necessary from the end of the zone to the beginning. (Nonrelocatable blocks are handled a bit differently, as described below.) As soon as it finds a free block big enough, it allocates the requested number of bytes from that block. That is, it uses the first free block it finds that's big enough to satisfy the request, instead of continuing to search for a better fit.

If a single free block can't be found that's big enough, the Memory Manager tries to create one by compacting the heap zone: moving allocated blocks together in order to collect the free space into a single larger free block (see Figure 5). Only relocatable, unlocked blocks can be moved. The compaction continues until either a free block of at least the requested size has been created or the entire heap zone has been compacted.

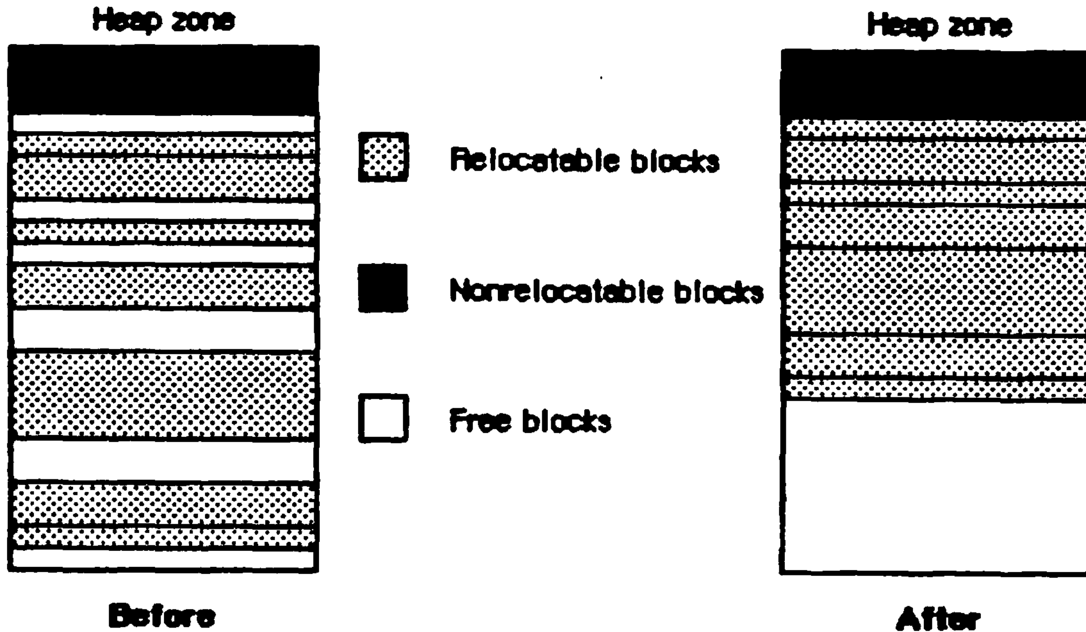


Figure 5. Heap Compaction

Notice that nonrelocatable blocks (and relocatable ones that are temporarily locked) tend to interfere with the compaction process by forming immovable "islands" in the heap. This can prevent free blocks from being collected together and lead to fragmentation of the available free space, as shown in Figure 6. To minimize this problem, the Memory Manager tries to keep all the nonrelocatable blocks together at the beginning of the heap zone. When you allocate a nonrelocatable block, the Memory Manager will do everything in its power to make room for the new block at the lowest available position in the zone, including moving other blocks upward, expanding the zone, or purging blocks from it (see below).

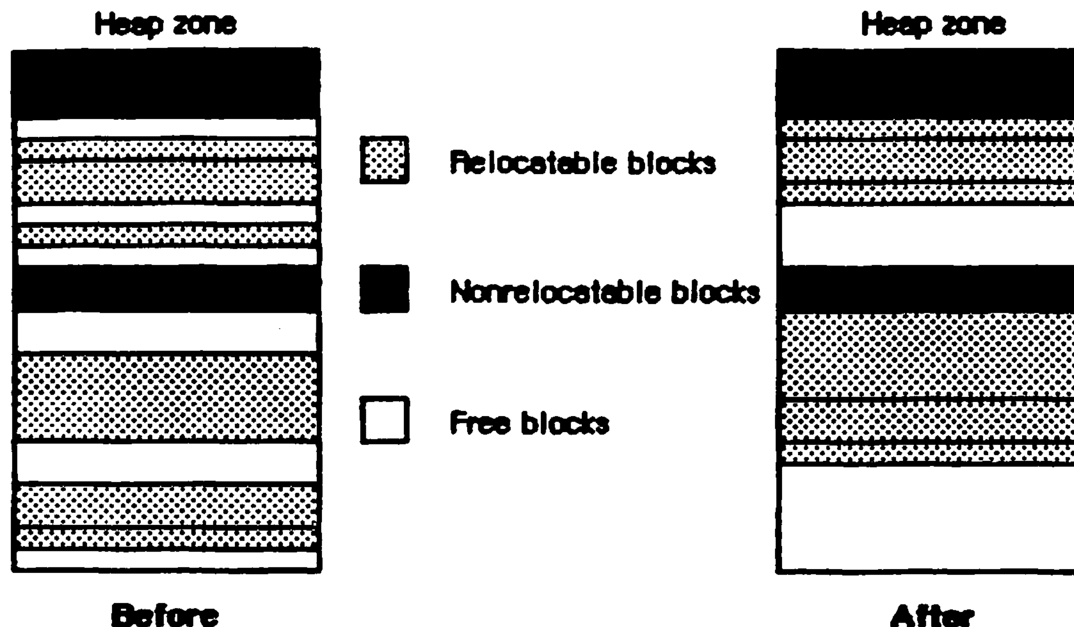


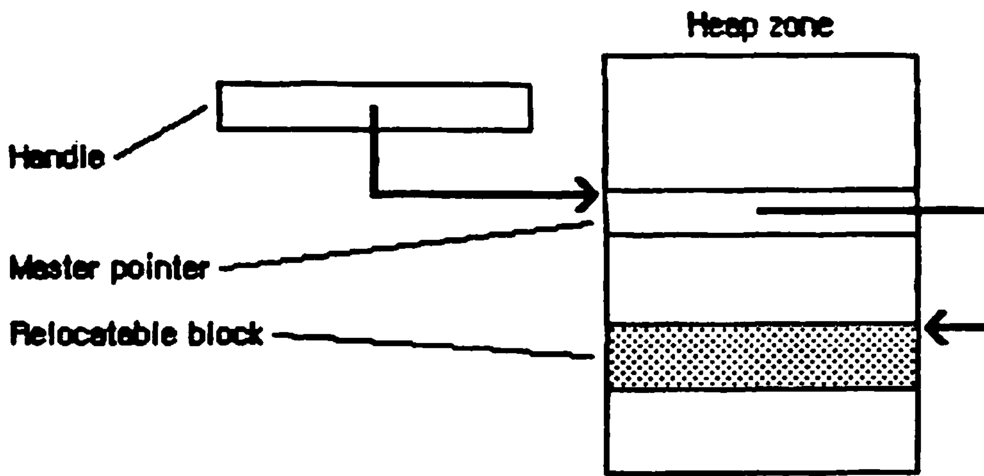
Figure 6. Fragmentation of Free Space

If the Memory Manager still can't satisfy the allocation request after compacting the entire heap zone, it next tries expanding the zone by the requested number of bytes, rounded upward to the nearest 1K. Only the original application zone can be expanded, and only up to a certain limit (discussed more fully under "The Stack and the Heap", below). If any other zone is current, or if the application zone has already reached or exceeded its limit, this step is skipped.

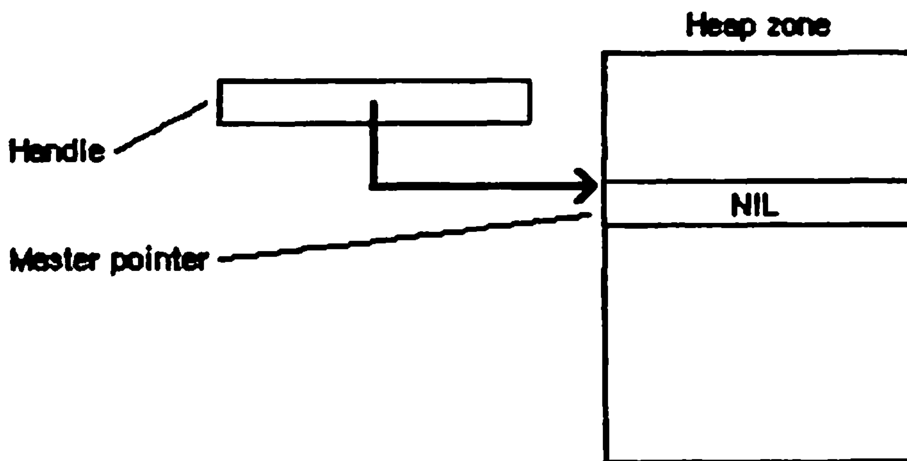
Next the Memory Manager tries to free space by <u>purging</u> blocks from the zone. Only relocatable blocks can be purged, and then only if they're explicitly marked as unlocked and purgeable. Purging a block removes it from its heap zone and frees the space it occupies. The block's master pointer is set to NIL, but the space occupied by the master pointer itself remains allocated. Any handles to the block now point to a NIL master pointer, and are said to be <u>empty</u>. If your program later needs to refer to the purged block, it can detect that the handle has become empty and ask the Memory Manager to <u>reallocate</u> the block. This operation updates the original master pointer, so that all handles to the block are left referring correctly to its new location (see Figure 7).

(eye)

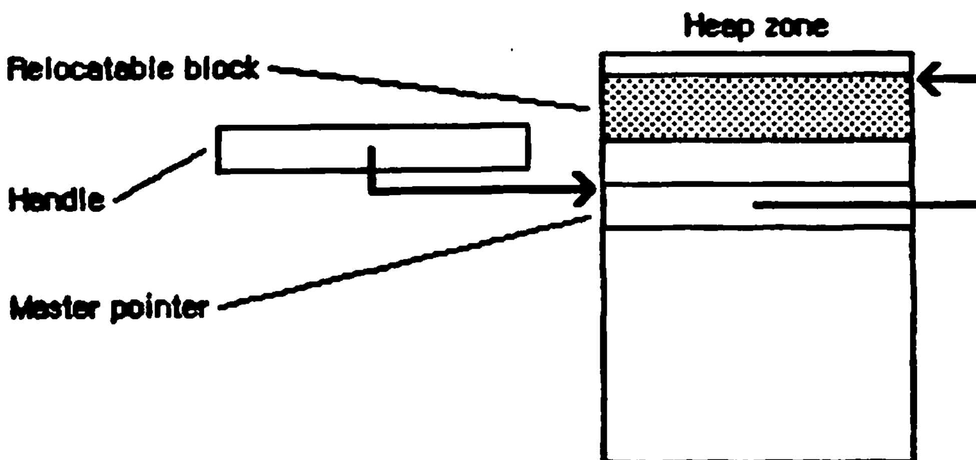
> Reallocating a block only recovers the space it occupies, not its contents. Any information the block contains is lost when the block is purged. It's up to your program to reconstitute the block's contents after reallocating it.



Before purging



After purging



After reallocating

Figure 7. Purging and Reallocating a Block

Finally, if all else fails, the Memory Manager calls the grow zone function, if any, for the current heap zone. This is an optional routine that you can provide to take any last-ditch measures your program may have at its disposal to try to free some space in the zone. The term "grow zone function" is misleading, since the function doesn't actually attempt to "grow" (expand) the zone. Rather, its purpose is to try to create additional free space within the existing zone (such as by purging blocks that were previously marked unpurgeable) or reduce the fragmentation of existing free space (such as by unlocking previously locked blocks). The Memory Manager will call the grow zone function repeatedly, compacting the heap again after each call, until either it finds the space it's looking for or the grow zone function reports that it can offer no further help. In the latter case, the Memory Manager will give up and report that it's unable to satisfy your allocation request.

THE STACK AND THE HEAP

The application heap zone and the application stack share the same area in memory, growing toward each other from opposite ends (see Figure 8). Naturally it would be disastrous for either to grow so far that it collides with and overwrites the other. To help prevent such collisions, the Memory Manager enforces a limit on how far the application heap zone can grow toward the stack. Your program can set this application heap limit to control the allotment of available space betwen the stack and the heap.

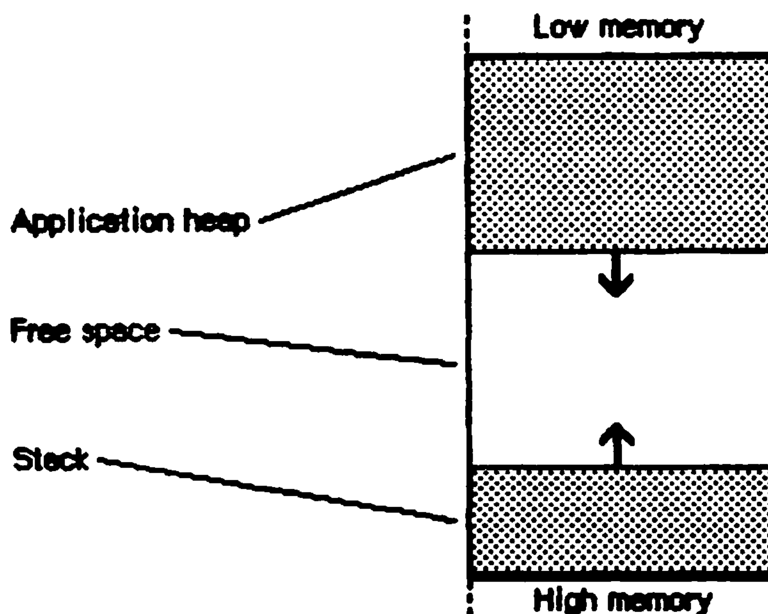


Figure 8. The Stack and the Heap

The application heap limit marks the boundary between the space available for the application heap zone and that reserved exclusively for the stack. At the start of each application program, the limit is initialized to allow 8K bytes for the stack. Depending on your program's needs, you can then adjust the limit to allow more heap space at the expense of the stack or vice versa.

Notice, however, that the limit applies only to expansion of the heap; it has no effect on how far the stack can expand. That is, although the heap can never expand beyond the limit into space reserved for the stack, there's nothing to prevent the stack from crossing the boundary and encroaching on space allotted for heap expansion--or even from overwriting part of the heap itself. It's up to you to set the limit low enough to allow for the maximum stack depth your program will ever need.

(hand)

> Regardless of the limit setting, the application zone is never allowed to grow to within 1K of the current end of the stack. This gives a little extra protection in case the stack is approaching the boundary or has crossed over onto the heap's side, and allows some safety margin for the stack to expand even further.

To help detect collisions between the stack and the heap, a "stack sniffer" routine is run sixty times a second, during the Macintosh's vertical retrace interrupt. This routine compares the current ends of the stack and the heap and opens an alert box on the screen in case of a collision. The stack sniffer can't prevent collisions, only detect them after the fact: a lot of computation can take place in a sixtieth of a second. In fact, the stack can easily expand into the heap, overwrite it, and then shrink back again before the next activation of the stack sniffer, escaping detection completely. The stack sniffer is useful mainly during software development; the alert box it displays can be confusing to your program's end user. Its purpose is to warn you, the programmer, that your program's stack and heap are colliding, so that you can adjust the heap limit to correct the problem before the user ever encounters it.

UTILITY DATA TYPES

The Memory Manager includes a number of type definitions for general-purpose use. For working with pointers and handles to allocated blocks, there are the following definitions:

```
TYPE SignedByte = -128..127;
     Byte       = 0..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;
```

SignedByte stands for an arbitrary byte in memory, just to give Ptr and Handle something to point to. You can define a buffer of bufSize

untyped memory bytes as a PACKED ARRAY [1..bufSize] OF SignedByte. Byte is an alternative definition that treats byte-length data as unsigned rather that signed quantities.

Because of Pascal's strong typing rules, you can't directly assign a value of type Ptr to a variable of some other pointer type. Instead, you have to use the Lisa Pascal functions ORD and POINTER to convert the pointer to an integer address and then back to a pointer. For example, after the declarations

```
VAR aPtr:          Ptr;
    somethingElse: ^Thing;
```

you can make somethingElse point to the same object as aPtr with the assignment

```
somethingElse := POINTER(ORD(aPtr))
```

This works because POINTER returns a generalized "pointer to anything" (like the Pascal pointer constant NIL) that can be assigned to any variable of pointer type or supplied as an argument value for any routine parameter of pointer type.

Type ProcPtr, defined as

```
TYPE ProcPtr = Ptr;
```

is useful for treating procedures and functions as data objects. If aProcPtr is a variable of type ProcPtr and myProc is a procedure (or function) defined in your program, you can make aProcPtr point to myProc by using Lisa Pascal's @ operator:

```
aProcPtr := @myProc
```

Like the POINTER function, the @ operator produces a "pointer to anything". Using it, you can assign procedures and functions to variables of type ProcPtr, embed them in data structures, and pass them as arguments to other routines. Notice, however, that a ProcPtr technically points to a SignedByte, not an actual routine. As a result, there's no way in Pascal to access the underlying routine in order to call it. Only routines written in assembly language (such as those in the Operating System and the Toolbox) can actually call the routine designated by a ProcPtr.

For specifying the sizes of blocks on the heap, the Memory Manager defines a special type called Size:

```
TYPE Size = LongInt;
```

All Memory Manager routines that deal with block sizes expect parameters of type Size or return them as results. To specify a size bigger than any existing block, you can use the constant maxSize:

```
CONST maxSize = $800000;
```

This is an enormous value, equivalent to 8 megabytes or 8,388,608 bytes —more than forty times the Macintosh's total memory capacity!

MEMORY MANAGER DATA STRUCTURES

This section contains detailed information on the Memory Manager's internal data structures. You won't need this information if you're just using the Memory Manager routinely to allocate and release blocks of memory from the application heap zone. The details are included here for programmers with unusual needs (or who are just curious about how the Memory Manager works).

Structure of Heap Zones

Each heap zone begins with a 52-byte zone header and ends with a 12-byte zone trailer (see Figure 9). The header contains all the information the Memory Manager needs about that heap zone; the trailer is just a minimum-size free block (described in the next section) placed at the end of the zone as a marker. All the remaining space between the header and trailer is available for allocation.

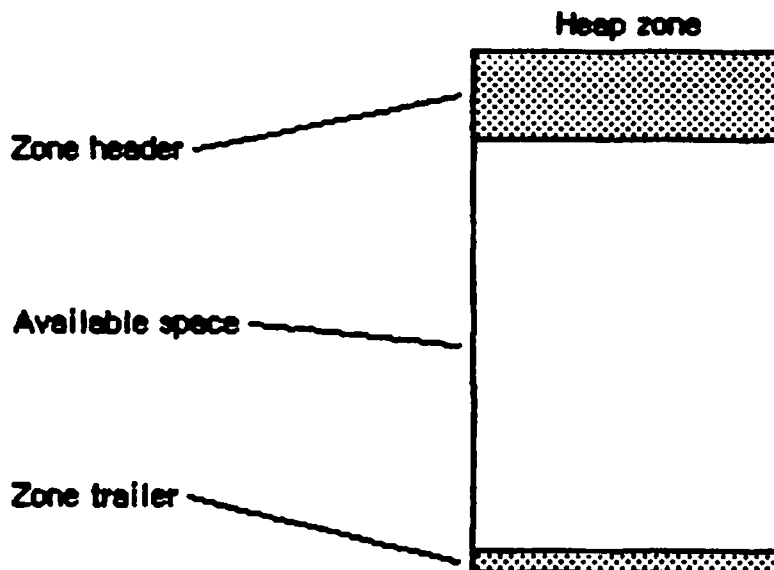


Figure 9. Structure of a Heap Zone

In Pascal, a heap zone is defined as a zone record of type Zone, reflecting the structure of the zone header. It's always referred to with a zone pointer of type THz ("the heap zone"):

```
TYPE THz  = ^Zone;
     Zone = RECORD
                  bkLim:      Ptr;
                  purgePtr:   Ptr;
                  hFstFree:   Ptr;
                  zcbFree:    LongInt;
                  gzProc:     ProcPtr;
                  moreMast:   INTEGER;
                  flags:      INTEGER;
                  cntRel:     INTEGER;
                  maxRel:     INTEGER;
                  cntNRel:    INTEGER;
                  maxNRel:    INTEGER;
                  cntEmpty:   INTEGER;
                  cntHandles: INTEGER;
                  minCBFree:  LongInt;
                  purgeProc:  ProcPtr;
                  sparePtr:   Ptr;
                  allocPtr:   Ptr;
                  heapData:   INTEGER
                END;
```

(eye)

> The fields of the zone header are for the Memory Manager's own internal use. You can examine the contents of the zone's fields, but in general it doesn't make sense for your program to try to change them. The few exceptions are noted below in the discussions of the specific fields.

BkLim is a pointer to the zone's trailer block. Since the trailer is the last block in the zone, this constitutes a <u>limit pointer</u> to the memory byte **following** the last byte of usable space in the zone.

PurgePtr and allocPtr are "roving pointers" into the heap zone that the Memory Manager maintains for its own internal use. When scanning the zone for a free block to satisfy an allocation request, the Memory Manager begins at the block pointed to by allocPtr instead of always starting from the beginning of the zone. When purging blocks from the zone, it starts from the block pointed to by purgePtr.

HFstFree is a pointer to the first free master pointer in the zone. Instead of just allocating space for one master pointer each time a relocatable block is created, the Memory Manager "preallocates" several master pointers at a time, themselves forming a nonrelocatable block within the zone. The moreMast field of the zone record tells the Memory Manager how many master pointers at a time to preallocate for this zone. Master pointers for the system heap zone are allocated 32 at a time; for the application zone, 64 at a time. For other heap zones, you specify the value of moreMast when you create the zone.

All master pointers that are allocated but not currently in use are linked together into a list beginning in the hFstFree field. When you

allocate a new relocatable block, the Memory Manager removes the first available master pointer from this list, sets it to point to the new block, and returns its address to you as a handle to the block. (If the list is empty, it allocates a fresh block of moreMast master pointers, uses one of them for the new relocatable block, and adds the rest to the list.) When you release a relocatable block, its master pointer isn't released, but linked onto the beginning of the list to be reused. Thus the amount of space devoted to master pointers can increase, but can never decrease unless the zone is reinitialized (for example, at the start of a new application program).

The zcbFree field always contains the number of free bytes remaining in the zone ("zcb" stands for "zone count of bytes"). As blocks are allocated and released, the Memory Manager adjusts zcbFree accordingly. This number represents an upper limit on the size of block you can allocate from this heap zone.

(eye)

> It may not actually be possible to allocate a block as big as zcbFree bytes. As space in a heap zone becomes fragmented, the free bytes typically don't remain contiguous but become scattered throughout the zone. Because nonrelocatable and locked blocks can't be moved, it isn't always possible to collect all the free space into a single block by compaction. (Even if the zone contains only relocatable blocks, the master pointers to these blocks are themselves nonrelocatable "islands" that can interfere with the compaction process.) So the maximum-size block you can actually allocate from the zone may be appreciably smaller than zcbFree bytes.

The gzProc field is a pointer to the zone's grow zone function, or NIL if there is none. You supply this pointer when you create a new heap zone and can change it at any time with the SetGrowZone procedure. The system and application heap zones initially have no grow zone function.

Flags contains a set of flag bits strictly for the Memory Manager's internal use; your program should never need to access this field.

CntRel, maxRel, cntNRel, maxNRel, cntEmpty, cntHandles, and minCBFree are not used by the ROM-based version of the Memory Manager. *** These fields are reserved for eventual use by a special RAM-based version that will gather statistics on a program's memory usage within each heap zone. CntRel and cntNRel will be used to count, respectively, the number of relocatable and nonrelocatable blocks currently allocated within the zone. MaxRel and maxNRel will record the "historical maximum" values attained by cntRel and cntNRel since the program was started. CntEmpty will count the current number of empty master pointers, cntHandles the total number of master pointers currently allocated. MinCBFree will record the historical minimum number of free bytes in the zone. ***

PurgeProc is a pointer to the zone's purge warning procedure (sometimes called a "purge hook"), or NIL if there is none. The Memory Manager

will call this procedure whenever it purges a block from the zone. You can "install" a purge warning procedure in this field to do optional housekeeping such as writing out a block's contents to a disk file before it's purged. In fact, this is exactly the way the Resource Manager keeps the contents of resources up to date if they're changed by your program. If you want to install your own purge hook, you have to be very careful not to interfere with the one the Resource Manager may have installed; see "Special Techniques", later in this manual, for further details.

SparePtr is an extra field included in the zone header for possible future expansion.

The last field of a zone record, heapData, is a dummy field marking the beginning of the zone's usable memory space. HeapData nominally contains an integer, but this integer has no significance in itself--it's just the first two bytes in the block header of the first block in the zone. The purpose of the heapData field is to give you a way of locating the effective beginning of the zone. For example, if myZone is a zone pointer, then

```
@(myZone^.heapData)
```

is a pointer to the first usable byte in the zone, just as

```
myZone^.bkLim
```

is a limit pointer to the byte following the last usable byte in the zone.

Structure of Blocks

Every memory block in a heap zone, whether allocated or free, has a block header that the Memory Manager uses to find its way around in the zone. Block headers are completely transparent to your program. All pointers and handles to allocated blocks point to the beginning of the block's contents, following the end of the header. Similarly, all block sizes seen by your program refer to the block's logical size (the number of bytes in its contents) rather than its physical size (the number of bytes it actually occupies in memory, including the header and any unused bytes at the end of the block).

Since your program shouldn't normally have to deal with block headers directly, there's no Pascal record type defining their structure. (It's possible to access block headers in assembly language, but be sure you know what you're doing!) A block header consists of 8 bytes, as shown in Figure 10.

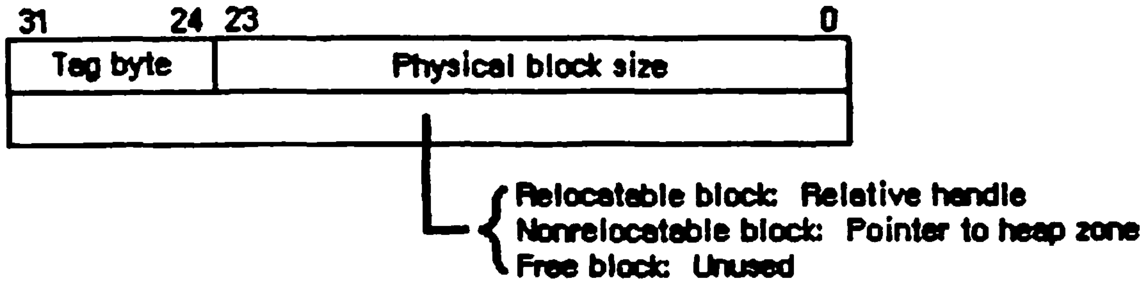


Figure 1Ø. Block Header

The first byte of the block header is the tag byte, discussed in detail below. The next 3 bytes contain the block's physical size in bytes. Adding this number to the block's address gives the address of the next block in the zone.

The contents of the second long word (4 bytes) in the block header depend on the type of block. For relocatable blocks, it contains the block's relative handle: a pointer to the block's master pointer, expressed as an offset relative to the start of the heap zone rather than as an absolute memory address. Adding the relative handle to the zone pointer produces a true handle for this block. For nonrelocatable blocks, the second long word of the header is just a pointer to the block's zone. For free blocks, these 4 bytes are unused.

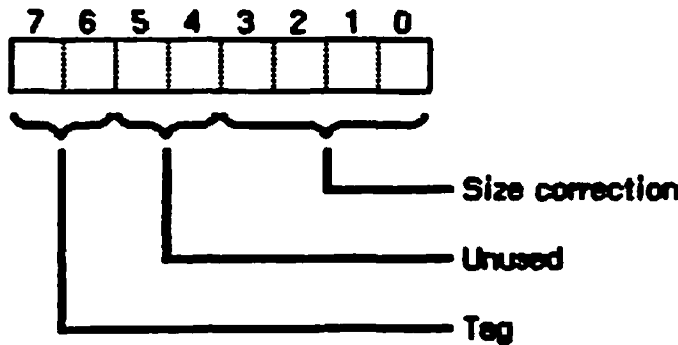


Figure 11. Tag Byte

The tag byte consists of a 2-bit tag, 2 unused bits, and a 4-bit size correction, as shown in Figure 11. The tag identifies the type of block:

Tag	Block type
ØØ	Free
Ø1	Nonrelocatable
1Ø	Relocatable

(A tag value of 11 is invalid.)

The size correction is the number of unused bytes at the end of the block, beyond the end of the block's contents. It's equal to the difference between the block's logical and physical sizes, excluding the 8 bytes of overhead for the block header:

sizeCorrection = physicalSize - logicalSize - 8

There are several reasons why a block may contain such unused bytes:

- The Memory Manager allocates space only in whole 16-bit words--that is, in even numbers of bytes. If the block's logical size is odd, an extra, unused byte is added at the end to keep the physical size even.

- Earlier versions of the Memory Manager used a block header of 12 bytes instead of 8. Although the header is now only 8 bytes long, the Memory Manager still enforces a minimum size of 12 bytes per block for compatibility with these earlier versions. If the logical size of a block is less than 4, enough extra bytes are allocated at the end of the block to bring its physical size up to 12.

- The 12-byte minimum applies to all blocks, free as well as allocated. If allocating the required number of bytes from a free block would leave a fragment of fewer than 12 free bytes, the leftover bytes are included unused at the end of the newly allocated block instead of being returned to free storage.

Putting all this together, the minimum overhead required for each allocated block is 8 bytes for the block header, plus an additional 4 bytes for the master pointer if the block is relocatable. The maximum possible overhead is 26 bytes, for a relocatable block with a logical size of Ø being allocated from a free block of 22 bytes: 8 bytes for the header, 4 for the master pointer, 4 to satisfy the 12-byte minimum, and a leftover fragment of 1Ø free bytes that's too small to return to free storage.

Structure of Master Pointers

The master pointer to a relocatable block has the structure shown in Figure 12. The low-order 3 bytes of the long word contain the address of the block's contents. The high-order byte contains some flag bits that specify the block's current status. Bit 7 of this byte is the <u>lock bit</u> (1 if the block is locked, Ø if it's unlocked); bit 6 is the <u>purge bit</u> (1 if the block is purgeable, Ø if it's unpurgeable). Bit 5

is used by the Resource Manager to identify blocks containing resource information for special treatment; such resource blocks are marked by a 1 in this bit.

(eye)
Before attempting to compare one master pointer with another or perform any arithmetic operation on it, don't forget to strip off the flag bits in the high-order byte.

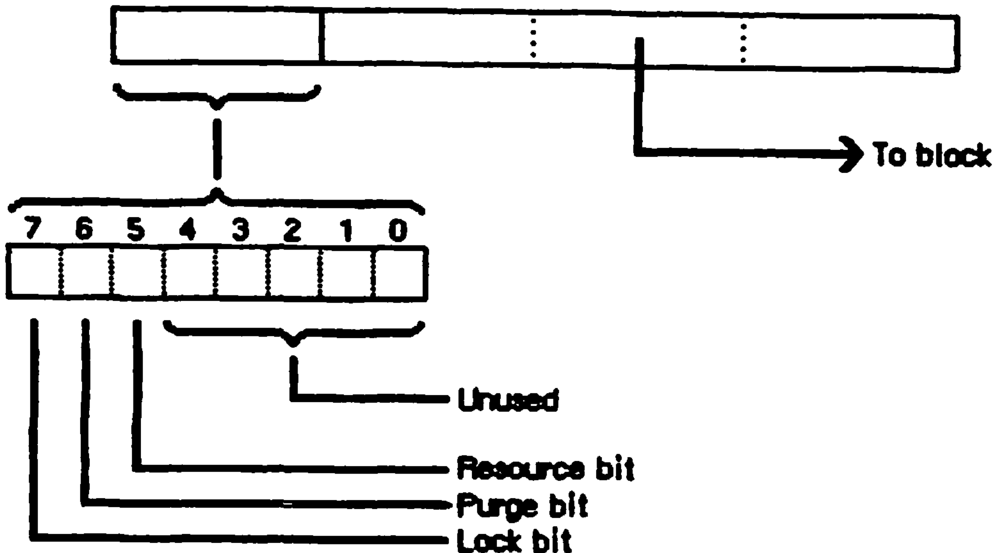


Figure 12. Structure of a Master Pointer

RESULT CODES

Like most other Operating System routines, Memory Manager routines generally return a result code in addition to their normal results. This is an integer code indicating whether the routine completed its task successfully or was prevented by some error condition. The type definition for result codes is

```
TYPE MemErr = INTEGER;
```

In the normal case that no error is detected, the result code is Ø; a nonzero result code signals an error:

```
CONST noErr        = Ø;     {no error}
      memFullErr   = -1Ø8;  {not enough room in zone}
      nilHandleErr = -1Ø9;  {NIL master pointer}
      memWZErr     = -111;  {attempt to operate on a free block}
      memPurErr    = -112;  {attempt to purge a locked block}
```

To inspect a result code from Pascal, call the Memory Manager function MemError. This function always returns the result code from the last Memory Manager call.

Assembly-language note: When called from assembly language via the trap mechanism, not all Memory Manager routines return a result code. Those that do always leave it as a word-length quantity in the low-order half of register DØ on return from the trap. However, some routines leave something else there instead: see the descriptions of individual routines for details. Just before returning, the trap dispatcher tests the lower half of DØ with a TST.W instruction, so that on return from the trap the condition codes reflect the status of the result code, if any.

The stack-based interface routines called from Pascal always produce a result code. If the underlying trap doesn't return one, the interface routine "manufactures" a result code of noErr and stores it where it can later be accessed with MemError.

The ROM-based version of the Memory Manager does only limited error checking. This manual describes only the result codes reported by the ROM version. *** There may eventually be a special RAM-based version that will do more extensive error checking. If so, any additional result codes reported by the RAM version will be documented at that time. ***

USING THE MEMORY MANAGER

This section discusses how the Memory Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Assembly-language note: If you're writing code that will be executed via a hardware interrupt, you can't use the Memory Manager. This is because an interrupt can occur unpredictably at any time. In particular, it can occur while the Memory Manager is in the middle of a heap compaction or in some other inconsistent internal state. To prevent catastrophes, interrupt routines are not allowed to allocate space from the heap.

There's ordinarily no need to initialize the Memory Manager before using it. The system heap zone is automatically initialized each time

the system is started up, and the application heap zone each time an application program is launched. In the unlikely event that you need to reinitialize the application zone while your program is running, you can use InitApplZone.

You can create additional heap zones for your program's own use, either from within the original application zone or from the stack, with InitZone. If you do maintain more than one heap zone, you can find out which zone is current at any given time with GetZone and switch from one to another with SetZone. Almost all Memory Manager operations implicitly apply to the current heap zone. To refer to the system heap zone or the (original) application heap zone, use the Memory Manager function SystemZone or ApplicZone. To find out which zone a particular block resides in, use HandleZone (if the block is relocatable) or PtrZone (if it's nonrelocatable).

(hand)

Most applications will just use the original application heap zone and never have to worry about which zone is current.

The main work of the Memory Manager is allocating and releasing blocks of memory. To allocate a new relocatable block, use NewHandle; for a nonrelocatable block, use NewPtr. These functions return a handle or a pointer, as the case may be, to the newly allocated block. You then use that handle or pointer whenever you need to refer to the block.

To release a block when you're finished with it, use DisposHandle or DisposPtr. You can also change the size of an already allocated block with SetHandleSize or SetPtrSize, and find out its current size with GetHandleSize or GetPtrSize. Use HLock and HUnlock to lock and unlock relocatable blocks.

(hand)

In general, you should use relocatable blocks whenever possible, to avoid unnecessary fragmentation of free space. Use nonrelocatable blocks only for things like I/O buffers, queues, and other objects that must have a fixed location in memory. For most applications, the only Memory Manager routines you'll ever need will be NewHandle, DisposHandle, and SetHandleSize.

(hand)

If you must lock a relocatable block, try to unlock it again at the earliest possible opportunity. Before allocating a block that you know will be locked for long periods of time, call ReservMem to make room for the block as near as possible to the beginning of the zone.

To speed up your program, you may sometimes want to convert the handle to a relocatable block into a copy of the master pointer it points to. This is called dereferencing the handle, and allows you to refer to the block by single instead of double indirection. Dereferencing a handle can be dangerous if you aren't careful; see "Special Techniques" for

further information. If you ever need to convert a dereferenced master pointer back into the original handle, use RecoverHandle.

Ordinarily, you shouldn't have to worry about compacting the heap or purging blocks from it; the Memory Manager automatically takes care of these chores for you. You can control which blocks are purgeable with HPurge and HNoPurge. If for some reason you want to compact or purge the heap explicitly, you can do so with CompactMem or PurgeMem. To explicitly purge a specific block, use EmptyHandle.

(eye)

> If you're working with purgeable blocks, **be careful!** Such blocks may be removed from the heap zone at any time in order to satisfy a memory allocation request. So before attempting to access any purgeable block, always check its handle to make sure the block is still allocated. If the handle is empty (that is, if h^ = NIL, where h is the handle), then the block has been purged: before accessing it, you have to reallocate it and update its master pointer by calling ReallocHandle. (If it's a resource block, use the Resource Manager procedure LoadResource instead.)

You can find out how much free space is left in a heap zone by calling FreeMem (to get the total number of free bytes) or MaxMem (to get the size of the largest single free block and the maximum amount by which the zone can grow). Beware, however: MaxMem also compacts and purges the entire zone before returning this information. To limit the growth of the application zone, use SetApplLimit; to install a grow zone function to help the Memory Manager allocate space in a zone, use SetGrowZone.

After calling any Memory Manager routine, you can examine its result code with MemError.

MEMORY MANAGER ROUTINES

This section describes all the Memory Manager procedures and functions. Each routine is presented first in its Pascal form (if there is one). For most routines, this is followed by a box containing information needed to use the routine from assembly language. Most Pascal programmers can just skip this box, although the list of result codes may be of interest to some. For general information on using the Memory Manager from assembly language, see "Using the Operating System from Assembly Language" *** (to be written) *** and also "Notes for Assembly-Language Programmers" in this manual.

Initialization and Allocation

PROCEDURE InitApplZone;

Trap macro	_InitApplZone
On exit	DØ: result code (integer)
Result codes	Ø $ØØØØ noErr No error

InitApplZone initializes the application heap zone and makes it the current zone. The contents of any previous application zone are completely wiped out; all previously existing blocks in that zone are discarded. InitApplZone is called by the Segment Loader when launching an application program; you shouldn't normally need to call it from within your own program.

(eye)

> Reinitializing the application zone from within a running program is tricky, since the program's code itself resides in the application zone. To do it safely, you have to move the code of the running program into the system heap zone, jump to it there, reinitialize the application zone, move the code back into the application zone, and jump to it again. Don't attempt this operation unless you're sure you know what you're doing.

The application zone has a standard initial size of 6K bytes, immediately following the end of the system heap zone, and can be expanded as needed in 1K increments. Space is initially allocated for 64 master pointers; should more be needed later, they will be added 64 at a time. The zone's grow zone function is set to NIL. After a call to InitApplZone, MemError will always return noErr.

PROCEDURE SetApplBase (startPtr: Ptr);

Trap macro	_SetApplBase	
On entry	AØ:	startPtr (pointer)
On exit	DØ:	result code (integer)
Result codes	Ø $ØØØØ noErr	No error

SetApplBase changes the starting address of the application heap zone to the address designated by startPtr, reinitializes the zone, and makes it the current zone. The contents of any previous application zone are completely wiped out; all previously existing blocks in that zone are discarded. SetApplBase is normally called only by the system itself; you should never need to call this procedure from within your own program.

Since the application heap zone begins immediately following the end of the system zone, changing its starting address has the effect of changing the size of the system zone. The system zone can be made larger, but never smaller; if startPtr points to an address lower than the current end of the system zone, it's ignored and the application zone's starting address is left unchanged.

In any case, SetApplBase reinitializes the application zone to its standard initial size of 6K bytes, which can later be expanded as needed in 1K increments. Space is initially allocated for 64 master pointers; should more be needed later, they will be added 64 at a time. The zone's grow zone function is set to NIL. After a call to SetApplBase, MemError will always return noErr.

(eye)

> Like InitApplZone, SetApplBase is a tricky operation, because the code of the program itself resides in the application heap zone. The recommended procedure for doing it safely is the same as for InitApplZone (see above); again, don't attempt it unless you know what you're doing.

PROCEDURE InitZone (growProc: ProcPtr; masterCount: INTEGER; limitPtr,
startPtr: Ptr);

<u>Trap macro</u> _InitZone

<u>On entry</u> AØ: pointer to parameter block

startPtr (4-byte pointer)
limitPtr (4-byte pointer)
masterCount (2-byte integer)
growProc (4-byte pointer)

<u>On exit</u> DØ: result code (integer)

<u>Result codes</u> Ø $ØØØØ noErr No error

InitZone creates a new heap zone, initializes its header and trailer, and makes it the current zone. The startPtr parameter is a pointer to the first byte of the new zone; limitPtr points to the byte **following** the end of the zone. That is, the new zone will occupy memory addresses from ORD(startPtr) to ORD(limitPtr) - 1.

MasterCount tells how many master pointers should be allocated at a time for the new zone. The specified number of master pointers are created initially; should more be needed later, they will be added in increments of this same number. For the system heap zone, masterCount is 32; for the application heap zone, it's 64.

The growProc parameter is a pointer to the grow zone function for the new zone, if any. If you're not defining a grow zone function for this one, supply a NIL value for growProc.

The new zone includes a 52-byte header and a 12-byte trailer, so its actual usable space runs from ORD(startPtr) + 52 through ORD(limitPtr) - 13. In addition, each master pointer occupies 4 bytes within this usable area. Thus the total available space in the zone, in bytes, is initially

ORD(limitPtr) - ORD(startPtr) - 64 - 4*masterCount

This number must not be less than Ø. Note that the amount of available space in the zone may decrease as more master pointers are allocated.

After a call to InitZone, MemError will always return noErr.

PROCEDURE SetApplLimit (zoneLimit: Ptr);

Trap macro	_SetApplLimit
On entry	AØ: zoneLimit (pointer)
On exit	DØ: result code (integer)
Result codes	Ø $ØØØØ noErr No error

SetApplLimit sets the application heap limit, beyond which the application heap zone can't be expanded. The actual expansion isn't under your program's control, but is done automatically by the Memory Manager when necessary in order to satisfy an allocation request. Only the original application zone can be expanded.

ZoneLimit is a limit pointer to a byte in memory beyond which the zone will not be allowed to grow. That is, the zone can grow to include the byte preceding zoneLimit in memory, but no farther. If the zone already extends beyond the specified limit it won't be cut back, but it will be prevented from growing any more.

(eye)

Notice that zoneLimit is not a byte count. To limit the application zone to a particular size (say 8K bytes), you have to write something like

```
SetApplLimit(POINTER(ORD(ApplicZone) + 8192))
```

After a call to SetApplLimit, MemError will always return noErr.

Heap Zone Access

FUNCTION GetZone : THz;

Trap macro	_GetZone
On exit	AØ: function result (pointer) Ø: result code (integer)
Result codes	Ø $ØØØØ noErr No error

GetZone returns a pointer to the current heap zone. After the call, MemError will always return noErr.

PROCEDURE SetZone (hz: THz);

Trap macro	_SetZone
On entry	AØ: hz (pointer)
On exit	DØ: result code (integer)
Result codes	Ø $ØØØØ noErr No error

SetZone sets the current heap zone to the zone pointed to by hz. After the call, MemError will always return noErr.

FUNCTION SystemZone : THz; [Pascal only]

Trap macro	None
Result codes	Ø $ØØØØ noErr No error

SystemZone returns a pointer to the system heap zone. After the call, MemError will always return noErr.

Assembly-language note: SystemZone is part of the Pascal interface to the Memory Manager, not part of the Memory Manager

itself. It doesn't reside in ROM and can't be called via a trap. To get a pointer to the system heap zone from assembly language, use the global variable sysZone.

FUNCTION ApplicZone : THz; [Pascal only]

Trap macro None

Result codes Ø $ØØØØ noErr No error

ApplicZone returns a pointer to the original application heap zone. After the call, MemError will always return noErr.

Assembly-language note: ApplicZone is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To get a pointer to the application heap zone from assembly language, use the global variable applZone.

Allocating and Releasing Relocatable Blocks

FUNCTION NewHandle (logicalSize: Size) : Handle;

Trap macro _NewHandle

On entry DØ: logicalSize (long integer)

On exit AØ: function result (handle)
Ø: result code (integer)

Result codes Ø $ØØØØ noErr No error
-1Ø8 $FF94 memFullErr Not enough room in zone

NewHandle allocates a new relocatable block from the current heap zone and returns a handle to it (or NIL if a block of that size can't be created). The new block will have a logical size of logicalSize bytes and will initially be marked unlocked and unpurgeable.

NewHandle will pursue all avenues open to it in order to create a free block of the requested size, including compacting the heap zone, increasing its size, purging blocks from it, and calling its grow zone function, if any. If all such attempts fail, or if the zone has run out of free master pointers and there's no room to allocate more, NewHandle returns NIL and MemError will return memFullErr after the call. If a new block was successfully allocated, NewHandle returns a handle to the new block and MemError will return noErr.

PROCEDURE DisposHandle (h: Handle);

Trap macro	_DisposHandle			
On entry	AØ:	h (handle)		
On exit	AØ:	Ø		
	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

DisposHandle releases the space occupied by the relocatable block whose handle is h. If the block is already free, MemError will return memWZErr after the call; otherwise it will return noErr.

(eye)

After a call to DisposHandle, all handles to the released block become invalid and should not be used again.

FUNCTION GetHandleSize (h: Handle) : Size;

Trap macro	_GetHandleSize			
On entry	AØ:	h (handle)		
On exit	DØ:	if >= Ø, function result (long integer) if < Ø, result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error [Pascal only]
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

GetHandleSize returns the logical size, in bytes, of the relocatable block whose handle is h. After the call, MemError will return

nilHandleErr if h points to a NIL master pointer, memWZErr if h is the handle of a free block, and noErr otherwise. In case of an error, GetHandleSize returns a result of Ø.

Assembly-language note: Recall that the trap dispatcher sets the condition codes before returning from a trap by testing the low-order half of register DØ with a TST.W instruction. Since the block size returned in DØ by _GetHandleSize is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of DØ, use your own TST.L instruction on return from the trap to test the full 32 bits of the register.

PROCEDURE SetHandleSize (h: Handle; newSize: Size);

Trap macro	_SetHandleSize			
On entry	AØ:	h (handle)		
	DØ:	newSize (long integer)		
On exit	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø8	$FF94	memFullErr	Not enough room to grow
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

SetHandleSize changes the logical size of the relocatable block whose handle is h to newSize bytes. After the call, MemError will return memFullErr if newSize is greater than the block's current size and enough room can't be found for the block to grow, nilHandleErr if h points to a NIL master pointer, memWZErr if h is the handle of a free block, and noErr otherwise.

FUNCTION HandleZone (h: Handle) : THz;

Trap macro	_HandleZone			
On entry	AØ:	h (handle)		
On exit	AØ:	function result (pointer)		
	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

HandleZone returns a pointer to the heap zone containing the relocatable block whose handle is h.

If handle h is empty (points to a NIL master pointer), HandleZone returns a pointer to the current heap zone and doesn't report an error: after the call, MemError will return noErr. If h is the handle of a free block, MemError will return memWZErr; in this case, the result returned by HandleZone is meaningless and should be ignored.

FUNCTION RecoverHandle (p: Ptr) : Handle;

Trap macro	_RecoverHandle			
On entry	AØ:	p (pointer)		
On exit	AØ:	function result (handle)		
	DØ:	unchanged (!)		
Result codes	Ø	$ØØØØ	noErr	No error [Pascal only]

RecoverHandle returns a handle to the relocatable block pointed to by p. If you've "dereferenced" a handle (converted it to a simple pointer) for efficiency, you can use this function to get back the original handle. After the call, MemError will always return noErr.

Assembly-language note: Through a minor oversight, the trap _RecoverHandle neglects to return a result code in register DØ; the previous contents of DØ are preserved unchanged. The stack-based interface routine called from Pascal always produces a result code of noErr.

PROCEDURE ReallocHandle (h: Handle; logicalSize: Size);

Trap macro	_ReallocHandle			
On entry	AØ:	h (handle)		
	DØ:	logicalSize (long integer)		
On exit	AØ:	original h or NIL		
	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø8	$FF94	memFullErr	Not enough room in zone
	-111	$FF91	memWZErr	Attempt to operate on a free block
	-112	$FF9Ø	memPurErr	Block is locked

ReallocHandle allocates a new relocatable block with a logical size of logicalSize bytes. It then updates handle h by setting its master pointer to point to the new block. The main use of this procedure is to reallocate space for a block that has been purged. Normally h is an empty handle, but it need not be: if it points to an existing block, that block is released before the new block is created.

After the call, MemError will return noErr if ReallocHandle succeeds in allocating a block of the requested size; if room can't be made for the requested block, it will return memFullErr. If h is the handle of an existing block, MemError will return memPurErr if the block is locked and memWZErr if it's already free. In case of an error, no new block is allocated and handle h is left unchanged.

Assembly-language note: On return from _ReallocHandle, register AØ contains the original handle h, or Ø (NIL) if no room could be found for the requested block.

Allocating and Releasing Nonrelocatable Blocks

FUNCTION NewPtr (logicalSize: Size) : Ptr;

Trap macro	_NewPtr			
On entry	DØ: logicalSize (long integer)			
On exit	AØ: function result (pointer) DØ: result code (integer)			
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø8	$FF94	memFullErr	Not enough room in zone

NewPtr allocates a new nonrelocatable block from the current heap zone and returns a pointer to it (or NIL if a block of that size can't be created). The new block will have a logical size of logicalSize bytes.

NewPtr will pursue all avenues open to it in order to create a free block of the requested size, including compacting the heap zone, increasing its size, purging blocks from it, and calling its grow zone function, if any. If all such attempts fail, NewPtr returns NIL and MemError will return memFullErr after the call. If a new block was successfully allocated, NewPtr returns a pointer to the new block and MemError will return noErr.

PROCEDURE DisposPtr (p: Ptr);

Trap macro	_DisposPtr			
On entry	AØ: p (pointer)			
On exit	AØ: Ø DØ: result code (integer)			
Result codes	Ø	$ØØØØ	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

DisposPtr releases the space occupied by the nonrelocatable block pointed to by p. If the block is already free, MemError will return memWZErr after the call; otherwise it will return noErr.

(eye)

After a call to DisposPtr, all pointers to the released block become invalid and should not be used again.

FUNCTION GetPtrSize (p: Ptr) : Size;

Trap macro	_GetPtrSize			
On entry	AØ: p (pointer)			
On exit	DØ: if >= Ø, function result (long integer) if < Ø, result code (integer)			
Result codes	Ø	$ØØØØ	noErr	No error [Pascal only]
	-111	$FF91	memWZErr	Attempt to operate on a free block

GetPtrSize returns the logical size, in bytes, of the nonrelocatable block pointed to by p. After the call, MemError will return memWZErr if p points to a free block and noErr otherwise. In case of an error, GetPtrSize returns a result of Ø.

Assembly-language note: Recall that the trap dispatcher sets the condition codes before returning from a trap by testing the low-order half of register DØ with a TST.W instruction. Since the block size returned in DØ by _GetPtrSize is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of DØ, use your own TST.L instruction on return from the trap to test the full 32 bits of the register.

PROCEDURE SetPtrSize (p: Ptr; newSize: Size);

Trap macro	_SetPtrSize			
On entry	AØ:	p (pointer)		
	DØ:	newSize (long integer)		
On exit	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø8	$FF94	memFullErr	Not enough room to grow
	-111	$FF91	memWZErr	Attempt to operate on a free block

SetPtrSize changes the logical size of the nonrelocatable block pointed to by p to newSize bytes. After the call, MemError will return memFullErr if newSize is greater than the block's current size and enough room can't be found for the block to grow, memWZErr if p points to a free block, and noErr otherwise.

FUNCTION PtrZone (p: Ptr) : THz;

Trap macro	_PtrZone			
On entry	AØ:	p (pointer)		
On exit	AØ:	function result (pointer)		
	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

PtrZone returns a pointer to the heap zone containing the nonrelocatable block pointed to by p. If p points to a free block, MemError will return memWZErr after the call; in this case, the result returned by PtrZone is meaningless and should be ignored.

Freeing Space on the Heap

FUNCTION FreeMem : LongInt;

Trap macro	_FreeMem				
On exit	DØ: function result (long integer)				
Result codes	Ø	$ØØØØ	noErr	No error	[Pascal only]

FreeMem returns the total amount of free space in the current heap zone, in bytes. Notice that it may not actually be possible to allocate a block of this size, because of fragmentation due to nonrelocatable or locked blocks. After a call to FreeMem, MemError will always return noErr.

FUNCTION MaxMem (VAR grow: Size) : Size;

Trap macro	_MaxMem				
On exit	DØ: function result (long integer) AØ: grow (long integer)				
Result codes	Ø	$ØØØØ	noErr	No error	[Pascal only]

MaxMem compacts the current heap zone and purges all purgeable blocks from the zone. It returns as its result the size in bytes of the largest contiguous free block in the zone after the compaction. If the current zone is the original application heap zone, the variable parameter grow is set to the maximum number of bytes by which the zone can grow. For any other heap zone, grow is set to Ø. MaxMem doesn't actually expand the zone or call its grow zone function. After the call, MemError will always return noErr.

FUNCTION CompactMem (cbNeeded: Size) : Size;

Trap macro	_CompactMem
On entry	DØ: cbNeeded (long integer)
On exit	DØ: function result (long integer) AØ: pointer to desired block or NIL
Result codes	Ø $ØØØØ noErr No error [Pascal only]

CompactMem compacts the current heap zone by moving relocatable blocks forward and collecting free space together until a contiguous block of at least cbNeeded free bytes is found or the entire zone is compacted. For each block that's moved, the master pointer is updated so that all handles to the block remain valid. CompactMem returns the size in bytes of the largest contiguous free block it finds, but doesn't actually allocate the block. After the call, MemError will always return noErr.

(hand)
To force a compaction of the entire heap zone, set cbNeeded equal to maxSize.

Assembly-language note: On return from _CompactMem, register AØ contains a pointer to a free block of at least cbNeeded bytes, or Ø (NIL) if no such block could be found.

FUNCTION ResrvMem (cbNeeded: Size);

Trap macro	_ResrvMem				
On entry	DØ: cbNeeded (long integer)				
On exit	AØ: pointer to desired block or NIL DØ: result code (integer)				
Result codes	Ø	$ØØØØ	noErr	No error	
	-1Ø8	$FF94	memFullErr	Not enough room in zone	

ResrvMem creates free space for a block of cbNeeded contiguous bytes at the lowest possible position in the current heap zone. It will try every available means to place the block as close as possible to the

beginning of the zone, including moving other blocks upward, expanding the zone, or purging blocks from it. If a free block of at least the requested size can't be created, MemError will return memFullErr after the call; otherwise it will return noErr. Notice that ResrvMem doesn't actually allocate the block.

(hand)
When you allocate a relocatable block that you know will be locked for long periods of time, call ResrvMem first. This reserves space for the block near the beginning of the heap zone, where it will interfere with compaction as little as possible. It isn't necessary to call ResrvMem for a nonrelocatable block; NewPtr calls it automatically.

Assembly-language note: On return from _ResrvMem, register A0 contains a pointer to the desired free block of at least cbNeeded bytes, or 0 (NIL) if no such block could be created.

FUNCTION PurgeMem (cbNeeded: Size);

Trap macro	_PurgeMem			
On entry	D0:	cbNeeded (long integer)		
On exit	A0:	pointer to desired block or NIL		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-108	$FF94	memFullErr	Not enough room in zone

PurgeMem purges blocks from the current heap zone until a contiguous block of at least cbNeeded free bytes is created or the entire zone is purged. Only relocatable, unlocked, purgeable blocks can be purged. If a free block of at least the requested size is found, MemError will return noErr after the call; if not, it will return memFullErr. Notice that PurgeMem doesn't actually allocate the block.

(hand)
To force a purge of the entire heap zone, set cbNeeded equal to maxSize.

Assembly-language note: On return from _PurgeMem, register AØ contains a pointer to a free block of at least cbNeeded bytes, or Ø (NIL) if no such block could be found.

PROCEDURE EmptyHandle (h: Handle);

Trap macro	_EmptyHandle			
On entry	AØ:	h (handle)		
On exit	AØ:	h (handle)		
	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block
	-112	$FF9Ø	memPurErr	Block is locked

EmptyHandle empties handle h: that is, it purges the relocatable block whose handle is h from its heap zone and sets its master pointer to NIL. If h is already empty, EmptyHandle does nothing.

(hand)

> The main use of this procedure is to release the space a block occupies without having to update every existing handle to the block. Since the space occupied by the master pointer itself remains allocated, all handles pointing to it remain valid but become empty. When you later reallocate space for the block with ReallocHandle, the master pointer will be updated, causing all existing handles to point correctly to the new block.

The block whose handle is h must be unlocked, but need not be purgeable: if you ask to purge an unpurgeable block, EmptyHandle assumes you know what you're doing and purges the block as requested. If the block is locked, EmptyHandle doesn't purge it; after the call, MemError will return memPurErr. If the block is already free, MemError will return memWZErr.

Properties of Relocatable Blocks

PROCEDURE HLock (h: Handle);

Trap macro	_HLock
On entry	AØ: h (handle)
On exit	DØ: result code (integer)

Result codes				
	Ø	$ØØØØ	noErr	No error
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HLock locks a relocatable block, preventing it from being moved within its heap zone. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already locked, HLock does nothing.

PROCEDURE HUnlock (h: Handle);

Trap macro	_HUnlock
On entry	AØ: h (handle)
On exit	DØ: result code (integer)

Result codes				
	Ø	$ØØØØ	noErr	No error
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HUnlock unlocks a relocatable block, allowing it to be moved within its heap zone. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already unlocked, HUnlock does nothing.

PROCEDURE HPurge (h: Handle);

Trap macro	_HPurge			
On entry	AØ:	h (handle)		
On exit	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HPurge marks a relocatable block as purgeable. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already purgeable, HPurge does nothing.

PROCEDURE HNoPurge (h: Handle);

Trap macro	_HNoPurge			
On entry	AØ:	h (handle)		
On exit	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HNoPurge marks a relocatable block as unpurgeable. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already unpurgeable, HNoPurge does nothing.

Grow Zone Functions

PROCEDURE SetGrowZone (growZone: ProcPtr);

Trap macro	_SetGrowZone
On entry	A0: growZone (pointer)
On exit	D0: result code (integer)
Result codes	0 $0000 noErr No error

SetGrowZone sets the current heap zone's grow zone function as designated by the growZone parameter. A NIL parameter value removes any grow zone function the zone may previously have had. After the call, MemError will always return noErr.

(hand)

> If your program presses the limits of the available heap space, it's a good idea to have a grow zone function of some sort. At the very least, the grow zone function should detect when the Memory Manager is about to run out of space at a critical time (see GZCritical, below) and take some graceful action--such as displaying an alert box with the message "Out of memory"--instead of just failing unpredictably. *** There may eventually be a default grow zone function that does this. ***

The Memory Manager calls the grow zone function as a last resort when trying to allocate space, after failing to create a block of the needed size by compacting the zone, increasing its size (in the case of the original application zone), or purging blocks from it. Memory Manager routines that may cause the grow zone function to be called are NewHandle, NewPtr, SetHandleSize, SetPtrSize, ReallocHandle, and ResrvMem.

The grow zone function should be of the form

```
FUNCTION GrowTheZone (cbNeeded: Size) : Size;
```

(Of course, the name GrowTheZone is only an example; you can give the function any name you like.) The cbNeeded parameter gives the physical size of the needed block in bytes, **including the block header.** The grow zone function should attempt to create a free block of at least this size. It should return as its result the number of additional bytes it has freed within the zone, but this number need not be accurate.

If the grow zone function returns Ø, the Memory Manager will give up trying to allocate the needed block and will signal failure with the result code memFullErr. Otherwise it will compact the heap zone and try again to allocate the block. If still unsuccessful, it will continue to call the grow zone function repeatedly, compacting the zone again after each call, until it either succeeds in allocating the needed block or receives a zero result and gives up.

The usual way for the grow zone function to free more space is to call EmptyHandle to purge blocks that were previously marked unpurgeable. Another possibility is to unlock blocks that were previously locked, in order to eliminate immovable "islands" that may have been interfering with the compaction process and fragmenting the existing free space.

(hand)

Although just unlocking blocks doesn't actually free any additional space in the zone, the grow zone function should still return a nonzero result in this case. This signals the Memory Manager to compact the heap and try again to allocate the needed block.

(eye)

Depending on the circumstances in which the grow zone function is called, there may be particular blocks within the heap zone that must not be purged or released. For instance, if your program is attempting to increase the size of a relocatable block with SetHandleSize, it would be disastrous to release the block being expanded. To deal with such cases safely, it's essential to understand the use of the functions GZCritical and GZSaveHnd (see below).

FUNCTION GZCritical : BOOLEAN; [Pascal only]

Trap macro None

Result codes None

GZCritical returns TRUE if the Memory Manager critically needs the requested space: for example, to create a new relocatable or nonrelocatable block or to reallocate a handle. It returns FALSE in less critical cases, such as ResrvMem trying to move a block in order to reserve space as low as possible in the heap zone or SetHandleSize trying to increase the size of a relocatable block by moving the block above it.

(eye)

If you're writing a grow zone function in Pascal, you should always call GZCritical and proceed only if the result is TRUE. All the information you need to handle

the critical cases safely is the value of GZSaveHnd (see below). The noncritical cases require additional information that isn't available from Pascal, so your grow zone function should just return Ø and not attempt to free any space.

Assembly-language note: GZCritical is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To find out whether a given grow zone call is critical, use the following magical incantation:

```
          MOVE.L  gzMoveHnd,DØ
          BEQ.S   Critical
          CMP.L   gzRootHnd,DØ
          BEQ.S   Critical

          CLR.L   4(SP)          ;If noncritical, just return Ø
          RTS

Critical  . . .                  ;Handle critical case
```

To handle the critical cases safely (and the noncritical ones if you choose to do more than just return Ø), see the note below under GZSaveHnd.

FUNCTION GZSaveHnd : Handle; [Pascal only]

Trap macro   None

Result codes   None

GZSaveHnd returns a handle to a relocatable block that mustn't be purged or released by the grow zone function, or NIL if there is no such block. The grow zone function will be safe if it avoids purging or releasing this block, **provided that the grow zone call was critical**. To handle noncritical cases safely, further information is needed that isn't available from Pascal.

Assembly-language note: GZSaveHnd is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. You can find the handle it returns in the global variable gzRootHnd. The "further information" that isn't available from

Pascal is the contents of two other global variables, gzRootPtr and gzMoveHnd, which may be nonzero in noncritical cases. If gzRootPtr is nonzero, it's a pointer to a nonrelocatable block that must not be released; gzMoveHnd is a handle to a relocatable block that must not be released but may be purged.

Utility Routines

PROCEDURE BlockMove (sourcePtr,destPtr: Ptr; byteCount: Size);

Trap macro	_BlockMove	
On entry	AØ:	sourcePtr (pointer)
	A1:	destPtr (pointer)
	DØ:	byteCount (long integer)
On exit	DØ:	result code (integer)
Result codes	Ø	$ØØØØ noErr No error

BlockMove moves a block of byteCount consecutive bytes from the address designated by sourcePtr to that designated by destPtr. No checking of any kind is done on the addresses; no pointers are updated. After the call, MemError will always return noErr.

FUNCTION TopMem : Ptr; [Pascal only]

Trap macro	None			
Result codes	Ø	$ØØØØ	noErr	No error

TopMem returns a pointer to the address following the last byte of physical memory. After the call, MemError will always return noErr.

Assembly-language note: TopMem is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To get a pointer to the end of physical memory from assembly language, use the global variable memTop.

FUNCTION MemError : MemErr; [Pascal only]

Trap macro None

Result codes None

MemError returns the result code produced by the last Memory Manager routine to be called.

Assembly-language note: MemError is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To get the a routine's result code from assembly language, look in register DØ on return from the routine.

SPECIAL TECHNIQUES

This section describes some special or unusual techniques that you may find useful.

Dereferencing a Handle

Accessing a block by double indirection, through a handle instead of a simple pointer, requires an extra memory reference. For efficiency, you may sometimes want to dereference the handle—that is, convert it to a copy of the master pointer, then use that pointer to access the block by single indirection. But **be careful!** Any operation that allocates space from the heap may cause the underlying block to be moved or purged. In that event, the master pointer itself will be correctly updated, but your copy of it will be left dangling.

One way to avoid this common type of program bug is to lock the block before dereferencing its handle: for example,

```
VAR aPointer: Ptr;
    aHandle: Handle;
    . . . ;
BEGIN
    . . . ;
    aHandle := NewHandle( . . . );   {create a relocatable block}
    . . . ;
    HLock(aHandle);                  {lock block before dereferencing}
    aPointer := aHandle^;            {convert handle to simple pointer}

    WHILE . . . DO
        BEGIN
            ...aPointer^...          {use simple pointer inside loop}
        END;

    HUnlock(aHandle);                {unlock block when finished}
    . . .
END
```

Assembly-language note: To dereference a handle in assembly language, just copy the master pointer into an address register and use it to access the block by single indirection. Remember that the master pointer points to the block's contents, not its header!

```
        MOVE.L  #blockSize,DØ ;set up block size for _NewHandle
        _NewHandle            ;create relocatable block
        MOVE.L  AØ,aHandle    ;save handle for later use
        . . .
        MOVE.L  aHandle,A1    ;get back handle
        MOVE.L  A1,AØ         ;lock block before dereferencing
        _HLock

        MOVE.L  (A1),A2       ;convert handle to simple pointer

LOOP    . . .
        MOVE    ...(A2)...    ;use simple pointer inside loop
        . . .
        Bcc.S   LOOP          ;loop back on some condition

        MOVE.L  A1,AØ         ;unlock block when finished
        _HUnlock
        . . .
```

Remember, however, that when you lock a block it becomes an "island" in the heap that may interfere with compaction and cause free space to become fragmented. It's recommended that you use this technique only in parts of your program where efficiency is critical, such as inside tight inner loops that are executed many times.

(eye)

Don't forget to unlock the block again when you're through with the dereferenced handle!

Instead of locking the block, you can update your copy of the master pointer after any "dangerous" operation (one that can invalidate the pointer by moving or purging the block it points to). Memory Manager routines that can move or purge blocks in the heap are NewHandle, NewPtr, SetHandleSize, SetPtrSize, ReallocHandle, ResrvMem, CompactMem, PurgeMem, and MaxMem. Since these routines can be called indirectly from other Operating System or Toolbox routines, you should assume that any call to the OS or Toolbox can potentially leave your dereferenced pointer dangling. *** Eventually there will be a technical note listing which OS and Toolbox routines are dangerous and which aren't. ***

(hand)

If you aren't performing any dangerous operations, you needn't worry about updating the pointer (or locking the block either, for that matter).

Subdividing the Application Heap Zone

In some applications, you may want to subdivide the original application heap zone into two or more independent zones to be used for different purposes. In doing this, it's important not to destroy any existing blocks in the original zone (such as those containing the code of your program). The recommended procedure is to allocate space for the subzones as nonrelocatable blocks within the original zone, then use InitZone to initialize them as independent zones. For example, to divide the available space in the application zone in half, you might write something like the following:

```
CONST minSize = 52 + 12 + 32*(12 + 4);   {zone header, zone trailer,}
                                          {   and 32 minimum-size blocks}
                                          {   with master pointers}

VAR myZone1, myZone2: THz;
    start, limit: Ptr;
    availSpace, zoneSize: Size;
    . . . ;
BEGIN
    . . . ;
    SetZone(ApplicZone);
    availSpace := CompactMem(maxSize);    {size of largest free block}
    zoneSize := 2 * (availSpace DIV 4);   {force new zone size to an}
                                          {   even number of bytes}
    IF zoneSize < (minSize + 8)           {need 8 bytes for}
                                          {   block header}
       THEN . . .                         {error--not enough room}
       ELSE
          BEGIN
             zoneSize := zoneSize - 8;    {adjust for block header}

             start := NewPtr(zoneSize);   {allocate a nonrel. block}
             limit := POINTER(ORD(start) + zoneSize);
             InitZone(NIL, 32, limit, start);
             myZone1 := POINTER(ORD(start));  {convert Ptr to THz}

             start := NewPtr(zoneSize);   {allocate a nonrel. block}
             limit := POINTER(ORD(start) + zoneSize);
             InitZone(NIL, 32, limit, start);
             myZone2 := POINTER(ORD(start))   {convert Ptr to THz}
          END;
    . . .
END
```

Assembly-language note: The equivalent assembly code might be

```
minSize   .EQU      52+12+<32*<12+4>> ;zone header and trailer, plus
                                      ;   32 minimum-size blocks
                                      ;   with master pointers
          . . .
          MOVE.L  applZone,AØ       ;get original application zone
          _SetZone                  ;make it current

          MOVE.L  #maxSize,DØ       ;compact entire zone
          _CompactMem               ;DØ has size of largest free block

          ASR.L   #2,DØ             ;force new zone size to an
          ASL.L   #1,DØ             ;   even number of bytes
          CMP.L   #minSize+8,DØ     ;need 8 bytes for block header
          BLO     NoRoom            ;error if < minimum size

          SUBQ.L  #8,DØ             ;adjust for block header
          MOVE.L  DØ,D1             ;save zone size
          _NewPtr                   ;allocate nonrelocatable block
          MOVE.L  AØ,myZone1        ;store zone pointer

          CLR.L   -(SP)             ;NIL grow zone function
          MOVE.W  #32,-(SP)         ;allocate 32 master pointers
          MOVE.L  AØ,-(SP)          ;AØ has zone pointer
          ADD.L   D1,(SP)           ;convert to limit pointer
          MOVE.L  AØ,-(SP)          ;push as start pointer

          MOVE.L  SP,AØ             ;point to argument block
          _InitZone                 ;create zone 1

          MOVE.L  D1,DØ             ;get back zone size
          _NewPtr                   ;allocate nonrelocatable block
          MOVE.L  AØ,myZone2        ;store zone pointer

          MOVE.L  AØ,4(SP)          ;move zone pointer to stack
          ADD.L   D1,(SP)           ;convert to limit pointer
          MOVE.L  AØ,(SP)           ;move to stack as start pointer

          MOVE.L  SP,AØ             ;point to argument block
          _InitZone                 ;create zone 2
          ADD.W   #14,SP            ;pop arguments off stack
          . . .
```

Creating a Heap Zone on the Stack

Another place you can get the space for a new heap zone is from the stack. For example,

```
CONST zoneSize = 2048;
VAR zoneArea: PACKED ARRAY [1..zoneSize] OF SignedByte;
    stackZone: THz;
    limit: Ptr;
    . . . ;
BEGIN
    . . . ;
    stackZone := @zoneArea;
    limit := POINTER(ORD(stackZone) + zoneSize);
    InitZone(NIL, 16, limit, @zoneArea);
    . . .
END
```

Assembly-language note: Here's how you might do the same thing in assembly language:

```
zoneSize  .EQU     2048
          . . .
          MOVE.L   SP,A2          ;save stack pointer for limit
          SUB.W    #zoneSize,SP   ;make room on stack
          MOVE.L   SP,A1          ;save stack pointer for start
          MOVE.L   A1,stackZone   ;store as zone pointer

          CLR.L    -(SP)          ;NIL grow zone function
          MOVE.W   #16,-(SP)      ;allocate 16 master pointers
          MOVE.L   A2,-(SP)       ;push limit pointer
          MOVE.L   A1,-(SP)       ;push start pointer

          MOVE.L   SP,A0          ;point to argument block
          _InitZone               ;create new zone
          ADD.W    #14,SP         ;pop arguments off stack
          . . .
```

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

General information about how to use the Macintosh Operating System from assembly language is *** (will be) *** given elsewhere. This section contains special notes of interest to programmers who will be using the Memory Manager from assembly language.

The primary aids to assembly-language programmers are files named SYSEQU.TEXT, SYSMACS.TEXT, SYSERR.TEXT, and HEAPDEFS.TEXT. If you use .INCLUDE to include these files when you assemble your program, all the Memory Manager constants, addresses of global variables, trap macros, error codes, and masks and offsets into fields of structured types will be available in symbolic form.

Constants

The file HEAPDEFS.TEXT defines a number of useful constants that you can use in your program as immediate data values. For example, to push the default master-point count onto the stack as an argument for _InitZone, you might write

```
MOVE.W  #dfltMasters,-(SP)
```

(hand)
It's a good idea to refer to these constants in your program by name instead of using the numeric value directly, since some of the values shown may be subject to change. Some of the constants are based on an eventual 512K memory configuration; the present Macintosh has 128K of RAM.

The following constants are defined in HEAPDEFS.TEXT:

```
minFree        .EQU    12          ;minimum block size
maxSize        .EQU    $7FFFF      ;maximum block size (512K - 1)
minAddr        .EQU    0           ;minimum legal address
maxAddr        .EQU    $80000      ;maximum legal address (512K)
dfltMasters    .EQU    32          ;default master-pointer count
maxMasters     .EQU    $1000       ;maximum master-pointer count (4K)
sysZoneSize    .EQU    $4000       ;size of system heap zone (16K)
applZoneSize   .EQU    $1800       ;initial size of application zone (6K)
minZone        .EQU    heapData+<4*minFree>+<8*dfltMasters>
                                   ;minimum size of application zone
dfltStackSize  .EQU    $00002000   ;initial space allotment for stack (8K)

tybkFree       .EQU    0           ;tag value for free block
tybkNRel       .EQU    1           ;tag value for nonrelocatable block
tybkRel        .EQU    2           ;tag value for relocatable block
```

One global constant pertinent to the Memory Manager is defined in SYSEQU.TEXT:

```
heapStart     .EQU    $0B00    ;start address of
                               ;   system heap zone (2816)
```

Global Variables

The Memory Manager's global variables are located in the system communication area and defined in the file SYSEQU.TEXT. To access a global variable, just refer to it by name as an absolute address. For example, to load a pointer to the current heap zone into register A2, write

```
MOVE.L  theZone,A2
```

The following global variables are used by the Memory Manager:

Variable	Contents
memTop	Limit address (end plus one) of physical memory
bufPtr	Base address of stack (grows downward from here)
minStack	Minimum space allotment for stack (1K)
defltStack	Default space allotment for stack (8K)
heapEnd	Current limit address of application heap zone
applLimit	Application heap limit
sysZone	Address of system heap zone
applZone	Address of application heap zone
theZone	Address of current heap zone

Trap Macros

All assembly-language trap macros for the Memory Manager (as well as the rest of the Operating System) are defined in the file SYSMACS.TEXT. To call a Memory Manager routine from assembly language via the trap mechanism, just use the name of the trap macro as the operation code of an instruction. For example, to find out the number of free bytes in the current heap zone, use the instruction

```
_FreeMem
```

As stated in the description of FreeMem above, the number of free bytes will be in register D0 on return from the trap.

Result Codes

The file SYSERR.TEXT contains constant definitions for all result codes returned by Operating System routines. You can use them in your program as immediate data values. For example, to test for the error code memFullErr on return from a trap, you might write

```
CMP.W   #memFullErr,DØ
BEQ     NoRoom
```

The Memory Manager uses the following error codes:

```
noErr          .EQU    Ø          ;no error
memFullErr     .EQU    -1Ø8       ;not enough room in zone
nilHandleErr   .EQU    -1Ø9       ;NIL master pointer
memWZErr       .EQU    -111       ;attempt to operate on a free block
memPurErr      .EQU    -112       ;attempt to purge a locked block
```

Offsets and Masks

Offsets to the fields of zone and block headers are defined as constants in the file HEAPDEFS.TEXT. To access a field, use the name of the offset constant as a displacement relative to an address register pointing to the first byte of the header. For example, if register A2 contains a pointer to a zone header, you can load the number of free bytes in the zone into D3 with the instruction

```
MOVE.L  gzProc(A2),D3
```

(eye)

Generally speaking, the offset and mask constants discussed here are intended for the Memory Manager's internal use. You shouldn't ordinarily be prowling around in a zone or block header unless you know what you're doing.

The following offset constants represent the fields of a zone header:

```
bkLim          .EQU    Ø          ;address of zone trailer (long)
purgePtr       .EQU    4          ;roving purge pointer (long)
hFstFree       .EQU    8          ;address of first free
                                  ;   master pointer (long)
zcbFree        .EQU    12         ;number of free bytes (long)
gzProc         .EQU    16         ;address of grow zone
                                  ;   function (long)
moreMasters    .EQU    2Ø         ;incremental master-pointer
                                  ;   count (word)
flags          .EQU    22         ;internal flags (word)
cntRel         .EQU    24         ;relocatable blocks (word)
```

```
maxRel       .EQU   26          ;max. cntRel so far (word)
cntNRel      .EQU   28          ;nonrelocatable blocks (word)
maxNRel      .EQU   30          ;max. cntNRel so far (word)
cntEmpty     .EQU   32          ;empty master pointers (word)
cntHandles   .EQU   34          ;total master pointers (word)
minCBFree    .EQU   36          ;min. zcbFree so far (long)
purgeProc    .EQU   40          ;address of purge warning
                                ;  procedure (long)
sparePtr     .EQU   44          ;spare pointer (long)
allocPtr     .EQU   48          ;roving allocation pointer (long)
heapData     .EQU   52          ;first usable byte in zone
```

The following offset constants represent the fields of a block header:

```
tagBC        .EQU   0           ;tag, size correction, and
                                ;   physical byte count (long)
handle       .EQU   4           ;reloc.:  relative handle (long)
                                ;nonreloc.:  zone pointer (long)
blkData      .EQU   8           ;first byte of block contents
```

HEAPDEFS.TEXT also defines the following mask constants for manipulating the fields of block headers and master pointers:

```
tagMask      .EQU   $C0000000 ;tag field
bcOffMask    .EQU   $0F000000 ;size correction
                              ;   ("byte count offset")
bcMask       .EQU   $00FFFFFF ;physical byte count
ptrMask      .EQU   $00FFFFFF ;address part of master pointer
                              ;   or zone pointer
handleMask   .EQU   $00FFFFFF ;relative handle
freeTag      .EQU   0         ;tag for free block
nRelTag      .EQU   $40000000 ;tag for nonrelocatable block
relTag       .EQU   $80000000 ;tag for relocatable block
```

(eye)

Remember, the pointer or handle you get from the Memory Manager when you allocate a block points to the block's contents, not its header. To get the address of the header, subtract the offset constant blkData, defined above. For example, if you have a handle to a block in register A2, the following code will set A3 to point to the block's header:

```
MOVE.L  (A2),A3        ;get pointer to block contents
SUBQ.L  #blkData,A3    ;offset back to header
```

Finally, SYSEQU.TEXT defines the following constants for the bit numbers of the various flag bits within the high-order byte of a master pointer:

```
lock         .EQU   7        ;lock bit
purge        .EQU   6        ;purge bit
resource     .EQU   5        ;resource bit
```

You can use these constants to access the flag bits directly, using the 68000 instructions BSET, BCLR, and BTST. For instance, if you have a handle to a relocatable block in register A2, you can mark the block as purgeable with the instruction

```
BSET.B  #purge,(A2)    ;set purge bit in master pointer
```

To branch on the current setting of the lock bit,

```
BTST.B  #lock,(A2)     ;test lock bit in master pointer
BNE     ItsLocked      ;   and branch on result
```

Handy Tricks

To save time in critical situations, here's a quick way to convert a dereferenced pointer to a relocatable block back into a handle without paying the overhead of a _RecoverHandle trap. Recall that the relative handle stored in the block's header is the offset of the block's master pointer relative to the start of its heap zone. So to convert a copy of the master pointer back into the original handle, find the relative handle and add it to the address of the zone. For example, if register A2 contains the master pointer of a block in the current heap zone, the following code will reconstruct the block's handle in A3:

```
MOVE.L  -4(A2),A3      ;relative handle is 4 bytes back
                       ;   from start of contents
ADD.L   theZone,A3     ;use as offset from start of zone
```

Conversely, given a true (absolute) handle to a relocatable block, you can find the zone the block belongs to by subtracting the relative handle from the absolute handle. If the absolute handle is in register A2, the following instructions will convert it into a pointer to the block's heap zone:

```
MOVE.L  (A2),A3        ;get pointer to block
SUB.L   -4(A3),A2      ;subtract relative handle
                       ;   to get zone pointer
```

For nonrelocatable blocks, the header contains a pointer directly back to the zone:

```
MOVE.L  -4(A2),A2      ;get zone pointer directly
```

SUMMARY OF THE MEMORY MANAGER

```
CONST noErr        = 0;     {no error}
      memFullErr   = -108;  {not enough room in zone}
      nilHandleErr = -109;  {NIL master pointer}
      memWZErr     = -111;  {attempt to operate on a free block}
      memPurErr    = -112;  {attempt to purge a locked block}

      maxSize = $800000;

TYPE SignedByte = -128..127;
     Byte       = 0..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;
     ProcPtr    = Ptr;

     Size   = LongInt;
     MemErr = INTEGER;

     THz  = ^Zone;
     Zone = RECORD
              bkLim:      Ptr;
              purgePtr:   Ptr;
              hFstFree:   Ptr;
              zcbFree:    LongInt;
              gzProc:     ProcPtr;
              moreMast:   INTEGER;
              flags:      INTEGER;
              cntRel:     INTEGER;
              maxRel:     INTEGER;
              cntNRel:    INTEGER;
              maxNRel:    INTEGER;
              cntEmpty:   INTEGER;
              cntHandles: INTEGER;
              minCBFree:  LongInt;
              purgeProc:  ProcPtr;
              sparePtr:   Ptr;
              allocPtr:   Ptr;
              heapData:   INTEGER
            END;
```

Initialization and Allocation

```
PROCEDURE InitApplZone;
PROCEDURE SetApplBase   (startPtr: Ptr);
PROCEDURE InitZone      (growProc: ProcPtr; masterCount: INTEGER;
                         limitPtr, startPtr: Ptr);
PROCEDURE SetApplLimit  (zoneLimit: Ptr);
```

Heap Zone Access

```
FUNCTION  GetZone :    THz;
PROCEDURE SetZone     (hz: THz);
FUNCTION  SystemZone : THz;  [Pascal only]
FUNCTION  ApplicZone : THz;  [Pascal only]
```

Allocating and Releasing Relocatable Blocks

```
FUNCTION  NewHandle     (logicalSize: Size) : Handle;
PROCEDURE DisposHandle  (h: Handle);
FUNCTION  GetHandleSize (h: Handle) : Size;
PROCEDURE SetHandleSize (h: Handle; newSize: Size);
FUNCTION  HandleZone    (h: Handle) : THz;
FUNCTION  RecoverHandle (p: Ptr) : Handle;
PROCEDURE ReallocHandle (h: Handle; logicalSize: Size);
```

Allocating and Releasing Nonrelocatable Blocks

```
FUNCTION  NewPtr     (logicalSize: Size) : Ptr;
PROCEDURE DisposPtr  (p: Ptr);
FUNCTION  GetPtrSize (p: Ptr) : Size;
PROCEDURE SetPtrSize (p: Ptr; newSize: Size);
FUNCTION  PtrZone    (p: Ptr) : THz;
```

Freeing Space on the Heap

```
FUNCTION  FreeMem :    LongInt;
FUNCTION  MaxMem      (VAR grow: Size) : Size;
FUNCTION  CompactMem  (cbNeeded: Size) : Size;
PROCEDURE ResrvMem    (cbNeeded: Size);
FUNCTION  PurgeMem    (cbNeeded: Size);
PROCEDURE EmptyHandle (h: Handle);
```

Properties of Relocatable Blocks

```
PROCEDURE HLock    (h: Handle);
PROCEDURE HUnlock  (h: Handle);
PROCEDURE HPurge   (h: Handle);
PROCEDURE HNoPurge (h: Handle);
```

Grow Zone Functions

```
PROCEDURE SetGrowZone (growZone: ProcPtr);
FUNCTION  GZCritical : BOOLEAN;  [Pascal only]
FUNCTION  GZSaveHnd :  Handle;   [Pascal only]
```

Utility Routines

```
PROCEDURE BlockMove (sourcePtr, destPtr: Ptr; byteCount: Size);
FUNCTION  TopMem :   Ptr;     [Pascal only]
FUNCTION  MemError : MemErr;  [Pascal only]
```

GLOSSARY

allocate: To reserve a block for use.

application heap zone: The heap zone provided by the Memory Manager for use by the application program.

block: An area of contiguous memory within a heap zone.

block contents: The area of a block available for use.

block header: The internal "housekeeping" information maintained by the Memory Manager at the beginning of each block in a heap zone.

compaction: The process of moving allocated blocks within a heap zone in order to collect the free space into a single block.

current heap zone: The heap zone currently under attention, to which most Memory Manager operations implicitly apply.

dereference: To convert a pointer into whatever it points to; specifically, to convert a handle into a copy of its corresponding master pointer.

empty handle: A handle that points to a NIL master pointer, signifying that the underlying relocatable block has been purged.

free block: A block containing space available for allocation.

grow zone function: A function supplied by the application program to help the Memory Manager create free space within a heap zone.

handle: A pointer to a master pointer, which designates a relocatable block by double indirection.

heap zone: An area of memory in which space can be allocated and released on demand, using the Memory Manager.

limit pointer: A pointer to the byte following the last byte of an area in memory, such as a block or a heap zone.

lock: To temporarily prevent a relocatable block from being moved during heap compaction.

lock bit: A bit in the master pointer to a relocatable block that indicates whether the block is currently locked.

logical size: The number of bytes in a block's contents; compare physical size.

master pointer: A single pointer to a relocatable block, maintained by the Memory Manager and updated whenever the block is moved, purged, or reallocated. All handles to a relocatable block refer to it by double indirection through the master pointer.

nonrelocatable block: A block whose location in its heap zone is fixed and can't be moved during heap compaction.

physical size: The actual number of bytes a block occupies within its heap zone.

purge: To remove a relocatable block from its heap zone, leaving its master pointer allocated but set to NIL.

purgeable block: A relocatable block that can be purged from its heap zone.

purge bit: A bit in the master pointer to a relocatable block that indicates whether the block is currently purgeable.

purge warning procedure: A procedure associated with a particular heap zone that is called whenever a block is purged from that zone.

reallocate: To allocate new space in a heap zone for a purged block, updating its master pointer to point to its new location.

relative handle: A handle to a relocatable block expressed as the offset of its master pointer within the heap zone, rather than as the absolute memory address of the master pointer.

release: To destroy an allocated block, freeing the space it occupies.

relocatable block: A block that can be moved within its heap zone during compaction.

result code: An integer code produced by a Memory Manager routine to signal the success of an operation or the reason for its failure.

size correction: The number of unused bytes included at the end of an allocated block; the difference between the block's logical and physical sizes, excluding the block header.

system heap zone: The heap zone provided by the Memory Manager for use by the Macintosh system software.

tag: A 2-bit code in the header of a block identifying it as relocatable, nonrelocatable, or free.

unlock: To allow a relocatable block to be moved during heap compaction.

unpurgeable block: A relocatable block that can't be purged from its heap zone.

zone header: The internal "housekeeping" information maintained by the Memory Manager at the beginning of each heap zone.

zone pointer: A pointer to a zone record.

zone record: A Pascal data structure representing the structure of a zone header.

zone trailer: A minimum-size free block marking the end of a heap zone.

MACINTOSH USER EDUCATION

The Menu Manager: A Programmer's Guide /MMGR/MENUS

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Toolbox Utilities: A Programmer's Guide

Modification History:

First Draft	P. Stanton-Wyman	
Second Draft	C. Espinosa	12/23/82
Updated (ROM 2.0)	C. Espinosa	1/24/83
Third Draft (ROM 3.0)	C. Espinosa & C. Rose	5/17/83
Fourth Draft (ROM 7)	C. Rose	11/1/83

ABSTRACT

The Macintosh User Interface frees the user from having to remember long strings of command words by placing all commands in menus. With the menu bar and pull-down menus, the user can at any time see all available menu choices. This manual describes the nature of pull-down menus and how to implement them with the Macintosh Menu Manager.

Summary of significant changes and additions since last version:

- The symbol for showing keyboard equivalents for menu items has changed from a solid apple to the Command key's symbol on the keyboard (page 6).

- The use of the "!" meta-character to indicate a marked menu item has changed (page 11).

- A new procedure, InsertResMenu, has been added (page 18).

- The predefined constant mCalcSize, for the menu definition procedure's message parameter, has been renamed mSizeMsg (page 27).

- For assembly-language programmers, the unconventional macro names for calling several of the Menu Manager routines are now listed under the descriptions of those routines, and some additional system globals are discussed.

TABLE OF CONTENTS

3 About This Manual
4 About the Menu Manager
4 The Menu Bar
5 Appearance of Menus
7 Menus and Resources
8 Menu Records
9 The Menu List
10 Creating a Menu
11 Separating Items
11 Items with Icons
11 Marked Items
12 Character Style of Items
12 Items with Keyboard Equivalents
13 Disabled Items
13 Using the Menu Manager
15 Menu Manager Routines
15 Initialization and Allocation
18 Forming the Menu Bar
20 Choosing From a Menu
22 Controlling Items' Appearance
25 Miscellaneous Utilities
26 Defining Your Own Menus
27 The Menu Definition Procedure
28 Formats of Resources for Menus
29 Menus in a Resource File
30 Menu Bars in a Resource File
31 Summary of the Menu Manager
35 Glossary

Copyright (c) Apple Computer, Inc. All rights reserved. Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Menu Manager, a major component of the Macintosh User Interface Toolbox. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. *** The Menu Manager allows you to create sets of menus, and allows the user to choose from the commands in those menus in a manner consistent with the Macintosh User Interface guidelines.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Menu Manager may not work as discussed here.

Like all documentation about the Toolbox, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly points, rectangles, and character style.

- Resources, as described in the Resource Manager manual.

- The Toolbox Event Manager. Some Menu Manager routines should be called only in response to certain events.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it. *** Some of that information refers to the "Toolbox equates" file (ToolEqu.Text), which the reader will have learned about in an earlier chapter of the final comprehensive manual. ***

The manual begins with an introduction to the Menu Manager and the appearance of menus on the Macintosh. It then discusses the basics of menus: the relationship between menus and resources, some internal structures related to menus, and information about how to create menus.

Next, a section on using the Menu Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Menu Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers: special information is provided for programmers who want to define their own menus, and the exact formats of resources related to menus are described.

Finally, there's a summary of the Menu Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE MENU MANAGER

The Menu Manager supports the use of menus, an integral part of the Macintosh User Interface. Menus allow users to examine all choices available to them at any time without being forced to choose one of them, and without having to remember command words or special keys. The Macintosh user simply positions the cursor in the menu bar and presses the mouse button over a menu title. The application then calls the Menu Manager, which highlights that title (by inverting it) and "pulls down" the menu below it. As long as the mouse button is held down, the menu is displayed. Dragging the mouse through the menu items causes each of the items to be highlighted in turn. If the mouse button is released over an item, that item is "chosen". The item blinks briefly to confirm the choice, and the menu disappears.

After a successful choice, the Menu Manager tells the application which item was chosen, and the application performs the corresponding action. When the application completes the action, it removes the highlighting from the menu title, indicating to the user that the operation is complete.

If the user moves the cursor out of the menu and releases the mouse button, no choice is made: the menu simply disappears and the application takes no action. The user is never forced to choose a command once a menu has been pulled down.

The Menu Bar

The menu bar always appears at the top of the Macintosh screen, 20 pixels high and as wide as the screen. It appears in front of all windows; nothing but the cursor ever appears in front of the menu bar. The menu bar is white and has a thin black lower border, and the menu titles in it are always in the system font and the system font size (see Figure 1).

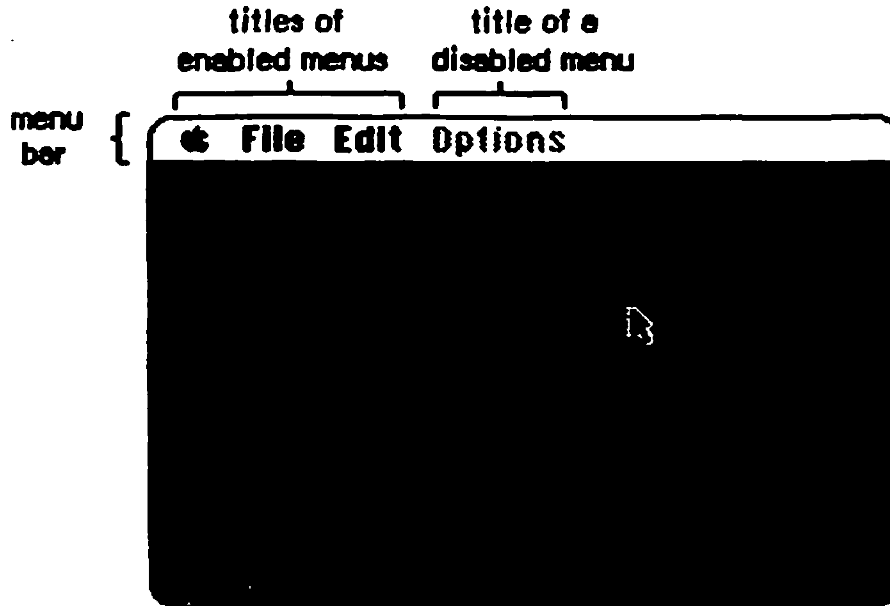


Figure 1. The Menu Bar

In applications that support desk accessories, the first menu should be the standard Apple menu (the menu whose title is an Apple symbol). This menu contains the names of all available desk accessories. When the user chooses a desk accessory, the title of a menu belonging to it may also appear in the menu bar, for as long as the accessory is active, or the entire menu bar may be occupied by menus belonging to the desk accessory. (Desk accessories are discussed in detail in the Desk Manager manual.)

A menu may temporarily be disabled, so that none of the items in the menu can be chosen. The title of a disabled menu and every item in it appear dimmed in the menu bar (that is, drawn in gray rather than black).

The maximum number of menu titles in the menu bar is 16; however, ten to twelve titles is usually all that will fit. If you're having trouble fitting your menus in the menu bar, you should review your menu organization and menu titles.

Appearance of Menus

A standard menu consists of a number of lines of text, listed vertically inside a shadowed rectangle (see Figure 2). Menus always appear in front of everything else (except the cursor); in Figure 2, the menu appears in front of a document window already on the screen.

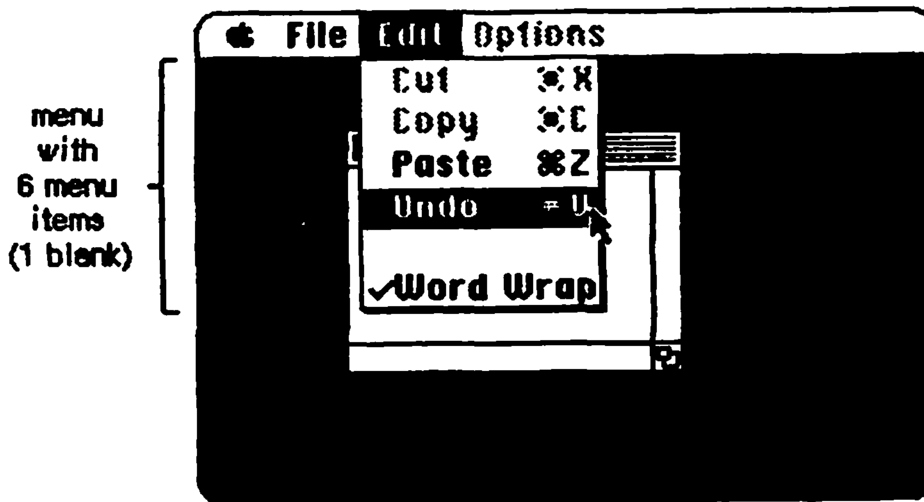


Figure 2. A Standard Menu

Each line of text is one menu item that the user can choose from that menu. The text always appears in the system font and the system font size. Each item can have a few visual variations from the standard appearance:

- An icon to the left of the item's text, to give a symbolic representation of the item's meaning or effect.

- A check mark or other character to the left of the item's text (or icon, if any), to denote the status of the item or of the mode it controls.

- The Command key symbol and another character to the right of the item's text, to show that the item may be invoked from the keyboard (that is, it has a keyboard equivalent).

- A character style other than the standard, such as bold, italic, underline, or a combination of these. (The QuickDraw manual gives a full discussion of character style.)

- A dimmed appearance, to indicate that the item is disabled.

(hand)

Special symbols or icons may have an unusual appearance when dimmed; notice the dimmed Command symbol in the Cut and Copy menu items in Figure 2.

The maximum number of menu items that will fit in a standard menu is 20 (minus 1 for every item that contains an icon). The fewer menu items you have, the simpler and clearer the menu appears to the user. To separate groups of items, you may use blank menu items or items consisting entirely of dashes.

If the standard menu doesn't suit your needs (for example, if you want more graphics or perhaps a nonlinear text arrangement), you can define a custom menu that, although visibly different to the user, responds to your application's Menu Manager calls just like a standard menu.

MENUS AND RESOURCES

The general definition of how a certain type of menu looks and behaves is determined by a menu definition procedure, which is usually stored as a resource in a resource file. Most applications will use the predefined menu definition procedure in the system resource file; others may write their own menu definition procedures (as described later in the section "Defining Your Own Menus").

One way to define the contents of your application's menus is to have your program create them manually, item by item. When you create a menu this way, the Menu Manager automatically sets it up to use the standard menu definition procedure and gets that procedure from the system resource file. The standard menu definition procedure has the capabilities described above: it lists the menu items vertically, and each one may have an icon, check mark, keyboard equivalent, different character style, or dimmed appearance.

You can also set up your application's menus by reading them in from a resource file. This is strongly recommended, for two reasons: it makes your application smaller, and it allows the menu items to be edited for documentation or translated to foreign languages without affecting the application's source code. The Menu Manager allows you to read not only individual menus but also complete menu bars from a resource file.

(hand)

> You can create menus and menu bars and store them in resource files with the aid of the Resource Editor *** eventually ***. The Resource Editor relieves you of having to know the exact formats of these resources in the file, but for interested programmers this information is given in the section "Formats of Resources for Menus". *** In the absence of the Resource Editor, you can write a small program to create your menus using the Menu Manager procedure AppendMenu, and store them in a resource file using the standard Resource Manager calls. You can also use the interim Resource Compiler; see the manual "Putting Together a Macintosh Application" for more information. ***

Even if you don't store entire menus in resource files, it's a good idea to store the text strings they contain as resources; you can call the Resource Manager directly to read them in. Icons in menus are read from resource files; in this case, the Menu Manager calls the Resource Manager.

There's one other interaction between menus and resources: a Menu Manager procedure that scans all open resource files for resources of a given type and install the names of all available resources of that type into a given menu. This is how you fill a menu with the names of all available desk accessories, for example.

MENU RECORDS

The Menu Manager keeps all the information it needs for its operations on a particular menu in a menu record. The menu record contains:

- The menu ID. For menus stored in resource files, this is the resource ID; for menus created by your application, it's any positive number (less than 32768) that you choose to identify the menu.
- The menu title.
- The contents of the menu; the text and other parts of each item.
- The horizontal and vertical dimensions of the menu, in pixels. The menu items appear inside the rectangle formed by these dimensions; the black border and shadow of the menu appear outside that rectangle.
- A handle to the menu definition procedure.
- Flags telling whether each menu item is enabled or disabled, and whether the menu itself is enabled or disabled.

The data type for a menu record is called MenuInfo. A menu is a dynamic, relocatable data structure and is referred to by a handle.

```
TYPE MenuPtr    = ^MenuInfo;
     MenuHandle = ^MenuPtr;
```

You can store into and access all the necessary fields of a menu record with Menu Manager routines, so normally you don't have to know its exact structure. Advanced users, however—particularly those who define their own types of menus—may need to know some of the field names.

```
TYPE MenuInfo = RECORD
                  menuID:      INTEGER;
                  menuWidth:   INTEGER;
                  menuHeight:  INTEGER;
                  menuProc:    Handle;
                  enableFlags: PACKED ARRAY [0..31] OF BOOLEAN;
                  menuData:    Str255
                END;
```

The menuID field contains the menu ID.

The menuWidth and menuHeight fields contain the menu's horizontal and vertical dimensions, respectively.

The menuProc field contains a handle to the menu definition procedure for this type of menu.

The Øth element of the enableFlags array is TRUE if the menu is enabled, or FALSE if it's disabled. The remaining elements similiarly determine whether each item in the menu is enabled or disabled.

The menuData field contains the menu title followed by variable-length data that defines the text and other parts of the menu items. The Str255 data type enables you to access the title from Pascal; there's actually additional data beyond the title that's inaccessible from Pascal and is not reflected in the MenuInfo data structure.

(eye)

> You can read the menu title directly from the menuData field, but do not change the title directly, or the data defining the menu items may be destroyed.

> Assembly-language note: The Toolbox equates file includes menuBlkSize, the length in bytes of all the fields of a menu record except menuData.

THE MENU LIST

The Menu Manager keeps a list of menu handles for all menus in the menu bar. The user can pull down and choose from any menu whose handle is in this menu list. The menu bar shows the titles, in order, of all menus in the menu list.

You can have menus that aren't in the menu list. These menus' titles don't appear in the menu bar, the menus can't be pulled down, and their items can't be chosen. Such menus are useful as "reserve" menus to hold items not normally available to the user; these items can be exchanged with items in other menus, or entire reserve menus can be added to the menu bar.

The Menu Manager provides all the necessary routines for manipulating the menu list, so there's no need to access it yourself directly. As a general rule, routines that deal specifically with menus in the menu list use the menu ID to refer to menus; those that deal with any menus, whether in the menu list or not, use the menu handle to refer to menus. Some routines refer to the menu list as a whole, with a handle.

Assembly-language note: The system global menuList contains a handle to the current menu list.

CREATING A MENU

For an application to create menus itself, rather than read them from a resource file, it must call the NewMenu and AppendMenu routines of the Menu Manager. NewMenu creates a new menu data structure, returning a handle to it. AppendMenu takes a string and a handle to a menu and adds the items in the string to the end of the menu.

The string passed to AppendMenu consists mainly of the text of the menu items (for a blank item, one or more spaces). Other characters interspersed in the string can have special meaning to the Menu Manager. These characters, called meta-characters, are used in conjunction with text to separate menu items or alter their appearance. The meta-characters do not appear in the menu.

Meta-character	Meaning
; or Return	Separates items
^	Item has an icon
!	Item has a check mark or other mark
<	Item has a special character style
/	Item has a keyboard equivalent
(	Item is disabled

None, any, or all of these meta-characters can appear in the AppendMenu string; they are described in detail below. To add one text-only item to a menu would require a simple string without any meta-characters:

```
AppendMenu(thisMenu,'Just Enough');
```

An extreme example could use many meta-characters:

```
AppendMenu(thisMenu,'(Too Much^1<B/T');
```

This example adds to the menu an item whose text is "Too Much", which is disabled, has icon number 1, is boldfaced, and can be invoked by Command-T. Your menu items should be much simpler than this.

(hand)

If you want any of the meta-characters to appear in the text of a menu item, you can include them by changing the text with the Menu Manager procedure SetItem.

Separating Items

Each call to AppendMenu can add one or many items to the menu. To add multiple items in the same call, use a semicolon (";") or a Return character to separate the items. The call

```
AppendMenu(thisMenu,'Cut;Copy');
```

has exactly the same effect as the calls

```
AppendMenu(thisMenu,'Cut');
AppendMenu(thisMenu,'Copy');
```

Items with Icons

A circumflex ("^") followed by a digit from 1 to 9 indicates that an icon should appear to the left of the menu item's text. The digit, which is called the <u>icon number</u>, yields the resource ID of the icon in the resource file. Resource IDs 257 through 511 are reserved for menu icons; thus the Menu Manager adds 256 to the icon number to get the proper resource ID.

If you need to install more than nine icons, you can use the SetItemIcon procedure.

(hand)
The Menu Manager gets the icon number by subtracting 48 from the ASCII code of the character following the "^" (since, for example, the ASCII code of "1" is 49). You can actually follow the "^" with any character that has an ASCII code greater than 48.

Marked Items

You can use an exclamation point ("!") to cause a check mark or any other character to be placed to the left of the menu item's text (or icon, if any). Follow the exclamation point with the character of your choice; note, however, that you may not be able to type a check mark or certain other special characters (such as the Apple symbol) from the keyboard. To specify one of these characters, you need to take special measures: Declare a string variable to have the length of the desired AppendMenu string, and assign it that string with a space following the exclamation point. Then separately store the special character in the position of the space. The following predefined constants may be useful:

```
CONST checkMark   = 18;    {check mark}
      appleSymbol = 20;    {Apple symbol}
```

For example, suppose you want to use AppendMenu to specify a menu item that has the text "Word Wrap" (nine characters) and a check mark to its left. You can declare the string variable

```
VAR s: STRING[11];
```

and do the following:

```
s := 'Word Wrap! '
s[11] := CHR(checkMark);
AppendMenu(thisMenu,s);
```

Character Style of Items

The system font is the only font available for menus; however, you can vary the character style for clarity and distinction. The meta-character used to specify the character style is the left angle bracket, "<". With AppendMenu, you can assign one and only one of the stylistic variations listed below.

<B	Bold
<I	Italic
<U	Underline
<O	Outline
<S	Shadow

The SetItemStyle procedure allows you to assign any character style to an item. For a further discussion of character style, see the QuickDraw manual.

Items with Keyboard Equivalents

Any menu item that can be chosen from a menu may also be associated with a key on the keyboard. Pressing this key while holding down the Command key invokes the item just as if it had been chosen from the menu.

A slash ("/") followed by a character associates that character with the item. The specified character (preceded by the Command key symbol) appears at the right of the item's text in the menu. For consistency between applications, the character should be uppercase if it's a letter. When invoking the item, the user can type the letter in either uppercase or lowercase. For example, if you specify 'Copy/C', the Copy command can be invoked by holding down the Command key and typing either C or c.

An application that receives a key down event with the Command key held down can call the Menu Manager with the typed character and receive the menu ID and item number of the item associated with that character.

Disabled Items

All items in a menu are usually choosable. There will be times when you don't want an item to be choosable, either initially or for the duration of your program (perhaps due to the program's incomplete state). The meta-character that disables an item is the left parenthesis "(". A disabled item cannot be chosen; it appears dimmed in the menu and is not highlighted when the cursor moves over it.

Blank items in a menu should always be disabled, as should any items used to separate groups of items. For example, the call

```
AppendMenu(thisMenu,'Undo;( ;Word Wrap');
```

adds two enabled menu items, Undo and Word Wrap, with a disabled blank item between them. Note that one or more spaces are required to specify a blank item.

You can change the enabled or disabled state of a menu item with the DisableItem and EnableItem procedures.

USING THE MENU MANAGER

This section discusses how the Menu Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

To use the Menu Manager, you must have previously called InitGraf to initialize QuickDraw, InitFonts to initialize the Font Manager, and InitWindows to initialize the Window Manager. The first Menu Manager routine to call is the initialization procedure InitMenus.

Your application can set up the menus it needs in any number of ways:

- Allocate the menus with NewMenu, fill them with items using AppendMenu, and place them in the menu bar using InsertMenu.

- Read the menus individually from a resource file using GetMenu, and place them in the menu bar using InsertMenu.

- Read an entire prepared menu list from a resource file with GetNewMBar, and place it in the menu bar with SetMenuBar.

- Allocate a menu with NewMenu, fill it with items using AddResMenu to get the names of all available resources of a given type, and place the menu in the menu bar using InsertMenu.

You can use AddResMenu or InsertResMenu to add items from resource files to any menu, regardless of how you created the menu or whether it already contains any items.

If you call NewMenu to allocate a menu, it will store a handle to the standard menu definition procedure in the window record; so if you want the menu to be one of your own design, you must replace that handle with a handle to your own menu definition procedure. For more information, see "Defining Your Own Menus".

At any time you can change or examine the appearance of an individual menu item with the SetItem and GetItem procedures (and similar procedures to set or get the item's icon, style, check mark, and so on). You can also change the number and order of menus in the menu list with InsertMenu and DeleteMenu, or change the entire menu list with ClearMenuBar, GetNewMBar, GetMenuBar, and SetMenuBar.

When your application receives a mouse down event, and the Window Manager's FindWindow function returns the predefined constant inMenuBar, your application should call the Menu Manager's MenuSelect function, supplying it with the point where the mouse button was pressed. MenuSelect will pull down the appropriate menu, and retain control--tracking the mouse, highlighting menu items, and pulling down other menus--until the user releases the mouse button. MenuSelect returns a long integer to the application: the high-order word contains the menu ID of the menu that was chosen, and the low-order word contains the <u>menu item number</u> of the item that was chosen. The menu item number is the index, starting from 1, of the item in the menu. The entire long integer is Ø if no item was chosen.

- If the long integer is Ø, your application should just continue to poll for further events.

- If the long integer is nonzero, the application should take the appropriate action for when the menu item specified by the low-order word is chosen from the menu whose ID is in the high-order word. Only after the action is completely finished (after all dialogs, alerts, or screen actions have been taken care of) should your application call HiliteMenu(Ø) to remove the highlighting from the menu bar, signaling the completion of the action.

Keyboard equivalents are handled in much the same manner. When your application receives a key down event with the Command key held down, it should call the MenuKey function, supplying it with the character that was typed. MenuKey will return a long integer with the same format as that of MenuSelect, and the application can handle the long integer in the manner described above.

(hand)

> You can use the Toolbox Utility routines LoWord and HiWord to extract the high-order and low-order words of a given long integer, as described in the Toolbox Utilities manual.

When you no longer need a menu, call DisposeMenu if you allocated it with NewMenu, or call the Resource Manager procedure ReleaseResource if you used GetMenu.

MENU MANAGER ROUTINES

This section describes all the Menu Manager procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Initialization and Allocation

```
PROCEDURE InitMenus;
```

InitMenus initializes system globals used by the Menu Manager, sets up its internal data structures, clears the menu list, and draws the (empty) menu bar. Call it once before all other Menu Manager routines. An application should never have to call this procedure more than once; to start afresh with all new menus, use ClearMenuBar.

(hand)

> InitWindows, which you previously called to initialize the Window Manager, will already have drawn the menu bar; InitMenus also draws the menu bar just in case it does happen to be called in mid-application.

```
FUNCTION NewMenu (menuID: INTEGER; menuTitle: Str255) : MenuHandle;
```

NewMenu allocates space for a new menu with the given menu ID and title, and returns a handle to it. The new menu (which is created empty) is not installed in the menu list. To use this menu, you must first call AppendMenu or AddResMenu to fill it with items, InsertMenu to place it in the menu list, and DrawMenuBar to update the menu bar to include the new title.

Application menus should always have positive menu IDs. Negative menu IDs are reserved for menus belonging to desk accessories. No menu should ever have a menu ID of Ø.

To set up the title of the Apple menu of desk accessory names, you can use the predefined constant appleSymbol (equal to 2Ø, the ASCII code of the Apple symbol). For example, you can declare the string variable

```
VAR myTitle: STRING[1];
```

and do the following:

```
myTitle := ' ';
myTitle[1] := CHR(appleSymbol);
```

(hand)

Once a menu is created with NewMenu, the only way to deallocate the memory it occupies is by calling DisposeMenu.

FUNCTION GetMenu (menuID: INTEGER) : MenuHandle;

GetMenu returns a menu handle for the menu having the given resource ID. If the menu isn't already in memory, GetMenu calls the Resource Manager to read it from the resource file into a menu record in memory. It stores the handle to the menu definition procedure in the menu record, reading the procedure from the resource file into memory if necessary. To use this menu, you must call InsertMenu to place it in the menu list and DrawMenuBar to update the menu bar to include the new title.

(hand)

To deallocate the memory occupied by a menu that you read from a resource file with GetMenu, use the Resource Manager procedure ReleaseResource.

Assembly-language note: The macro you invoke to call GetMenu from assembly language is named _GetRMenu.

PROCEDURE DisposeMenu (menu: MenuHandle);

Call DisposeMenu to deallocate the memory occupied by a menu that you allocated with NewMenu. (For menus read from a resource file with GetMenu, use the Resource Manager procedure ReleaseResource instead.) This is useful if you've created temporary menus that you no longer need.

(eye)

Make sure you remove the menu from the menu list (with DeleteMenu) before disposing of it. Also be careful not to use the menu handle after disposing of the menu.

Assembly-language note: The macro you invoke to call DisposeMenu from assembly language is named _DisposMenu.

```
PROCEDURE AppendMenu (menu: MenuHandle; data: Str255);
```

AppendMenu adds an item or items to the end of the given menu, which must previously have been allocated by NewMenu or read from a resource file by GetMenu. The data string consists of the text of the menu item; it may be blank but should not be the null string. As described in the section "Creating a Menu", the following meta-characters may be embedded in the data string:

Meta-character	Usage
; or Return	Separates multiple items
^	Followed by an icon number, adds that icon to the item
!	Followed by a character, marks the item with that character
<	Followed by B, I, U, O, or S, sets the character style of the item
/	Followed by a character, associates a keyboard equivalent with the item
(	Disables the item

Once items have been appended to a menu, they cannot be removed or rearranged. AppendMenu works properly whether or not the menu is in the menu list.

```
PROCEDURE AddResMenu (menu: MenuHandle; theType: ResType);
```

AddResMenu searches all open resource files for resources of type theType and appends the names of all resources it finds to the given menu. Each resource name appears in the menu as an enabled item, without an icon or mark, and in the normal character style. The standard Menu Manager calls can be used to get the name or change its appearance, as described below under "Controlling Items' Appearance".

(hand)

So that you can have resources of the given type that won't appear in the menu, AddResMenu does not append any resource names that begin with a period (".").

Use this procedure to fill a menu with the names of all available fonts or desk accessories. For example, if you declare a variable as

```
VAR fontMenu: MenuHandle;
```

you can set up a menu containing all font names as follows:

```
fontMenu := NewMenu(5,'Fonts');
AddResMenu(fontMenu,'FONT');
```

PROCEDURE InsertResMenu (menu: MenuHandle; theType: ResType; afterItem: INTEGER);

InsertResMenu is the same as AddResMenu (above) except that it inserts the resource names in the menu where specified by the afterItem parameter: if afterItem is Ø, the names are inserted before the first menu item; if it's the item number of an item in the menu, they're inserted after that item; if it's equal to or greater than the last item number, they're appended to the menu as by AddResMenu.

(hand)

InsertResMenu inserts the names in the reverse of the order that AddResMenu appends them. For consistency in the appearance of menus between applications, use AddResMenu instead of InsertResMenu if possible.

Forming the Menu Bar

PROCEDURE InsertMenu (menu: MenuHandle; beforeID: INTEGER);

InsertMenu inserts a menu into the menu list before the menu whose menu ID equals beforeID. If beforeID is Ø (or isn't the ID of any menu in the menu list), the new menu is added after all others. If the menu is already in the menu list, InsertMenu does nothing. Be sure to call DrawMenuBar to update the menu bar.

PROCEDURE DrawMenuBar;

DrawMenuBar redraws the menu bar according to the menu list, incorporating any changes since the last call to DrawMenuBar. Any highlighted menu title remains highlighted when drawn by DrawMenuBar. This procedure should always be called after a sequence of InsertMenu or DeleteMenu calls, and after ClearMenuBar, SetMenuBar, or any other routine that changes the menu list.

PROCEDURE DeleteMenu (menuID: INTEGER);

DeleteMenu deletes a menu from the menu list. If there's no menu with the given menu ID in the menu list, DeleteMenu has no effect. Be sure to call DrawMenuBar to update the menu bar; the menu titles following the deleted menu will move over to fill the vacancy.

(hand)

DeleteMenu simply removes the menu from the list of currently available menus; it doesn't deallocate the menu data structure.

PROCEDURE ClearMenuBar;

Call ClearMenuBar to remove all menus from the menu list when you want to start afresh with all new menus. Be sure to call DrawMenuBar to update the menu bar.

(hand)
ClearMenuBar, like DeleteMenu, doesn't deallocate the menu data structures; it merely removes them from the menu list.

You don't have to call ClearMenuBar at the beginning of your program, because InitMenus clears the menu list for you.

FUNCTION GetNewMBar (menuBarID: INTEGER) : Handle;

GetNewMBar creates a menu list as defined by the menu bar resource having the given resource ID, and returns a handle to it. If the resource isn't already in memory, GetNewMBar reads it into memory from the resource file. It calls GetMenu to get each of the individual menus.

To make the menu list the current menu list, call SetMenuBar. To dispose of the memory occupied by the menu list, use the Memory Manager procedure DisposHandle.

(eye)
You don't have to know the individual menu IDs to use GetNewMBar, but that doesn't mean you don't have to know them at all: to do anything further with a particular menu, you have to know its ID or its handle (which you can get by passing the ID to GetMHandle, as described below under "Miscellaneous Utilities").

FUNCTION GetMenuBar : Handle;

GetMenuBar creates a copy of the current menu list and returns a handle to the copy. You can then add or remove menus from the menu list (with InsertMenu, DeleteMenu, or ClearMenuBar), and later restore the saved menu list with SetMenuBar. To dispose of the memory occupied by the saved menu list, use the Memory Manager procedure DisposHandle.

(eye)
GetMenuBar doesn't copy the menus themselves, only a list of their handles. Do not dispose of any menus that might be in a saved menu list!

PROCEDURE SetMenuBar (menuBar: Handle);

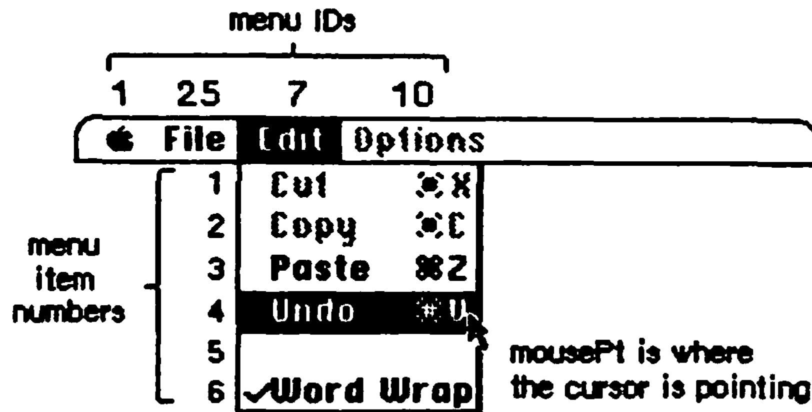
Given a handle to a menu list, SetMenuBar makes it the current menu list. You can use this procedure to restore a menu list previously

saved by GetMenuBar, or pass it a handle returned by GetNewMBar. Be sure to call DrawMenuBar to update the menu bar.

Choosing From a Menu

FUNCTION MenuSelect (startPt: Point) : LongInt;

When a mouse down event occurs in the menu bar, you should call MenuSelect with startPt (in global coordinates) equal to the point where the mouse button was pressed. MenuSelect tracks the mouse, pulling down menus as needed and highlighting menu items under the cursor. When the mouse button is released over an enabled item in an application menu, MenuSelect returns a long integer whose high-order word is the menu ID of the menu, and whose low-order word is the menu item number for the item chosen (see Figure 3). It leaves the selected menu title highlighted. After performing the chosen task, your application should call HiliteMenu(Ø) to remove the highlighting from the menu title.



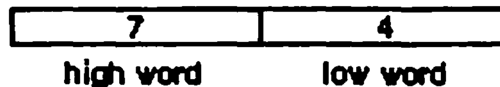


Figure 3. MenuSelect and MenuKey

MenuSelect returns Ø if no choice is made; this includes the case where the mouse button is released over a disabled menu item (such as the blank item in Figure 3) or over any menu title.

If the mouse button is released over an enabled item in a menu belonging to a desk accessory, MenuSelect passes the menu ID and item number to the Desk Manager procedure SystemMenu for processing and returns Ø to your application.

Assembly-language note: If the system global mBarEnable is nonzero, MenuSelect knows that every menu currently in the menu bar belongs to a desk accessory. (See the Desk Manager manual for more information.) The system global menuHook normally contains Ø; you can store in it the address of a routine having no parameters, and MenuSelect will call that routine repeatedly while the mouse button is down.

FUNCTION MenuKey (ch: CHAR) : LongInt;

MenuKey maps the given character to the associated menu and item for that character. When you get a key down event with the Command key held down, call MenuKey with the character that was typed (which can be found in the low-order byte of the event message). MenuKey highlights the appropriate menu title and returns a long integer just as MenuSelect does. This long integer contains the menu ID in its high-order word and the menu item number in its low-order word (see Figure 3 above). After performing the chosen task, your application should call HiliteMenu(Ø) to remove the highlighting from the menu title.

MenuKey returns Ø if the given character isn't associated with any enabled menu item currently in the menu list.

If the given character invokes a menu item in a menu belonging to a desk accessory, MenuKey (like MenuSelect) passes the menu ID and item number to the Desk Manager procedure SystemMenu for processing and returns Ø to your application.

(hand)

There should never be more than one item in the menu list with the same keyboard equivalent, but if there is, MenuKey returns the first such item encountered (scanning the menus from left to right and their items from top to bottom).

PROCEDURE HiliteMenu (menuID: INTEGER);

HiliteMenu highlights the title of the given menu, or does nothing if the title is already highlighted. Since only one menu title can be highlighted at a time, it unhighlights any previously highlighted menu title. If menuID is Ø (or isn't the ID of any menu in the menu list), HiliteMenu simply unhighlights whichever menu title is highlighted.

After MenuSelect or MenuKey, your application should perform the chosen task and then call HiliteMenu(Ø) to unhighlight the chosen menu title.

Assembly-language note: The system global theMenu contains the menu ID of the currently highlighted menu.

Controlling Items' Appearance

```
PROCEDURE SetItem (menu: MenuHandle; item: INTEGER; itemString:
          Str255);
```

SetItem changes the text of the given menu item to itemString. It doesn't recognize the meta-characters used in AppendMenu; if you include them in itemString, they will appear in the text of the menu item. The attributes already in effect for this item--its character style, icon, and so on--remain in effect. ItemString may be blank but should not be the null string.

Use SetItem to flip between two alternative menu items--for example, to change "Show Clipboard" to "Hide Clipboard" when the Clipboard is already showing.

(hand)
We heartily recommend against capricious changing of menu items.

```
PROCEDURE GetItem (menu: MenuHandle; item: INTEGER; VAR itemString:
          Str255);
```

GetItem returns the text of the given menu item in itemString. It doesn't place any meta-characters in the string. This procedure is useful for getting the name of a menu item that was installed with AddResMenu or InsertResMenu.

```
PROCEDURE DisableItem (menu: MenuHandle; item: INTEGER);
```

Given a menu item number in the item parameter, DisableItem disables that menu item; given Ø in the item parameter, it disables the entire menu.

Disabled menu items appear dimmed and are not highlighted when the cursor moves over them. MenuSelect and MenuKey return Ø if the user attempts to invoke a disabled item. Use DisableItem to disable all menu choices that aren't appropriate at a given time (such as a Cut command when there's no text selection).

All menu items are initially enabled unless you specify otherwise (such as by using the "(" meta-character in a call to AppendMenu).

Every menu item in a disabled menu is dimmed. The menu title is also dimmed, but you must call DrawMenuBar to update the menu bar to show the dimmed title.

PROCEDURE EnableItem (menu: MenuHandle; item: INTEGER);

Given a menu item number in the item parameter, EnableItem enables the item; given Ø in the item parameter, it enables the entire menu. (The item or menu may have been disabled with the DisableItem procedure, or the item may have been disabled with the "(" meta-character in the AppendMenu string.) The item or menu title will no longer appear dimmed and can be chosen like any other enabled item or menu.

PROCEDURE CheckItem (menu: MenuHandle; item: INTEGER; checked:
            BOOLEAN);

CheckItem places or removes a check mark at the left of the given menu item. After you call CheckItem with checked=TRUE, a check mark will appear each subsequent time the menu is pulled down. Calling CheckItem with checked=FALSE removes the check mark from the menu item (or, if it's marked with a different character, removes that mark).

Menu items are initially unmarked unless you specify otherwise (such as with the "!" meta-character in a call to AppendMenu).

PROCEDURE SetItemIcon (menu: MenuHandle; item: INTEGER; icon: INTEGER);

SetItemIcon associates the given menu item with an icon. It sets the item's icon number to the given value (an integer from 1 to 255). The Menu Manager adds 256 to the icon number to get the icon's resource ID, which it passes to the Resource Manager to get the corresponding icon.

(eye)

> If you deal directly with the Resource Manager to read or store menu icons, be sure to adjust your icon numbers accordingly.

Menu items initially have no icons unless you specify otherwise (such as with the "^" meta-character in a call to AppendMenu).

> Assembly-language note: The macro you invoke to call SetItemIcon from assembly language is named _SetItmIcon.

PROCEDURE GetItemIcon (menu: MenuHandle; item: INTEGER; VAR icon:
INTEGER);

GetItemIcon returns the icon number associated with the given menu item, as an integer from 1 to 255, or Ø if the item has not been associated with an icon. The icon number is 256 less than the icon's resource ID.

Assembly-language note: The macro you invoke to call GetItemIcon from assembly language is named _GetItmIcon.

PROCEDURE SetItemStyle (menu: MenuHandle; item: INTEGER; chStyle:
Style);

SetItemStyle changes the character style of the given menu item to chStyle. For example:

```
SetItemStyle(thisMenu,1,[bold,italic]);     {bold and italic}
```

Menu items are initially in the normal character style unless you specify otherwise (such as with the "<" meta-character in a call to AppendMenu).

Assembly-language note: The macro you invoke to call SetItemStyle from assembly language is named _SetItmStyle.

PROCEDURE GetItemStyle (menu: MenuHandle; item: INTEGER; VAR chStyle:
Style);

GetItemStyle returns the character style of the given menu item in chStyle.

Assembly-language note: The macro you invoke to call GetItemStyle from assembly language is named _GetItmStyle.

```
PROCEDURE SetItemMark (menu: MenuHandle; item: INTEGER; markChar:
        CHAR);
```

SetItemMark marks the given menu item in a more general manner than CheckItem. It allows you to place any character in the system font, not just the check mark, to the left of the item. You can specify some useful values for the markChar parameter with the following predefined constants:

```
CONST checkMark   = 18;    {check mark}
      appleSymbol = 2Ø;    {Apple symbol}
      noMark      = Ø;     {nothing, to remove a mark}
```

Assembly-language note: The macro you invoke to call SetItemMark from assembly language is named _SetItmMark.

```
PROCEDURE GetItemMark (menu: MenuHandle; item: INTEGER; VAR markChar:
        CHAR);
```

GetItemMark returns in markChar whatever character the given menu item is marked with, or the NUL character (ASCII code Ø) if no mark is present.

Assembly-language note: The macro you invoke to call GetItemMark from assembly language is named _GetItmMark.

Miscellaneous Utilities

```
PROCEDURE SetMenuFlash (menu: MenuHandle; count: INTEGER);
```

When the mouse button is released over an enabled menu item, the item blinks briefly to confirm the choice. Normally your application need not be concerned about the duration of the blinking, but for special situations SetMenuFlash allows you to control the duration for all items in the given menu. Calling SetMenuFlash with a count of Ø disables blinking; calling it with a count of 2 (the default value) will cause items to blink for about Ø.1 second. A count of 3 is appropriate for naive user applications. Values greater than 3 can be annoyingly slow.

Assembly-language note: The macro you invoke to call SetMenuFlash from assembly language is named _SetMFlash. The current count is stored in the system global menuFlash.

(hand)
Items in both standard and nonstandard menus blink when chosen. The appearance of the blinking for a nonstandard menu depends on the menu definition procedure, as described under "Defining Your Own Menus".

PROCEDURE CalcMenuSize (menu: MenuHandle);

You can use CalcMenuSize to recalculate the horizontal and vertical dimensions of a menu whose contents have been changed (and store them in the appropriate fields of the menu record). CalcMenuSize is called automatically after every AppendMenu, SetItem, SetItemIcon, and SetItemStyle call.

FUNCTION CountMItems (menu: MenuHandle) : INTEGER;

CountMItems returns the number of menu items in the given menu.

FUNCTION GetMHandle (menuID: INTEGER) : MenuHandle;

Given the menu ID of a menu currently installed in the menu list, GetMHandle returns a handle to that menu; given any other menu ID, it returns NIL.

PROCEDURE FlashMenuBar (menuID: INTEGER);

If menuID is Ø (or isn't the ID of any menu in the menu list), FlashMenuBar inverts the entire menu bar; otherwise, it inverts the title of the given menu.

DEFINING YOUR OWN MENUS

Normally when you create a menu you get the standard type of Macintosh menu, as described in this manual. You may, however, want to define your own type of menu, such as one with more graphics or perhaps a nonlinear text arrangement. QuickDraw and the Menu Manager make it possible for you to do this.

To define your own type of menu, you must write a menu definition procedure. The menu definition procedure defines the menu by

performing basic operations such as drawing the menu. When the Menu Manager needs to perform one of these operations, it calls the menu definition procedure with a parameter that identifies the operation, and the menu definition procedure in turn takes the appropriate action.

Usually you'll store the menu definition procedure as a resource in a resource file. If you won't be sharing it with other applications, you may want to include it with your application code instead.

When you create a menu with NewMenu, it stores a handle to the standard menu definition procedure in the menu record's menuProc field; you must replace this with a handle to your own menu definition procedure. If your definition procedure is in a resource file, you get the handle by calling the Resource Manager to read it from the resource file into memory.

Instead of creating menus with NewMenu, your application may read the menus from a resource file with GetMenu (or GetNewMBar, which calls GetMenu). A menu in a resource file contains the resource ID of its menu definition procedure. If you store the resource ID of your own menu definition procedure in a menu in a resource file, GetMenu will take care of reading the procedure into memory and storing a handle to it in the menuProc field of the menu record.

The Menu Definition Procedure

The menu definition procedure may be written in Pascal or assembly language; the only requirement is that its entry point be at the beginning. You may choose any name you wish for the procedure. Here's how you would declare one named MyMenu:

```
PROCEDURE MyMenu (message: INTEGER; menu: MenuHandle; menuRect:
                  Rect; hitPt: Point; VAR whichItem: INTEGER);
```

The message parameter identifies the operation to be performed. Its value will be one of the following predefined constants:

```
CONST mDrawMsg   = 0;   {draw the menu}
      mChooseMsg = 1;   {tell which menu item was chosen and}
                        { highlight it}
      mSizeMsg   = 2;   {calculate the menu's dimensions}
```

The menu parameter indicates the menu that the operation will affect, and menuRect is the rectangle (in global coordinates) in which the menu is located.

The message mDrawMsg tells the menu definition procedure to draw the menu inside menuRect; the grafPort will be set up properly for this. (For details on drawing, see the QuickDraw manual.) The standard menu definition procedure figures out how to draw the menu items by looking in the menu record at the data that defines them; this data is described in detail under "Formats of Resources for Menus" below. For menus of your own definition, you may set up the data defining the menu

items any way you like, or even omit it altogether (in which case all the information necessary to draw the menu would be in the menu definition procedure itself).

(eye)

Be sure that any text in the menu is drawn in the system font.

When the menu definition procedure receives the message mChooseMsg, the hitPt parameter is the point (in global coordinates) where the mouse button was pressed, and the whichItem parameter is the item number of the last item that was chosen from this menu. The procedure should test whether hitPt is inside menuRect and respond accordingly:

- If hitPt is inside menuRect, unhighlight whichItem, highlight the newly chosen item, and return the item number of that item in whichItem.

- If hitPt isn't inside menuRect, unhighlight whichItem and return Ø.

(hand)

When the Menu Manager needs to make a chosen menu item blink, it repeatedly calls the menu definition procedure with the message mChooseMsg, causing the item to be alternately highlighted and unhighlighted.

Finally, the message mSizeMsg tells the menu definition procedure to calculate the horizontal and vertical dimensions of the menu and store them in the menuWidth and menuHeight fields of the menu record.

FORMATS OF RESOURCES FOR MENUS

The resource type for a menu definition procedure is 'MDEF'. The standard menu definition procedure has a resource ID of Ø, so your own such procedures must have resource IDs other than Ø. The resource data is simply the assembled code of the procedure.

Icons in menus must be stored in a resource file under the resource type 'ICON' with resource IDs from 257 to 511. Strings in resource files have the resource type 'STR '—but note that if you follow the recommendation of storing entire menus in resource files, you'll never have to store the strings they contain separately.

The formats of menus and menu bars in resource files are given below.

Menus in a Resource File

The resource type for a menu is 'MENU'. The resource ID must be negative for menus belonging to desk accessories and positive for other menus; it should never be Ø. The resource data for a menu has the format shown below. Once read into memory, this data is stored in a menu record (described earlier in the "Menu Records" section).

Number of bytes	Contents
2 bytes	Menu ID (resource ID of this menu)
2 bytes	Ø; placeholder for menu width
2 bytes	Ø; placeholder for menu height
2 bytes	Resource ID of menu definition procedure
2 bytes	Ø (see comment below)
4 bytes	Same as enableFlags field of menu record
1 byte	Length of following title in bytes
n bytes	Characters of menu title
For each menu item:	
1 byte	Length of following text in bytes
m bytes	Text of menu item
1 byte	Icon number, or Ø if no icon
1 byte	Keyboard equivalent, or Ø if none
1 byte	Character marking menu item, or Ø if none
1 byte	Character style of item's text
1 byte	Ø, indicating end of menu items

The four bytes beginning with the resource ID of the menu definition procedure serve as a placeholder for the handle to the procedure: When GetMenu is called to read the menu from the resource file, it also reads in the menu definition procedure if necessary, and replaces these four bytes with a handle to the procedure. The resource ID of the standard menu definition procedure is:

```
CONST textMenuProc = Ø;
```

The resource data for a nonstandard menu can define menu items in any way whatsoever, or not at all, depending on the requirements of its menu definition procedure. If the appearance of the items is basically the same as the standard, the resource data might be as shown above, but in fact everything following "For each menu item" can have any desired format or can be omitted altogether. Similarly, all bits beyond the first of the enableFlags array may be set and used in any way desired by the menu definition procedure; the first bit applies to the entire menu and must reflect whether it's enabled or disabled.

If your menu definition procedure does use the enableFlags array, menus of that type may contain no more than 31 items (1 per available bit); otherwise, the number of items they may contain is limited only by the amount of room on the screen.

(hand)
See "Using the Toolbox from Assembly Language" for the exact format of the character style byte. *** (Currently

it's in "Using QuickDraw from Assembly Language" in the QuickDraw manual.) ***

(eye)
Menus in resource files must not be purgeable.

Menu Bars in a Resource File

The resource type for the contents of a menu bar is 'MBAR' and the resource data has the following format:

Number of bytes	Contents
2 bytes	Number of menus
For each menu:	
2 bytes	Resource ID of menu

SUMMARY OF THE MENU MANAGER

Constants

```
CONST noMark      = Ø;
      checkMark   = 18;   {check mark}
      applesymbol = 2Ø;   {Apple symbol}

      mDrawMsg   = Ø;   {draw the menu}
      mChooseMsg = 1;   {tell which item was chosen and highlight it}
      mSizeMsg   = 2;   {calculate the menu's dimensions}

      textMenuProc = Ø;
```

Data Structures

```
TYPE MenuPtr    = ^MenuInfo;
     MenuHandle = ^MenuPtr;
     MenuInfo   = RECORD
                    menuID:      INTEGER;
                    menuWidth:   INTEGER;
                    menuHeight:  INTEGER;
                    menuProc:    Handle;
                    enableFlags: PACKED ARRAY [Ø..31] OF BOOLEAN;
                    menuData:    Str255
                  END;
```

Routines

Initialization and Allocation

```
PROCEDURE InitMenus;
FUNCTION  NewMenu       (menuID: INTEGER; menuTitle: Str255) :
                         MenuHandle;
FUNCTION  GetMenu       (menuID: INTEGER) : MenuHandle;
PROCEDURE DisposeMenu   (menu: MenuHandle);
PROCEDURE AppendMenu    (menu: MenuHandle; data: Str255);
PROCEDURE AddResMenu    (menu: MenuHandle; theType: ResType);
PROCEDURE InsertResMenu (menu: MenuHandle; theType: ResType; afterItem:
                         INTEGER);
```

Forming the Menu Bar

```
PROCEDURE InsertMenu    (menu: MenuHandle; beforeID: INTEGER);
PROCEDURE DrawMenuBar;
PROCEDURE DeleteMenu    (menuID: INTEGER);
```

```
PROCEDURE ClearMenuBar;
FUNCTION  GetNewMBar    (menuBarID: INTEGER) : Handle;
FUNCTION  GetMenuBar :   Handle;
PROCEDURE SetMenuBar    (menuBar: Handle);
```

Choosing from a Menu

```
FUNCTION  MenuSelect (startPt: Point) : LongInt;
FUNCTION  MenuKey    (ch: CHAR) : LongInt;
PROCEDURE HiliteMenu (menuID: INTEGER);
```

Controlling Items' Appearance

```
PROCEDURE SetItem      (menu: MenuHandle; item: INTEGER; itemString:
                        Str255);
PROCEDURE GetItem      (menu: MenuHandle; item: INTEGER; VAR itemString:
                        Str255);
PROCEDURE DisableItem  (menu: MenuHandle; item: INTEGER);
PROCEDURE EnableItem   (menu: MenuHandle; item: INTEGER);
PROCEDURE CheckItem    (menu: MenuHandle; item: INTEGER; checked:
                        BOOLEAN);
PROCEDURE SetItemIcon  (menu: MenuHandle; item: INTEGER; icon: INTEGER);
PROCEDURE GetItemIcon  (menu: MenuHandle; item: INTEGER; VAR icon:
                        INTEGER);
PROCEDURE SetItemStyle (menu: MenuHandle; item: INTEGER; chStyle: Style);
PROCEDURE GetItemStyle (menu: MenuHandle; item: INTEGER; VAR chStyle:
                        Style);
PROCEDURE SetItemMark  (menu: MenuHandle; item: INTEGER; markChar: CHAR);
PROCEDURE GetItemMark  (menu: MenuHandle; item: INTEGER; VAR markChar:
                        CHAR);
```

Miscellaneous Utilities

```
PROCEDURE SetMenuFlash (menu: MenuHandle; count: INTEGER);
PROCEDURE CalcMenuSize (menu: MenuHandle);
FUNCTION  CountMItems  (menu: MenuHandle) : INTEGER;
FUNCTION  GetMHandle   (menuID: INTEGER) : MenuHandle;
PROCEDURE FlashMenuBar (menuID: INTEGER);
```

Meta-Characters for AppendMenu

Meta-character	Usage
; or Return	Separates multiple items
^	Followed by an icon number, adds that icon to the item
!	Followed by a character, marks the item with that character
<	Followed by B, I, U, O, or S, sets the character style of the item
/	Followed by a character, associates a keyboard equivalent with the item
(	Disables the item

Menu Definition Procedure

```
PROCEDURE MyMenu (message: INTEGER; menu: MenuHandle; menRect: Rect;
                  hitPt: Point; VAR whichItem: INTEGER);
```

Assembly-Language Information

Constants

```
noMark        .EQU    0
checkMark     .EQU    18      ;check mark
appleSymbol   .EQU    20      ;Apple symbol

mDrawMsg      .EQU    0       ;draw the menu
mChooseMsg    .EQU    1       ;tell which item was chosen and
                              ; highlight it
mSizeMsg      .EQU    2       ;calculate the menu's dimensions
```

Menu Record Data Structure

menuID	Menu ID
menuWidth	Menu width
menuHeight	Menu height
menuDefHandle	Handle to menu definition procedure
menuEnable	Enable flags
menuData	Menu title followed by data defining the items
menuBlkSize	Length of all the above fields except menuData

Special Macro Names

Routine name	Macro name
DisposeMenu	_DisposMenu
GetItemIcon	_GetItmIcon
GetItemMark	_GetItmMark
GetItemStyle	_GetItmStyle
GetMenu	_GetRMenu
SetItemIcon	_SetItmIcon
SetItemMark	_SetItmMark
SetItemStyle	_SetItmStyle
SetMenuFlash	_SetMFlash

System Globals

Name	Size	Contents
menuList	4 bytes	Handle to current menu list
mBarEnable	2 bytes	Nonzero if menu bar belongs to a desk accessory
menuHook	4 bytes	Hook for routine to be called during MenuSelect
theMenu	2 bytes	Menu ID of currently highlighted menu
menuFlash	2 bytes	Count for duration of menu item blinking

GLOSSARY

character style: A set of stylistic variations, such as bold, italic, and underline. The empty set indicates normal text (no stylistic variations).

dimmed: Drawn in gray rather than black.

disabled: A disabled menu item or menu is one that cannot be chosen; the menu item or menu title appears dimmed.

icon: A 32-by-32 bit image that graphically represents an object, concept, or message.

icon number: A digit from 1 to 9 to which the Menu Manager adds 256 to get the resource ID of an icon associated with a menu item.

keyboard equivalent: A way of invoking a menu item from the keyboard, by holding down the Command key and typing a character.

menu: A list of menu items that appears when the user points to and presses a menu title in the menu bar. Dragging through the menu and releasing over an enabled menu item chooses that item.

menu bar: The horizontal strip at the top of the Macintosh screen that contains the menu titles of all menus in the menu list.

menu definition procedure: A procedure called by the Menu Manager when it needs to perform basic operations on a particular type of menu, such as drawing the menu.

menu ID: For menus defined in resource files, the resource ID of the menu; for application menus, a positive number that you choose to identify the menu.

menu item: A choice in a menu, usually a command to the current application; in a standard Macintosh menu, a line containing text and possibly an icon.

menu item number: The index, starting from 1, of a menu item in a menu.

menu list: A list of menu handles for all menus in the menu bar, kept internally by the Menu Manager.

menu record: The internal representation of a menu, where the Menu Manager stores all the information it needs for its operations on that menu.

menu title: A word or phrase in the menu bar that designates one menu.

meta-character: One of the characters ; ^ ! < / (or Return appearing in the string passed to the Menu Manager routine AppendMenu, to

separate menu items or alter their appearance.

9-March-83
LAK

The OS Event Manager

The Event Manager core routines manipulate events on the system event queue. These consist of functions such as adding and retrieving events from the system event queue, polling for available events, and removing events from the queue. The system queue is initialized to contain 30 22-byte elements.

(ToolEvents contain the higher-level ToolBox event handling calls EventAvail and GetNextEvent: these will be documented separately with other ToolBox documentation, although some ToolEvents-defined events are briefly covered here. ToolEvents makes calls to OSEventAvail and GetOSEvent, adding Activate and Update events, and supports journaling. Most application programs will just make calls to ToolEvents.)

Four routines are associated with the event manager: PostEvent, OSEventAvail, GetOSEvent, and FlushEvents. PostEvent may be called from an interrupt or completion routine; all other routines in the event manager must be called from the main thread of execution. Additionally, the system event mask may be read and set via the OS routines GetSysParam and SetSysParam.

The Event Manager manages its own private buffer to get storage for the event queueing elements. It does this because PostEvent runs at interrupt level and thus cannot call the standard storage allocater.

Events

The Macintosh operating system uses the metaphor of an "event" to report to user programs the occurance of keyboard keypresses, mouse button state changes, and other relatively slow and irregular things which the system detects and the user program is interested in. Faster input/output, such as receipt of a character on one of the serial port , is handled via the "I/O driver" model in the I/O and File subsystems.

Event Mask, Event Number

Events are posted and selected subject to event masks; an event mask is a word-long bitmap of all possible events: a 1 in the bit position of an event enables that event. Possible events by event number, bit position in event mask, and name are:

0	$0001	Null Event
1	$0002	Mouse button down
2	$0004	Mouse button up
3	$0008	Key down
4	$0010	Key up
5	$0020	Auto-key
6	$0040	Update event
7	$0080	Disk Inserted

8	$0100	Activate/Deactivate event
9	$0200	Abort event
10	$0400	Network event
11	$0800	IO Driver event
12	$1000	application defined
13	$2000	application defined
14	$4000	application defined
15	$8000	application defined

Event Queue Element, Event Record

The basic data structure for events is a 22-byte buffer called an EVENT QUEUE ELEMENT, in which events are buffered by the Event Manager. Events are communicated to users via EVENT RECORDS, which are structured like event queue elements, minus the six-byte queue link and type fields. The SYSTEM EVENT BUFFER has room enough for 30 event queue elements.

Event Queue Element:

```
       (0)     Queue link to next element, zero for last element (32-bit)
       (4)     Queue type field, set to $0004 (16-bit)
       (6)     Event Record (16-byte)
```

Event Record:

```
       (0)     Event Number (16-bit)
       (2)     Event-defined message (32-bit)
       (6)     TICKS value when event occurred (32-bit) (TICKS is a 32-bit
                 variable which is incremented every 1/60 second)
      (10)     Mouse position when event occurred (32-bit)
      (14)     Meta-key flags (8-bit) as follows (bit=1 when key is down):
                 bit 7-4: undefined
                       3: option key
                       2: alpha-lock key
                       1: shift key
                       0: command
      (15)     Mouse button state (8-bit):
                 bit   7: down=0,up=1
                     6-0: undefined (toolevents uses bits 0-1 to distinguish
                              activate from deactivate, and sys-appl change).
```

Event-defined messages are as follows (including ToolEvents-defined events):

Null Event	none (0)
Mouse button down	none (0)
Mouse button up	none (0)
Key down	byte0=byte1=0,byte2=raw keycode,byte3=ASCII code
Key up	byte0=byte1=0,byte2=raw keycode,byte3=ASCII code
Auto-key	byte0=byte1=0,byte2=raw keycode,byte3=ASCII code
Disk Inserted	drive number: 1 internal, 2 external
Update event	32-bit windowPtr of window to be updated
Activate/Deactivate	32-bit windowPtr

Events are generally posted as they occur and are self-explanatory;

some notable exceptions are the null event and auto-key events.
For this discussion, events will be classified into standard events (1-4 and 7-8), auto-key events, and the null event. An event is available only when it is enabled by the user-specified event mask.

The null event is returned by OSEventAvail and GetOSEvent when no standard or auto-key events are available. Null events are always enabled (i.e., generation of null events is not subject to a mask), and they are never posted into the system event queue.

Auto-key events are posted into the event queue by OSEventAvail when there are no standard events available, there is a repeatable key down, the repeat time thresholds have been satisfied, and auto-key events are enabled by both the user-specified mask and the system event mask. The posted auto-key is returned like a standard event (and dequeued if posted by a GetOSEvent call to OSEventAvail).

The null event returns the current state of the mouse button, mouse position, keyboard meta-keys, and the current value of TICKS.

Routine: PostEvent

Arguments:
A0 (input) -- event number (16-bit)
D0 (input) -- event message (32-bit)
D0 (output) -- result: 0=event posted, 1=not posted

Function: This routine adds an element to the system event queue. The specified event number and event message are logged for the event. The current time, mouse position, state of command key, option key, shift key, alpha lock key, and mouse button are also logged. An event is only posted if enabled by the system event mask; if not enabled, a result code of 1 is returned. This routine will delete the first element of the event queue (the oldest element), if the queue is full, to make room for the new event: this guarantees that an enabled event will be posted.

Calling sequence:

```
MOVE.W  #EventNumber,A0
MOVE.L  #Message,D0
_PostEvent
```

Routine: OSEventAvail

Arguments:
A0 (input) -- pointer to user event record (32-bit)
D0 (input) -- set of events desired (event mask)
D0 (output) -- 0=non-null event returned, -1=null event returned

Function: This routine polls for availability of certain types of events.

If no events are available, the null event is returned along with a -1 result code in D0. Note that an event which is reported as available may disappear (i.e., not be accessible by a later call to GetOSEvent or OSEventAvail) in a busy environment due to the event buffer wraparound performed by PostEvent.

Calling sequence:

```
MOVE.W  #EventMask,D0
LEA     EventBuffer,A0
_OSEventAvail
```

Routine: GetOSEvent

Arguments:

```
A0 (input)   --  pointer to user event buffer (32-bit)
D0 (input)   --  type of event desired (event mask)
D0 (output)  --  0=non-null event returned, -1=null event
                 returned
```

Function: This routine returns the next event in the system event queue. The returned event is dequeued, thereby freeing up the space which holds that queue element (except for update events, which are never queued up). If no events of the types enabled by the mask are enabled, the null event is returned.

Calling sequence:

```
MOVE.W  #EventMask,D0
LEA     EventBuffer,A0
_GetOSEvent
```

Routine: FlushEvents

Arguments:

```
D0 (input)   --  low word: events to remove (event mask)
                high word: events on which to stop (event mask)
D0 (output)  --  event type of event which terminated search
```

Function: This routine removes events of type specified by the caller. On entry, D0 contains a long word of two 16-bit event masks. The low-order 16 bits contains a mask of events to remove, and the high-order 16 bits contains a mask of events that, once encountered, terminates the event removal process. D0 returns 0 if all events were deleted from the queue and, if not, the event number of the event which terminated the flush.

Calling sequence:

```
MOVE.L  #EventMasks,D0
_FlushEvents
```

MACINTOSH USER EDUCATION

Macintosh Packages: A Programmer's Guide /PACKAGES/PACK

See Also: The Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
The Memory Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Event Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
Putting Together a Macintosh Application
The Structure of a Macintosh Application

Modification History: First Draft (ROM 7) B. Hacker & C. Rose 2/29/84

ABSTRACT

Packages are sets of data structures and routines that are stored as resources and brought into memory only when needed. There's a package for presenting the standard user interface when a file is to be saved or opened, and others for doing less common operations such as floating-point arithmetic. This manual describes packages and the Package Manager, the part of the Macintosh User Interface Toolbox that provides access to packages.

Erratum:

The SFListPtr data type has been removed from the Standard File Package. The typeList parameter has the data type SFTypeList.

TABLE OF CONTENTS

3 About This Manual
4 The Package Manager
6 The Standard File Package
6 About the Standard File Package
7 Using the Standard File Package
8 Standard File Package Routines
17 The Disk Initialization Package
17 Using the Disk Initialization Package
18 Disk Initialization Package Routines
23 Summary of the Package Manager
24 Summary of the Standard File Package
26 Summary of the Disk Initialization Package
27 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes packages and the Package Manager. The Macintosh packages include one for presenting the standard user interface when a file is to be saved or opened, and others for doing less common operations such as floating-point arithmetic. The Package Manager is the part of the Macintosh User Interface Toolbox that provides access to packages. *** Eventually, this will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

You should already be familiar with the Macintosh User Interface Guidelines, Lisa Pascal, the Macintosh Operating System's Memory Manager, and the Resource Manager. Using the various packages may require that you be familiar with other parts of the Toolbox and Operating System as well.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with a discussion of the Package Manager and packages in general. This is followed by a series of sections on the individual packages *** (only two for now; more will be added) ***. You'll only need to read the sections about the packages that interest you. Each section describes the package briefly, tells how its routines fit into the flow of your application program, and then gives detailed descriptions of the package's routines.

Finally, there are summaries of the Package Manager and the individual packages, for quick reference, followed by a glossary of terms used in this manual.

THE PACKAGE MANAGER

The Package Manager is the part of the Macintosh User Interface Toolbox that enables you to access packages. Packages are sets of data structures and routines that are stored as resources and brought into memory only when needed. They serve as extensions to the Macintosh Operating System and User Interface Toolbox, for the most part performing less common operations.

The Macintosh packages, which are stored in the system resource file, include the following:

- The Standard File Package, for presenting the standard user interface when a file is to be saved or opened.
- The Disk Initialization Package, for initializing and naming new disks. This package is called by the Standard File Package; you'll only need to call it in nonstandard situations.
- *** more to be added ***

Packages have the resource type 'PACK' and the following resource IDs:

```
CONST dskInit = 2;  {Disk Initialization}
      stdFile = 3;  {Standard File}
      flPoint = 4;  {Floating-Point Arithmetic}
      trFunc  = 5;  {Transcendental Functions}
      intUtil = 6;  {International Utilities}
      bdConv  = 7;  {Binary/Decimal Conversion}
```

Assembly-language note: All macros for calling package routines expand to invoke one macro, _PackN, where N is the resource ID of the package. The package determines which routine to execute from the routine selector, an integer that's passed to it on the stack. For example, the routine selector for the Standard File Package procedure SFPutFile is 1, so invoking the macro _SFPutFile pushes 1 onto the stack and invokes _Pack3.

There are two Package Manager routines that you can call directly from Pascal: one that lets you access a specified package and one that lets you access all packages. The latter will already have been called when your application starts up, so normally you won't ever have to call the Package Manager yourself. Its procedures are described below for advanced programmers who may want to use them in unusual situations.

PROCEDURE InitPack (packID: INTEGER);

InitPack enables you to use the package specified by packID, which is the package's resource ID. (It gets a handle that will be used later to read the package into memory.)

PROCEDURE InitAllPacks;

InitAllPacks enables you to use all Macintosh packages (as though InitPack were called for each one). It will already have been called when your application starts up.

Assembly-language note: The macro you invoke to call InitAllPacks from assembly language is named _InitMath.

THE STANDARD FILE PACKAGE

The Standard File Package provides the standard user interface for specifying a file to be saved or opened. It allows the file to be on a disk in either drive (if an external drive is attached to the Macintosh), and lets a currently inserted disk be ejected so that another one can be inserted.

*** In the final, comprehensive manual, the documentation of this package will be at the end of the volume that describes the Toolbox. ***

You should already be familiar with the following:

- the basic concepts and structures behind QuickDraw, particularly points and rectangles
- the Toolbox Event Manager, the Control Manager, and the Dialog Manager
- the Package Manager and packages in general

About the Standard File Package

Standard Macintosh applications should have a File menu from which the user can save and open documents, via the Save, Save As, and Open commands. In response to these commands, the application can call the Standard File Package to find out the document name and let the user switch disks if desired. As described below, a dialog box is presented for this purpose. (More details and illustrations are given later in the descriptions of the individual routines.)

When the user chooses Save As, or Save when the document is untitled, the application needs a name for the document. The corresponding dialog box lets the user enter the document name and click a button labeled "Save" (or just click "Cancel" to abort the command). By convention, the dialog box comes up displaying the current document name, if any, so the user can edit it.

In response to an Open command, the application needs to know which document to open. The corresponding dialog box displays the names of all documents that might be opened, and the user chooses one by clicking it and then clicking a button labeled "Open". A vertical scroll bar allows scrolling through the names if there are more than can be shown at once.

Both of these dialog boxes let the user:

- insert a disk in an external drive connected to the Macintosh
- eject a disk from either drive and insert another

- initialize and name an inserted disk if it's uninitialized

- switch between the internal and external drives

On the right in the dialog box, separated from the rest of the box by a gray line, there's a disk name with one or two buttons below it; Figure 1 shows what this looks like when an external drive is connected to the Macintosh but currently has no disk in it. Notice that the Drive button is inactive (dimmed). After the user inserts a disk in the external drive (and, if necessary, initializes and names it), the Drive button becomes active. If there's no external drive, the Drive button isn't displayed at all.

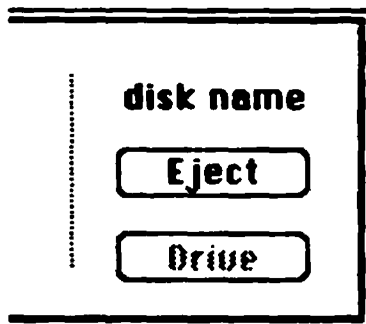


Figure 1. Partial Dialog Box

The disk name displayed in the dialog box is the name of the current disk, initially the disk in the internal drive. The user can click Eject to eject the current disk and insert another, which then becomes the current disk. If there's an external drive, clicking the Drive button changes the current disk from the one in the external drive to the one in the internal drive or vice versa. The Drive button is inactive whenever there's only one disk inserted.

If an uninitialized or otherwise unreadable disk is inserted, the Standard File Package calls the Disk Initialization Package to provide the standard user interface for initializing and naming a disk.

Using the Standard File Package

This section discusses how the routines in the Standard File Package fit into the general flow of an application program, and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

The Standard File Package and the resources it uses are automatically read into memory when one of its routines is called. It in turn reads the Disk Initialization Package into memory when needed; together they occupy about 5K bytes.

Call SFPutFile when your application is to save to a file and needs to get the name of the file from the user. Standard applications should do this when the user chooses Save As from the File menu, or Save when the document is untitled. SFPutFile displays a dialog box allowing the

user to enter a file name.

Similarly, SFGetFile is useful whenever your application is to open a file and needs to know which one, such as when the user chooses the Open command from a standard application's File menu. SFGetFile displays a dialog box with a list of file names to choose from.

You pass these routines a reply record, as shown below, and they fill it with information about the user's reply.

```
TYPE SFReply = RECORD
                 good:     BOOLEAN;     {ignore command if FALSE}
                 copy:     BOOLEAN;     {not used}
                 fType:    OSType;      {file type or not used}
                 vRefNum:  INTEGER;     {volume reference number}
                 version:  INTEGER;     {file's version number}
                 fName:    STRING[63]   {file name}
               END;
```

The first field of this record determines whether the file operation should take place or the command should be ignored (because the user clicked the Cancel button in the dialog box). The fType field is used by SFGetFile to store the file's type. The vRefNum, version, and fName fields identify the file chosen by the user; the application passes their values on to the File Manager routine that does the actual file operation. VRefNum contains the volume reference number of the volume containing the file. Currently the version field always contains 0.

Both SFPutFile and SFGetFile allow you to use a nonstandard dialog box; two additional routines, SFPPutFile and SFPGetFile, provide an even more convenient and powerful way of doing this.

Standard File Package Routines

Assembly-language note: The macros for calling the Standard File Package routines push one of the following routine selectors onto the stack and then invoke _Pack3:

Routine	Selector
SFPutFile	1
SFPPutFile	3
SFGetFile	2
SFPGetFile	4

PROCEDURE SFPutFile (where: Point; prompt: Str255; origName: Str255;
dlgHook: ProcPtr; VAR reply: SFReply);

SFPutFile displays a dialog box allowing the user to specify a file to which data will be written (as during a Save or Save As command). It then repeatedly gets and handles events until the user either confirms the command after entering an appropriate file name or aborts the command by clicking Cancel in the dialog. It reports the user's reply by filling the fields of the reply record specified by the reply parameter, as described above; the fType field of this record isn't used.

The general appearance of the standard SFPutFile dialog box is shown in Figure 2. The where parameter specifies the location of the top left corner of the dialog box in global coordinates. The prompt parameter is a line of text to be displayed as a statText item in the dialog box, where shown in Figure 2. The origName parameter contains text that appears as an enabled, selected editText item; for the standard document-saving commands, it should be the current name of the document, or the empty string (to display an insertion point) if the document hasn't been named yet.

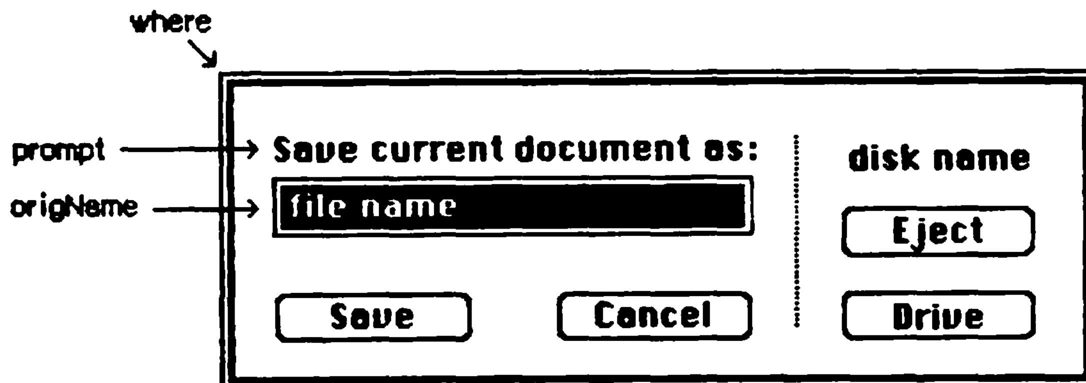


Figure 2. Standard SFPutFile Dialog

If you want to use the standard SFPutFile dialog box, pass NIL for dlgHook; otherwise, see the information for advanced programmers below.

SFPutFile repeatedly calls the Dialog Manager procedure ModalDialog. After an event involving an enabled dialog item occurs, ModalDialog returns the item number, and SFPutFile responds as follows:

- If the Eject or Drive button is clicked, or a disk is inserted, SFPutFile responds as described above under "About the Standard File Package".

- Text entered into the editText item is stored in the fName field of the reply record. (SFPutFile keeps track of whether there's currently any text in the item, and makes the Save button inactive if not.)

- If the Save button is clicked, SFPutFile determines whether the file name in the fName field of the reply record is appropriate. If so, it returns control to the application with the first field of the reply record set to TRUE; otherwise, it responds accordingly, as described below.

- If the Cancel button in the dialog is clicked, SFPutFile returns control to the application with the first field of the reply record set to FALSE.

(note)

Notice that disk insertion is one of the user actions listed above, even though ModalDialog normally ignores disk inserted events. The reason this works is that SFPutFile calls ModalDialog with a filterProc function that checks for a disk inserted event and returns a "fake", very large item number if one occurs; SFPutFile recognizes this item number as an indication that a disk was inserted.

The situations that may cause an entered name to be inappropriate, and SFPutFile's response to each, are as follows:

- If a file with the specified name already exists on the disk and is different from what was passed in the origName parameter, the alert in Figure 3 is displayed. If the user clicks Yes, the file name is appropriate.

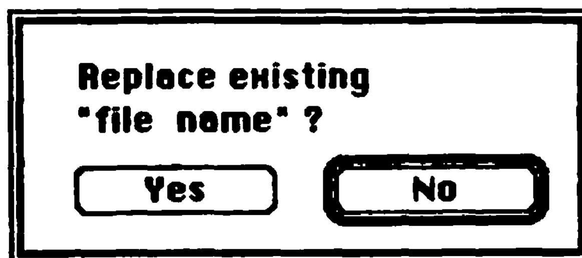


Figure 3. Alert for Existing File

- If the disk to which the file should be written is locked, the alert in Figure 4 is displayed. If a system error occurs, a similar alert is displayed, with a corresponding message explaining the problem.

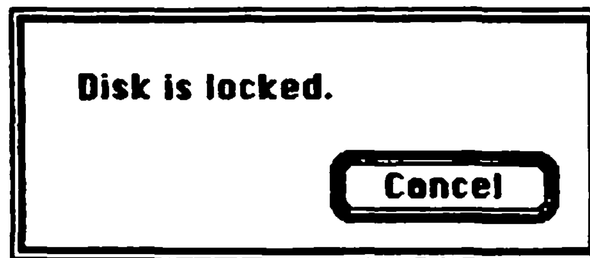


Figure 4. Alert for Locked Disk

(note)
The user may specify a disk name (preceding the file name and separated from it by a colon). If the disk isn't currently in a drive, an alert similar to the one in Figure 4 is displayed. The ability to specify a disk name is supported for historical reasons only; users should not be encouraged to do it.

After the user clicks No or Cancel in response to one of these alerts, SFPutFile dismisses the alert box and continues handling events (so a different name may be entered).

Advanced programmers: You can create your own dialog box rather than use the standard SFPutFile dialog. To do this, you must provide your own dialog template and store it in your application's resource file with the same resource ID that the standard template has in the system resource file:

```
CONST putDlgID = -3999;  {SFPutFile dialog template ID}
```

(note)
The SFPPutFile procedure, described below, lets you use any resource ID for your nonstandard dialog box.

Your dialog template must specify that the dialog window be invisible, and your dialog must contain all the standard items, as listed below. The appearance and location of these items in your dialog may be different. You can make an item "invisible" by giving it a display rectangle that's off the screen. The display rectangle for each item in the standard dialog box is given below. The rectangle for the standard dialog box itself is (Ø, Ø, 3Ø4, 1Ø4).

Item number	Item	Standard display rectangle
1	Save button	(12, 74, 82, 92)
2	Cancel button	(114, 74, 184, 92)
3	Prompt string (statText)	(12, 12, 184, 28)
4	UserItem for disk name	(209, 16, 295, 34)
5	Eject button	(217, 43, 287, 61)
6	Drive button	(217, 74, 287, 92)
7	EditText item for file name	(14, 34, 182, 50)
8	UserItem for gray line	(200, 16, 201, 88)

If your dialog has additional items beyond the the standard ones, or if you want to handle any of the standard items in a nonstandard manner, you must write your own dlgHook function and point to it with dlgHook. Your dlgHook function should have two parameters and return an integer value. For example, this is how it would be declared if it were named MyDlg:

```
FUNCTION MyDlg (item: INTEGER; dialog: DialogPtr) : INTEGER;
```

SFPutFile passes information about every event in an enabled dialog item to your dlgHook function (which is called after ModalDialog but before SFPutFile responds to events). In the dialog parameter it passes a pointer to the dialog record describing your dialog box, and in the item parameter it passes the item number of the item. Using these two parameters, your dlgHook function should determine how to handle the event. There are predefined constants for the item numbers of standard enabled items, as follows:

```
CONST putSave   = 1;  {Save button}
      putCancel = 2;  {Cancel button}
      putEject  = 5;  {Eject button}
      putDrive  = 6;  {Drive button}
      putName   = 7;  {editText item for file name}
```

After handling the event (or, perhaps, after ignoring it) the dlgHook function must return an item number to SFPutFile. If the item number corresponds to one of the standard items, SFPutFile responds as described above; otherwise, it does nothing.

```
PROCEDURE SFPPutFile (where: Point; prompt: Str255; origName: Str255;
          dlgHook: ProcPtr; VAR reply: SFReply; dlgID: INTEGER;
          filterProc: ProcPtr);
```

SFPPutFile is an alternative to SFPutFile for advanced programmers who want to use a nonstandard dialog box. It's the same as SFPutFile except for the two additional parameters dlgID and filterProc.

DlgID is the resource ID of the dialog template to be used instead of the standard one (so you can use whatever ID you wish rather than the same one as the standard).

The filterProc parameter determines how ModalDialog will filter events when called by SFPPutFile. If filterProc is NIL, ModalDialog does the

standard filtering that it does when called by SFPutFile; otherwise, filterProc should point to a function for ModalDialog to execute after doing the standard filtering. The function must be the same as one you'd pass directly to ModalDialog in its filterProc parameter.

```
PROCEDURE SFGetFile (where: Point; prompt: Str255; fileFilter: ProcPtr;
            numTypes: INTEGER; typeList: SFListPtr; dlgHook: ProcPtr;
            VAR reply: SFReply);
```

SFGetFile displays a dialog box listing the names of a specific group of files from which the user can select one to be opened (as during an Open command). It then repeatedly gets and handles events until the user either confirms the command after choosing a file name or aborts the command by clicking Cancel in the dialog. It reports the user's reply by filling the fields of the reply record specified by the reply parameter, as described above under "Using the Standard File Package".

The general appearance of the standard SFGetFile dialog box is shown in Figure 5. In this case there are more file names than can be displayed at one time, so the scroll bar is active; if all the names can be displayed at once, the scroll bar is inactive (there's no scroll box and the gray area is white).

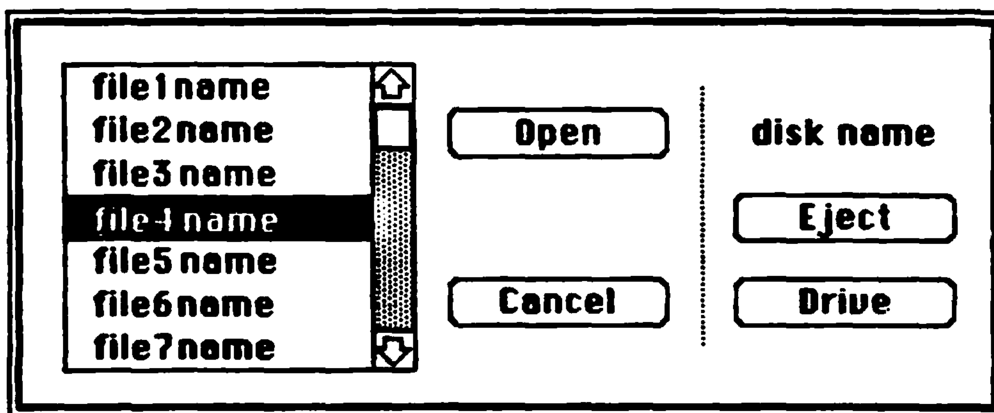


Figure 5. Standard SFGetFile Dialog

The where parameter specifies the location of the top left corner of the dialog box in global coordinates. The prompt parameter is ignored; it's there for historical purposes only.

The fileFilter, numTypes, and typeList parameters determine which files appear in the dialog box. SFGetFile first looks at numTypes and typeList to determine what types of files to display, then it executes the function pointed to by fileFilter (if any) to do additional filtering on which files to display. File types are discussed in the manual The Structure of a Macintosh Application. For example, if the application is concerned only with pictures, you won't want to display the names of any text files.

Pass -1 for numTypes to display all types of files; otherwise, pass the number of file types you want to display, and pass the types themselves in typeList. The SFListPtr data type is defined as follows:

```
TYPE SFListPtr  = ^SFTypeList;
     SFTypeList = ARRAY [0..3] OF OSType;
```

(note)
This array is declared for a reasonable maximum number of types (four). If you need to specify more than four types, declare your own array type with the desired number of entries (and use the @ operator to pass a pointer to it).

If fileFilter isn't NIL, SFGetFile executes the function it points to for each file, to determine whether the file should be displayed. The fileFilter function has one parameter and returns a Boolean value. For example:

```
FUNCTION MyFileFilter (paramBlock: ParmBlkPtr) : BOOLEAN;
```

SFGetFile passes this function the file information it gets by calling the File Manager procedure PBGetFInfo (see the *** forthcoming *** File Manager manual for details). The function selects which files should appear in the dialog by returning FALSE for every file that should be shown and TRUE for every file that shouldn't be shown.

If you want to use the standard SFGetFile dialog box, pass NIL for dlgHook; otherwise, see the information for advanced programmers below.

Like SFPutFile, SFGetFile repeatedly calls the Dialog Manager procedure ModalDialog. After an event involving an enabled dialog item occurs, ModalDialog returns the item number, and SFGetFile responds as follows:

- If the Eject or Drive button is clicked, or a disk is inserted, SFGetFile responds as described above under "About the Standard File Package".

- If clicking or dragging occurs in the scroll bar, the contents of the dialog box are redrawn accordingly.

- If a file name is clicked, it's stored in the fName field of the reply record. (SFGetFile keeps track of whether a file name is currently selected, and makes the Open button inactive if not.)

- If the Open button is clicked, SFGetFile returns control to the application with the first field of the reply record set to TRUE.

- If a file name is double-clicked, SFGetFile responds as if the user clicked the file name and then the Open button.

- If the Cancel button in the dialog is clicked, SFGetFile returns control to the application with the first field of the reply record set to FALSE.

Advanced programmers: You can create your own dialog box rather than use the standard SFGetFile dialog. To do this, you must provide your own dialog template and store it in your application's resource file with the same resource ID that the standard template has in the system resource file:

```
CONST getDlgID = -4000;  {SFGetFile dialog template ID}
```

(note)
The SFPGetFile procedure, described below, lets you use any resource ID for your nonstandard dialog box.

Your dialog template must specify that the dialog window be invisible, and your dialog must contain all the standard items, as listed below. The appearance and location of these items in your dialog may be different. You can make an item "invisible" by giving it a display rectangle that's off the screen. The display rectangle for each in the standard dialog box is given below. The rectangle for the standard dialog box itself is (0, 0, 348, 136).

Item number	Item	Standard display rectangle
1	Open button	(152, 28, 232, 46)
2	Invisible button	(1152, 59, 1232, 77)
3	Cancel button	(152, 90, 232, 108)
4	UserItem for disk name	(248, 28, 344, 46)
5	Eject button	(256, 59, 336, 77)
6	Drive button	(256, 90, 336, 108)
7	UserItem for file name list	(12, 11, 125, 125)
8	UserItem for scroll bar	(124, 11, 140, 125)
9	UserItem for gray line	(244, 20, 245, 116)
10	Invisible text (statText)	(1044, 20, 1145, 116)

If your dialog has additional items beyond the the standard ones, or if you want to handle any of the standard items in a nonstandard manner, you must write your own dlgHook function and point to it with dlgHook. Your dlgHook function should have two parameters and return an integer value. For example, this is how it would be declared if it were named MyDlg:

```
FUNCTION MyDlg (item: INTEGER; dialog: DialogPtr) : INTEGER;
```

SFGetFile passes information about every event in an enabled dialog item to your dlgHook function (which is called after ModalDialog but before SFGetFile responds to events). In the dialog parameter it passes a pointer to the dialog record describing your dialog box, and in the item parameter it passes the item number of the item. Using these two parameters, your dlgHook function should determine how to handle the event. There are predefined constants for the item numbers of standard enabled items, as follows:

```
CONST getOpen   = 1;  {Open button}
      getCancel = 3;  {Cancel button}
      getEject  = 5;  {Eject button}
      getDrive  = 6;  {Drive button}
      getNmList = 7;  {userItem for file name list}
      getScroll = 8;  {userItem for scroll bar}
```

After handling the event (or, perhaps, after ignoring it) your dlgHook function must return an item number to SFGetFile. If the item number corresponds to one of the standard items, SFGetFile responds as described above; otherwise, it does nothing.

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255; fileFilter:
            ProcPtr; numTypes: INTEGER; typeList: SFListPtr; dlgHook:
            ProcPtr; VAR reply: SFReply; dlgID: INTEGER; filterProc:
            ProcPtr);
```

SFPGetFile is an alternative to SFGetFile for advanced programmers who want to use a nonstandard dialog box. It's the same as SFGetFile except for the two additional parameters dlgID and filterProc.

DlgID is the resource ID of the dialog template to be used instead of the standard one (so you can use whatever ID you wish rather than the same one as the standard).

The filterProc parameter determines how ModalDialog will filter events when called by SFPGetFile. If filterProc is NIL, ModalDialog does the standard filtering that it does when called by SFGetFile; otherwise, filterProc should point to a function for ModalDialog to execute after doing the standard filtering. The function must be the same as one you'd pass directly to ModalDialog in its filterProc parameter.

THE DISK INITIALIZATION PACKAGE

The Disk Initialization Package provides routines for initializing disks to be accessed with the Macintosh Operating System's File Manager and Disk Driver. A single routine lets you easily present the standard user interface for initializing and naming a disk; the Standard File Package calls this routine when the user inserts an uninitialized disk. You can also use the Disk Initialization Package to perform each of the three steps of initializing a disk separately if desired.

*** In the final, comprehensive manual, the documentation of this package will be at the end of the volume that describes the Operating System. ***

You should already be familiar with the following:

- the basic concepts and structures behind QuickDraw, particularly points

- the Toolbox Event Manager

- the File Manager *** up-to-date documentation about the File Manager isn't yet available, but see the File System Core Routines and File and I/O Core Routines sections of the Macintosh OS Reference Manual ***

- the Package Manager and packages in general

Using the Disk Initialization Package

This section discusses how the routines in the Disk Initialization package fit into the general flow of an application program, and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

The Disk Initialization Package and the resources it uses are automatically read into memory from the system resource file when one of the routines in the package is called. If the disk containing the system resource file isn't currently in a Macintosh disk drive, the user will be asked to switch disks and so may have to remove the one to be initialized. To avoid this, you can use the DILoad procedure, which explicitly reads the necessary resources into memory and makes them unpurgeable. You would need to call DILoad before explicitly ejecting the system disk or before any situations where it may be switched with another disk (except for situations handled by the Standard File Package, which calls DILoad itself).

(note)

> The resources used by the Disk Initialization Package consist of a single dialog and its associated items, even though the package may present what seem to be a number of different dialogs. A special technique was used to

allow the single dialog to contain all possible dialog items with only some of them visible at one time. *** This technique will be documented in the next revision of the Dialog Manager manual. ***

When you no longer need to have the Disk Initialization Package in memory, call DIUnload. The Standard File Package calls DIUnload before returning.

When a disk inserted event occurs, the system attempts to mount the volume (by calling the File Manager function MountVol) and returns MountVol's result code in the high-order word of the event message. In response to such an event, your application can examine the result code in the event message and call DIBadMount if an error occurred (that is, if the volume could not be mounted). If the error is one that can be corrected by initializing the disk, DIBadMount presents the standard user interface for initializing and naming the disk, and then mounts the volume itself. For other errors, it justs ejects the disk; these errors are rare, and may reflect a problem in your program.

(note)

Disk inserted events during standard file saving and opening are handled by the Standard File Package. You'll call DIBadMount only in other, less common situations (for example, if your program explicitly ejects disks, or if you want to respond to the user's inserting an uninitialized disk when not expected).

Disk initialization consists of three steps, each of which can be performed separately by the functions DIFormat, DIVerify, and DIZero. Normally you won't call these in a standard application, but they may be useful in special utility programs that have a nonstandard interface.

Disk Initialization Package Routines

Assembly-language note: The macros for calling the Disk Initialization Package routines push one of the following routine selectors onto the stack and then invoke _Pack2:

Routine	Selector
DILoad	2
DIUnload	4
DIBadMount	0
DIFormat	6
DIVerify	8
DIZero	10

PROCEDURE DILoad;

DILoad reads the Disk Initialization Package, and its associated dialog and dialog items, from the system resource file into memory and makes them unpurgeable.

(note)

DIFormat, DIVerify, and DIZero don't need the dialog, so if you use only these routines you can call the Resource Manager function GetResource to read just the package resource into memory (and the Memory Manager procedure HNoPurge to make it unpurgeable).

PROCEDURE DIUnload;

DIUnload makes the Disk Initialization Package (and its associated dialog and dialog items) purgeable.

FUNCTION DIBadMount (where: Point; evtMessage: LongInt) : INTEGER;

Call DIBadMount when a disk inserted event occurs if the result code in the high-order word of the associated event message indicates an error (that is, the result code is other than noErr). Given the event message in evtMessage, DIBadMount evaluates the result code and either ejects the disk or lets the user initialize and name it. The low-order word of the event message contains the drive number. The where parameter specifies the location (in global coordinates) of the top left corner of the dialog box displayed by DIBadMount.

If the result code passed is extFSErr, mFulErr, nsDrvErr, paramErr, or volOnLinErr, DIBadMount simply ejects the disk from the drive and returns the result code. If the result code ioErr, badMDBErr, or noMacDskErr is passed, the error can be corrected by initializing the disk; DIBadMount displays a dialog box that describes the problem and asks whether the user wants to initialize the disk. For the result code ioErr, the dialog box shown in Figure 6 is displayed. (This happens if the disk is brand new.) For badMDBErr and noMacDskErr, DIBadMount displays a similar dialog box in which the description of the problem is "This disk is damaged" and "This is not a Macintosh disk", respectively.

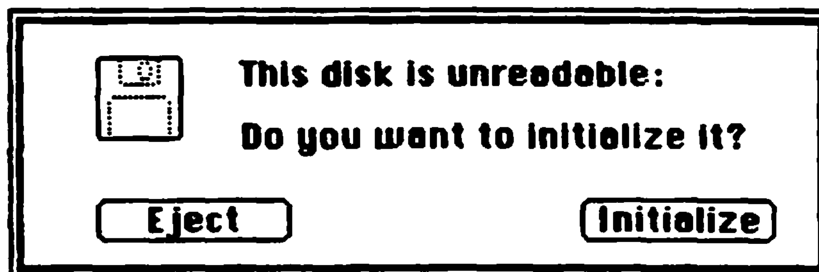


Figure 6. Disk Initialization Dialog for IOErr

(note)
Before presenting the disk initialization dialog, DIBadMount checks whether the drive contains an already mounted volume; if so, it ejects the disk and returns 2 as its result. This will happen rarely and may reflect an error in your program (for example, you forgot to call DILoad and the user had to switch to the disk containing the system resource file).

If the user responds to the disk initialization dialog by clicking the Eject button, DIBadMount ejects the disk and returns 1 as its result. If the Initialize button is clicked, a box displaying the message "Initializing disk..." appears, and DIBadMount attempts to initialize the disk. If initialization fails, the disk is ejected and the user is informed as shown in Figure 7; after the user clicks OK, DIBadMount returns a negative result code ranging from firstDskErr to lastDskErr, indicating that a low-level disk error occurred.

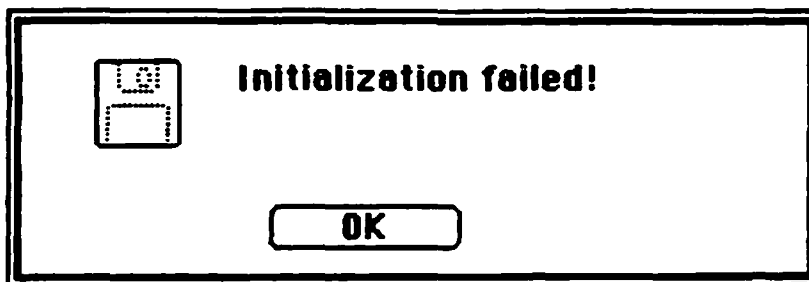


Figure 7. Initialization Failure Dialog

If the disk is successfully initialized, the dialog box in Figure 8 appears. After the user names the disk and clicks OK, DIBadMount mounts the volume by calling the File Manager function MountVol and returns MountVol's result code (noErr if no error occurs).

Figure 8. Dialog for Naming a Disk

Result codes		
	noErr	No error
	extFSErr	External file system
	mFulErr	Memory full
	nsDrvErr	No such drive
	paramErr	Bad drive number
	volOnLinErr	Volume already on-line
	firstDskErr through lastDskErr	Low-level disk error
Other results	1	User clicked Eject
	2	Mounted volume in drive

FUNCTION DIFormat (drvNum: INTEGER) : OSErr;

DIFormat formats the disk in the drive specified by the given drive number and returns a result code indicating whether the formatting was completed successfully or failed. Formatting a disk consists of writing special information onto it so that the Disk Driver can read from and write to the disk.

Result codes		
	noErr	No error
	firstDskErr through lastDskErr	Low-level disk error

FUNCTION DIVerify (drvNum: INTEGER) : OSErr;

DIVerify verifies the format of the disk in the drive specified by the given drive number; it reads each bit from the disk and returns a result code indicating whether all bits were read successfully or not.

Result codes		
	noErr	No error
	firstDskErr through lastDskErr	Low-level disk error

FUNCTION DIZero (drvNum: INTEGER; volName: Str255) : OSErr;

On the unmounted volume in the drive specified by the given drive number, DIZero writes the volume information, a block map, and a file directory as for a volume with no files; the volName parameter specifies the volume name to be included in the volume information. This is the last step in initialization (after formatting and verifying) and makes any files that are already on the volume permanently inaccessible. If the operation fails, DIZero returns a result code indicating that a low-level disk error occurred; otherwise, it mounts the volume by calling the File Manager function MountVol and returns MountVol's result code (noErr if no error occurs).

Result codes		
	noErr	No error
	badMDBErr	Bad master directory block
	extFSErr	External file system
	ioErr	Disk I/O error
	mFulErr	Memory full
	noMacDskErr	Not a Macintosh volume
	nsDrvErr	No such drive
	paramErr	Bad drive number
	volOnLinErr	Volume already on-line
	firstDskErr through lastDskErr	Low-level disk error

SUMMARY OF THE PACKAGE MANAGER

Constants

```
CONST dskInit = 2;  {Disk Initialization}
      stdFile = 3;  {Standard File}
      flPoint = 4;  {Floating-Point Arithmetic}
      trFunc  = 5;  {Transcendental Functions}
      intUtil = 6;  {International Utilities}
      bdConv  = 7;  {Binary/Decimal Conversion}
```

Routines

```
PROCEDURE InitPack (packNumber: INTEGER);
PROCEDURE InitAllPacks;
```

Assembly-Language Information

Constants

```
dskInit    .EQU    2   ;Disk Initialization
stdFile    .EQU    3   ;Standard File
flPoint    .EQU    4   ;Floating-Point Arithmetic
trFunc     .EQU    5   ;Transcendental Functions
intUtil    .EQU    6   ;International Utilities
bdConv     .EQU    7   ;Binary/Decimal Conversion
```

Special Macro Names

Routine name	Macro name
InitAllPacks	_InitMath

SUMMARY OF THE STANDARD FILE PACKAGE

Constants

```
CONST = putDlgID = -3999;  {SFPutFile dialog template ID}

        putSave   = 1;  {Save button}
        putCancel = 2;  {Cancel button}
        putEject  = 5;  {Eject button}
        putDrive  = 6;  {Drive button}
        putName   = 7;  {editText item for file name}

        getDlgID = -4000;  {SFGetFile dialog template ID}

        getOpen   = 1;  {Open button}
        getCancel = 3;  {Cancel button}
        getEject  = 5;  {Eject button}
        getDrive  = 6;  {Drive button}
        getNmList = 7;  {userItem for file name list}
        getScroll = 8;  {userItem for scroll bar}
```

Data Structures

```
TYPE SFReply    = RECORD
                    good:    BOOLEAN;     {ignore command if FALSE}
                    copy:    BOOLEAN;     {not used}
                    fType:   OSType;      {file type or not used}
                    vRefNum: INTEGER;     {volume reference number}
                    version: INTEGER;     {file's version number}
                    fName:   STRING[63]   {file name}
                  END;

     SFListPtr  = ^SFTypeList;
     SFTypeList = ARRAY [0..3] OF OSType;
```

Routines

```
PROCEDURE SFPutFile  (where: Point; prompt: Str255; origName: Str255;
                      dlgHook: ProcPtr; VAR reply: SFReply);
PROCEDURE SFPPutFile (where: Point; prompt: Str255; origName: Str255;
                      dlgHook: ProcPtr; VAR reply: SFReply; dlgID:
                      INTEGER; filterProc: ProcPtr);
PROCEDURE SFGetFile  (where: Point; prompt: Str255; fileFilter:
                      ProcPtr; numTypes: INTEGER; typeList: SFListPtr;
                      dlgHook: ProcPtr; VAR reply: SFReply);
PROCEDURE SFPGetFile (where: Point; prompt: Str255; fileFilter:
                      ProcPtr; numTypes: INTEGER; typeList: SFListPtr;
                      dlgHook: ProcPtr; VAR reply: SFReply; dlgID:
                      INTEGER; filterProc: ProcPtr);
```

DlgHook Function

```
FUNCTION MyDlg (item: INTEGER; dialog: DialogPtr) : INTEGER;
```

FileFilter Function

```
FUNCTION MyFileFilter (paramBlock: ParmBlkPtr) : BOOLEAN;
```

Assembly-Language Information

Constants

```
putDlgID     .EQU    -3999   ;SFPutFile dialog template ID
putSave      .EQU    1       ;Save button
putCancel    .EQU    2       ;Cancel button
putEject     .EQU    5       ;Eject button
putDrive     .EQU    6       ;Drive button
putName      .EQU    7       ;editText item for file name

getDlgID     .EQU    -4000   ;SFGetFile dialog template ID
getOpen      .EQU    1       ;Open button
getCancel    .EQU    3       ;Cancel button
getEject     .EQU    5       ;Eject button
getDrive     .EQU    6       ;Drive button
getNmList    .EQU    7       ;userItem for file name list
getScroll    .EQU    8       ;userItem for scroll bar
```

Reply Record Data Structure

rGood	Ignore command if FALSE
rType	File type
rVolume	Volume reference number
rVersion	File's version number
rName	File name

Routine Selectors

Routine	Selector
SFPutFile	1
SFPPutFile	3
SFGetFile	2
SFPGetFile	4

SUMMARY OF THE DISK INITIALIZATION PACKAGE

Routines

```
PROCEDURE DILoad;
PROCEDURE DIUnload;
FUNCTION  DIBadMount (where: Point; evtMessage: LongInt) : INTEGER;
FUNCTION  DIFormat   (drvNum: INTEGER) : INTEGER;
FUNCTION  DIVerify   (drvNum: INTEGER) : INTEGER;
FUNCTION  DIZero     (drvNum: INTEGER; volName: Str255) : INTEGER;
```

Assembly-Language Information

Routine Selectors

Routine	Selector
DILoad	2
DIUnload	4
DIBadMount	Ø
DIFormat	6
DIVerify	8
DIZero	1Ø

Result Codes

Name	Value	Meaning
badMDBErr	-6Ø	Bad master directory block
extFSErr	-58	External file system
firstDskErr	-84	First of the range of low-level disk errors
ioErr	-36	Disk I/O error
lastDskErr	-64	Last of the range of low-level disk errors
mFulErr	-41	Memory full
noErr	Ø	No error
noMacDskErr	-57	Not a Macintosh disk
nsDrvErr	-56	No such drive
paramErr	-5Ø	Bad drive number
volOnLinErr	-55	Volume already on-line

GLOSSARY

package: A set of data structures and routines that's stored as a resource and brought into memory only when needed.

routine selector: An integer that's pushed onto the stack before the _PackN macro is invoked, to identify which routine to execute. (N is the resource ID of a package; all macros for calling routines in the package expand to invoke _PackN.)

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

ALAIN ROSSMANN
Ext 4088

November 10, 1983
Revised December 12, 1983
Revised January 30, 1984

Changes : Added tips on word order and length.
Updated definition of INTL0 and INTL1

This note details all the things to keep in mind when writing an application to make it easily localizable. It assumes a certain familiarity with the technical documentation of Macintosh. Some of the features mentioned here may not be fully documented at this time in the technical documentation.

DRAFT

SOFTWARE LOCALIZATION GUIDELINES

The time of global markets has come. Why not increase the sales of your program by selling it into international markets ? These markets are developing very fast and are expected to grow at a higher rate than the US market.

To reach these markets your product needs to be fully localized. It can achieve a good penetration in international markets only if its users can understand it : having localized products is the key to success in international markets.

Macintosh unique localization technology allows you to sharply reduce the cost of localizing your product to foreign markets. Macintosh provides you with the ideal set of tools to create international products.

In order to get the maximum benefit from these tools, your program must be conceived from the ground up with international markets in mind. In particular, your program must make systematic use of Resources for Menus, Dialogs. Alerts, and formats.

The tools that Macintosh provides you are : predefined Resources for Menu Bars and Menu Items, Dialog Boxes, Alert Boxes, formats (number, currency, time, date), resource editor, keyboard editor, software packages for compares,

time and date display, number and currency display, number input.

GENERAL DESIGN TIPS

Macintosh is a graphically oriented machine. The use of icons greatly enhances and simplifies the interaction with the user. From an international standpoint, it also simplifies the localization process. Icons are international, they don't have to be translated.

Use icons as much as possible. A good example is MacPaint where most of the commands are accessible by clicking on an Icon. Macintosh provides you with an Icon editor to create your own Icons, which can then be stored in your application resource file.

MENU ITEMS, MENU BARS, DIALOG BOXES, ALERT BOXES

Translating software can be a difficult and expensive process: the translator has to "dive" into the code to translate a program and do the layout changes required. Macintosh totally solves this problem by using resources.

TOOLS

Resources enable you to save most of the country dependent features of your program into a separate entity. This resource can then be easily modified through the resource editor. The resource editor is not only a very powerful development tool, but also the most flexible localization tool.

With the Resource Editor any non technical person can access the Dialogs, Alerts, and Menus of your program and modify them on the screen, in real time. The Resource Editor is completely graphically oriented, for example dialog zones can be grown by selecting them, clicking in their grow box, and draging the box. There are no coordinates to compute, no counting of dots. ** Will be available early 84, alpha versions for dialogs exist**

SOFTWARE DESIGN

Menu Items, Menu Bars, Dialogs, and Alert Boxes should always be put into your application resource file. This will allow a non technical person to translate, resize, and

change the layout of dialog boxes, without knowing anything about the actual code of your application. Text in Dialogs, Alerts, and Menus can be edited the same way any text is edited on Macintosh. Translation simply consists in selecting the text, replacing it by the translated text, changing the layout to fit the size of the new text. Upon exiting of the resource editor, the localized program is immediately functional, no recompilation is necessary.

Your application should never rely on the length or position of strings in Menus or Dialogs. Different languages will have different word length and different word order. If your program is dependent on some string length or string order it won't work properly when translated.

Using only standard resource types will make your program easier to translate. Only standard resource types are supported by the Resource Editor.

LOCALIZATION

If the localization is performed by a third party, you don't have to give them your source code. This helps you keep complete control over your product. The resource editor runs on a Macintosh: no other equipment is needed for the localization phase of the program. Once a dictionary of common terms has been built, most programs can be fully translated in less than an hour by a non technical person.

CHARACTER SET AND KEYBOARDS

Foreign countries use characters that do not appear in the standard ASCII character set. This can lead to lots of problems when selling products in International markets.

TOOLS

On Macintosh, there is only one character set for all the countries which use Latin characters. This character set is common to Macintosh and Lisa. It contains all the characters needed for the major countries plus special characters and mathematical symbols (See Appendix 1 for the character set).

The character set defines the one byte codes that internally represent each character. Thus all the Macintoshes in all the countries use the same internal code for all the characters. Note that all the bits are used in the one byte code that defines a character.

Fonts contain the bit pattern that defines the shape of a specific character on the screen. Fonts are contained in a special type of resource. Some fonts may not include bit patterns for all the characters **the foreign characters may not be always there**. The system font will always have all the characters.

If you feel strongly that your application must have complete fonts, it can have its private fonts.

The only thing that differs from country to country is the way characters are generated. Each country has its own keyboard (See Appendix 2 for keyboard layouts). The whole character set can be generated from any keyboard. The only difference is that the keys to type in order to generate a certain character may not be the same in different countries.

The option key is used to access the characters which are not shown on the keycaps. Accents are not characters by themselves. They are generated by pressing dead keys. When a dead key is pressed it does not generated any character. If the accute accent is typed, for example, nothing happens until the next character is typed. This character will be accented with an accute accent.

Please note that keyboards have no way of identifying themselves. Only the keyboard mapping is changed to go from one keyboard to another.

The keyboard desk accessory allows you to look at the option keyboard. It also enables you to remap your keyboard, that is to redefine what character is generated when a certain combination of keys is pressed ** remapping is not yet implemented**.

SOFTWARE DESIGN

Do not use the 8th bit in the ASCII code to store information, it is used in our extended character set. Your program should not access the keyboard directly. As long as it uses the toolbox routines to do so, foreign keyboards will be transparent to your application.

Some menu commands may have keyboard equivalents in your application. Be sure to put these commands in the Menu Item definition Resource so that it can be easily edited through the Resource Editor.

LOCALIZATION

It is easy to overlook keyboard equivalents during the translation process. Be sure to explain how they have been chosen to the person who will translate your program.

ROM COMPARE

Having accented characters in the character set poses specific problems when comparing strings. There is a compare routine in ROM to handle compares.

TOOLS

The routine comes in four flavors determined by two boolean flags. The first flag is "ignore case", the second is " ignore diacritical marks".

If the "ignore case" flag is set, the comparison will be true regardless of the cases of the two characters compared. Likewise, if the "ignore diacritical" flag is set, the compare will be true regardless of the accentuation of the two characters being compared. If both flags are reset, the compare is an ASCII compare, if both flags are set it matches characters regardless of their cases and accentuation.

SOFTWARE DESIGN

Be sure to use the right kind of compare although it may not make a difference in your language. For example a simple Word processor may provide an "ignore everything" compare although there are no diacritical marks in your language.

FORMATS

Different countries use different formats for numbers, currencies, time, date, measure units. They also have different ways of sorting lists. This can lead to lots of problems when an application is ported to another country. Macintosh gives a very elegant,yet powerful solution to this problem through the use of the resources INTL0 and INTL1.

TOOLS

These resources contain information concerning number format, currency format, date format, time format, use of metric or English format, sorting. This information is stored in two predefined resource types : INTL0 and INTL1. (See Appendix 4 for the map of INTL0 and INTL1).

INTL1 contains the information needed to display expanded dates and to sort. You can save space by omitting INTL1 when expanded dates and sorting are not needed.

A set of routine which allow you to add INTL0 and INTL1 to your program are provided to make it easier to implement INTL0 or INTL1. They may be completed by a set of access routines to allow Pascal programs to easily access the information in INTL0 and INTL1 **May not be written**.

SOFTWARE DESIGN

Each time your application uses anything related to these items , it must either call the appropriate routine provided in our software packages or directly look into the resources to get the necessary information.

If you use our packages, you don't need to know the detailed structure of INTL0 and INTL1. We provide packages which cover : number and currency display, time and date display, magnitude compare for sorting, number input. Number and currency input and output are included in the arithmetic package **May not be written**.

Be careful to look into INTL0 to get the number separator if you don't use our number input package. The number separator should also be used for list of numbers as they may appear in a function having multiple arguments.

As for any other resource, INTL0 and INTL1 can live at any of these three levels : in the System Resource File, in the Application Resource File, and in the Document Resource File. Resources Files will automatically be searched for INTL0 or INTL1 in the following order : Document Resource File, Application Resource File, System Resource Files.

At the developer's choice there can be an INTL0 or INTL1 resource in the application resource or in the document resource. There will always be a copy of INTL0 and INTL1 in the system resource file so that it can be used as a default if your application does not have its private version of INTL0 and INTL1.

Having your own version of INTL0 or INTL1 in your application's Resource File, allows your application to remember its formats independently from the disk it is moved to. If documents have their own version of INTL0 or INTL1,

they will keep the same format even if they are used under a foreign version of the application.

A calendar for example may not need to have its private version of INTL0 or INTL1, It is sufficient for this type of application to look into the System Resource to get the necessary format information.

A program using dates may want to display dates according to the language used in its Dialogs. To ac eve that it must have an INTL0 in its Resource File, so that the date formats are linked to the application and not to the disk.

A Spreadsheet where numbers can be displayed as currencies would put INTL0 with each document. Data integrity has to be preserved : it is not acceptable to have amounts labeled in Dollars suddenly displayed as Francs because a different version of the application is used. There must be an INTL0 in each document resource file. Upon creation of the document, the default formats can be read from the application INTL0.

If the structure of your documents depends on the sorting sequence, INTL1 must be included in the document resource file. An example of that is a binary tree where part of the structure information is in the file itself and the rest in the magnitude compare.

LOCALIZATION

INTL0 and INTL1 (except for sorting) are easily modified through the resource editor. This allows user to customize their formats, provided that your program makes systematic use of these resources.

It allows you to produce masters for each country that have the proper formats on them (See Appendix 3 for the most common formats in each country).

Of course if you feel that users of your program must be able to modify these formats "on the fly" , you can include a routine that directly modifies INTL0 or INTL1 from within your program. An example of that would be for a wordprocessor to offer the capability to switch from English to Metric rulers from within the application.

The Character Set

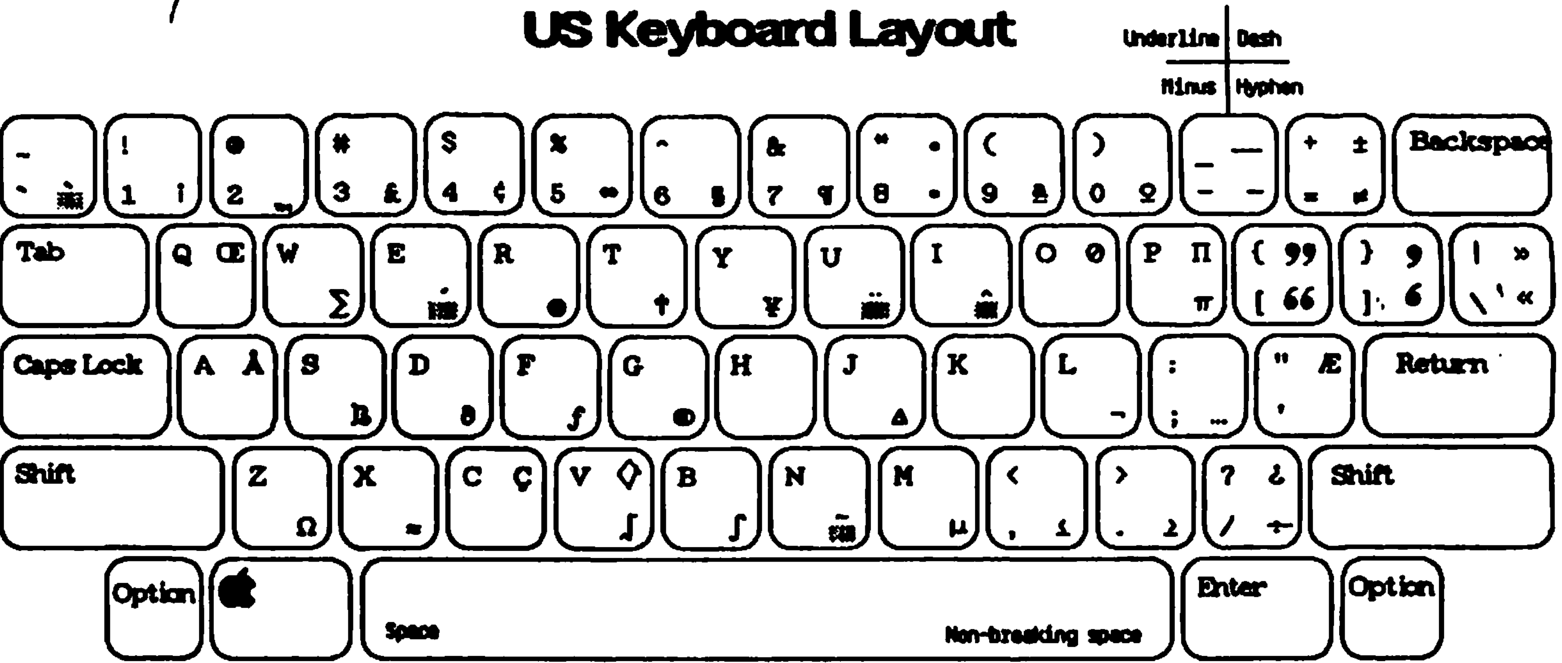
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”		
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷		
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	∑	«	◊		
8	BS	CAN	(	8	H	X	h	x	à	ò	®	∏	»	ÿ		
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	⎵			
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	À			
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã			
D	CR	GS	–	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ			
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ			

NON-PRINTING (columns 0–1)

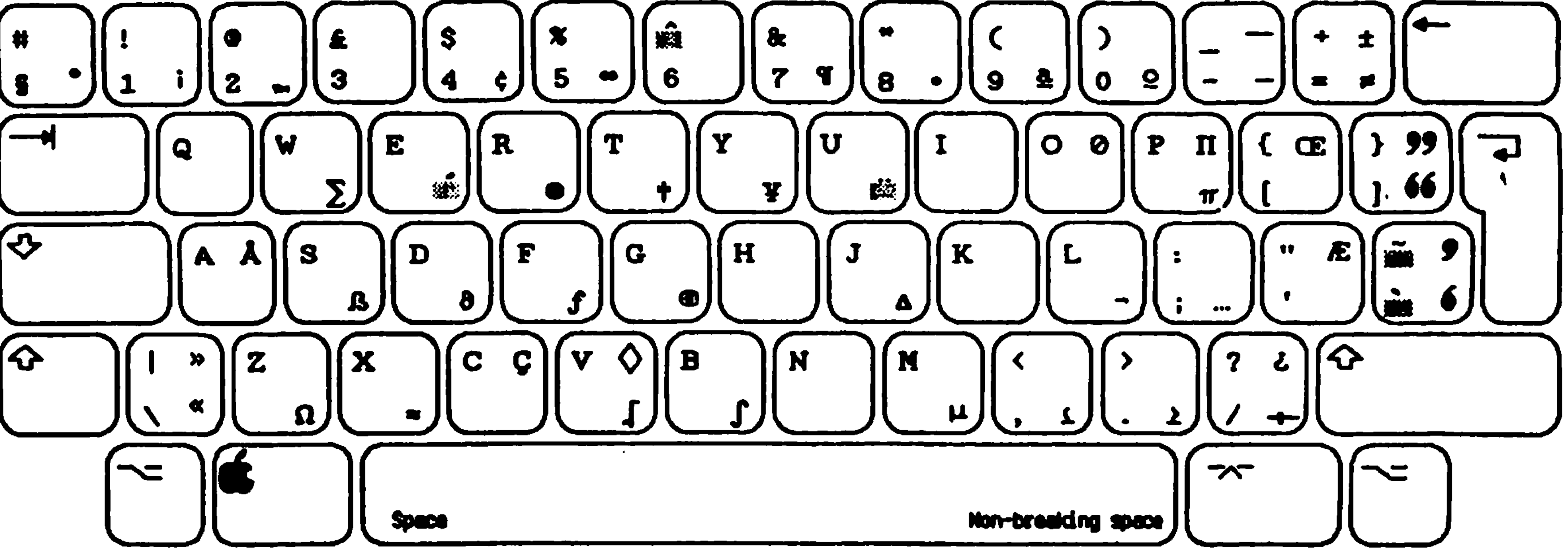
ASCII (columns 0–7) — NON-ASCII (columns 8–F)

⎵ Symbol for a non-breaking space. Prints a blank character, same width as numbers.

(blank box)	Lisa 1.0 Character Set	8 NOV 82
(dotted box)	Revision 2	1 SEP 83


US Keyboard Layout
Underline
Dash
Minus
Hyphen
Backspace
Tab
Caps Lock
Return
Shift
Shift
Option
Space
Non-breaking space
Enter
Option

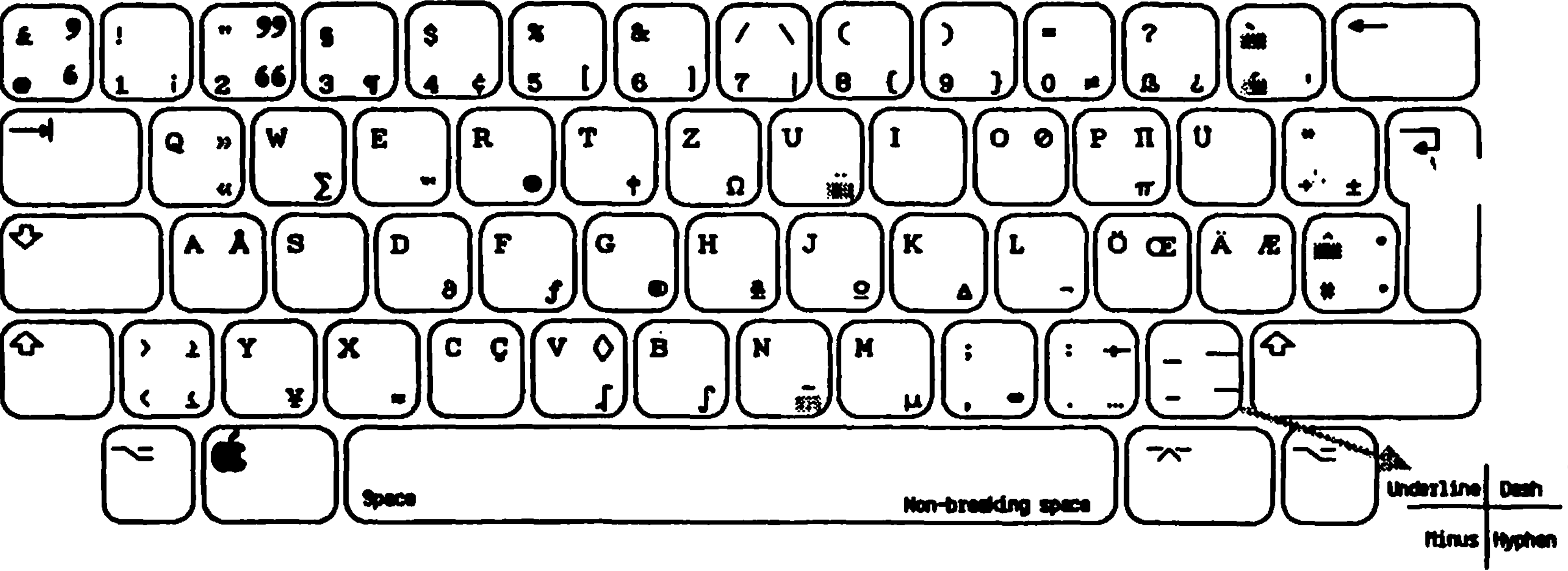
UK Keyboard Layout



DD 8/31/83

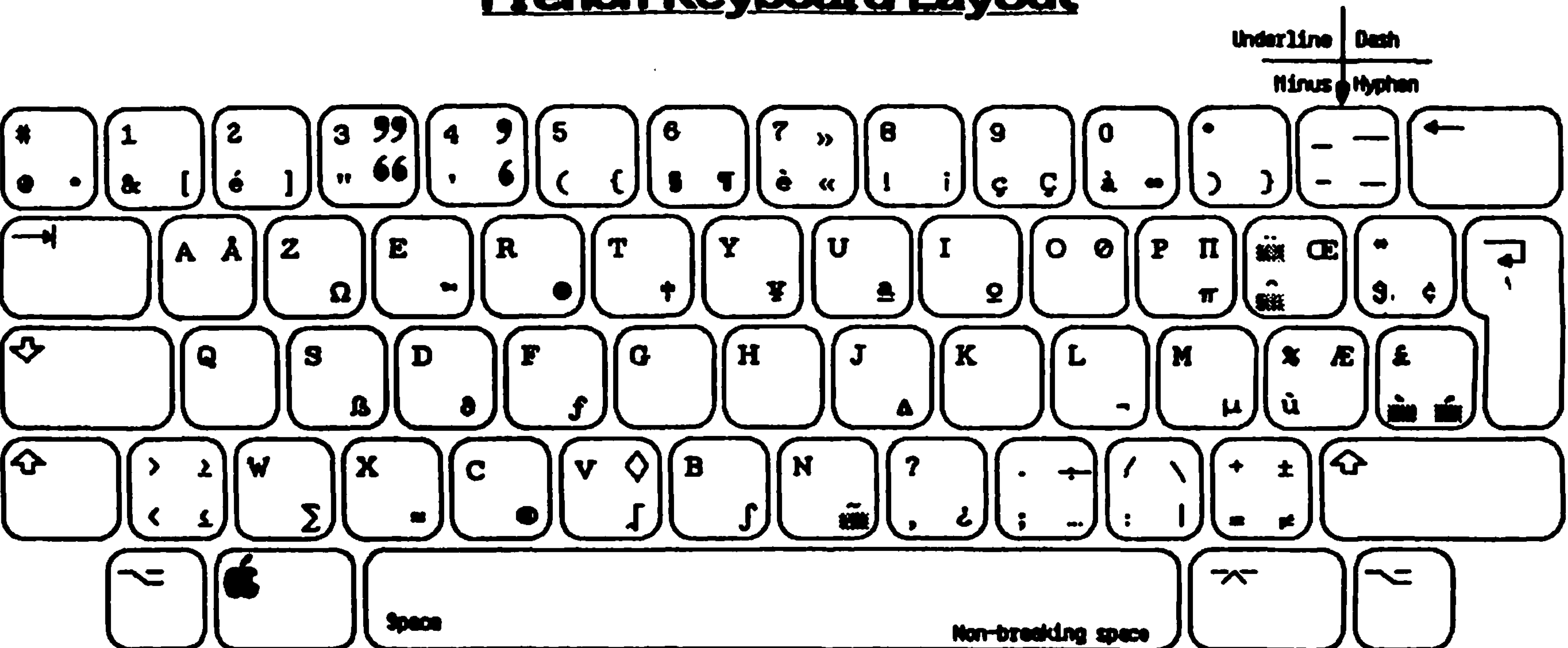
Revised 6/15/83

German Keyboard Layout



DD 8/31/83

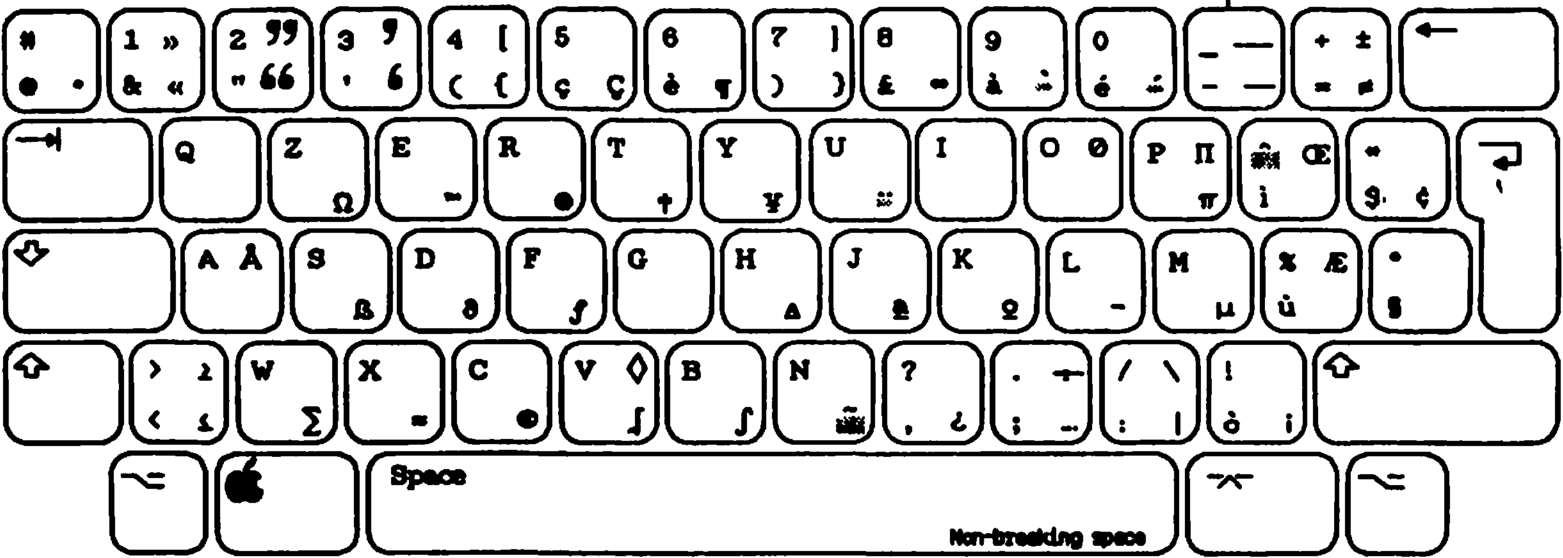
French Keyboard Layout



ATTENTION: Caps look affects number keys too!

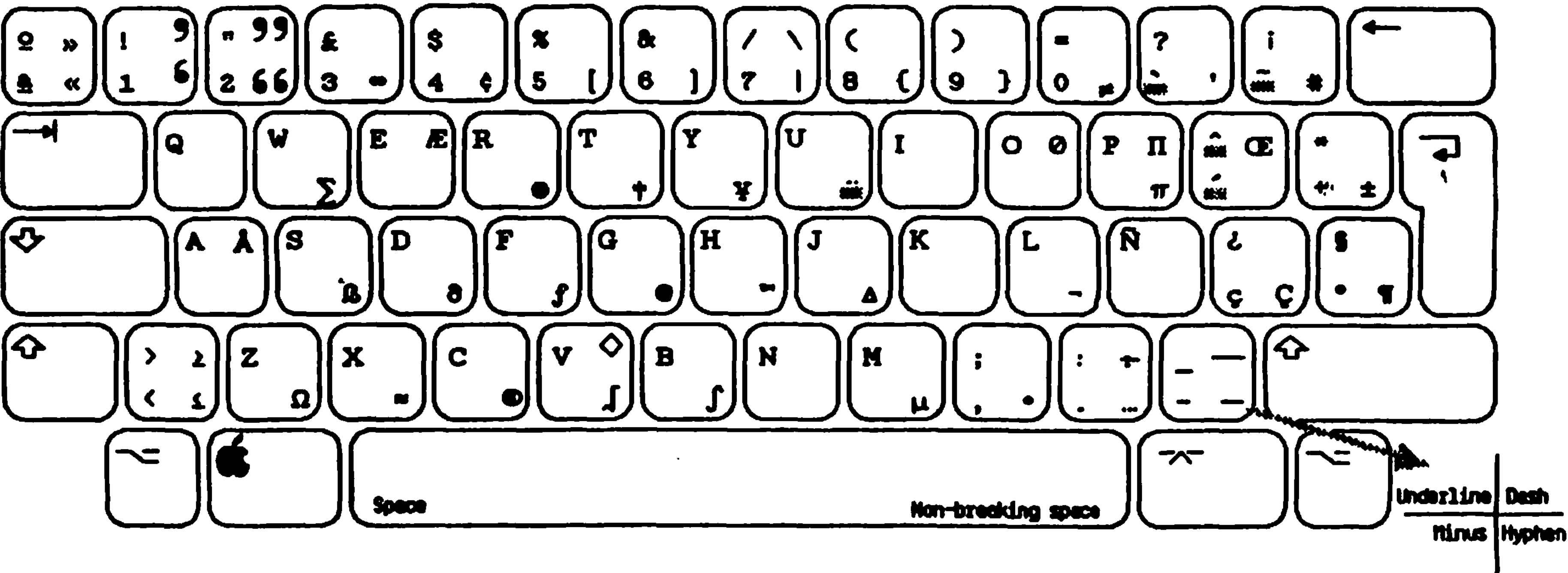
Italian Keyboard Layout

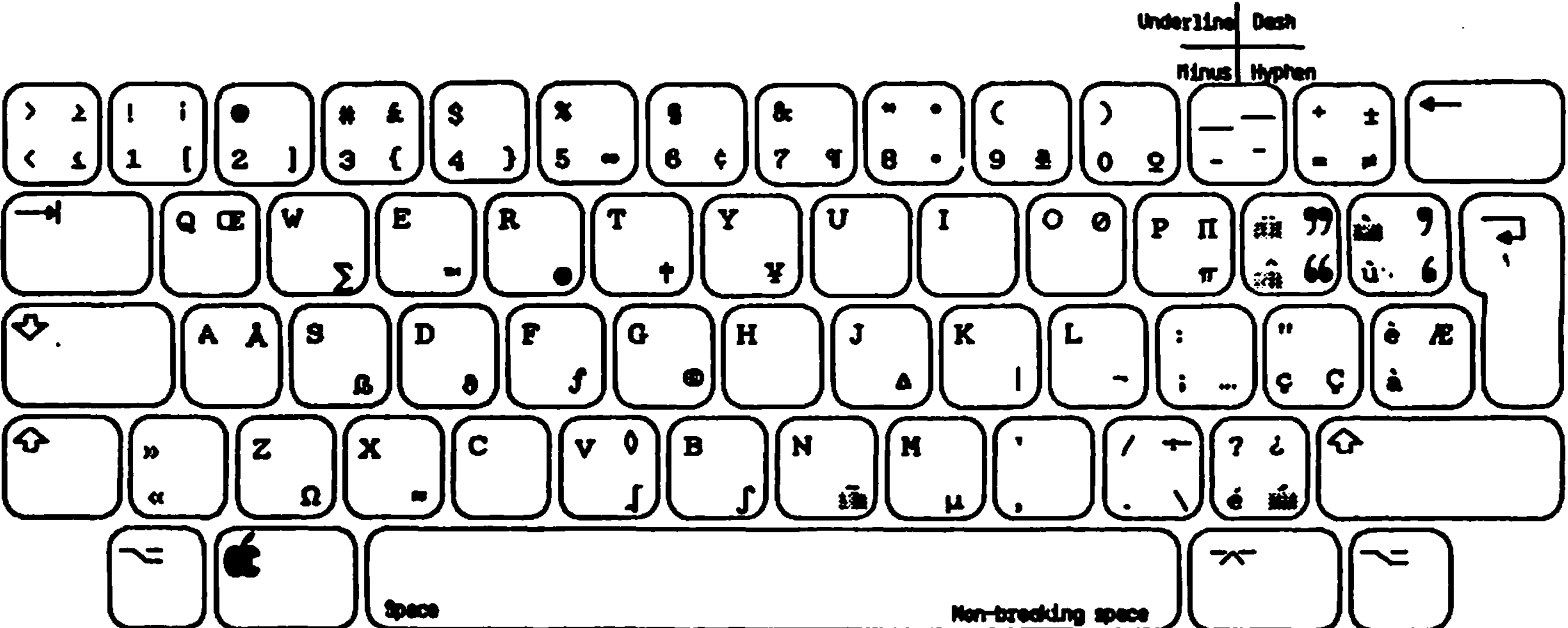
Underline	Dash
Minus	Hyphen



DD 8/31/83

Spanish/Latin American Keyboard Layout





French/Canadian Keyboard Layout
Underline
Dash
Minus
Hyphen
Space
Non-breaking space

APPENDIX 3

Number formats

	US	UK	Germany/Italy	France
Number	1,234,567.89	1,234,567.89	1.234.567,89	1 234 567,89
Decimal number	0.9876	0.9876	0,9876	0,9876
Separator (assumes that numbers are entered without separators other than the decimal point)	,	,	;	;

Example: Number1=4132.2

Number2=3.14159

Separator= "," (US)
";" (Germany)

4123.2 , 3.14159 entered from the keyboard for US

4123,2 ; 3,14159 entered from the keyboard for Germany

APPENDIX 3

Currency representation

	US	UK	Germany	France	Italy
Currency symbol	$0.23	£0.23	0,23 DM	0,23 F	L. 0,23
Negative	($0.23)	(£ 0.23)	- 0,23 DM	- 0,23 F	LIT. -0,23
Without decimal	$345.00	£ 345	325,00 DM	325 F	L. 345

Note :

Thousand separators and decimal point are the same in currency representation as in numbers

APPENDIX 3

Time Formats

	US	UK	Germany	France	Italy
Time	11:30 AM	11:30	11.30 Uhr	11 H 30	11,30
	9:05 Am	09:05	9.05 Uhr	9 H 05	9,05
	11:20 PM	23:20	23.20 Uhr	23 H 20	23,20

APPENDIX 3

Dates Formats

		Date		Expanded date
US	mm/dd/yy	3/31/83	6/3/83	Thursday March 31, 1983
UK	dd/mm/yy	31/03/83	03/06/83	Thursday March 31, 1983
Germany	dd.mm.yy	31.03.83	3.06.83	Donnerstag den 31. März 1983
France	dd.mm.yy	31.03.83	3.06.83	Jeudi 31 mars 1983
Italy	dd/mm/aa	31/03/83	03/06/83	giovedì 31 Maggio 1983

	US/UK	Germany	France	Italy
Days	Monday	Montag	Lundi	lunedì
	Tuesday	Dienstag	Mardi	martedì
	Wednesday	Mittwoch	Mercredi	mercoledì
	Thursday	Donnersdag	Jeudi	giovedì
	Friday	Freitag	Vendredi	venerdì
	Saturday	Sonnabend	Samedi	sabato
	Sunday	Sonntag	Dimanche	domenica
Month	January	Januar	janvier	Gennaio
	February	Februar	février	Febbraio
	March	März	mars	Marzo
	April	April	avril	Aprile
	May	Mai	mai	Maggio
	June	Juni	juin	Giugno
	July	Juli	juillet	Luglio
	August	August	aout	Agosto
	September	September	septembre	Settembre
	October	OKtober	octobre	Ottobre
	November	November	novembre	Novembre
	December	Dezember	décembre	Dicembre

INTLO FORMAT

Offset

0 Byte containing the ASCII character for decimal point
1 Byte containing the ASCII character for thousand separator
2 Byte containing the ASCII character used as a list separator
(must be different from decimal point)

3 3 bytes containing the ASCII for the currency symbol
(equal to 0 if not used)
6 Byte containing currency format flags
Bit 7: set = leading zero, reset = no leading zero
Bit 6: set = trailing zero, reset = no trailing zero
Bit 5: set = minus sign, reset = brackets (for negative amounts)
Bit 4: set = trailing symbol, reset = leading symbol
7 Byte containing short date format
DMY = 2
YMD = 1
MDY = 0
8 Byte containing date elements format flags
Bit 7 : set = leading zero, reset =no leading zero for the Month
Bit 6 : set = leading zero, reset =no leading zero for the Day
Bit 5 : set = Century , reset =no Century
9 Byte containing the ASCII character for the date separator

10 Byte containing the flag for 12 h or 24 h cycle
Byte = FF : 12 hour cycle, Byte = 0 24 hour cycle
11 Byte containing time elements format flags
Bit 7 : set = leading zero, reset = no leading zero for Hours
Bit 6 : set = leading zero, reset = no leading zero for Minutes
Bit 5 : set = leading zero, reset = no leading zero for Seconds
12 4 bytes containing ASCII for trailing string from 0:00 to 11:59
(equal 0 if unused)
16 4 bytes containing ASCII for trailing string from 12:00 to 23:59
(equal 0 if unused)

20 Byte containing ASCII for the time separator

21 8 bytes containing suffix string used in 24 hr mode

29 Byte containing the flag for metric system
Byte = FF : Metric system, Byte = 0 : English system

30 One word to store the version : High byte = country, Low byte = version

INTL1 FORMAT

```
0       1 byte containing the number of characters to be displayed
        15 bytes containing the word for Sunday
.
.
.
96      16 bytes containing the length and word for Monday
        (zero if unused)
112     16 bytes containing the length and word for January
.
.
288     16 bytes containing the length and word for December
        (zero if unused)

304     Byte contaning the flag to suppress the day
            Byte = 00  : Full expanded date
            Byte = FF  : No day of the week in date

305     Byte containing  the flag for expanded date format
            Byte = FF : st0 D st1 M st2 # st3 Y st4
            Byte = 0  : st0 D st1 # st2 M st3 Y st4
306     Byte contaning the flag for day# leading zero
            Byte = FF  : Leading zero
            Byte = 0   : No leading zero
307     Byte containing the month length for short-expanded date
308     4 bytes contaning the string st0
312     4 bytes containing the string st1
316     4 bytes containing the string st2
320     4 bytes containing the string st3
324     4 bytes containing the string st4

328     Version word

330     Routine to handle exceptions for magnitude compare (RTS for US)
```

```
Type INTL = HEXA
     .0 (32)
2C2E3B
46000000
80802E
007F
00000000
00000000
4B
00000000
00000000
FF
0000

      .1 (32)
0344696D616E63686500000000000000
034C756E646900000000000000000000
03546172646900000000000000000000
034D6572637265646900000000000000
034A6575646900000000000000000000
0356656E647265646900000000000000
0353616D656469000000000000000000
074A616E766965720000000000000000
07466576726965720000000000000000
044D6172730000000000000000000000
05417672696C00000000000000000000
034D6169000000000000000000000000
044A75696E0000000000000000000000
074A75696C6C65740000000000000000
04417F9E740000000000000000000000
0953657074656D627265000000000000
074F63746F6272650000000000000000
084E6F76656D62726500000000000000
08446563656D62726500000000000000
00000003
00000000
20000000
20000000
20000000
00000000
0001
4E75
```

APPENDIX 3

Dates Formats

		Date		Expanded date
US	mm/dd/yy	3/31/83	6/3/83	Thursday March 31, 1983
UK	dd/mm/yy	31/03/83	03/06/83	Thursday March 31, 1983
Germany	dd.mm.yy	31.03.83	3.06.83	Donnerstag den 31. März 1983
France	dd.mm.yy	31.03.83	3.06.83	Jeudi 31 mars 1983
Italy	dd/mm/aa	31/03/83	03/06/83	giovedì 31 Maggio 1983

	US/UK	Germany	France	Italy
Days	Monday	Montag	Lundi	lunedì
	Tuesday	Dienstag	Mardi	martedì
	Wednesday	Mittwoch	Mercredi	mercoledì
	Thursday	Donnersdag	Jeudi	giovedì
	Friday	Freitag	Vendredi	venerdì
	Saturday	Sonnabend	Samedi	sabato
	Sunday	Sonntag	Dimanche	domenica
Month	January	Januar	janvier	Gennaio
	February	Februar	février	Febbraio
	March	März	mars	Marzo
	April	April	avril	Aprile
	May	Mai	mai	Maggio
	June	Juni	juin	Giugno
	July	Juli	juillet	Luglio
	August	August	aout	Agosto
	September	September	septembre	Settembre
	October	Oktober	octobre	Ottobre
	November	November	novembre	Novembre
	December	Dezember	décembre	Dicembre

INTLO FORMAT

```
Offset

0       Byte containing the ASCII character for decimal point
1       Byte containing the ASCII character for thousand separator
2       Byte containing the ASCII character used as a list separator
        (must be different from decimal point)

3       3 bytes containing the ASCII for the currency symbol
        (equal to 0 if not used)
6       Byte containing currency format flags
             Bit 7: set = leading zero, reset = no leading zero
             Bit 6: set = trailing zero, reset = no trailing zero
             Bit 5: set = minus sign, reset = brackets (for negative amounts)
Bit 4: set = trailing symbol, reset = leading symbol
7       Byte containing short date format
             DMY = 2
             YMD = 1
             MDY = 0
8       Byte containing date elements format flags
             Bit 7 : set = leading zero, reset =no leading zero for the Month
             Bit 6 : set = leading zero, reset =no leading zero for the Day
             Bit 5 : set = Century , reset =no Century
9       Byte containing the ASCII character for the date separator

10      Byte containing the flag for 12 h or 24 h cycle
             Byte = FF : 12 hour cycle, Byte = 0 24 hour cycle
11      Byte containing time elements format flags
             Bit 7 : set = leading zero, reset = no leading zero  for Hours
             Bit 6 : set = leading zero, reset = no leading zero  for Minutes
             Bit 5 : set = leading zero, reset = no leading zero  for Seconds
12      4 bytes containing ASCII for trailing string from 0:00 to 11:59
         (equal 0 if unused)
16      4 bytes containing ASCII for trailing string from 12:00 to 23:59
         (equal 0 if unused)

20      Byte containing ASCII for the time separator

21      8 bytes containing suffix string used in 24 hr mode

29      Byte containing the flag for metric system
             Byte = FF : Metric system, Byte = 0 :  English system

30     One word to store the version : High byte = country, Low byte = version
```

INTL1 FORMAT

```
0       1 byte containing the number of characters to be displayed
        15 bytes containing the word for Sunday
.
.
.
96      16 bytes containing the length and word for Monday
        (zero if unused)
112     16 bytes containing the length and word for January
.
.
288     16 bytes containing the length and word for December
        (zero if unused)

304     Byte contaning the flag to suppress the day
            Byte = 00  : Full expanded date
            Byte = FF   : No day of the week in date

305    Byte containing  the flag for expanded date format
             Byte = FF : st0 D st1 M st2 # st3 Y st4
             Byte = 0   : st0 D st1 # st2 M st3 Y st4
306    Byte contaning the flag for day# leading zero
             Byte = FF   : Leading zero
             Byte = 0    : No leading zero
307    Byte containing the month length for short-expanded date
308     4 bytes contaning the string st0
312     4 bytes containing the string st1
316     4 bytes containing the string st2
320     4 bytes containing the string st3
324     4 bytes containing the string st4

328     Version word

330     Routine to handle exceptions for magnitude compare (RTS for US)
```

MACINTOSH PUBLICATIONS

QuickDraw: A Programmer's Guide	/QUICK/QUIKDRAW

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
The Window Manager: A Programmer's Guide

Modification History:			
	First Draft	C. Espinosa	11/27/81
	Revised and Edited	C. Espinosa	2/15/82
	Revised and Edited	C. Rose	8/16/82
	Errata Added	C. Rose	8/19/82
	Revised	C. Rose	11/15/82
	Revised for ROM 2.1	C. Rose	3/2/83

ABSTRACT

This document describes the QuickDraw graphics package, heart of the Macintosh User Interface Toolbox routines. It describes the conceptual and physical data types used by QuickDraw and gives details of the procedures and functions available in QuickDraw.

Summary of significant changes and additions since last version:

- "Font" no longer includes type size. There is a new grafPort field (txSize) and a procedure (TextSize) for specifying the size (pages 25, 43). Some other grafPort fields were reordered and some global variables were moved to the grafPort (page 18).

- The character style data type was renamed Style and now includes two new variations, condense and extend (page 23).

- You can set up your application now to produce color output when devices supporting it are available in the future (pages 30, 45).

- The Polygon data type was changed (page 33), and the PolyNext procedure was removed.

- There are two new grafPort routines, InitPort and ClosePort (pages 35, 36), and three new calculation routines, EqualRect and EmptyRect (page 48) and EqualPt (page 65).

- XferRgn and XferRect were removed; use CopyBits, PaintRgn, FillRgn, PaintRect, or FillRect. CursorVis was also removed; use HideCursor or ShowCursor.

- A section on customizing QuickDraw operations was added (page 70).

TABLE OF CONTENTS

3 About This Manual
4 About QuickDraw
5 How To Use QuickDraw
6 The Mathematical Foundation of QuickDraw
6 The Coordinate Plane
7 Points
8 Rectangles
9 Regions
11 Graphic Entities
12 The Bit Image
13 The BitMap
15 Patterns
15 Cursors
17 The Drawing Environment: GrafPort
21 Pen Characteristics
22 Text Characteristics
25 Coordinates in GrafPorts
27 General Discussion of Drawing
29 Transfer Modes
30 Drawing in Color
31 Pictures and Polygons
31 Pictures
32 Polygons
34 QuickDraw Routines
34 GrafPort Routines
39 Cursor-Handling Routines
40 Pen and Line-Drawing Routines
43 Text-Drawing Routines
45 Drawing in Color
46 Calculations with Rectangles
49 Graphic Operations on Rectangles
50 Graphic Operations on Ovals
51 Graphic Operations on Rounded-Corner Rectangles
52 Graphic Operations on Arcs and Wedges
54 Calculations with Regions
58 Graphic Operations on Regions
59 Bit Transfer Operations
61 Pictures
62 Calculations with Polygons
64 Graphic Operations on Polygons
65 Calculations with Points
67 Miscellaneous Utilities
70 Customizing QuickDraw Operations
73 Using QuickDraw from Assembly Language
78 Summary of QuickDraw
87 Glossary

ABOUT THIS MANUAL

This manual describes QuickDraw, a set of graphics procedures, functions, and data types that allow a Pascal or assembly-language programmer of Macintosh to perform highly complex graphic operations very easily and very quickly. It covers the graphic concepts behind QuickDraw, as well as the technical details of the data types, procedures, and functions you will use in your programs.

(hand)

This manual describes version 2.1 of the ROM. In earlier versions, QuickDraw may not work as discussed here.

We assume that you are familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's memory management. This graphics package is for programmers, not end users. Although QuickDraw may be used from either Pascal or assembly language, this manual gives all examples in their Pascal form, to be clear, concise, and more intuitive; a section near the end describes the details of the assembly-language interface to QuickDraw.

The manual begins with an introduction to QuickDraw and what you can do with it. It then steps back a little and looks at the mathematical concepts that form the foundation for QuickDraw: coordinate planes, points, and rectangles. Once you understand these concepts, read on about the graphic entities based on those concepts -- how the mathematical world of planes and rectangles is translated into the physical phenomena of light and shadow.

Then comes some discussion of how to use several graphics ports, a summary of the basic drawing process, and a discussion of two more parts of QuickDraw, pictures and polygons.

Next, there's the detailed description of all QuickDraw procedures and functions, their parameters, calling protocol, effects, side effects, and so on -- all the technical information you'll need each time you write a program for Macintosh.

Following these descriptions are sections that will not be of interest to all readers. Special information is given for programmers who want to customize QuickDraw operations by overriding the standard drawing procedures, and for those who will be using QuickDraw from assembly language.

Finally, there's a summary of the QuickDraw data structures and routine calls, for quick reference, and a glossary that explains terms that may be unfamiliar to you.

ABOUT QUICKDRAW

QuickDraw allows you to divide the Macintosh screen into a number of individual areas. Within each area you can draw many things, as illustrated in Figure 1.

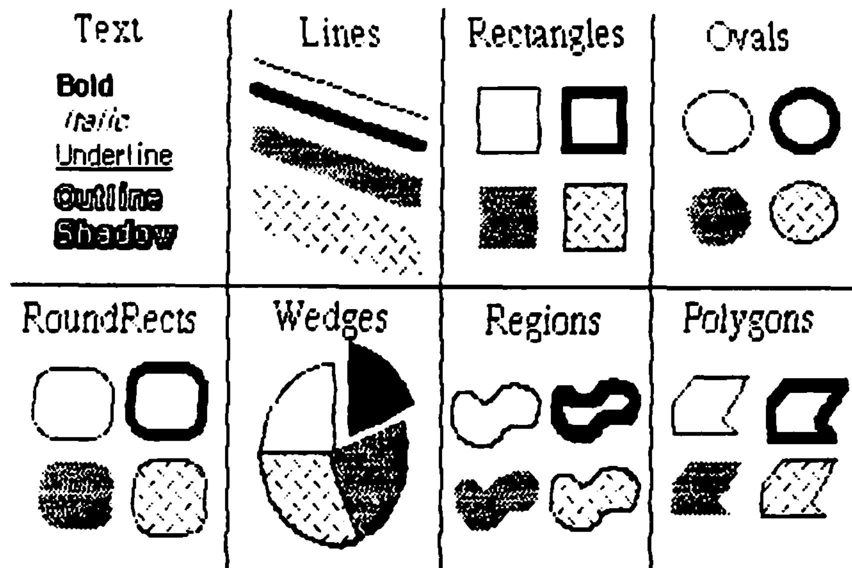


Figure 1. Samples of QuickDraw's Abilities

You can draw:

- Text characters in a number of proportionally-spaced fonts, with variations that include boldfacing, italicizing, underlining, and outlining.

- Straight lines of any length and width.

- A variety of shapes, either solid or hollow, including: rectangles, with or without rounded corners; full circles and ovals or wedge-shaped sections; and polygons.

- Any other arbitrary shape or collection of shapes, again either solid or hollow.

- A picture consisting of any combination of the above items, with just a single procedure call.

In addition, QuickDraw has some other abilities that you won't find in many other graphics packages. These abilities take care of most of the "housekeeping" -- the trivial but time-consuming and bothersome overhead that's necessary to keep things in order.

- The ability to define many distinct "ports" on the screen, each with its own complete drawing environment -- its own coordinate system, drawing location, character set, location on the screen, and so on. You can easily switch from one such port to another.

- Full and complete "clipping" to arbitrary areas, so that drawing will occur only where you want. It's like a super-duper coloring book that won't let you color outside the lines. You don't have to worry about accidentally drawing over something else on the screen, or drawing off the screen and destroying memory.

- Off-screen drawing. Anything you can draw on the screen, you can draw into an off-screen buffer, so you can prepare an image for an output device without disturbing the screen, or you can prepare a picture and move it onto the screen very quickly.

And QuickDraw lives up to its name! It's very fast. The speed and responsiveness of the Macintosh user interface is due primarily to the speed of the QuickDraw package. You can do good-quality animation, fast interactive graphics, and complex yet speedy text displays using the full features of QuickDraw. This means you don't have to bypass the general-purpose QuickDraw routines by writing a lot of special routines to improve speed.

How To Use QuickDraw

QuickDraw can be used from either Pascal or MC68000 machine language. It has no user interface of its own; you must write and compile (or assemble) a Pascal (or assembly-language) program that includes the proper QuickDraw calls, link the resulting object code with the QuickDraw code, and execute the linked object file.

Some programming models are available through your Macintosh software coordinator; they show the structure of a properly organized QuickDraw program. What's best for beginners is to obtain a machine-readable version of the text of one of these programs, read through the text, and, using the superstructure of the program as a "shell", modify it to suit your own purposes. Once you get the hang of writing programs inside the presupplied shell, you can work on changing the shell itself.

QuickDraw is stored permanently in the ROM memory. All access is made through an indirection table in low RAM. When you write a program that uses QuickDraw, you link it with this indirection table. Each time you call a QuickDraw procedure or function, or load a predefined constant, the request goes through the table into QuickDraw. You'll never access any QuickDraw address directly, nor will you have to code constant addresses into your program. The linker will make sure all address references get straightened out.

QuickDraw is an independent unit; it doesn't use any other units, not even HeapZone (the Pascal interface to the Operating System's memory management routines). This means it cannot use the data types Ptr and Handle, because they are defined in HeapZone. Instead, QuickDraw defines two data types that are equivalent to Ptr and Handle, QDPtr and QDHandle.

```
TYPE QDByte    = -128..127;
     QDPtr     = ^QDByte;
     QDHandle  = ^QDPtr;
```

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Keyboard input, mouse input, and larger user-interface constructs such as windows and menus are implemented in separate packages that use QuickDraw but are linked in as separate units. You don't need these units in order to use QuickDraw; however, you'll probably want to read the documentation for windows and menus and learn how to use them with your Macintosh programs.

THE MATHEMATICAL FOUNDATION OF QUICKDRAW

To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics will be, too. QuickDraw defines some clear mathematical constructs that are widely used in its procedures, functions, and data types: the coordinate plane, the point, the rectangle, and the region.

The Coordinate Plane

All information about location, placement, or movement that you give to QuickDraw is in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid, as illustrated in Figure 2.

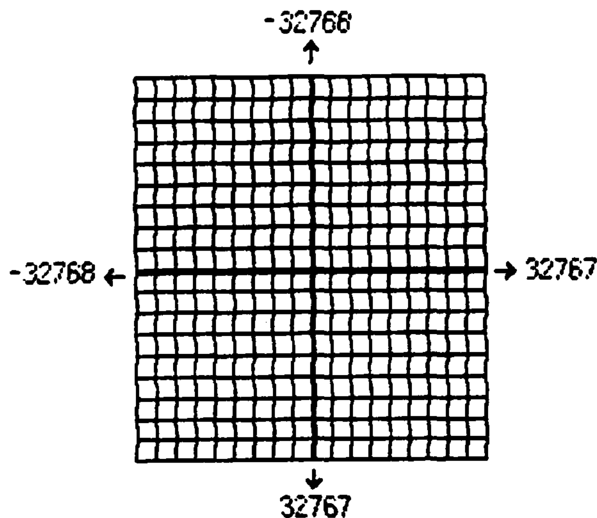


Figure 2. The Coordinate Plane

There are two distinctive features of the QuickDraw coordinate plane:

- All grid coordinates are integers.

- All grid lines are infinitely thin.

These concepts are important! First, they mean that the QuickDraw plane is finite, not infinite (although it's very large). Horizontal coordinates range from -32768 to +32767, and vertical coordinates have the same range. (An auxiliary package is available that maps real Cartesian space, with X, Y, and Z coordinates, onto QuickDraw's two-dimensional integer coordinate system.)

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that grid lines are infinitely thin, you'll never have "endpoint paranoia" -- the confusion that results from not knowing whether that last dot is included in the line.

Points

On the coordinate plane are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Macintosh screen: when using QuickDraw you associate small parts of the grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate system.

The coordinate origin (Ø,Ø) is in the middle of the grid. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. This is the way both a TV screen and a page of English text are scanned: from the top left to the bottom right.

You can store the coordinates of a point into a Pascal variable whose type is defined by QuickDraw. The type Point is a record of two integers, and has this structure:

```
TYPE VHSelect = (V,H);
     Point    = RECORD CASE INTEGER OF

                Ø:  (v:  INTEGER;
                     h:  INTEGER);

                1:  (vh:  ARRAY [VHSelect] OF INTEGER)

                END;
```

The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable goodPt were declared to be of type Point, the following would all refer to the coordinate parts of the point:

```
goodPt.v              goodPt.h
goodPt.vh[V]          goodPt.vh[H]
```

Rectangles

Any two points can define the top left and bottom right corners of a rectangle. As these points are infinitely small, the borders of the rectangle are infinitely thin (see Figure 3).

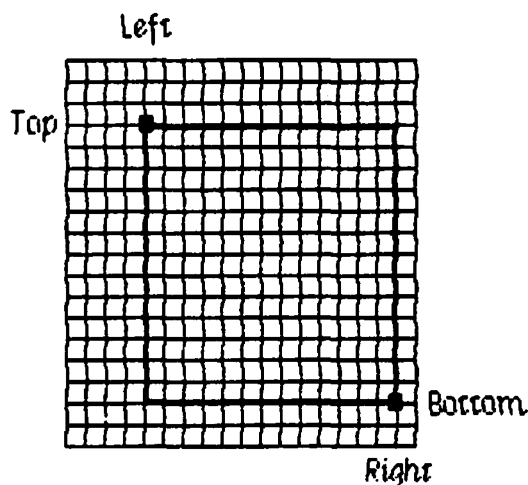


Figure 3. A Rectangle

Rectangles are used to define active areas on the screen, to assign coordinate systems to graphic entities, and to specify the locations and sizes for various drawing commands. QuickDraw also allows you to perform many mathematical calculations on rectangles -- changing their sizes, shifting them around, and so on.

(hand)

> Remember that rectangles, like points, are mathematical concepts that have no direct representation on the screen. The association between these conceptual elements and their physical representations is made by a bitMap, described below.

The data type for rectangles is called Rect, and consists of four integers or two points:

```
TYPE Rect = RECORD CASE INTEGER OF

                 0:  (top:      INTEGER;
                      left:     INTEGER;
                      bottom:   INTEGER;
                      right:    INTEGER);

                 1:  (topLeft:  Point;
                      botRight: Point)

                 END;
```

Again, the record variant allows you to access a variable of type Rect either as four boundary coordinates or as two diagonally opposing corner points. Combined with the record variant for points, all of the following references to the rectangle named bRect are legal:

```
bRect                                             {type Rect}

bRect.topLeft            bRect.botRight           {type Point}

bRect.top                bRect.left               {type INTEGER}
bRect.topLeft.v          bRect.topLeft.h          {type INTEGER}
bRect.topLeft.vh[V]      bRect.topLeft.vh[H]      {type INTEGER}

bRect.bottom             bRect.right              {type INTEGER}
bRect.botRight.v         bRect.botRight.h         {type INTEGER}
bRect.botRight.vh[V]     bRect.botRight.vh[H]     {type INTEGER}
```

(eye)

If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is equal to or less than the left, the rectangle is an empty rectangle (i.e., one that contains no bits).

Regions

Unlike most graphics packages that can manipulate only simple geometric structures (usually rectilinear, at that), QuickDraw has the unique and amazing ability to gather an arbitrary set of spatially coherent points into a structure called a region, and perform complex yet rapid manipulations and calculations on such structures. This remarkable feature not only will make your standard programs simpler and faster, but will let you perform operations that would otherwise be nearly impossible; it is fundamental to the Macintosh user interface.

You define a region by drawing lines, shapes such as rectangles and ovals, or even other regions. The outline of a region should be one or more closed loops. A region can be concave or convex, can consist of one area or many disjoint areas, and can even have "holes" in the middle. In Figure 4, the region on the left has a hole in the middle, and the region on the right consists of two disjoint areas.

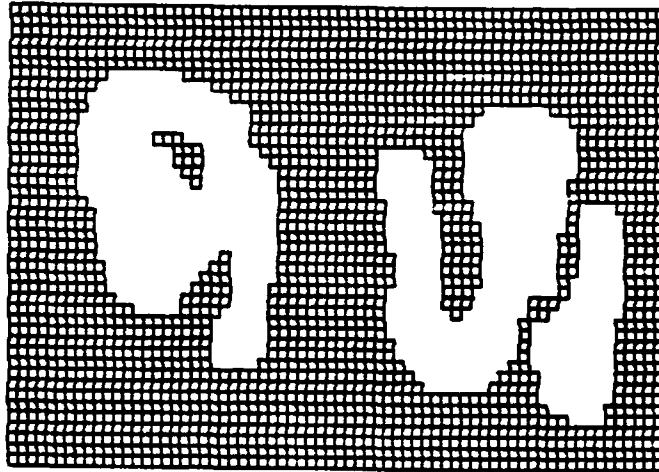


Figure 4. Regions

Because a region can be any arbitrary area or set of areas on the coordinate plane, it takes a variable amount of information to store the outline of a region. The data structure for a region, therefore, is a variable-length entity with two fixed fields at the beginning, followed by a variable-length data field:

```
TYPE Region = RECORD
                rgnSize:  INTEGER;
                rgnBBox:  Rect;
                {optional region definition data}
              END;
```

The rgnSize field contains the size, in bytes, of the region variable. The rgnBBox field is a rectangle which completely encloses the region.

The simplest region is a rectangle. In this case, the rgnBBox field defines the entire region, and there is no optional region data. For rectangular regions (or empty regions), the rgnSize field contains 1Ø.

The region definition data for nonrectangular regions is stored in a compact way which allows for highly efficient access by QuickDraw procedures.

As regions are of variable size, they are stored dynamically on the heap, and the Operating System's memory management moves them around as their sizes change. Being dynamic, a region can be accessed only through a pointer; but when a region is moved, all pointers referring to it must be updated. For this reason, all regions are accessed through handles, which point to one master pointer which in turn points to the region.

```
TYPE RgnPtr    = ^Region;
     RgnHandle = ^RgnPtr;
```

When the memory management relocates a region's data in memory, it updates only the RgnPtr master pointer to that region. The references through the master pointer can find the region's new home, but any references pointing directly to the region's previous position in memory would now point at dead bits. To access individual fields of a region, use the region handle and double indirection:

```
myRgn^^.rgnSize          {size of region whose handle is myRgn}
myRgn^^.rgnBBox          {rectangle enclosing the same region}
myRgn^^.rgnBBox.top      {minimum vertical coordinate of all
                          points in the region}

myRgn^.rgnBBox           {syntactically incorrect; will not compile
                         if myRgn is a rgnHandle}
```

Regions are created by a QuickDraw function which allocates space for the region, creates a master pointer, and returns a rgnHandle. When you're done with a region, you dispose of it with another QuickDraw routine which frees up the space used by the region. Only these calls allocate or deallocate regions; do NOT use the Pascal procedure NEW to create a new region!

You specify the outline of a region with procedures that draw lines and shapes, as described in the section "QuickDraw Routines". An example is given in the discussion of CloseRgn under "Calculations with Regions" in that section.

Many calculations can be performed on regions. A region can be "expanded" or "shrunk" and, given any two regions, QuickDraw can find their union, intersection, difference, and exclusive-OR; it can also determine whether a given point or rectangle intersects a given region, and so on. There is of course a set of graphic operations on regions to draw them on the screen.

GRAPHIC ENTITIES

Coordinate planes, points, rectangles, and regions are all good mathematical models, but they aren't really graphic elements -- they don't have a direct physical appearance. Some graphic entities that do have a direct graphic interpretation are the bit image, bitMap, pattern, and cursor. This section describes the data structure of these graphic entities and how they relate to the mathematical constructs described above.

The Bit Image

A bit image is a collection of bits in memory which have a rectilinear representation. Take a collection of words in memory and lay them end to end so that bit 15 of the lowest-numbered word is on the left and bit Ø of the highest-numbered word is on the far right. Then take this array of bits and divide it, on word boundaries, into a number of equal-size rows. Stack these rows vertically so that the first row is on the top and the last row is on the bottom. The result is a matrix like the one shown in Figure 5 — rows and columns of bits, with each row containing the same number of bytes. The number of bytes in each row of the bit image is called the <u>row width</u> of that image.

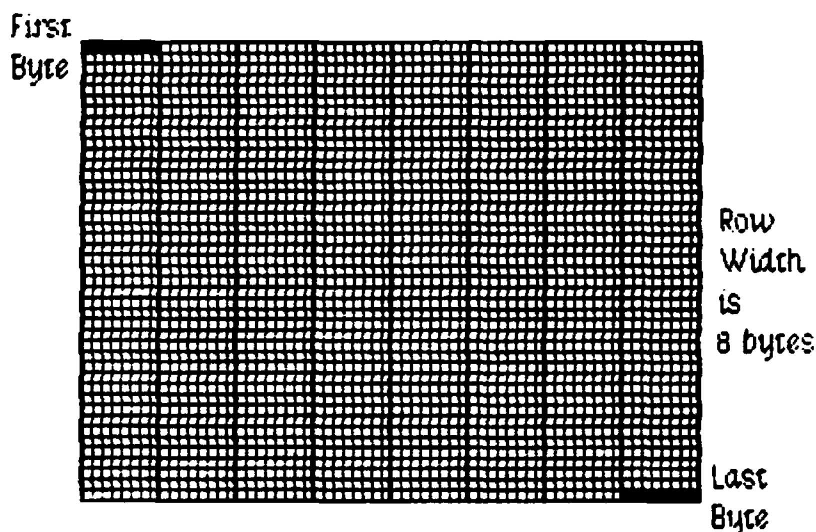


Figure 5. A Bit Image

A bit image can be stored in any static or dynamic variable, and can be of any length that is a multiple of the row width.

The Macintosh screen itself is one large visible bit image. The upper 21,888 bytes of memory are displayed as a matrix of 175,1Ø4 <u>pixels</u> on the screen, each bit corresponding to one pixel. If a bit's value is Ø, its pixel is white; if the bit's value is 1, the pixel is black.

The screen is 342 pixels tall and 512 pixels wide, and the row width of its bit image is 64 bytes. Each pixel on the screen is square; there are 72 pixels per inch in each direction.

(hand)

> Since each pixel on the screen represents one bit in a bit image, wherever this document says "bit", you can substitute "pixel" if the bit image is the Macintosh screen. Likewise, this document often refers to pixels on the screen where the discussion applies equally to bits in an off-screen bit image.

The BitMap

When you combine the physical entity of a bit image with the conceptual entities of the coordinate plane and rectangle, you get a bitMap. A bitMap has three parts: a pointer to a bit image, the row width (in bytes) of that image, and a boundary rectangle which gives the bitMap both its dimensions and a coordinate system. Notice that a bitMap does not actually include the bits themselves: it points to them.

There can be several bitMaps pointing to the same bit image, each imposing a different coordinate system on it. This important feature is explained more fully in "Coordinates in GrafPorts", below.

As shown in Figure 6, the data structure of a bitMap is as follows:

```
TYPE BitMap = RECORD
                baseAddr:  QDPtr;
                rowBytes:  INTEGER;
                bounds:    Rect
              END;
```

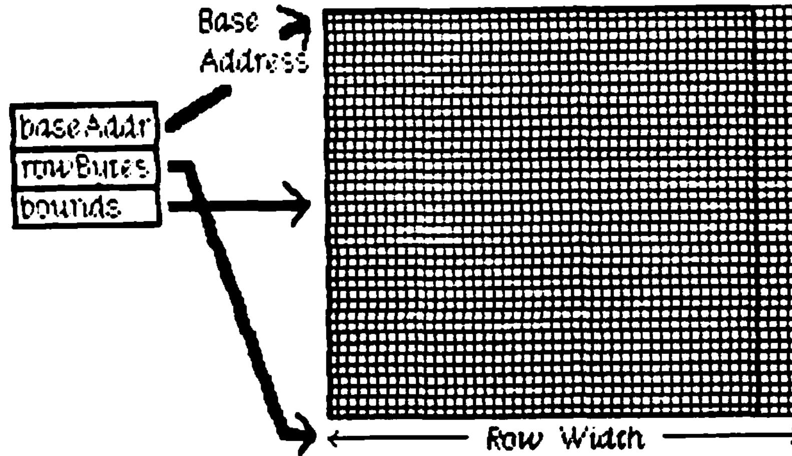


Figure 6. A BitMap

The baseAddr field is a pointer to the beginning of the bit image in memory, and the rowBytes field is the number of bytes in each row of the image. Both of these should always be even: a bitMap should always begin on a word boundary and contain an integral number of words in each row.

The bounds field is a boundary rectangle that both encloses the active area of the bit image and imposes a coordinate system on it. The relationship between the boundary rectangle and the bit image in a bitMap is simple yet very important. First, a few general rules:

- Bits in a bit image fall between points on the coordinate plane.

- A rectangle divides a bit image into two sets of bits: those bits inside the rectangle and those outside the rectangle.

- A rectangle that is H points wide and V points tall encloses exactly (H-1)*(V-1) bits.

The top left corner of the boundary rectangle is aligned around the first bit in the bit image. The width of the rectangle determines how many bits of one row are logically owned by the bitMap; the relationship

```
8*map.rowBytes >= map.bounds.right-map.bounds.left
```

must always be true. The height of the rectangle determines how many rows of the image are logically owned by the bitMap; the relationship

```
SIZEOF(map.baseAddr^) >= (map.bounds.bottom-map.bounds.top)
                                  * map.rowBytes
```

must always be true to ensure that the number of bits in the logical bitMap area is not larger than the number of bits in the bit image.

Normally, the boundary rectangle completely encloses the bit image: the width of the boundary rectangle is equal to the number of bits in one row of the image, and the height of the rectangle is equal to the number of rows in the image. If the rectangle is smaller than the dimensions of the image, the least significant bits in each row, as well as the last rows in the image, are not affected by any operations on the bitMap.

The bitMap also imposes a coordinate system on the image. Because bits fall between coordinate points, the coordinate system assigns integer values to the lines that border and separate bits, not to the bit positions themselves. For example, if a bitMap is assigned the boundary rectangle with corners (10,-8) and (34,8), the bottom right bit in the image will be between horizontal coordinates 33 and 34, and between vertical coordinates 7 and 8 (see Figure 7).

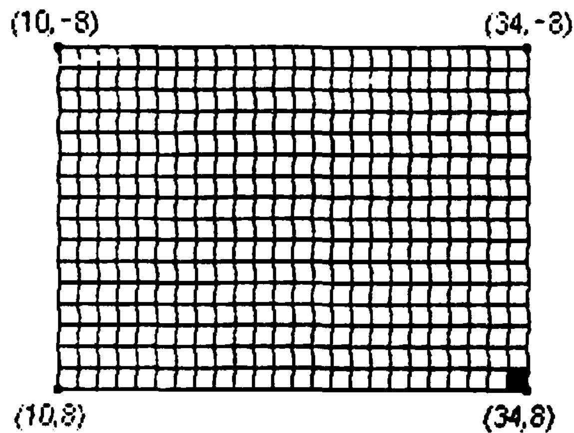


Figure 7. Coordinates and BitMaps

Patterns

A pattern is a 64-bit image, organized as an 8-by-8-bit square, which is used to define a repeating design (such as stripes) or tone (such as gray). Patterns can be used to draw lines and shapes or to fill areas on the screen.

When a pattern is drawn, it is aligned such that adjacent areas of the same pattern in the same graphics port will blend with it into a continuous, coordinated pattern. QuickDraw provides the predefined patterns white, black, gray, ltGray, and dkGray. Any other 64-bit variable or constant can be used as a pattern, too. The data type definition for a pattern is as follows:

```
TYPE Pattern = PACKED ARRAY [Ø..7] OF Ø..255;
```

The row width of a pattern is 1 byte.

Cursors

A cursor is a small image that appears on the screen and is controlled by the mouse. (It appears only on the screen, and never in an off-screen bit image.)

(hand)

Other Macintosh documentation calls this image a "pointer", since it points to a location on the screen. To avoid confusion with other meanings of "pointer" in this manual and other Toolbox documentation, we use the alternate term "cursor".

A cursor is defined as a 256-bit image, a 16-by-16-bit square. The row width of a cursor is 2 bytes. Figure 8 illustrates four cursors.

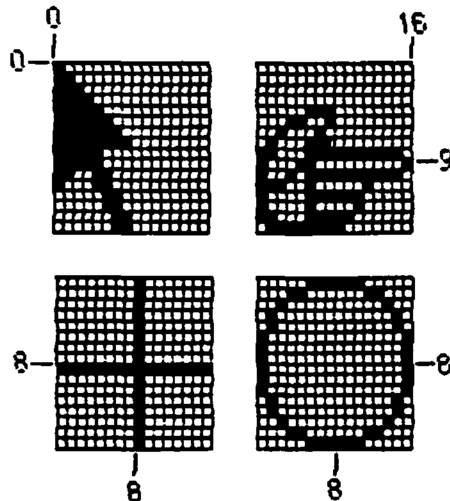


Figure 8. Cursors

A cursor has three fields: a 16-word data field that contains the image itself, a 16-word mask field that contains information about the screen appearance of each bit of the cursor, and a hotSpot point that aligns the cursor with the position of the mouse.

```
TYPE Cursor = RECORD
                data:     ARRAY [Ø..15] OF INTEGER;
                mask:     ARRAY [Ø..15] OF INTEGER;
                hotSpot:  Point
              END;
```

The data for the cursor must begin on a word boundary.

The cursor appears on the screen as a 16-by-16-bit square. The appearance of each bit of the square is determined by the corresponding bits in the data and mask and, if the mask bit is Ø, by the pixel "under" the cursor (the one already on the screen in the same position as this bit of the cursor):

Data	Mask	Resulting pixel on screen
Ø	1	White
1	1	Black
Ø	Ø	Same as pixel under cursor
1	Ø	Inverse of pixel under cursor

Notice that if all mask bits are Ø, the cursor is completely transparent, in that the image under the cursor can still be viewed: pixels under the white part of the cursor appear unchanged, while under the black part of the cursor, black pixels show through as white.

The hotSpot aligns a point in the image (not a bit, a point!) with the mouse position. Imagine the rectangle with corners (Ø,Ø) and (16,16) framing the image, as in each of the examples in Figure 8; the hotSpot is defined in this coordinate system. A hotSpot of (Ø,Ø) is at the top left of the image. For the arrow in Figure 8 to point to the mouse position, (Ø,Ø) would be its hotSpot. A hotSpot of (8,8) is in the exact center of the image; the center of the plus sign or circle in Figure 8 would coincide with the mouse position if (8,8) were the hotSpot for that cursor. Similarly, the hotSpot for the pointing hand would be (16,9).

Whenever you move the mouse, the low-level interrupt-driven mouse routines move the cursor's hotSpot to be aligned with the new mouse position.

(hand)

> The mouse position is always linked to the cursor position. You can't reposition the cursor through software; the only control you have is whether it's visible or not, and what shape it will assume. Think of it as being hard-wired: if the cursor is visible, it always follows the mouse over the full size of the screen.

QuickDraw supplies a predefined arrow cursor, an arrow pointing north-northwest.

THE DRAWING ENVIRONMENT: GRAFPORT

A grafPort is a complete drawing environment that defines how and where graphic operations will have their effect. It contains all the information about one instance of graphic output that is kept separate from all other instances. You can have many grafPorts open at once, and each one will have its own coordinate system, drawing pattern, background pattern, pen size and location, character font and style, and bitMap in which drawing takes place. You can instantly switch from one port to another. GrafPorts are the structures on which a program builds windows, which are fundamental to the Macintosh "overlapping windows" user interface.

A grafPort is a dynamic data structure, defined as follows:

```
TYPE GrafPtr  = ^GrafPort;
     GrafPort = RECORD
                  device:      INTEGER;
                  portBits:    BitMap;
                  portRect:    Rect;
                  visRgn:      RgnHandle;
                  clipRgn:     RgnHandle;
                  bkPat:       Pattern;
                  fillPat:     Pattern;
                  pnLoc:       Point;
                  pnSize:      Point;
                  pnMode:      INTEGER;
                  pnPat:       Pattern;
                  pnVis:       INTEGER;
                  txFont:      INTEGER;
                  txFace:      Style;
                  txMode:      INTEGER;
                  txSize:      INTEGER;
                  spExtra:     INTEGER;
                  fgColor:     LongInt;
                  bkColor:     LongInt;
                  colrBit:     INTEGER;
                  patStretch:  INTEGER;
                  picSave:     QDHandle;
                  rgnSave:     QDHandle;
                  polySave:    QDHandle;
                  grafProcs:   QDProcsPtr
                END;
```

All QuickDraw operations refer to grafPorts via grafPtrs. You create a grafPort with the Pascal procedure NEW and use the resulting pointer in calls to QuickDraw. You could, of course, declare a static VAR of type grafPort, and obtain a pointer to that static structure (with the @ operator), but as most grafPorts will be used dynamically, their data structures should be dynamic also.

(hand)

You can access all fields and subfields of a grafPort normally, but you should not store new values directly into them. QuickDraw has procedures for altering all fields of a grafPort, and using these procedures ensures that changing a grafPort produces no unusual side effects.

The device field of a grafPort is the number of the logical output device that the grafPort will be using. The Font Manager uses this information, since there are physical differences in the same logical font for different output devices. The default device number is 0, for the Macintosh screen. For more information about device numbers, see the *** not yet existing *** Font Manager documentation.

The portBits field is the bitMap that points to the bit image to be used by the grafPort. All drawing that is done in this grafPort will take place in this bit image. The default bitMap uses the entire Macintosh screen as its bit image, with rowBytes of 64 and a boundary rectangle of (Ø,Ø,512,342). The bitMap may be changed to indicate a different structure in memory: all graphics procedures work in exactly the same way regardless of whether their effects are visible on the screen. A program can, for example, prepare an image to be printed on a printer without ever displaying the image on the screen, or develop a picture in an off-screen bitMap before transferring it to the screen. By altering the coordinates of the portBits.bounds rectangle, you can change the coordinate system of the grafPort; with a QuickDraw procedure call, you can set an arbitrary coordinate system for each grafPort, even if the different grafPorts all use the same bit image (e.g., the full screen).

The portRect field is a rectangle that defines a subset of the bitMap for use by the grafPort. Its coordinates are in the system defined by the portBits.bounds rectangle. All drawing done by the application occurs inside this rectangle. The portRect usually defines the "writable" interior area of a window, document, or other object on the screen.

The visRgn field is manipulated by the Window Manager; users and programmers will normally never change a grafPort's visRgn. It indicates that region (remember, an arbitrary area or set of areas) which is actually visible on the screen. For example, if you move one window in front of another, the Window Manager logically removes the area of overlap from the visRgn of the window in the back. When you draw into the back window, whatever's being drawn is clipped to the visRgn so that it doesn't run over onto the front window. The default visRgn is set to the portRect. The visRgn has no effect on images that are not displayed on the screen.

The clipRgn is an arbitrary region that the application can use to limit drawing to any region within the portRect. If, for example, you want to draw a half circle on the screen, you can set the clipRgn to half the square that would enclose the whole circle, and go ahead and draw the whole circle. Only the half within the clipRgn will actually be drawn in the grafPort. The default clipRgn is set arbitrarily large, and you have full control over its setting. Notice that unlike the visRgn, the clipRgn affects the image even if it is not displayed on the screen.

Figure 9 illustrates a typical bitMap (as defined by portBits), portRect, visRgn, and clipRgn.

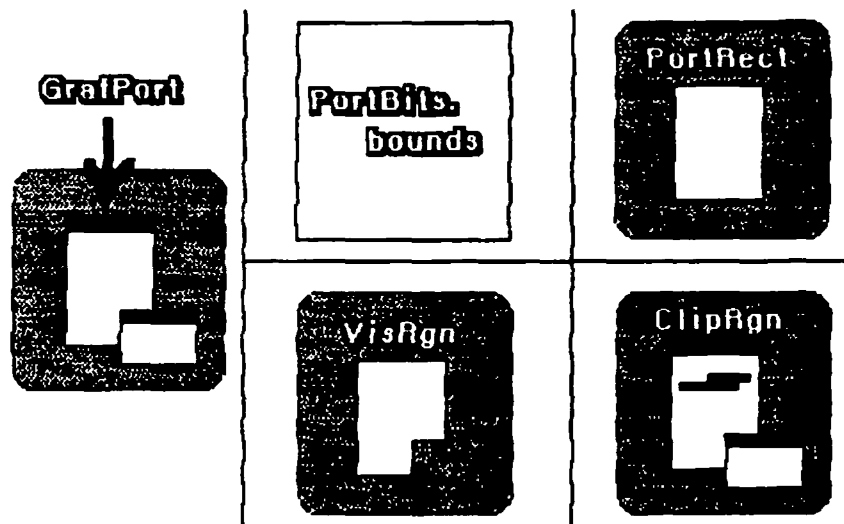


Figure 9. GrafPort Regions

The bkPat and fillPat fields of a grafPort contain patterns used by certain QuickDraw routines. BkPat is the "background" pattern that is used when an area is erased or when bits are scrolled out of it. When asked to fill an area with a specified pattern, QuickDraw stores the given pattern in the fillPat field and then calls a low-level drawing routine which gets the pattern from that field. The various graphic operations are discussed in detail later in the descriptions of individual QuickDraw routines.

Of the next ten fields, the first five determine characteristics of the graphics pen and the last five determine characteristics of any text that may be drawn; these are described in subsections below.

The fgColor, bkColor, and colrBit fields contain values related to drawing in color, a capability that will be available in the future when Apple supports color output devices for the Macintosh. FgColor is the grafPort's foreground color and bkColor is its background color. ColrBit tells the color imaging software which plane of the color picture to draw into. For more information, see "Drawing in Color" in the general discussion of drawing.

The patStretch field is used during output to a printer to expand patterns if necessary. The application should not change its value.

The picSave, rgnSave, and polySave fields reflect the state of picture, region, and polygon defintion, respectively. To define a region, for example, you "open" it, call routines that draw it, and then "close" it. If no region is open, rgnSave contains NIL; otherwise, it contains a handle to information related to the region definition. The application should not be concerned about exactly what information the handle leads to; you may, however, save the current value of rgnSave, set the field to NIL to disable the region definition, and later restore it to the saved value to resume the region definition. The

picSave and polySave fields work similarly for pictures and polygons.

Finally, the grafProcs f[illegible]y point to a special data structure that the application stores i[illegible]it wants to customize QuickDraw drawing procedures or use QuickD[illegible]other advanced, highly specialized ways. (For more information, s[illegible]tomizing QuickDraw Operations".) If grafProcs is NIL, QuickDraw responds in the standard ways described in this manual.

Pen Characteristics

The pnLoc, pnSize, pnMode, pnPat, and pnVis fields of a grafPort deal with the graphics pen. Each grafPort has one and only one graphics pen, which is used for drawing lines, shapes, and text. As illustrated in Figure 1Ø, the pen has four characteristics: a location, a size, a drawing mode, and a drawing pattern.

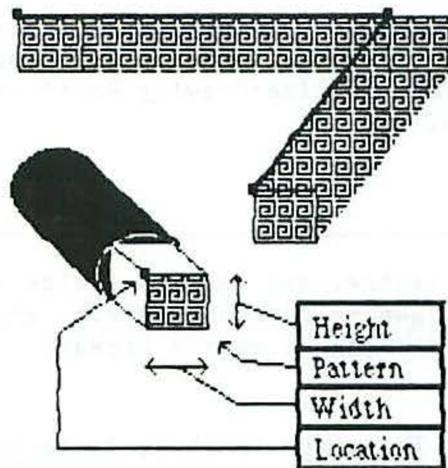


Figure 1Ø. A Graphics Pen

The pen location is a point in the coordinate system of the grafPort, and is where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane: there are no restrictions on the movement or placement of the pen. Remember that the pen location is a point on the coordinate plane, not a pixel in a bit image!

The pen is rectangular in shape, and has a user-definable width and height. The default size is a 1-by-1-bit square; the width and height can range from (Ø,Ø) to (32767,32767). If either the pen width or the pen height is less than 1, the pen will not draw on the screen.

- The pen appears as a rectangle with its top left corner at the pen location; it hangs below and to the right of the pen location.

The pnMode and pnPat fields of a grafPort determine how the bits under the pen are affected when lines or shapes are drawn. The pnPat is a pattern that is used like the "ink" in the pen. This pattern, like all other patterns drawn in the grafPort, is always aligned with the port's coordinate system: the top left corner of the pattern is aligned with the top left corner of the portRect, so that adjacent areas of the same pattern will blend into a continuous, coordinated pattern. Five patterns are predefined (white, black, and three shades of gray); you can also create your own pattern and use it as the pnPat. (A utility procedure, called StuffHex, allows you to fill patterns easily.)

The pnMode field determines how the pen pattern is to affect what's already on the bitMap when lines or shapes are drawn. When the pen draws, QuickDraw first determines what bits of the bitMap will be affected and finds their corresponding bits in the pattern. It then does a bit-by-bit evaluation based on the pen mode, which specifies one of eight boolean operations to perform. The resulting bit is placed into its proper place in the bitMap. The pen modes are described under "Transfer Modes" in the general discussion of drawing below.

The pnVis field determines the pen's visibility, that is, whether it draws on the screen. For more information, see the descriptions of HidePen and ShowPen under "Pen and Line-Drawing Routines" in the "QuickDraw Routines" section.

Text Characteristics

The txFont, txFace, txMode, txSize, and spExtra fields of a grafPort determine how text will be drawn -- the font, style, and size of characters and how they will be placed on the bitMap.

(hand)

> In the Macintosh User Interface Toolbox, character style means stylistic variations such as bold, italic, and underline; font means the complete set of characters of one typeface, such as Helvetica, and does not include the character style or size.

QuickDraw can draw characters as quickly and easily as it draws lines and shapes, and in many prepared fonts. Figure 11 shows two QuickDraw characters and some terms you should become familiar with.

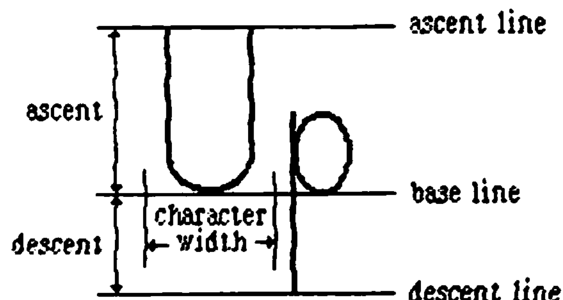


Figure 11. QuickDraw Characters

QuckDraw can display characters in any size, as well as boldfaced, italicized, outlined, or shadowed, all without changing fonts. It can also underline the characters, or draw them closer together or farther apart.

The txFont field is a font number that identifies the character font to be used in the grafPort. The font number Ø represents the system font. For more information about the system font, the other font numbers recognized by the Font Manager, and the construction, layout, and loading of fonts, see the *** not yet existing *** Font Manager documentation.

A character font is defined as a collection of bit images: these images make up the individual characters of the font. The characters can be of unequal widths, and they're not restricted to their "cells": the lower curl of a lowercase j, for example, can stretch back under the previous character (typographers call this kerning). A font can consist of up to 256 distinct characters, yet not all characters need be defined in a single font. Each font contains a missing symbol to be drawn in case of a request to draw a character that is missing from the font.

The txFace field controls the appearance of the font with values from the set defined by the Style data type:

```
       TYPE StyleItem = (bold, italic, underline, outline, shadow,
                         condense, extend);
            Style     = SET OF StyleItem;
```

You can apply these either alone or in combination (see Figure 12). Most combinations usually look good only for large fonts.

Normal Characters
Bold Characters
Italic Characters
Underlined Characters xyz
Outlined Characters
Shadowed Characters
Condensed Characters
Extended Characters
Bold Italic Characters
Bold Outlined Underlined

... and in other fonts, too!

Figure 12. Character Styles

If you specify bold, each character is repeatedly drawn one bit to the right an appropriate number of times for extra thickness.

Italic adds an italic slant to the characters. Character bits above the base line are skewed right; bits below the base line are skewed left.

Underline draws a line below the base line of the characters. If part of a character descends below the base line (as "y" in Figure 12), the underline is not drawn through the pixel on either side of the descending part.

You may specify either outline or shadow. Outline makes a hollow, outlined character rather than a solid one. With shadow, not only is the character hollow and outlined, but the outline is thickened below and to the right of the character to achieve the effect of a shadow. If you specify bold along with outline or shadow, the hollow part of the character is widened.

Condense and extend affect the horizontal distance between all characters, including spaces. Condense decreases the distance between characters and extend increases it, by an amount which the Font Manager determines is appropriate.

The txMode field controls the way characters are placed on a bit image. It functions much like a pnMode: when a character is drawn, QuickDraw determines which bits of the bit image will be affected, does a bit-by-bit comparison based on the mode, and stores the resulting bits into the bit image. These modes are described under "Transfer Modes" in the general discussion of drawing below. Only three of them -- srcOr, srcXor, and srcBic -- should be used for drawing text.

The txSize field specifies the type size for the font, in points (where "point" here is a printing term meaning 1/72 inch). Any size may be specified. If the Font Manager does not have the font in a specified size, it will scale a size it does have as necessary to produce the size desired. A value of Ø in this field directs the Font Manager to choose the size from among those it has for the font; it will choose whichever size is closest to the system font size.

Finally, the spExtra field is useful when a line of characters is to be drawn justified such that it is aligned with both a left and a right margin (sometimes called "full justification"). SpExtra is the number of pixels by which each space character should be widened to fill out the line.

COORDINATES IN GRAFPORTS

Each grafPort has its own local coordinate system. All fields in the grafPort are expressed in these coordinates, and all calculations and actions performed in QuickDraw use the local coordinate system of the currently selected port.

Two things are important to remember:

- Each grafPort maps a portion of the coordinate plane into a similarly-sized portion of a bit image.

- The portBits.bounds rectangle defines the local coordinates for a grafPort.

The top left corner of portBits.bounds is always aligned around the first bit in the bit image; the coordinates of that corner "anchor" a point on the grid to that bit in the bit image. This forms a common reference point for multiple grafPorts using the same bit image (such as the screen). Given a portBits.bounds rectangle for each port, you know that their top left corners coincide.

The interrelationship between the portBits.bounds and portRect rectangles is very important. As the portBits.bounds rectangle establishes a coordinate system for the port, the portRect rectangle indicates the section of the coordinate plane (and thus the bit image) that will be used for drawing. The portRect usually falls inside the portBits.bounds rectangle, but it's not required to do so.

When a new grafPort is created, its bitMap is set to point to the entire Macintosh screen, and both the portBits.bounds and the portRect rectangles are set to 512-by-342-bit rectangles, with the point (Ø,Ø) at the top left corner of the screen.

You can redefine the local coordinates of the top left corner of the grafPort's portRect, using the SetOrigin procedure. This changes the local coordinate system of the grafPort, recalculating the coordinates of all points in the grafPort to be relative to the new corner

coordinates. For example, consider these procedure calls:

```
SetPort(gamePort);
SetOrigin(40,80);
```

The call to SetPort sets the current grafPort to gamePort; the call to SetOrigin changes the local coordinates of the top left corner of that port's portRect to (40,80) (see Figure 13).

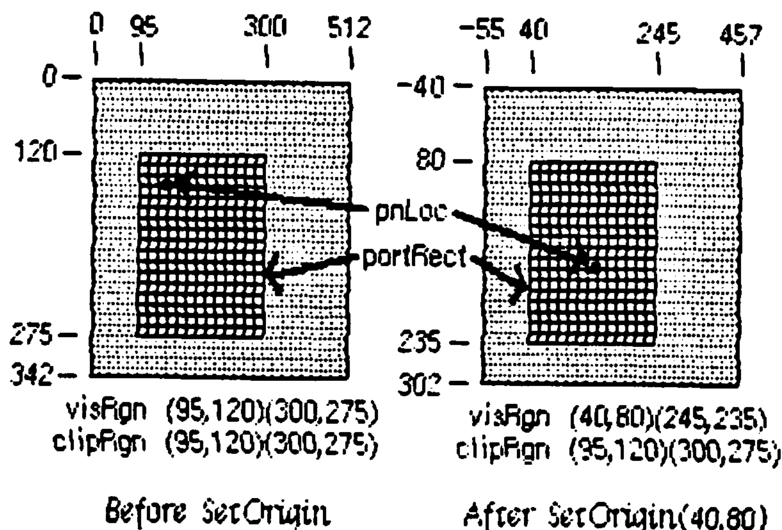


Figure 13. Changing Local Coordinates

This recalculates the coordinate components of the following elements:

```
gamePort^.portBits.bounds        gamePort^.portRect

gamePort^.visRgn
```

These elements are always kept "in sync", so that all calculations, comparisons, or operations that seem right, work right.

Notice that when the local coordinates of a grafPort are offset, the visRgn of that port is offset also, but the clipRgn is not. A good way to think of it is that if a document is being shown inside a grafPort, the document "sticks" to the coordinate system, and the port's structure "sticks" to the screen. Suppose, for example, that the visRgn and clipRgn in Figure 13 before SetOrigin are the same as the portRect, and a document is being shown. After the SetOrigin call, the top left corner of the clipRgn is still (95,120), but this location has moved down and to the right, and the location of the pen within the document has similarly moved. The locations of portBits.bounds, portRect, and visRgn did not change; their coordinates were offset. As always, the top left corner of portBits.bounds remains aligned around the first bit in the bit image (the first pixel on the screen).

If you are moving, comparing, or otherwise dealing with mathematical items in different grafPorts (for example, finding the intersection of

two regions in two different grafPorts), you must adjust to a common coordinate system before you perform the operation. A QuickDraw procedure, LocalToGlobal, lets you convert a point's local coordinates to a global system where the top left corner of the bit image is (0,0); by converting the various local coordinates to global coordinates, you can compare and mix them with confidence. For more information, see the description of this procedure under "Calculations with Points" in the section "QuickDraw Routines".

GENERAL DISCUSSION OF DRAWING

Drawing occurs:

- Always inside a grafPort, in the bit image and coordinate system defined by the grafPort's bitMap.

- Always within the intersection of the grafPort's portBits.bounds and portRect, and clipped to its visRgn and clipRgn.

- Always at the grafPort's pen location.

- Usually with the grafPort's pen size, pattern, and mode.

With QuickDraw procedures, you can draw lines, shapes, and text. Shapes include rectangles, ovals, rounded-corner rectangles, wedge-shaped sections of ovals, regions, and polygons.

Lines are defined by two points: the current pen location and a destination location. When drawing a line, QuickDraw moves the top left corner of the pen along the mathematical trajectory from the current location to the destination. The pen hangs below and to the right of the trajectory (see Figure 14).

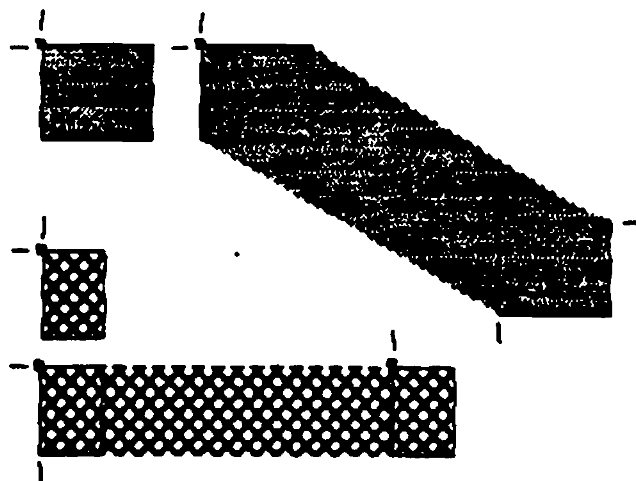


Figure 14. Drawing Lines

(hand)

No mathematical element (such as the pen location) is ever affected by clipping; clipping only determines what appears where in the bit image. If you draw a line to a location outside your grafPort, the pen location will move there, but only the portion of the line that is inside the port will actually be drawn. This is true for all drawing procedures.

Rectangles, ovals, and rounded-corner rectangles are defined by two corner points. The shapes always appear inside the mathematical rectangle defined by the two points. A region is defined in a more complex manner, but also appears only within the rectangle enclosing it. Remember, these enclosing rectangles have infinitely thin borders and are not visible on the screen.

As illustrated in Figure 15, shapes may be drawn either solid (filled in with a pattern) or framed (outlined and hollow).

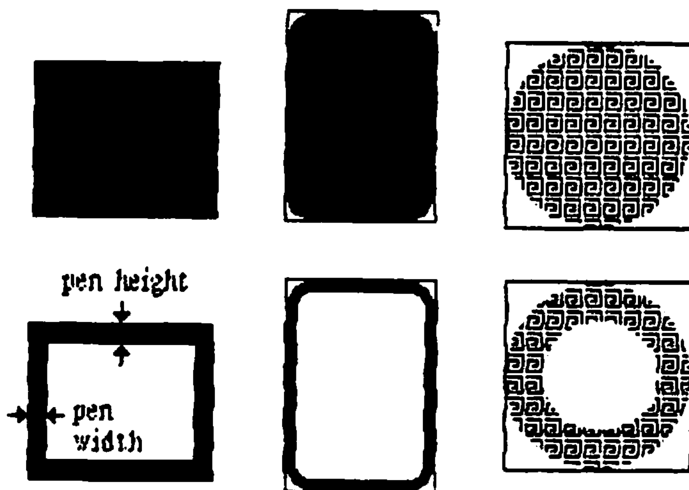


Figure 15. Solid Shapes and Framed Shapes

In the case of framed shapes, the outline appears completely within the enclosing rectangle -- with one exception -- and the vertical and horizontal thickness of the outline is determined by the pen size. The exception is polygons, as discussed in "Pictures and Polygons" below.

The pen pattern is used to fill in the bits that are affected by the drawing operation. The pen mode defines how those bits are to be affected by directing QuickDraw to apply one of eight boolean operations to the bits in the shape and the corresponding pixels on the screen.

Text drawing does not use the pnSize, pnPat, or pnMode, but it does use the pnLoc. Each character is placed to the right of the current pen location, with the left end of its base line at the pen's location. The pen is moved to the right to the location where it will draw the

next character. No wrap or carriage return is performed automatically.

The method QuickDraw uses in placing text is controlled by a mode similar to the pen mode. This is explained in "Transfer Modes", below. Clipping of text is performed in exactly the same manner as all other clipping in QuickDraw.

Transfer Modes

When lines or shapes are drawn, the pnMode field of the grafPort determines how the drawing is to appear in the port's bit image; similarly, the txMode field determines how text is to appear. There is also a QuickDraw procedure that transfers a bit image from one bitMap to another, and this procedure has a mode parameter that determines the appearance of the result. In all these cases, the mode, called a transfer mode, specifies one of eight boolean operations: for each bit in the item to be drawn, QuickDraw finds the corresponding bit in the destination bit image, performs the boolean operation on the pair of bits, and stores the resulting bit into the bit image.

There are two types of transfer mode:

- Pattern transfer modes, for drawing lines or shapes with a pattern.

- Source transfer modes, for drawing text or transferring any bit image between two bitMaps.

For each type of mode, there are four basic operations -- Copy, Or, Xor, and Bic. The Copy operation simply replaces the pixels in the destination with the pixels in the pattern or source, "painting" over the destination without regard for what is already there. The Or, Xor, and Bic operations leave the destination pixels under the white part of the pattern or source unchanged, and differ in how they affect the pixels under the black part: Or replaces those pixels with black pixels, thus "overlaying" the destination with the black part of the pattern or source; Xor inverts the pixels under the black part; and Bic erases them to white.

Each of the basic operations has a variant in which every pixel in the pattern or source is inverted before the operation is performed, giving eight operations in all. Each mode is defined by name as a constant in QuickDraw (see Figure 16).

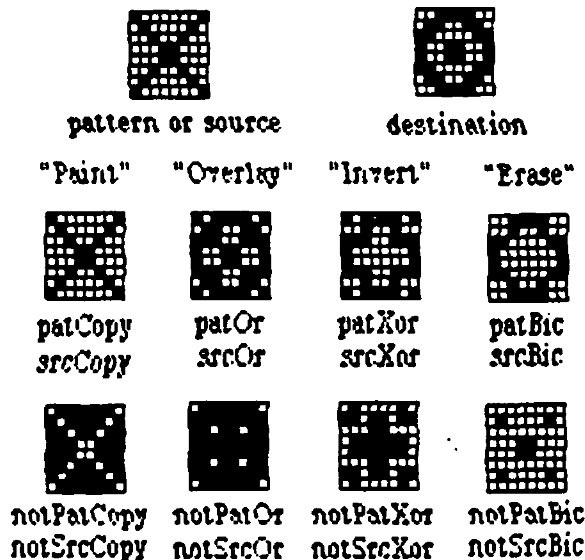


Figure 16. Transfer Modes

Pattern transfer mode	Source transfer mode	Action on each pixel in destination: If black pixel in pattern or source	If white pixel in pattern or source
patCopy	srcCopy	Force black	Force white
patOr	srcOr	Force black	Leave alone
patXor	srcXor	Invert	Leave alone
patBic	srcBic	Force white	Leave alone
notPatCopy	notSrcCopy	Force white	Force black
notPatOr	notSrcOr	Leave alone	Force black
notPatXor	notSrcXor	Leave alone	Invert
notPatBic	notSrcBic	Leave alone	Force white

Drawing in Color

Currently you can only look at QuickDraw output on a black-and-white screen or printer. Eventually, however, Apple will support color output devices. If you want to set up your application now to produce color output in the future, you can do so by using QuickDraw procedures to set the foreground color and the background color. Eight standard colors may be specified with the following predefined constants: blackColor, whiteColor, redColor, greenColor, blueColor, cyanColor, magentaColor, and yellowColor. Initially, the foreground color is blackColor and the background color is whiteColor. If you specify a color other than whiteColor, it will appear as black on a black-and-white output device.

To apply the table in the "Transfer Modes" section above to drawing in color, make the following translation: where the table shows "Force black", read "Force foreground color", and where it shows "Force white", read "Force background color". When you eventually receive the

color output device, you'll find out the effect of inverting a color on it.

(hand)

> QuickDraw can support output devices that have up to 32 bits of color information per pixel. A color picture may be thought of, then, as having up to 32 planes. At any one time, QuickDraw draws into only one of these planes. A QuickDraw routine called by the color-imaging software specifies which plane.

PICTURES AND POLYGONS

QuickDraw lets you save a sequence of drawing commands and "play them back" later with a single procedure call. There are two such mechanisms: one for drawing any picture to scale in a destination rectangle that you specify, and another for drawing polygons in all the ways you can draw other shapes in QuickDraw.

Pictures

A picture in QuickDraw is a transcript of calls to routines which draw something -- anything -- on a bitMap. Pictures make it easy for one program to draw something defined in another program, with great flexibility and without knowing the details about what's being drawn.

For each picture you define, you specify a rectangle that surrounds the picture; this rectangle is called the picture frame. When you later call the procedure that draws the saved picture, you supply a destination rectangle, and QuickDraw scales the picture so that its frame is completely aligned with the destination rectangle. Thus, the picture may be expanded or shrunk to fit its destination rectangle. For example, if the picture is a circle inside a square picture frame, and the destination rectangle is not square, the picture is drawn as an oval.

Since a picture may include any sequence of drawing commands, its data structure is a variable-length entity. It consists of two fixed fields followed by a variable-length data field:

```
TYPE Picture = RECORD
                 picSize:   INTEGER;
                 picFrame:  Rect;
                 {picture definition data}
               END;
```

The picSize field contains the size, in bytes, of the picture variable. The picFrame field is the picture frame which surrounds the picture and gives a frame of reference for scaling when the picture is drawn. The rest of the structure contains a compact representation of the drawing

commands that define the picture.

All pictures are accessed through handles, which point to one master pointer which in turn points to the picture.

```
TYPE PicPtr    = ^Picture;
     PicHandle = ^PicPtr;
```

To define a picture, you call a QuickDraw function that returns a picHandle and then call the routines that draw the picture. There is a procedure to call when you've finished defining the picture, and another for when you're done with the picture altogether.

QuickDraw also allows you to intersperse picture comments in with the definition of a picture. These comments, which do not affect the picture's appearance, may be used to provide additional information about the picture when it's played back. This is especially valuable when pictures are transmitted from one application to another. There are two standard types of comment which, like parentheses, serve to group drawing commands together (such as all the commands that draw a particular part of a picture):

```
CONST picLParen = 0;
      picRParen = 1;
```

The application defining the picture can use these standard comments as well as comments of its own design.

To include a comment in the definition of a picture, the application calls a QuickDraw procedure that specifies the comment with three parameters: the comment kind, which identifies the type of comment; a handle to additional data if desired; and the size of the additional data, if any. When playing back a picture, QuickDraw passes any comments in the picture's definition to a low-level procedure accessed indirectly through the grafProcs field of the grafPort (see "Customizing QuickDraw Operations" for more information). To process comments, the application must include a procedure to do the processing and store a pointer to it in the data structure pointed to by the grafProcs field.

(hand)
The standard low-level procedure for processing picture comments simply ignores all comments.

Polygons

Polygons are similar to pictures in that you define them by a sequence of calls to QuickDraw routines. They are also similar to other shapes that QuickDraw knows about, since there is a set of procedures for performing graphic operations and calculations on them.

A polygon is simply any sequence of connected lines (see Figure 17). You define a polygon by moving to the starting point of the polygon and

drawing lines from there to the next point, from that point to the next, and so on.

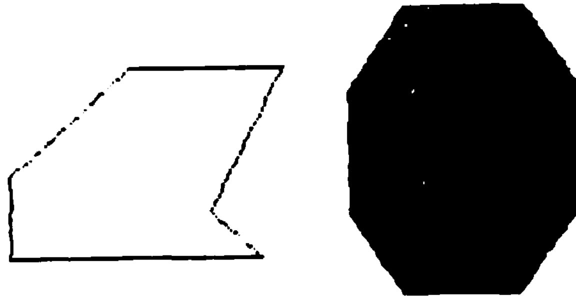


Figure 17. Polygons

The data structure for a polygon is a variable-length entity. It consists of two fixed fields followed by a variable-length array:

```
TYPE Polygon = RECORD
                 polySize:    INTEGER;
                 polyBBox:    Rect;
                 polyPoints:  ARRAY [0..0] OF Point
               END;
```

The polySize field contains the size, in bytes, of the polygon variable. The polyBBox field is a rectangle which just encloses the entire polygon. The polyPoints array expands as necessary to contain the points of the polygon -- the starting point followed by each succesive point to which a line is drawn.

Like pictures and regions, polygons are accessed through handles.

```
TYPE PolyPtr    = ^Polygon;
     PolyHandle = ^PolyPtr;
```

To define a polygon, you call a QuickDraw function that returns a polyHandle and then form the polygon by calling procedures that draw lines. You call a procedure when you've finished defining the polygon, and another when you're done with the polygon altogether.

Just as for other shapes that QuickDraw knows about, there is a set of graphic operations on polygons to draw them on the screen. QuickDraw draws a polygon by moving to the starting point and then drawing lines to the remaining points in succession, just as when the routines were called to define the polygon. In this sense it "plays back" those routine calls. As a result, polygons are not treated exactly the same

as other QuickDraw shapes. For example, the procedure that frames a polygon draws outside the actual boundary of the polygon, because QuickDraw line-drawing routines draw below and to the right of the pen location. The procedures that fill a polygon with a pattern, however, stay within the boundary of the polygon; they also add an additional line between the ending point and the starting point if those points are not the same, to complete the shape.

There is also a difference in the way QuickDraw scales a polygon and a similarly-shaped region if it's being drawn as part of a picture: when stretched, a slanted line is drawn more smoothly if it's part of a polygon rather than a region. You may find it helpful to keep in mind the conceptual difference between polygons and regions: a polygon is treated more as a continuous shape, a region more as a set of bits.

QUICKDRAW ROUTINES

This section describes all the procedures and functions in QuickDraw, their parameters, and their operation. They are presented in their Pascal form; for information on using them from assembly language, see "Using QuickDraw from Assembly Language".

GrafPort Routines

```
PROCEDURE InitGraf (globalPtr: QDPtr);
```

Call InitGraf once and only once at the beginning of your program to initialize QuickDraw. It initializes the QuickDraw global variables listed below.

Variable	Type	Initial setting
thePort	GrafPtr	NIL
white	Pattern	all-white pattern
black	Pattern	all-black pattern
gray	Pattern	50% gray pattern
ltGray	Pattern	25% gray pattern
dkGray	Pattern	75% gray pattern
arrow	Cursor	pointing arrow cursor
screenBits	BitMap	Macintosh screen, (0,0,512,342)
randSeed	LongInt	1

The globalPtr parameter tells QuickDraw where to store its global variables, beginning with thePort. From Pascal programs, this parameter should always be set to @thePort; assembly-language programmers may choose any location, as long as it can accommodate the number of bytes specified by GRAFSIZE in GRAFTYPES.TEXT (see "Using QuickDraw from Assembly Language").

(hand)
To initialize the cursor, call InitCursor (described under "Cursor-Handling Routines" below).

PROCEDURE OpenPort (gp: GrafPtr);

OpenPort allocates space for the given grafPort's visRgn and clipRgn, initializes the fields of the grafPort as indicated below, and makes the grafPort the current port (see SetPort). You must call OpenPort before using any grafPort; first perform a NEW to create a grafPtr and then use that grafPtr in the OpenPort call.

Field	Type	Initial setting
device	INTEGER	Ø (Macintosh screen)
portBits	BitMap	screenBits (see InitGraf)
portRect	Rect	screenBits.bounds (Ø,Ø,512,342)
visRgn	RgnHandle	handle to the rectangular region (Ø,Ø,512,342)
clipRgn	RgnHandle	handle to the rectangular region (-3ØØØØ,-3ØØØØ,3ØØØØ,3ØØØØ)
bkPat	Pattern	white
fillPat	Pattern	black
pnLoc	Point	(Ø,Ø)
pnSize	Point	(1,1)
pnMode	INTEGER	patCopy
pnPat	Pattern	black
pnVis	INTEGER	Ø (visible)
txFont	INTEGER	Ø (system font)
txFace	Style	normal
txMode	INTEGER	srcOr
txSize	INTEGER	Ø (Font Manager decides)
spExtra	INTEGER	Ø
fgColor	LongInt	blackColor
bkColor	LongInt	whiteColor
colrBit	INTEGER	Ø
patStretch	INTEGER	Ø
picSave	QDHandle	NIL
rgnSave	QDHandle	NIL
polySave	QDHandle	NIL
grafProcs	QDProcsPtr	NIL

PROCEDURE InitPort (gp: GrafPtr);

Given a pointer to a grafPort that has been opened with OpenPort, InitPort reinitializes the fields of the grafPort and makes it the current port (if it's not already).

(hand)
InitPort does everything OpenPort does except allocate space for the visRgn and clipRgn.

PROCEDURE ClosePort (gp: GrafPtr);

ClosePort deallocates the space occupied by the given grafPort's visRgn and clipRgn. When you are completely through with a grafPort, call this procedure and then dispose of the grafPort (with a DISPOSE of the grafPtr).

(eye)
If you do not call ClosePort before disposing of the grafPort, the memory used by the visRgn and clipRgn will be unrecoverable.

(eye)
After calling ClosePort, be sure not to use any copies of the visRgn or clipRgn handles that you may have made.

PROCEDURE SetPort (gp: GrafPtr);

SetPort sets the grafPort indicated by gp to be the current port. The global pointer thePort always points to the current port. All QuickDraw drawing routines affect the bitMap thePort^.portBits and use the local coordinate system of thePort^. Note that OpenPort and InitPort do a SetPort to the given port.

(eye)
Never do a SetPort to a port that has not been opened with OpenPort.

Each port possesses its own pen and text characteristics which remain unchanged when the port is not selected as the current port.

PROCEDURE GetPort (VAR gp: GrafPtr);

GetPort returns a pointer to the current grafPort. If you have a program that draws into more than one grafPort, it's extremely useful to have each procedure save the current grafPort (with GetPort), set its own grafPort, do drawing or calculations, and then restore the previous grafPort (with SetPort). The pointer to the current grafPort is also available through the global pointer thePort, but you may prefer to use GetPort for better readability of your program text. For example, a procedure could do a GetPort(savePort) before setting its own grafPort and a SetPort(savePort) afterwards to restore the previous port.

PROCEDURE GrafDevice (device: INTEGER);

GrafDevice sets thePort^.device to the given number, which identifies the logical output device for this grafPort. The Font Manager uses this information. The initial device number is Ø, which represents the Macintosh screen.

PROCEDURE SetPortBits (bm: BitMap);

SetPortBits sets thePort^.portBits to any previously defined bitMap. This allows you to perform all normal drawing and calculations on a buffer other than the Macintosh screen -- for example, a 640-by-7 output buffer for a C. Itoh printer, or a small off-screen image for later "stamping" onto the screen.

Remember to prepare all fields of the bitMap before you call SetPortBits.

PROCEDURE PortSize (width,height: INTEGER);

PortSize changes the size of the current grafPort's portRect. THIS DOES NOT AFFECT THE SCREEN; it merely changes the size of the "active area" of the grafPort.

(hand)
This procedure is normally called only by the Window Manager.

The top left corner of the portRect remains at its same location; the width and height of the portRect are set to the given width and height. In other words, PortSize moves the bottom right corner of the portRect to a position relative to the top left corner.

PortSize does not change the clipRgn or the visRgn, nor does it affect the local coordinate system of the grafPort: it changes only the portRect's width and height. Remember that all drawing occurs only in the intersection of the portBits.bounds and the portRect, clipped to the visRgn and the clipRgn.

PROCEDURE MovePortTo (leftGlobal,topGlobal: INTEGER);

MovePortTo changes the position of the current grafPort's portRect. THIS DOES NOT AFFECT THE SCREEN; it merely changes the location at which subsequent drawing inside the port will appear.

(hand)
This procedure is normally called only by the Window Manager.

The leftGlobal and topGlobal parameters set the distance between the top left corner of portBits.bounds and the top left corner of the new portRect. For example,

```
MovePortTo(256,171);
```

will move the top left corner of the portRect to the center of the screen (if portBits is the Macintosh screen) regardless of the local coordinate system.

Like PortSize, MovePortTo does not change the clipRgn or the visRgn, nor does it affect the local coordinate system of the grafPort.

PROCEDURE SetOrigin (h,v: INTEGER);

SetOrigin changes the local coordinate system of the current grafPort. THIS DOES NOT AFFECT THE SCREEN; it does, however, affect where subsequent drawing and calculation will appear in the grafPort. SetOrigin updates the coordinates of the portBits.bounds, the portRect, and the visRgn. All subsequent drawing and calculation routines will use the new coordinate system.

The h and v parameters set the coordinates of the top left corner of the portRect. All other coordinates are calculated from this point. All relative distances among any elements in the port will remain the same; only their absolute local coordinates will change.

(hand)

> SetOrigin does not update the coordinates of the clipRgn or the pen; these items stick to the coordinate system (unlike the port's structure, which sticks to the screen).

SetOrigin is useful for adjusting the coordinate system after a scrolling operation. (See ScrollRect under "Bit Transfer Operations" below.)

PROCEDURE SetClip (rgn: RgnHandle);

SetClip changes the clipping region of the current grafPort to a region equivalent to the given region. Note that this does not change the region handle, but affects the clipping region itself. Since SetClip makes a copy of the given region, any subsequent changes you make to that region will not affect the clipping region of the port.

You can set the clipping region to any arbitrary region, to aid you in drawing inside the grafPort. The initial clipRgn is an arbitrarily large rectangle.

PROCEDURE GetClip (rgn: RgnHandle);

GetClip changes the given region to a region equivalent to the clipping region of the current grafPort. This is the reverse of what SetClip does. Like SetClip, it does not change the region handle.

PROCEDURE ClipRect (r: Rect);

ClipRect changes the clipping region of the current grafPort to a rectangle equivalent to given rectangle. Note that this does not change the region handle, but affects the region itself.

PROCEDURE BackPat (pat: Pattern);

BackPat sets the background pattern of the current grafPort to the given pattern. The background pattern is used in ScrollRect and in all QuickDraw routines that perform an "erase" operation.

Cursor-Handling Routines

PROCEDURE InitCursor;

InitCursor sets the current cursor to the predefined arrow cursor, an arrow pointing north-northwest, and sets the cursor level to Ø, making the cursor visible. The cursor level, which is initialized to Ø when the system is booted, keeps track of the number of times the cursor has been hidden to compensate for nested calls to HideCursor and ShowCursor (below).

Before you call InitCursor, the cursor is undefined (or, if set by a previous process, it's whatever that process set it to).

PROCEDURE SetCursor (crsr: Cursor);

SetCursor sets the current cursor to the 16-by-16-bit image in crsr. If the cursor is hidden, it remains hidden and will attain the new appearance when it's uncovered; if the cursor is already visible, it changes to the new appearance immediately.

The cursor image is initialized by InitCursor to a north-northwest arrow, visible on the screen. There is no way to retrieve the current cursor image.

PROCEDURE HideCursor;

HideCursor removes the cursor from the screen, restoring the bits under it, and decrements the cursor level (which InitCursor initialized to Ø). Every call to HideCursor should be balanced by a subsequent call to ShowCursor.

PROCEDURE ShowCursor;

ShowCursor increments the cursor level, which may have been decremented by HideCursor, and displays the cursor on the screen if the level becomes Ø. A call to ShowCursor should balance each previous call to HideCursor. The level is not incremented beyond Ø, so extra calls to ShowCursor don't hurt.

QuickDraw low-level interrupt-driven routines link the cursor with the mouse position, so that if the cursor level is Ø (visible), the cursor

automatically follows the mouse. You don't need to do anything but a ShowCursor to have a cursor track the mouse. There is no way to "disconnect" the cursor from the mouse; you can't force the cursor to a certain position, nor can you easily prevent the cursor from entering a certain area of the screen.

If the cursor has been changed (with SetCursor) while hidden, ShowCursor presents the new cursor.

The cursor is initialized by InitCursor to a north-northwest arrow, not hidden.

PROCEDURE ObscureCursor;

ObscureCursor hides the cursor until the next time the mouse is moved. Unlike HideCursor, it has no effect on the cursor level and must not be balanced by a call to ShowCursor.

Pen and Line-Drawing Routines

The pen and line-drawing routines all depend on the coordinate system of the current grafPort. Remember that each grafPort has its own pen; if you draw in one grafPort, change to another, and return to the first, the pen will have remained in the same location.

PROCEDURE HidePen;

HidePen decrements the current grafPort's pnVis field, which is initialized to Ø by OpenPort; whenever pnVis is negative, the pen does not draw on the screen. PnVis keeps track of the number of times the pen has been hidden to compensate for nested calls to HidePen and ShowPen (below). HidePen is called by OpenRgn, OpenPicture, and OpenPoly so that you can define regions, pictures, and polygons without drawing on the screen.

PROCEDURE ShowPen;

ShowPen increments the current grafPort's pnVis field, which may have been decremented by HidePen; if pnVis becomes Ø, QuickDraw resumes drawing on the screen. Extra calls to ShowPen will increment pnVis beyond Ø, so every call to ShowPen should be balanced by a subsequent call to HidePen. ShowPen is called by CloseRgn, ClosePicture, and ClosePoly.

PROCEDURE GetPen (VAR pt: Point);

GetPen returns the current pen location, in the local coordinates of the current grafPort.

PROCEDURE GetPenState (VAR pnState: PenState);

GetPenState saves the pen location, size, pattern, and mode into a storage variable, to be restored later with SetPenState (below). This is useful when calling short subroutines that operate in the current port but must change the graphics pen: each such procedure can save the pen's state when it's called, do whatever it needs to do, and restore the previous pen state immediately before returning.

The PenState data type is not useful for anything except saving the pen's state.

PROCEDURE SetPenState (pnState: PenState);

SetPenState sets the pen location, size, pattern, and mode in the current grafPort to the values stored in pnState. This is usually called at the end of a procedure that has altered the pen parameters and wants to restore them to their state at the beginning of the procedure. (See GetPenState, above.)

PROCEDURE PenSize (width,height: INTEGER);

PenSize sets the dimensions of the graphics pen in the current grafPort. All subsequent calls to Line, LineTo, and the procedures that draw framed shapes in the current grafPort will use the new pen dimensions.

The pen dimensions can be accessed in the variable thePort^.pnSize, which is of type Point. If either of the pen dimensions is set to a negative value, the pen assumes the dimensions (Ø,Ø) and no drawing is performed. For a discussion of how the pen draws, see the "General Discussion of Drawing" earlier in this manual.

PROCEDURE PenMode (mode: INTEGER);

PenMode sets the transfer mode through which the pnPat is transferred onto the bitMap when lines or shapes are drawn. The mode may be any one of the pattern transfer modes:

patCopy	patXor	notPatCopy	notPatXor
patOr	patBic	notPatOr	notPatBic

If the mode is one of the source transfer modes (or negative), no drawing is performed. The current pen mode can be obtained in the variable thePort^.pnMode. The initial pen mode is patCopy, in which the pen pattern is copied directly to the bitMap.

PROCEDURE PenPat (pat: Pattern);

PenPat sets the pattern that is used by the pen in the current grafPort. The standard patterns white, black, gray, ltGray, and dkGray are predefined; the initial pnPat is black. The current pen pattern can be obtained in the variable thePort^.pnPat, and this value can be assigned (but not compared!) to any other variable of type Pattern.

PROCEDURE PenNormal;

PenNormal resets the initial state of the pen in the current grafPort, as follows:

Field	Setting
pnSize	(1,1)
pnMode	patCopy
pnPat	black

The pen location is not changed.

PROCEDURE MoveTo (h,v: INTEGER);

MoveTo moves the pen to location (h,v) in the local coordinates of the current grafPort. No drawing is performed.

PROCEDURE Move (dh,dv: INTEGER);

This procedure moves the pen a distance of dh horizontally and dv vertically from its current location; it calls MoveTo(h+dh,v+dv), where (h,v) is the current location. The positive directions are to the right and down. No drawing is performed.

PROCEDURE LineTo (h,v: INTEGER);

LineTo draws a line from the current pen location to the location specified (in local coordinates) by h and v. The new pen location is (h,v) after the line is drawn. See the general discussion of drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the pnSize, pnMode, or pnPat. (See OpenRgn and OpenPoly.)

PROCEDURE Line (dh,dv: INTEGER);

This procedure draws a line to the location that is a distance of dh horizontally and dv vertically from the current pen location; it calls LineTo(h+dh,v+dv), where (h,v) is the current location. The positive directions are to the right and down. The pen location becomes the coordinates of the end of the line after the line is drawn. See the

general discussion of drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the pnSize, pnMode, or pnPat. (See OpenRgn and OpenPoly.)

Text-Drawing Routines

Each grafPort has its own text characteristics, and all these procedures deal with those of the current port.

```
PROCEDURE TextFont (font: INTEGER);
```

TextFont sets the current grafPort's font (thePort^.txFont) to the given font number. The initial font number is Ø, which represents the system font.

```
PROCEDURE TextFace (face: Style);
```

TextFace sets the current grafPort's character style (thePort^.txFace). The Style data type allows you to specify a set of one or more of the following predefined constants: bold, italic, underline, outline, shadow, condense, and extend. For example:

```
TextFace([bold]);                      {bold}
TextFace([bold,italic]);               {bold and italic}
TextFace(thePort^.txFace+[bold]);      {whatever it was plus bold}
TextFace(thePort^.txFace-[bold]);      {whatever it was but not bold}
TextFace([]);                          {normal}
```

```
PROCEDURE TextMode (mode: INTEGER);
```

TextMode sets the current grafPort's transfer mode for drawing text (thePort^.txMode). The mode should be srcOr, srcXor, or srcBic. The initial transfer mode for drawing text is srcOr.

```
PROCEDURE TextSize (size: INTEGER);
```

TextSize sets the current grafPort's type size (thePort^.txSize) to the given number of points. Any size may be specified, but the result will look best if the Font Manager has the font in that size (otherwise it will scale a size it does have). The next best result will occur if the given size is an even multiple of a size available for the font. If Ø is specified, the Font Manager will choose one of the available sizes -- whichever is closest to the system font size. The initial txSize setting is Ø.

PROCEDURE SpaceExtra (extra: INTEGER);

SpaceExtra sets the current grafPort's spExtra field, which specifies the number of pixels by which to widen each space in a line of text. This is useful when text is being fully justified (that is, aligned with both a left and a right margin). Consider, for example, a line that contains three spaces; if there would normally be six pixels between the end of the line and the right margin, you would call SpaceExtra(2) to print the line with full justification. The initial spExtra setting is Ø.

(hand)

SpaceExtra will also take a negative argument, but be careful not to narrow spaces so much that the text is unreadable.

PROCEDURE DrawChar (ch: CHAR);

DrawChar places the given character to the right of the pen location, with the left end of its base line at the pen's location, and advances the pen accordingly. If the character is not in the font, the font's missing symbol is drawn.

PROCEDURE DrawString (s: Str255);

DrawString performs consecutive calls to DrawChar for each character in the supplied string; the string is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

PROCEDURE DrawText (textBuf: QDPtr; firstByte,byteCount: INTEGER);

DrawText draws text from an arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. The string of text is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

FUNCTION CharWidth (ch: CHAR) : INTEGER;

CharWidth returns the value that will be added to the pen horizontal coordinate if the specified character is drawn. CharWidth includes the effects of the stylistic variations set with TextFace; if you change these after determining the character width but before actually drawing the character, the predetermined width may not be correct. If the character is a space, CharWidth also includes the effect of SpaceExtra.

FUNCTION StringWidth (s: Str255) : INTEGER;

StringWidth returns the width of the given text string, which it calculates by adding the CharWidths of all the characters in the string (see above). This value will be added to the pen horizontal coordinate if the specified string is drawn.

FUNCTION TextWidth (textBuf: QDPtr; firstByte,byteCount: INTEGER) : INTEGER;

TextWidth returns the width of the text stored in the arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. It calculates the width by adding the CharWidths of all the characters in the text. (See CharWidth, above.)

PROCEDURE GetFontInfo (VAR info: FontInfo);

GetFontInfo returns the following information about the current grafPort's character font, taking into consideration the style and size in which the characters will be drawn: the ascent, descent, maximum character width (the greatest distance the pen will move when a character is drawn), and leading (the vertical distance between the descent line and the ascent line below it), all in pixels. The FontInfo data structure is defined as:

```
TYPE FontInfo = RECORD
                  ascent:  INTEGER;
                  descent: INTEGER;
                  widMax:  INTEGER;
                  leading: INTEGER
                END;
```

Drawing in Color

These routines will enable applications to do color drawing in the future when Apple supports color output devices for the Macintosh. All nonwhite colors will appear as black on black-and-white output devices.

PROCEDURE ForeColor (color: LongInt);

ForeColor sets the foreground color for all drawing in the current grafPort (^thePort.fgColor) to the given color. The following standard colors are predefined: blackColor, whiteColor, redColor, greenColor, blueColor, cyanColor, magentaColor, and yellowColor. The initial foreground color is blackColor.

PROCEDURE BackColor (color: LongInt);

BackColor sets the background color for all drawing in the current grafPort (^thePort.bkColor) to the given color. Eight standard colors are predefined (see ForeColor above). The initial background color is whiteColor.

PROCEDURE ColorBit (whichBit: INTEGER);

ColorBit is called by printing software for a color printer, or other color-imaging software, to set the current grafPort's colrBit field to whichBit; this tells QuickDraw which plane of the color picture to draw into. QuickDraw will draw into the plane corresponding to bit number whichBit. Since QuickDraw can support output devices that have up to 32 bits of color information per pixel, the possible range of values for whichBit is Ø through 31. The initial value of the colrBit field is Ø.

Calculations with Rectangles

Calculation routines are independent of the current coordinate system; a calculation will operate the same regardless of which grafPort is active.

(hand)

> Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocalToGlobal and GlobalToLocal under "Calculations with Points" below.

PROCEDURE SetRect (VAR r: Rect; left,top,right,bottom: INTEGER);

SetRect assigns the four boundary coordinates to the rectangle. The result is a rectangle with coordinates (left,top,right,bottom).

This procedure is supplied as a utility to help you shorten your program text. If you want a more readable text at the expense of length, you can assign integers (or points) directly into the rectangle's fields. There is no significant code size or execution speed advantage to either method; one's just easier to write, and the other's easier to read.

PROCEDURE OffsetRect (VAR r: Rect; dh,dv: INTEGER);

OffsetRect moves the rectangle by adding dh to each horizontal coordinate and dv to each vertical coordinate. If dh and dv are

positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it's merely moved on the coordinate plane. This does not affect the screen unless you subsequently call a routine to draw within the rectangle.

```
PROCEDURE InsetRect (VAR r: Rect; dh,dv: INTEGER);
```

InsetRect shrinks or expands the rectangle. The left and right sides are moved in by the amount specified by dh; the top and bottom are moved towards the center by the amount specified by dv. If dh or dv is negative, the appropriate pair of sides is moved outwards instead of inwards. The effect is to alter the size by 2*dh horizontally and 2*dv vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (Ø,Ø,Ø,Ø).

```
FUNCTION SectRect (srcRectA,srcRectB: Rect; VAR dstRect: Rect) :
            BOOLEAN;
```

SectRect calculates the rectangle that is the intersection of the two input rectangles, and returns TRUE if they indeed intersect or FALSE if they do not. Rectangles that "touch" at a line or a point are not considered intersecting, because their intersection rectangle (really, in this case, an intersection line or point) does not enclose any bits on the bitMap.

If the rectangles do not intersect, the destination rectangle is set to (Ø,Ø,Ø,Ø). SectRect works correctly even if one of the source rectangles is also the destination.

```
PROCEDURE UnionRect (srcRectA,srcRectB: Rect; VAR dstRect: Rect);
```

UnionRect calculates the smallest rectangle which encloses both input rectangles. It works correctly even if one of the source rectangles is also the destination.

```
FUNCTION PtInRect (pt: Point; r: Rect) : BOOLEAN;
```

PtInRect determines whether the pixel below and to the right of the given coordinate point is enclosed in the specified rectangle, and returns TRUE if so or FALSE if not.

```
PROCEDURE Pt2Rect (ptA,ptB: Point; VAR: dstRect: Rect);
```

Pt2Rect returns the smallest rectangle which encloses the two input points.

PROCEDURE PtToAngle (r: Rect; pt: Point; VAR angle: INTEGER);

PtToAngle calculates an integer angle between a line from the center of the rectangle to the given point and a line from the center of the rectangle pointing straight up (12 o'clock high). The angle is in degrees from 0 to 359, measured clockwise from 12 o'clock, with 90 degrees at 3 o'clock, 180 at 6 o'clock, and 270 at 9 o'clock. Other angles are measured relative to the rectangle: If the line to the given point goes through the top right corner of the rectangle, the angle returned is 45 degrees, even if the rectangle is not square; if it goes through the bottom right corner, the angle is 135 degrees, and so on (see Figure 18).

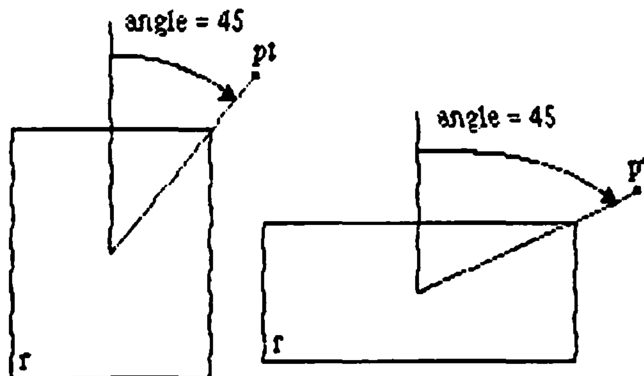


Figure 18. PtToAngle

The angle returned might be used as input to one of the procedures that manipulate arcs and wedges, as described below under "Graphic Operations on Arcs and Wedges".

FUNCTION EqualRect (rectA,rectB: Rect) : BOOLEAN;

EqualRect compares the two rectangles and returns TRUE if they are equal or FALSE if not. The two rectangles must have identical boundary coordinates to be considered equal.

FUNCTION EmptyRect (r: Rect) : BOOLEAN;

EmptyRect returns TRUE if the given rectangle is an empty rectangle or FALSE if not. A rectangle is considered empty if the bottom coordinate is equal to or less than the top or the right coordinate is equal to or less than the left.

Graphic Operations on Rectangles

These procedures perform graphic operations on rectangles. See also ScrollRect under "Bit Transfer Operations".

PROCEDURE FrameRect (r: Rect);

FrameRect draws a hollow outline just inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

PROCEDURE PaintRect (r: Rect);

PaintRect paints the specified rectangle with the current grafPort's pen pattern and mode. The rectangle on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseRect (r: Rect);

EraseRect paints the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

PROCEDURE InvertRect (r: Rect);

InvertRect inverts the pixels enclosed by the specified rectangle: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillRect (r: Rect; pat: Pattern);

FillRect fills the specified rectangle with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Graphic Operations on Ovals

Ovals are drawn inside rectangles that you specify. If the rectangle you specify is square, QuickDraw draws a circle.

```
PROCEDURE FrameOval (r: Rect);
```

FrameOval draws a hollow outline just inside the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

```
PROCEDURE PaintOval (r: Rect);
```

PaintOval paints an oval just inside the specified rectangle with the current grafPort's pen pattern and mode. The oval on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

```
PROCEDURE EraseOval (r: Rect);
```

EraseOval paints an oval just inside the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertOval (r: Rect);
```

InvertOval inverts the pixels enclosed by an oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillOval (r: Rect; pat: Pattern);
```

FillOval fills an oval just inside the specified rectangle with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Graphic Operations on Rounded-Corner Rectangles

PROCEDURE FrameRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

FrameRoundRect draws a hollow outline just inside the specified rounded-corner rectangle, using the current grafPort's pen pattern, mode, and size. OvalWidth and ovalHeight specify the diameters of curvature for the corners (see Figure 19). The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

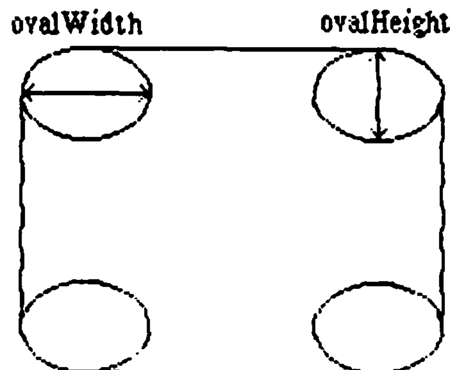


Figure 19. Rounded-Corner Rectangle

If a region is open and being formed, the outside outline of the new rounded-corner rectangle is mathematically added to the region's boundary.

PROCEDURE PaintRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

PaintRoundRect paints the specified rounded-corner rectangle with the current grafPort's pen pattern and mode. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The rounded-corner rectangle on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

EraseRoundRect paints the specified rounded-corner rectangle with the current grafPort's background pattern bkPat (in patCopy mode).

OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

```
PROCEDURE InvertRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
```

InvertRoundRect inverts the pixels enclosed by the specified rounded-corner rectangle: every white pixel becomes black and every black pixel becomes white. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

```
PROCEDURE FillRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER; pat:
            Pattern);
```

FillRoundRect fills the specified rounded-corner rectangle with the given pattern (in patCopy mode). OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Graphic Operations on Arcs and Wedges

These procedures perform graphic operations on arcs and wedge-shaped sections of ovals. See also PtToAngle under "Calculations with Rectangles".

```
PROCEDURE FrameArc (r: Rect; startAngle,arcAngle: INTEGER);
```

FrameArc draws an arc of the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. StartAngle indicates where the arc begins and is treated mod 360. ArcAngle defines the extent of the arc. The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90 (or -270) is at 3 o'clock, 180 (or -180) is at 6 o'clock, and 270 (or -90) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top right corner is at 45 degrees, even if the rectangle is not square; a line through the bottom right corner is at 135 degrees, and so on (see Figure 20).

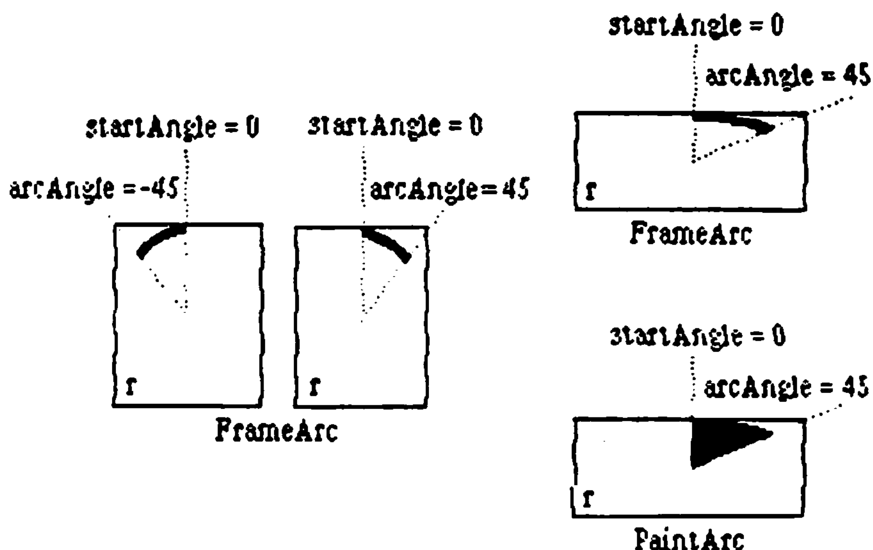


Figure 2Ø. Operations on Arcs and Wedges

The arc is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

(eye)

FrameArc differs from other QuickDraw procedures that frame shapes in that the arc is not mathematically added to the boundary of a region that is open and being formed.

PROCEDURE PaintArc (r: Rect; startAngle,arcAngle: INTEGER);

PaintArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's pen pattern and mode. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The wedge on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseArc (r: Rect; startAngle,arcAngle: INTEGER);

EraseArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

PROCEDURE InvertArc (r: Rect; startAngle,arcAngle: INTEGER);

InvertArc inverts the pixels enclosed by a wedge of the oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillArc (r: Rect; startAngle,arcAngle: INTEGER; pat: Pattern);

FillArc fills a wedge of the oval just inside the specified rectangle with the given pattern (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Calculations with Regions

(hand)
Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocaltoGlobal and GlobalToLocal under "Calculations with Points" below.

FUNCTION NewRgn : RgnHandle;

NewRgn allocates space for a new, dynamic, variable-size region, initializes it to the empty region (Ø,Ø,Ø,Ø), and returns a handle to the new region. Only this function creates new regions; all other procedures just alter the size and shape of regions you create. OpenPort calls NewRgn to allocate space for the port's visRgn and clipRgn.

(eye)
Except when using visRgn or clipRgn, you MUST call NewRgn before specifying a region's handle in any drawing or calculation procedure.

(eye)
Never refer to a region without using its handle.

PROCEDURE DisposeRgn (rgn: RgnHandle);

DisposeRgn deallocates space for the region whose handle is supplied, and returns the memory used by the region to the free memory pool. Use

this only after you are completely through with a temporary region.

(eye)
Never use a region once you have deallocated it, or you will risk being hung by dangling pointers!

PROCEDURE CopyRgn (srcRgn,dstRgn: RgnHandle);

CopyRgn copies the mathematical structure of srcRgn into dstRgn; that is, it makes a duplicate copy of srcRgn. Once this is done, srcRgn may be altered (or even disposed of) without affecting dstRgn. COPYRGN DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call CopyRgn.

PROCEDURE SetEmptyRgn (rgn: RgnHandle);

SetEmptyRgn destroys the previous structure of the given region, then sets the new structure to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE SetRectRgn (rgn: RgnHandle; left,top,right,bottom: INTEGER);

SetRectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by left, top, right, and bottom.

If the specified rectangle is empty (i.e., left>=right or top>=bottom), the region is set to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE RectRgn (rgn: RgnHandle; r: Rect);

RectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by r. This is operationally synonymous with SetRectRgn, except the input rectangle is defined by a rectangle rather than by four boundary coordinates.

PROCEDURE OpenRgn;

OpenRgn tells QuickDraw to allocate temporary space and start saving lines and framed shapes for later processing as a region definition. While a region is open, all calls to Line, LineTo, and the procedures that draw framed shapes (except arcs) affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition; the pen mode, pattern, and size do not affect it. In fact, OpenRgn calls HidePen, so no drawing occurs on the screen while the region is open (unless you called ShowPen just after OpenRgn, or you called ShowPen previously without balancing it by a call to HidePen). Since the pen hangs below and to the right of the pen location, drawing lines with even the smallest pen will change bits that lie outside the region you define.

The outline of a region is mathematically defined and infinitely thin, and separates the bitMap into two groups of bits: those within the region and those outside it. A region should consist of one or more closed loops. Each framed shape itself constitutes a loop. Any lines drawn with Line or LineTo should connect with each other or with a framed shape. Even though the on-screen presentation of a region is clipped, the definition of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

When a region is open, the current grafPort's rgnSave field contains a handle to information related to the region definition. If you want to temporarily disable the collection of lines and shapes, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the region definition.

(eye)

Do not call OpenRgn while another region is already open. All open regions but the most recent will behave strangely.

PROCEDURE CloseRgn (dstRgn: RgnHandle);

CloseRgn stops the collection of lines and framed shapes, organizes them into a region definition, and saves the resulting region into the region indicated by dstRgn. You should perform one and only one CloseRgn for every OpenRgn. CloseRgn calls ShowPen, balancing the HidePen call made by OpenRgn.

Here's an example of how to create and open a region, define a barbell shape, close the region, and draw it:

```
barbell := NewRgn;                       {make a new region}
OpenRgn;                                 {begin collecting stuff}
   SetRect(tempRect,20,20,30,50);        {form the left weight}
   FrameOval(tempRect);
   SetRect(tempRect,30,30,80,40);        {form the bar}
   FrameRect(tempRect);
   SetRect(tempRect,80,20,90,50);        {form the right weight}
   FrameOval(tempRect);
CloseRgn(barbell);                       {we're done; save in barbell}
FillRgn(barbell,black);                  {draw it on the screen}
DisposeRgn(barbell);                     {we don't need you anymore...}
```

PROCEDURE OffsetRgn (rgn: RgnHandle; dh,dv: INTEGER);

OffsetRgn moves the region on the coordinate plane, a distance of dh horizontally and dv vertically. This does not affect the screen unless you subsequently call a routine to draw the region. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape.

(hand)

OffsetRgn is an especially efficient operation, because most of the data defining a region is stored relative to rgnBBox and so isn't actually changed by OffsetRgn.

PROCEDURE InsetRgn (rgn: RgnHandle; dh,dv: INTEGER);

InsetRgn shrinks or expands the region. All points on the region boundary are moved inwards a distance of dv vertically and dh horizontally; if dh or dv is negative, the points are moved outwards in that direction. InsetRgn leaves the region "centered" at the same position, but moves the outline in (for positive values of dh and dv) or out (for negative values of dh and dv). InsetRgn of a rectangular region works just like InsetRect.

PROCEDURE SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

SectRgn calculates the intersection of two regions and places the intersection in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call SectRgn. The dstRgn can be one of the source regions, if desired.

If the regions do not intersect, or one of the regions is empty, the destination is set to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE UnionRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

UnionRgn calculates the union of two regions and places the union in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call UnionRgn. The dstRgn can be one of the source regions, if desired.

If both regions are empty, the destination is set to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE DiffRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

DiffRgn subtracts srcRgnB from srcRgnA and places the difference in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call DiffRgn. The dstRgn can be one of the source regions, if desired.

If the first source region is empty, the destination is set to the empty region (Ø,Ø,Ø,Ø).

PROCEDURE XorRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

XorRgn calculates the difference between the union and the intersection of two regions and places the result in a third region. THIS DOES NOT

CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call XorRgn. The dstRgn can be one of the source regions, if desired.

If the regions are coincident, the destination is set to the empty region (Ø,Ø,Ø,Ø).

FUNCTION PtInRgn (pt: Point; rgn: RgnHandle) : BOOLEAN;

PtInRgn checks whether the pixel below and to the right of the given coordinate point is within the specified region, and returns TRUE if so or FALSE if not.

FUNCTION RectInRgn (r: Rect; rgn: RgnHandle) : BOOLEAN;

RectInRgn checks whether the given rectangle intersects the specified region, and returns TRUE if the intersection encloses at least one bit or FALSE if not.

FUNCTION EqualRgn (rgnA,rgnB: RgnHandle) : BOOLEAN;

EqualRgn compares the two regions and returns TRUE if they are equal or FALSE if not. The two regions must have identical sizes, shapes, and locations to be considered equal. Any two empty regions are always equal.

FUNCTION EmptyRgn (rgn: RgnHandle) : BOOLEAN;

EmptyRgn returns TRUE if the region is an empty region or FALSE if not. Some of the circumstances in which an empty region can be created are: a NewRgn call; a CopyRgn of an empty region; a SetRectRgn or RectRgn with an empty rectangle as an argument; CloseRgn without a previous OpenRgn or with no drawing after an OpenRgn; OffsetRgn of an empty region; InsetRgn with an empty region or too large an inset; SectRgn of nonintersecting regions; UnionRgn of two empty regions; and DiffRgn or XorRgn of two identical or nonintersecting regions.

Graphic Operations on Regions

These routines all depend on the coordinate system of the current grafPort. If a region is drawn in a different grafPort than the one in which it was defined, it may not appear in the proper position inside the port.

PROCEDURE FrameRgn (rgn: RgnHandle);

FrameRgn draws a hollow outline just inside the specified region, using the current grafPort's pen pattern, mode, and size. The outline is as

wide as the pen width and as tall as the pen height; under no circumstances will the frame go outside the region boundary. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the region being framed is mathematically added to that region's boundary.

PROCEDURE PaintRgn (rgn: RgnHandle);

PaintRgn paints the specified region with the current grafPort's pen pattern and pen mode. The region on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseRgn (rgn: RgnHandle);

EraseRgn paints the specified region with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

PROCEDURE InvertRgn (rgn: RgnHandle);

InvertRgn inverts the pixels enclosed by the specified region: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);

FillRgn fills the specified region with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Bit Transfer Operations

PROCEDURE ScrollRect (r: Rect; dh,dv: INTEGER; updateRgn: RgnHandle);

ScrollRect shifts ("scrolls") those bits inside the intersection of the specified rectangle, visRgn, clipRgn, portRect, and portBits.bounds. The bits are shifted a distance of dh horizontally and dv vertically. The positive directions are to the right and down. No other bits are affected. Bits that are shifted out of the scroll area are lost; they are neither placed outside the area nor saved. The grafPort's background pattern bkPat fills the space created by the scroll. In addition, updateRgn is changed to the area filled with bkPat (see Figure 21).

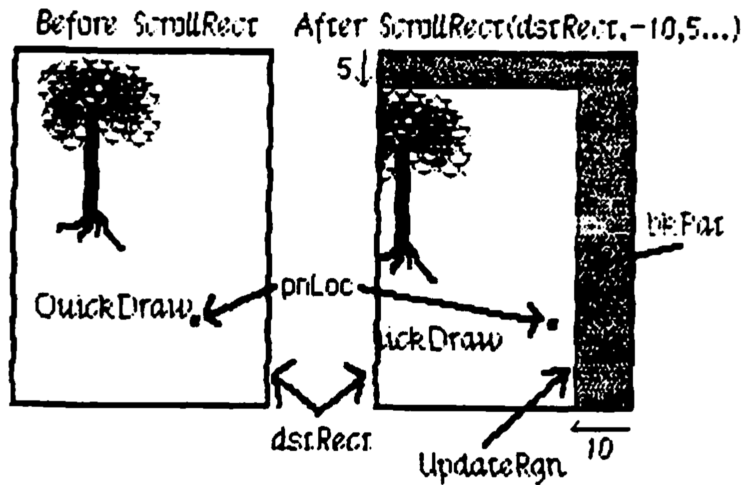


Figure 21. Scrolling

Figure 21 shows that the pen location after a ScrollRect is in a different position relative to what was scrolled in the rectangle. The entire scrolled item has been moved to different coordinates. To restore it to its coordinates before the ScrollRect, you can use the SetOrigin procedure. For example, suppose the dstRect here is the portRect of the grafPort and its top left corner is at (95,120). SetOrigin(105,115) will offset the coordinate system to compensate for the scroll. Since the clipRgn and pen location are not offset, they move down and to the left.

```
PROCEDURE CopyBits (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
            mode: INTEGER; maskRgn: RgnHandle);
```

CopyBits transfers a bit image between any two bitMaps and clips the result to the area specified by the maskRgn parameter. The transfer may be performed in any of the eight source transfer modes. The result is always clipped to the maskRgn and the boundary rectangle of the destination bitMap; if the destination bitMap is the current grafPort's portBits, it is also clipped to the intersection of the grafPort's clipRgn and visRgn. If you do not want to clip to a maskRgn, just pass NIL for the maskRgn parameter.

The dstRect and maskRgn coordinates are in terms of the dstBits.bounds coordinate system, and the srcRect coordinates are in terms of the srcBits.bounds coordinates.

The bits enclosed by the source rectangle are transferred into the destination rectangle according to the rules of the chosen mode. The source transfer modes are as follows:

srcCopy	srcXor	notSrcCopy	notSrcXor
srcOr	srcBic	notSrcOr	notSrcBic

The source rectangle is completely aligned with the destination rectangle; if the rectangles are of different sizes, the bit image is expanded or shrunk as necessary to fit the destination rectangle. For example, if the bit image is a circle in a square source rectangle, and the destination rectangle is not square, the bit image appears as an oval in the destination (see Figure 22).

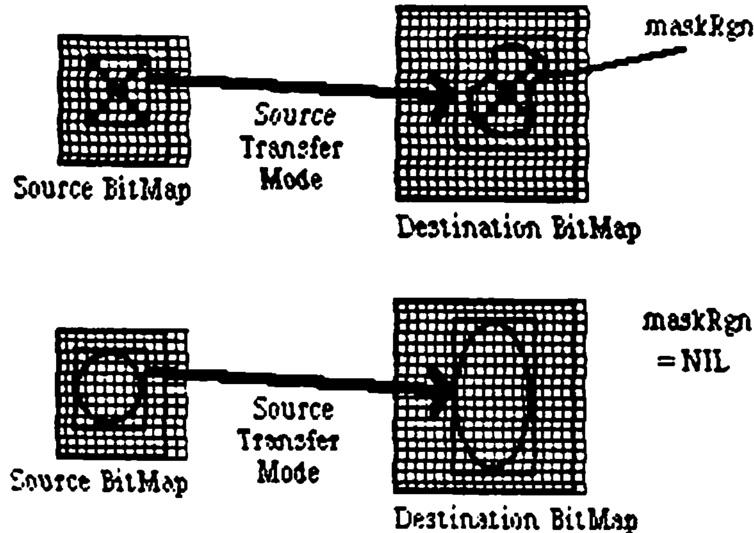


Figure 22. Operation of CopyBits

Pictures

FUNCTION OpenPicture (picFrame: Rect) : PicHandle;

OpenPicture returns a handle to a new picture which has the given rectangle as its picture frame, and tells QuickDraw to start saving as the picture definition all calls to drawing routines and all picture comments (if any).

OpenPicture calls HidePen, so no drawing occurs on the screen while the picture is open (unless you call ShowPen just after OpenPicture, or you called ShowPen previously without balancing it by a call to HidePen).

When a picture is open, the current grafPort's picSave field contains a handle to information related to the picture definition. If you want to temporarily disable the collection of routine calls and picture comments, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the picture definition.

(eye)

Do not call OpenPicture while another picture is already open.

PROCEDURE ClosePicture;

ClosePicture tells QuickDraw to stop saving routine calls and picture comments as the definition of the currently open picture. You should perform one and only one ClosePicture for every OpenPicture. ClosePicture calls ShowPen, balancing the HidePen call made by OpenPicture.

PROCEDURE PicComment (kind,dataSize: INTEGER; dataHandle: QDHandle);

PicComment inserts the specified comment into the definition of the currently open picture. Kind identifies the type of comment. DataHandle is a handle to additional data if desired, and dataSize is the size of that data in bytes. If there is no additional data for the comment, dataHandle should be NIL and dataSize should be Ø. The application that processes the comment must include a procedure to do the processing and store a pointer to the procedure in the data structure pointed to by the grafProcs field of the grafPort (see "Customizing QuickDraw Operations").

PROCEDURE DrawPicture (myPicture: PicHandle; dstRect: Rect);

DrawPicture draws the given picture to scale in dstRect, expanding or shrinking it as necessary to align the borders of the picture frame with dstRect. DrawPicture passes any picture comments to the procedure accessed indirectly through the grafProcs field of the grafPort (see PicComment above).

PROCEDURE KillPicture (myPicture: PicHandle);

KillPicture deallocates space for the picture whose handle is supplied, and returns the memory used by the picture to the free memory pool. Use this only when you are completely through with a picture.

Calculations with Polygons

FUNCTION OpenPoly : PolyHandle;

OpenPoly returns a handle to a new polygon and tells QuickDraw to start saving the polygon definition as specified by calls to line-drawing routines. While a polygon is open, all calls to Line and LineTo affect the outline of the polygon. Only the line endpoints affect the polygon definition; the pen mode, pattern, and size do not affect it. In fact, OpenPoly calls HidePen, so no drawing occurs on the screen while the polygon is open (unless you call ShowPen just after OpenPoly, or you called ShowPen previously without balancing it by a call to HidePen).

A polygon should consist of a sequence of connected lines. Even though the on-screen presentation of a polygon is clipped, the definition of a polygon is not; you can define a polygon anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

When a polygon is open, the current grafPort's polySave field contains a handle to information related to the polygon definition. If you want to temporarily disable the polygon definition, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the polygon definition.

(eye)

Do not call OpenPoly while another polygon is already open.

PROCEDURE ClosePoly;

ClosePoly tells QuickDraw to stop saving the definition of the currently open polygon and computes the polyBBox rectangle. You should perform one and only one ClosePoly for every OpenPoly. ClosePoly calls ShowPen, balancing the HidePen call made by OpenPoly.

Here's an example of how to open a polygon, define it as a triangle, close it, and draw it:

```
triPoly := OpenPoly;        {save handle and begin collecting stuff}
  MoveTo(300,100);          { move to first point and }
  LineTo(400,200);          {        form             }
  LineTo(200,200);          {         the             }
  LineTo(300,100);          {       triangle          }
ClosePoly;                  {stop collecting stuff}
FillPoly(triPoly,gray);     {draw it on the screen}
KillPoly(triPoly);          {we're all done}
```

PROCEDURE KillPoly (poly: PolyHandle);

KillPoly deallocates space for the polygon whose handle is supplied, and returns the memory used by the polygon to the free memory pool. Use this only after you are completely through with a polygon.

PROCEDURE OffsetPoly (poly: PolyHandle; dh,dv: INTEGER);

OffsetPoly moves the polygon on the coordinate plane, a distance of dh horizontally and dv vertically. This does not affect the screen unless you subsequently call a routine to draw the polygon. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The polygon retains its shape and size.

(hand)
OffsetPoly is an especially efficient operation, because the data defining a polygon is stored relative to polyStart and so isn't actually changed by OffsetPoly.

Graphic Operations on Polygons

PROCEDURE FramePoly (poly: PolyHandle);

FramePoly plays back the line-drawing routine calls that define the given polygon, using the current grafPort's pen pattern, mode, and size. The pen will hang below and to the right of each point on the boundary of the polygon; thus, the polygon drawn will extend beyond the right and bottom edges of poly^^.polyBBox by the pen width and pen height, respectively. All other graphic operations occur strictly within the boundary of the polygon, as for other shapes. You can see this difference in Figure 23, where each of the polygons is shown with its polyBBox.

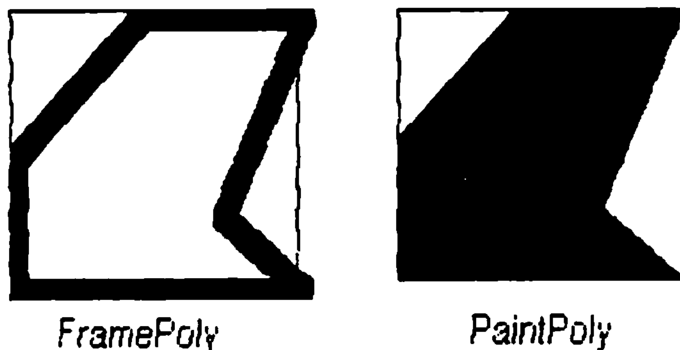


Figure 23. Drawing Polygons

If a polygon is open and being formed, FramePoly affects the outline of the polygon just as if the line-drawing routines themselves had been called. If a region is open and being formed, the outside outline of the polygon being framed is mathematically added to the region's boundary.

PROCEDURE PaintPoly (poly: PolyHandle);

PaintPoly paints the specified polygon with the current grafPort's pen pattern and pen mode. The polygon on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The

pen location is not changed by this procedure.

PROCEDURE ErasePoly (poly: PolyHandle);

ErasePoly paints the specified polygon with the current grafPort's background pattern bkPat (in patCopy mode). The pnPat and pnMode are ignored; the pen location is not changed.

PROCEDURE InvertPoly (poly: PolyHandle);

InvertPoly inverts the pixels enclosed by the specified polygon: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);

FillPoly fills the specified polygon with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Calculations with Points

PROCEDURE AddPt (srcPt: Point; VAR dstPt: Point);

AddPt adds the coordinates of srcPt to the coordinates of dstPt, and returns the result in dstPt.

PROCEDURE SubPt (srcPt: Point; VAR dstPt: Point);

SubPt subtracts the coordinates of srcPt from the coordinates of dstPt, and returns the result in dstPt.

PROCEDURE SetPt (VAR pt: Point; h,v: INTEGER);

SetPt assigns two integer coordinates to a variable of type Point.

FUNCTION EqualPt (ptA,ptB: Point) : BOOLEAN;

EqualPt compares the two points and returns true if they are equal or FALSE if not.

PROCEDURE LocalToGlobal (VAR pt: Point);

LocalToGlobal converts the given point from the current grafPort's local coordinate system into a global coordinate system with the origin (Ø,Ø) at the top left corner of the port's bit image (such as the screen). This global point can then be compared to other global points, or be changed into the local coordinates of another grafPort.

Since a rectangle is defined by two points, you can convert a rectangle into global coordinates by performing two LocalToGlobal calls. You can also convert a rectangle, region, or polygon into global coordinates by calling OffsetRect, OffsetRgn, or OffsetPoly. For examples, see GlobalToLocal below.

PROCEDURE GlobalToLocal (VAR pt: Point);

GlobalToLocal takes a point expressed in global coordinates (with the top left corner of the bitMap as coordinate (Ø,Ø)) and converts it into the local coordinates of the current grafPort. The global point can be obtained with the LocalToGlobal call (see above). For example, suppose a game draws a "ball" within a rectangle named ballRect, defined in the grafPort named gamePort (as illustrated below in Figure 24). If you want to draw that ball in the grafPort named selectPort, you can calculate the ball's selectPort coordinates like this:

```
SetPort(gamePort);                   {start in origin port}
selectBall := ballRect;              {make a copy to be moved}
LocalToGlobal(selectBall.topLeft);   {put both corners into }
LocalToGlobal(selectBall.botRight);  {  global coordinates  }

SetPort(selectPort);                 {switch to destination port}
GlobalToLocal(selectBall.topLeft);   {put both corners into     }
GlobalToLocal(selectBall.botRight);  {  these local coordinates }
FillOval(selectBall,ballColor);      {now you have the ball!}
```

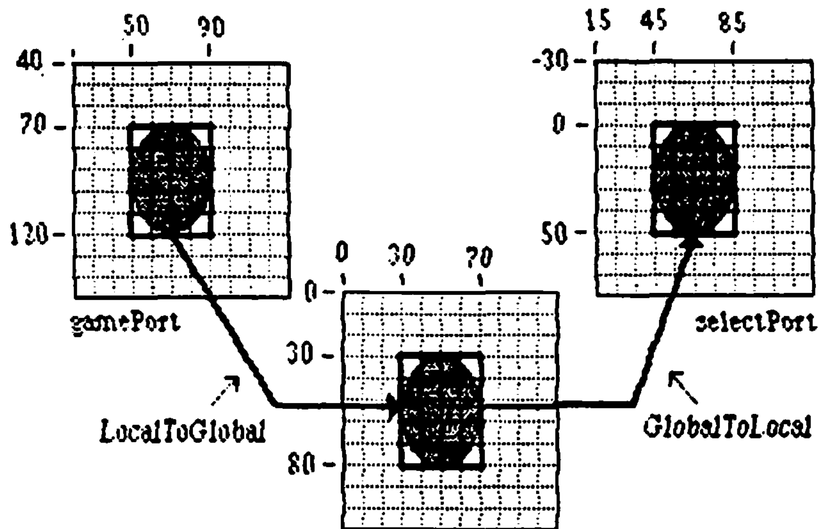


Figure 24. Converting between Coordinate Systems

You can see from Figure 24 that LocalToGlobal and GlobalToLocal simply offset the coordinates of the rectangle by the coordinates of the top left corner of the local grafPort's boundary rectangle. You could also do this with OffsetRect. In fact, the way to convert regions and polygons from one coordinate system to another is with OffsetRgn or OffsetPoly rather than LocalToGlobal and GlobalToLocal. For example, if myRgn were a region enclosed by a rectangle having the same coordinates as ballRect in gamePort, you could convert the region to global coordinates with

```
OffsetRgn(myRgn, -2Ø, -4Ø);
```

and then convert it to the coordinates of the selectPort grafPort with

```
OffsetRgn(myRgn, 15, -3Ø);
```

Miscellaneous Utilities

FUNCTION Random : INTEGER;

This function returns an integer, uniformly distributed pseudo-random, in the range from -32768 through 32767. The value returned depends on the global variable randSeed, which InitGraf initializes to 1; you can start the sequence over again from where it began by resetting randSeed to 1.

FUNCTION GetPixel (h,v: INTEGER) : BOOLEAN;

GetPixel looks at the pixel associated with the given coordinate point and returns TRUE if it is black or FALSE if it is white. The selected pixel is immediately below and to the right of the point whose coordinates are given in h and v, in the local coordinates of the current grafPort. There is no guarantee that the specified pixel actually belongs to the port, however; it may have been drawn by a port overlapping the current one. To see if the point indeed belongs to the current port, perform a PtInRgn(pt,thePort^.visRgn).

PROCEDURE StuffHex (thingPtr: QDPtr; s: Str255);

StuffHex pokes bits (expressed as a string of hexadecimal digits) into any data structure. This is a good way to create cursors, patterns, or bit images to be "stamped" onto the screen with CopyBits. For example,

```
StuffHex(@stripes,'0102040810204080')
```

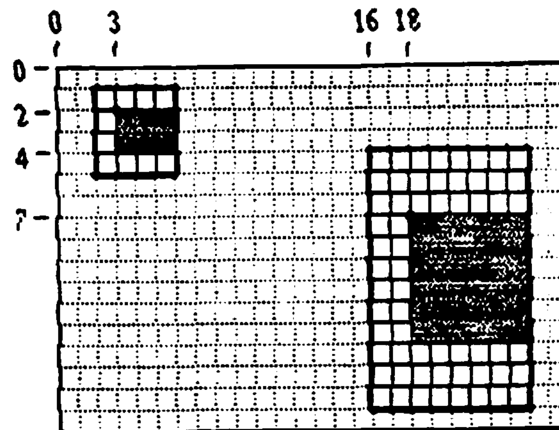
places a striped pattern into the pattern variable stripes.

(eye)

There is no range checking on the size of the destination variable. It's easy to overrun the variable and destroy something if you don't know what you're doing.

PROCEDURE ScalePt (VAR pt: Point; srcRect,dstRect: Rect);

A width and height are passed in pt; the horizontal component of pt is the width, and the vertical component of pt is the height. ScalePt scales these measurements as follows and returns the result in pt: it multiplies the given width by the ratio of dstRect's width to srcRect's width, and multiplies the given height by the ratio of dstRect's height to srcRect's height. In Figure 25, where dstRect's width is twice srcRect's width and its height is three times srcRect's height, the pen width is scaled from 3 to 6 and the pen height is scaled from 2 to 6.



ScalePt scales pen size (3,2) to (6,6).
MapPt maps point (3,2) to (18,7).

Figure 25. ScalePt and MapPt

PROCEDURE MapPt (VAR pt: Point; srcRect,dstRect: Rect);

Given a point within srcRect, MapPt maps it to a similarly located point within dstRect (that is, to where it would fall if it were part of a drawing being expanded or shrunk to fit dstRect). The result is returned in pt. A corner point of srcRect would be mapped to the corresponding corner point of dstRect, and the center of srcRect to the center of dstRect. In Figure 25 above, the point (3,2) in srcRect is mapped to (18,7) in dstRect. FromRect and dstRect may overlap, and pt need not actually be within srcRect.

(eye)

Remember, if you are going to draw inside the rectangle in dstRect, you will probably also want to scale the pen size accordingly with ScalePt.

PROCEDURE MapRect (VAR r: Rect; srcRect,dstRect: Rect);

Given a rectangle within srcRect, MapRect maps it to a similarly located rectangle within dstRect by calling MapPt to map the top left and bottom right corners of the rectangle. The result is returned in r.

PROCEDURE MapRgn (rgn: RgnHandle; srcRect,dstRect: Rect);

Given a region within srcRect, MapRgn maps it to a similarly located region within dstRect by calling MapPt to map all the points in the region.

PROCEDURE MapPoly (poly: PolyHandle; srcRect,dstRect: Rect);

Given a polygon within srcRect, MapPoly maps it to a similarly located polygon within dstRect by calling MapPt to map all the points that define the polygon.

CUSTOMIZING QUICKDRAW OPERATIONS

For each shape that QuickDraw knows how to draw, there are procedures that perform these basic graphic operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the FrameOval, PaintOval, EraseOval, InvertOval, and FillOval procedures all call a low-level routine that draws the oval. For each type of object QuickDraw can draw, including text and lines, there is a pointer to such a routine. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified parameters as necessary.

Other low-level routines that you can install in this way are:

- The procedure that does bit transfer and is called by CopyBits.
- The function that measures the width of text and is called by CharWidth, StringWidth, and TextWidth.
- The procedure that processes picture comments and is called by DrawPicture. The standard such procedure ignores picture comments.
- The procedure that saves drawing commands as the definition of a picture, and the one that retrieves them. This enables the application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

The grafProcs field of a grafPort determines which low-level routines are called; if it contains NIL, the standard routines are called, so that all operations in that grafPort are done in the standard ways described in this manual. You can set the grafProcs field to point to a record of pointers to routines. The data type of grafProcs is QDProcsPtr:

```
TYPE QDProcsPtr = ^QDProcs;
     QDProcs    = RECORD
                    textProc:    QDPtr;  {text drawing}
                    lineProc:    QDPtr;  {line drawing}
                    rectProc:    QDPtr;  {rectangle drawing}
                    rRectProc:   QDPtr;  {roundRect drawing}
                    ovalProc:    QDPtr;  {oval drawing}
                    arcProc:     QDPtr;  {arc/wedge drawing}
                    polyProc:    QDPtr;  {polygon drawing}
                    rgnProc:     QDPtr;  {region drawing}
                    bitsProc:    QDPtr;  {bit transfer}
                    commentProc: QDPtr;  {picture comment processing}
                    txMeasProc:  QDPtr;  {text width measurement}
                    getPicProc:  QDPtr;  {picture retrieval}
                    putPicProc:  QDPtr   {picture saving}
                  END;
```

To assist you in setting up a QDProcs record, QuickDraw provides the following procedure:

```
PROCEDURE SetStdProcs (VAR procs: QDProcs);
```

This procedure sets all the fields of the given QDProcs record to point to the standard low-level routines. You can then change the ones you wish to point to your own routines. For example, if your procedure that processes picture comments is named MyComments, you will store @MyComments in the commentProc field of the QDProcs record.

The routines you install must of course have the same calling sequences as the standard routines, which are described below. The standard drawing routines tell which graphic operation to perform from a parameter of type GrafVerb.

```
TYPE GrafVerb = (frame, paint, erase, invert, fill);
```

When the grafVerb is fill, the pattern to use when filling is passed in the fillPat field of the grafPort.

```
PROCEDURE StdText (byteCount: INTEGER; textBuf: QDPtr; numer,denom:
          INTEGER);
```

StdText is the standard low-level routine for drawing text. It draws text from the arbitrary structure in memory specified by textBuf, starting from the first byte and continuing for byteCount bytes. Numer and denom specify the scaling, if any: numer.v over denom.v gives the vertical scaling, and numer.h over denom.h gives the horizontal scaling.

```
PROCEDURE StdLine (newPt: Point);
```

StdLine is the standard low-level routine for drawing a line. It draws a line from the current pen location to the location specified (in

local coordinates) by newPt.

PROCEDURE StdRect (verb: GrafVerb; r: Rect);

StdRect is the standard low-level routine for drawing a rectangle. It draws the given rectangle according to the specified grafVerb.

PROCEDURE StdRRect (verb: GrafVerb; r: Rect; ovalwidth,ovalHeight:
INTEGER);

StdRRect is the standard low-level routine for drawing a rounded-corner rectangle. It draws the given rounded-corner rectangle according to the specified grafVerb. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

PROCEDURE StdOval (verb: GrafVerb; r: Rect);

StdOval is the standard low-level routine for drawing an oval. It draws an oval inside the given rectangle according to the specified grafVerb.

PROCEDURE StdArc (verb: GrafVerb; r: Rect; startAngle,arcAngle:
INTEGER);

StdArc is the standard low-level routine for drawing an arc or a wedge. It draws an arc or wedge of the oval that fits inside the given rectangle. The grafVerb specifies the graphic operation; if it's the frame operation, an arc is drawn; otherwise, a wedge is drawn.

PROCEDURE StdPoly (verb: GrafVerb; poly: PolyHandle);

StdPoly is the standard low-level routine for drawing a polygon. It draws the given polygon according to the specified grafVerb.

PROCEDURE StdRgn (verb: GrafVerb; rgn: RgnHandle);

StdRgn is the standard low-level routine for drawing a region. It draws the given region according to the specified grafVerb.

PROCEDURE StdBits (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
mode: INTEGER; maskRgn: RgnHandle);

StdBits is the standard low-level routine for doing bit transfer. It transfers a bit image between the given bitMap and thePort^.portBits, just as if CopyBits were called with the same parameters and with a destination bitMap equal to thePort^.portBits.

PROCEDURE StdComment (kind,dataSize: INTEGER; dataHandle: QDHandle);

StdComment is the standard low-level routine for processing a picture comment. Kind identifies the type of comment. DataHandle is a handle to additional data, and dataSize is the size of that data in bytes. If there is no additional data for the command, dataHandle will be NIL and dataSize will be 0. StdComment simply ignores the comment.

FUNCTION StdTxMeas (byteCount: INTEGER; textBuf: QDPtr; VAR numer,denom: Point; VAR info: FontInfo) : INTEGER;

StdTxMeas is the standard low-level routine for measuring text width. It returns the width of the text stored in the arbitrary structure in memory specified by textBuf, starting with the first byte and continuing for byteCount bytes. Numer and denom specify the scaling as in the StdText procedure; note that StdTxMeas may change them.

PROCEDURE StdGetPic (dataPtr: QDPtr; byteCount: INTEGER);

StdGetPic is the standard low-level routine for retrieving information from the definition of a picture. It retrieves the next byteCount bytes from the definition of the currently open picture and stores them in the data structure pointed to by dataPtr.

PROCEDURE StdPutPic (dataPtr: QDPtr; byteCount: INTEGER);

StdPutPic is the standard low-level routine for saving information as the definition of a picture. It saves as the definition of the currently open picture the drawing commands stored in the data structure pointed to by dataPtr, starting with the first byte and continuing for the next byteCount bytes.

USING QUICKDRAW FROM ASSEMBLY LANGUAGE

All Macintosh User Interface Toolbox routines can be called from assembly-language programs as well as from Pascal. When you write an assembly-language program to use these routines, though, you must emulate Pascal's parameter passing and variable transfer protocols.

This section discusses how to use the QuickDraw constants, global variables, data types, procedures, and functions from assembly language.

The primary aid to assembly-language programmers is a file named GRAFTYPES.TEXT. If you use .INCLUDE to include this file when you assemble your program, all the QuickDraw constants, offsets to locations of global variables, and offsets into the fields of structured types will be available in symbolic form.

Constants

QuickDraw constants are stored in the GRAFTYPES.TEXT file, and you can use the constant values symbolically. For example, if you've loaded the effective address of the thePort^.txMode field into address register A2, you can set that field to the srcXor mode with this statement:

```
MOVE.W  #SRCXOR,(A2)
```

To refer to the number of bytes occupied by the QuickDraw global variables, you can use the constant GRAFSIZE. When you call the InitGraf procedure, you must pass a pointer to an area at least that large.

Data Types

Pascal's strong typing ability lets you write Pascal programs without really considering the size of a variable. But in assembly language, you must keep track of the size of every variable. The sizes of the standard Pascal data types are as follows:

Type	Size
INTEGER	Word (2 bytes)
LongInt	Long (4 bytes)
BOOLEAN	Word (2 bytes)
CHAR	Word (2 bytes)
REAL	Long (4 bytes)

INTEGERs and LongInts are in two's complement form; BOOLEANs have their boolean value in bit 8 of the word (the low-order bit of the byte at the same location); CHARs are stored in the high-order byte of the word; and REALs are in the KCS standard format.

The QuickDraw simple data types listed below are constructed out of these fundamental types.

Type	Size
QDPtr	Long (4 bytes)
QDHandle	Long (4 bytes)
Word	Long (4 bytes)
Str255	Page (256 bytes)
Pattern	8 bytes
Bits16	32 bytes

Other data types are constructed as records of variables of the above types. The size of such a type is the sum of the sizes of all the fields in the record; the fields appear in the variable with the first field in the lowest address. For example, consider the data type BitMap, which is defined like this:

```
TYPE BitMap = RECORD
                baseAddr: QDPtr;
                rowBytes: INTEGER;
                bounds:   Rect
              END;
```

This data type would be arranged in memory as seven words: a long for the baseAddr, a word for the rowBytes, and four words for the top, left, right, and bottom parts of the bounds rectangle. To assist you in referring to the fields inside a variable that has a structure like this, the GRAFTYPES.TEXT file defines constants that you can use as offsets into the fields of a structured variable. For example, to move a bitMap's rowBytes value into D3, you would execute the following instruction:

```
MOVE.W  MYBITMAP+ROWBYTES,D3
```

Displacements are given in the GRAFTYPES.TEXT file for all fields of all data types defined by QuickDraw.

To do double indirection, you perform an LEA indirectly to obtain the effective address from the handle. For example, to get at the top coordinate of a region's enclosing rectangle:

```
MOVE.L  MYHANDLE,A1             ; Load handle into A1
MOVE.L  (A1),A1                 ; Use handle to get pointer
MOVE.W  RGNBBOX+TOP(A1),D3      ; Load value using pointer
```

(eye)

For regions (and all other variable-length structures with handles), you must not move the pointer into a register once and just continue to use that pointer; you must do the double indirection each time. Every QuickDraw, Toolbox, or memory management call you make can possibly trigger a heap compaction that renders all pointers to movable heap items (like regions) invalid. The handles will remain valid, but pointers you've obtained through handles can be rendered invalid at any subroutine call or trap in your program.

Global Variables

Global variables are stored in a special section of Macintosh low memory; register A5 always points to this section of memory. The GRAFTYPES.TEXT file defines a constant GRAFGLOB that points to the beginning of the QuickDraw variables in this space, and other constants that point to the individual variables. To access one of the variables, put GRAFGLOB in an address register, sum the constants, and index off of that register. For example, if you want to know the horizontal coordinate of the pen location for the current grafPort, which the global variable thePort points to, you can give the following instructions:

```
MOVE.L  GRAFGLOB(A5),AØ       ; Point to QuickDraw globals
MOVE.L  THEPORT(AØ),A1        ; Get current grafPort
MOVE.W  PNLOC+H(A1),DØ        ; Get thePort^.pnLoc.h
```

Procedures and Functions

To call a QuickDraw procedure or function, you must push all parameters to it on the stack, then JSR to the function or procedure. When you link your program with QuickDraw, these JSRs are adjusted to refer to the jump table in low RAM, so that a JSR into the table redirects you to the actual location of the procedure or function.

The only difficult part about calling QuickDraw procedures and functions is stacking the parameters. You must follow some strict rules:

- Save all registers you wish to preserve BEFORE you begin pushing parameters. Any QuickDraw procedure or function can destroy the contents of the registers AØ, A1, DØ, D1, and D2, but the others are never altered.

- Push the parameters in the order that they appear in the Pascal procedural interface.

- For booleans, push a byte; for integers and characters, push a word; for pointers, handles, long integers, and reals, push a long.

- For any structured variable longer than four (4) bytes, push a pointer to the variable.

- For all VAR parameters, regardless of size, push a pointer to the variable.

- When calling a function, FIRST push a null entry equal to the size of the function result, THEN push all other parameters. The result will be left on the stack after the function returns to you.

This makes for a lengthy interface, but it also guarantees that you can mock up a Pascal version of your program, and later translate it into assembly code that works the same. For example, the Pascal statement

```
blackness := GetPixel(5Ø,mousePos.v);
```

would be written in assembly language like this:

```
CLR.W   -(SP)                 ; Save space for boolean result
MOVE.W  #5Ø,-(SP)             ; Push constant 5Ø (decimal)
MOVE.W  MOUSEPOS+V,-(SP)      ; Push the value of mousePos.v
JSR     GETPIXEL              ; Call routine
MOVE.W  (SP)+,BLACKNESS       ; Fetch result from stack
```

This is a simple example, pushing and pulling word-long constants. Normally, you'll be pushing more pointers, using the PEA (Push Effective Address) instruction:

```
FillRoundRect(myRect,1,thePort^.pnSize.v,white);

PEA     MYRECT               ; Push pointer to myRect
MOVE.W  #1,-(SP)             ; Push constant 1
MOVE.L  GRAFGLOB(A5),A0      ; Point to QuickDraw globals
MOVE.L  THEPORT(A0),A1       ; Get current grafPort
MOVE.W  PNSIZE+V(A1),-(SP)   ; Push value of thePort^.pnSize.v
PEA     WHITE(A0)            ; Push pointer to global variable white
JSR     FILLROUNDRECT        ; Call the subroutine
```

To call the TextFace procedure, push a word in which each of seven bits represents a stylistic variation: set bit 0 for bold, bit 1 for italic, bit 2 for underline, bit 3 for outline, bit 4 for shadow, bit 5 for condense, and bit 6 for extend.

SUMMARY OF QUICKDRAW

```
CONST srcCopy     = 0;
      srcOr       = 1;
      srcXor      = 2;
      srcBic      = 3;
      notSrcCopy  = 4;
      notSrcOr    = 5;
      notSrcXor   = 6;
      notSrcBic   = 7;
      patCopy     = 8;
      patOr       = 9;
      patXor      = 10;
      patBic      = 11;
      notPatCopy  = 12;
      notPatOr    = 13;
      notPatXor   = 14;
      notPatBic   = 15;

      blackColor   = 33;
      whiteColor   = 30;
      redColor     = 205;
      greenColor   = 341;
      blueColor    = 409;
      cyanColor    = 273;
      magentaColor = 137;
      yellowColor  = 69;

      picLParen = 0;
      picRParen = 1;

TYPE QDByte    = -128..127;
     QDPtr     = ^QDByte;
     QDHandle  = ^QDPtr;
     Str255    = STRING[255];
     Pattern   = PACKED ARRAY [0..7] OF 0..255;
     Bits16    = ARRAY [0..15] OF INTEGER;
     GrafVerb  = (frame, paint, erase, invert, fill);

     StyleItem = (bold, italic, underline, outline, shadow, condense,
                  extend);
     Style      = SET OF StyleItem;

     FontInfo = RECORD
                  ascent:  INTEGER;
                  descent: INTEGER;
                  widMax:  INTEGER;
                  leading: INTEGER
                END;
```

```
VHSelect = (v,h);
Point    = RECORD CASE INTEGER OF

              Ø: (v: INTEGER;
                  h: INTEGER);

              1: (vh: ARRAY[VHSelect] OF INTEGER)

            END;

Rect = RECORD CASE INTEGER OF

         Ø: (top:      INTEGER;
             left:     INTEGER;
             bottom:   INTEGER;
             right:    INTEGER);

         1: (topLeft:  Point;
             botRight: Point)

       END;

BitMap = RECORD
           baseAddr: QDPtr;
           rowBytes: INTEGER;
           bounds:   Rect
         END;

Cursor = RECORD
           data:    Bits16;
           mask:    Bits16;
           hotSpot: Point
         END;

PenState = RECORD
             pnLoc:    Point;
             pnSize:   Point;
             pnMode:   INTEGER;
             pnPat:    Pattern
           END;

RgnHandle = ^RgnPtr;
RgnPtr    = ^Region;
Region    =  RECORD
               rgnSize:   INTEGER;
               rgnBBox:   Rect;
               {more data if not rectangular}
             END;
```

```
    PicHandle = ^PicPtr;
    PicPtr    = ^Picture;
    Picture   = RECORD
                  picSize:  INTEGER;
                  picFrame: Rect;
                  {picture definition data}
                END;

    PolyHandle = ^PolyPtr;
    PolyPtr    = ^Polygon;
    Polygon    = RECORD
                   polySize:   INTEGER;
                   polyBBox:   Rect;
                   polyPoints: ARRAY [0..0] OF Point
                 END;

QDProcsPtr = ^QDProcs;
QDProcs    = RECORD
               textProc:    QDPtr;
               lineProc:    QDPtr;
               rectProc:    QDPtr;
               rRectProc:   QDPtr;
               ovalProc:    QDPtr;
               arcProc:     QDPtr;
               polyProc:    QDPtr;
               rgnProc:     QDPtr;
               bitsProc:    QDPtr;
               commentProc: QDPtr;
               txMeasProc:  QDPtr;
               getPicProc:  QDPtr;
               putPicProc:  QDPtr
             END;
```

```
     GrafPtr  = ^GrafPort;
     GrafPort = RECORD
                  device:     INTEGER;
                  portBits:   BitMap;
                  portRect:   Rect;
                  visRgn:     RgnHandle;
                  clipRgn:    RgnHandle;
                  bkPat:      Pattern;
                  fillPat:    Pattern;
                  pnLoc:      Point;
                  pnSize:     Point;
                  pnMode:     INTEGER;
                  pnPat:      Pattern;
                  pnVis:      INTEGER;
                  txFont:     INTEGER;
                  txFace:     Style;
                  txMode:     INTEGER;
                  txSize:     INTEGER;
                  spExtra:    INTEGER;
                  fgColor:    LongInt;
                  bkColor:    LongInt;
                  colrBit:    INTEGER;
                  patStretch: INTEGER;
                  picSave:    QDHandle;
                  rgnSave:    QDHandle;
                  polySave:   QDHandle;
                  grafProcs:  QDProcsPtr
                END;

VAR thePort:     GrafPtr;
    white:       Pattern;
    black:       Pattern;
    gray:        Pattern;
    ltGray:      Pattern;
    dkGray:      Pattern;
    arrow:       Cursor;
    screenBits:  BitMap;
    randSeed:    LongInt;
```

GrafPort Routines

```
PROCEDURE InitGraf    (globalPtr: QDPtr);
PROCEDURE OpenPort    (gp: GrafPtr);
PROCEDURE InitPort    (gp: GrafPtr);
PROCEDURE ClosePort   (gp: GrafPtr);
PROCEDURE SetPort     (gp: GrafPtr);
PROCEDURE GetPort     (VAR gp: GrafPtr);
PROCEDURE GrafDevice  (device: INTEGER);
PROCEDURE SetPortBits (bm: BitMap);
PROCEDURE PortSize    (width,height: INTEGER);
PROCEDURE MovePortTo  (leftGlobal,topGlobal: INTEGER);
PROCEDURE SetOrigin   (h,v: INTEGER);
```

```
PROCEDURE SetClip      (rgn: RgnHandle);
PROCEDURE GetClip      (rgn: RgnHandle);
PROCEDURE ClipRect     (r: Rect);
PROCEDURE BackPat      (pat: Pattern);
```

Cursor Handling

```
PROCEDURE InitCursor;
PROCEDURE SetCursor       (crsr: Cursor);
PROCEDURE HideCursor;
PROCEDURE ShowCursor;
PROCEDURE ObscureCursor;
```

Pen and Line Drawing

```
PROCEDURE HidePen;
PROCEDURE ShowPen;
PROCEDURE GetPen       (VAR pt: Point);
PROCEDURE GetPenState (VAR pnState: PenState);
PROCEDURE SetPenState (pnState: PenState);
PROCEDURE PenSize      (width,height: INTEGER);
PROCEDURE PenMode      (mode: INTEGER);
PROCEDURE PenPat       (pat: Pattern);
PROCEDURE PenNormal;
PROCEDURE MoveTo       (h,v: INTEGER);
PROCEDURE Move         (dh,dv: INTEGER);
PROCEDURE LineTo       (h,v: INTEGER);
PROCEDURE Line         (dh,dv: INTEGER);
```

Text Drawing

```
PROCEDURE TextFont     (font: INTEGER);
PROCEDURE TextFace     (face: Style);
PROCEDURE TextMode     (mode: INTEGER);
PROCEDURE TextSize     (size: INTEGER);
PROCEDURE SpaceExtra   (extra: INTEGER);
PROCEDURE DrawChar     (ch: CHAR);
PROCEDURE DrawString   (s: Str255);
PROCEDURE DrawText     (textBuf: QDPtr; firstByte,byteCount: INTEGER);
FUNCTION  CharWidth    (ch: CHAR) : INTEGER;
FUNCTION  StringWidth (s: Str255) : INTEGER;
FUNCTION  TextWidth    (textBuf: QDPtr; firstByte,byteCount: INTEGER) :
                        INTEGER;
PROCEDURE GetFontInfo (VAR info: FontInfo);
```

Drawing in Color

```
PROCEDURE ForeColor (color: LongInt);
PROCEDURE BackColor (color: LongInt);
PROCEDURE ColorBit  (whichBit: INTEGER);
```

Calculations with Rectangles

```
PROCEDURE SetRect    (VAR r: Rect; left,top,right,bottom: INTEGER);
PROCEDURE OffsetRect (VAR r: Rect; dh,dv: INTEGER);
PROCEDURE InsetRect  (VAR r: Rect; dh,dv: INTEGER);
FUNCTION  SectRect   (srcRectA,srcRectB: Rect; VAR dstRect: Rect) :
                      BOOLEAN;
PROCEDURE UnionRect  (srcRectA,srcRectB: Rect; VAR dstRect: Rect)
FUNCTION  PtInRect   (pt: Point; r: Rect) : BOOLEAN;
PROCEDURE Pt2Rect    (ptA,ptB: Point; VAR dstRect: Rect);
PROCEDURE PtToAngle  (r: Rect; pt: Point; VAR angle: INTEGER);
FUNCTION  EqualRect  (rectA,rectB: Rect) : BOOLEAN;
FUNCTION  EmptyRect  (r: Rect) : BOOLEAN;
```

Graphic Operations on Rectangles

```
PROCEDURE FrameRect  (r: Rect);
PROCEDURE PaintRect  (r: Rect);
PROCEDURE EraseRect  (r: Rect);
PROCEDURE InvertRect (r: Rect);
PROCEDURE FillRect   (r: Rect; pat: Pattern);
```

Graphic Operations on Ovals

```
PROCEDURE FrameOval  (r: Rect);
PROCEDURE PaintOval  (r: Rect);
PROCEDURE EraseOval  (r: Rect);
PROCEDURE InvertOval (r: Rect);
PROCEDURE FillOval   (r: Rect; pat: Pattern);
```

Graphic Operations on Rounded-Corner Rectangles

```
PROCEDURE FrameRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE PaintRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE EraseRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE InvertRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE FillRoundRect   (r: Rect; ovalWidth,ovalHeight: INTEGER;
                            pat: Pattern);
```

Graphic Operations on Arcs and Wedges

```
PROCEDURE FrameArc  (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE PaintArc  (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE EraseArc  (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE InvertArc (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE FillArc   (r: Rect; startAngle,arcAngle: INTEGER; pat:
                     Pattern);
```

Calculations with Regions

```
FUNCTION  NewRgn :     RgnHandle;
PROCEDURE DisposeRgn  (rgn: RgnHandle);
PROCEDURE CopyRgn     (srcRgn,dstRgn: RgnHandle);
PROCEDURE SetEmptyRgn (rgn: RgnHandle);
PROCEDURE SetRectRgn  (rgn: RgnHandle; left,top,right,bottom: INTEGER);
PROCEDURE RectRgn     (rgn: RgnHandle; r: Rect);
PROCEDURE OpenRgn;
PROCEDURE CloseRgn    (dstRgn: RgnHandle);
PROCEDURE OffsetRgn   (rgn: RgnHandle; dh,dv: INTEGER);
PROCEDURE InsetRgn    (rgn: RgnHandle; dh,dv: INTEGER);
PROCEDURE SectRgn     (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE UnionRgn    (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE DiffRgn     (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE XorRgn      (srcRgnA,srcRgnB,dstRgn: RgnHandle);
FUNCTION  PtInRgn     (pt: Point; rgn: RgnHandle) : BOOLEAN;
FUNCTION  RectInRgn   (r: Rect; rgn: RgnHandle) : BOOLEAN;
FUNCTION  EqualRgn    (rgnA,rgnB: RgnHandle) : BOOLEAN;
FUNCTION  EmptyRgn    (rgn: RgnHandle) : BOOLEAN;
```

Graphic Operations on Regions

```
PROCEDURE FrameRgn  (rgn: RgnHandle);
PROCEDURE PaintRgn  (rgn: RgnHandle);
PROCEDURE EraseRgn  (rgn: RgnHandle);
PROCEDURE InvertRgn (rgn: RgnHandle);
PROCEDURE FillRgn   (rgn: RgnHandle; pat: Pattern);
```

Bit Transfer Operations

```
PROCEDURE ScrollRect (r: Rect; dh,dv: INTEGER; updateRgn: RgnHandle);
PROCEDURE CopyBits   (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
                      mode: INTEGER; maskRgn: RgnHandle);
```

Pictures

```
FUNCTION  OpenPicture   (picFrame: Rect) : PicHandle;
PROCEDURE PicComment    (kind,dataSize: INTEGER; dataHandle: QDHandle);
PROCEDURE ClosePicture;
PROCEDURE DrawPicture   (myPicture: PicHandle; dstRect: Rect);
PROCEDURE KillPicture   (myPicture: PicHandle);
```

Calculations with Polygons

```
FUNCTION  OpenPoly :  PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE KillPoly   (poly: PolyHandle);
PROCEDURE OffsetPoly (poly: PolyHandle; dh,dv: INTEGER);
```

Graphic Operations on Polygons

```
PROCEDURE FramePoly  (poly: PolyHandle);
PROCEDURE PaintPoly  (poly: PolyHandle);
PROCEDURE ErasePoly  (poly: PolyHandle);
PROCEDURE InvertPoly (poly: PolyHandle);
PROCEDURE FillPoly   (poly: PolyHandle; pat: Pattern);
```

Calculations with Points

```
PROCEDURE AddPt         (srcPt: Point; VAR dstPt: Point);
PROCEDURE SubPt         (srcPt: Point; VAR dstPt: Point);
PROCEDURE SetPt         (VAR pt: Point; h,v: INTEGER);
FUNCTION  EqualPt       (ptA,ptB: Point) : BOOLEAN;
PROCEDURE LocalToGlobal (VAR pt: Point);
PROCEDURE GlobalToLocal (VAR pt: Point);
```

Miscellaneous Utilities

```
FUNCTION  Random :  INTEGER;
FUNCTION  GetPixel (h,v: INTEGER) : BOOLEAN;
PROCEDURE StuffHex (thingPtr: QDPtr; s: Str255);
PROCEDURE ScalePt  (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapPt    (VAR pt: Point; srcRect,dstRect: Rect);
PROCEDURE MapRect  (VAR r: Rect; srcRect,dstRect: Rect);
PROCEDURE MapRgn   (rgn: RgnHandle; srcRect,dstRect: Rect);
PROCEDURE MapPoly  (poly: PolyHandle; srcRect,dstRect: Rect);
```

Customizing QuickDraw Operations

```
PROCEDURE SetStdProcs (VAR procs: QDProcs);
PROCEDURE StdText     (byteCount: INTEGER; textAddr: QDPtr; numer,denom:
                       Point);
PROCEDURE StdLine     (newPt: Point);
PROCEDURE StdRect     (verb: GrafVerb; r: Rect);
PROCEDURE StdRRect    (verb: GrafVerb; r: Rect; ovalwidth,ovalHeight:
                       INTEGER);
PROCEDURE StdOval     (verb: GrafVerb; r: Rect);
PROCEDURE StdArc      (verb: GrafVerb; r: Rect; startAngle,arcAngle:
                       INTEGER);
PROCEDURE StdPoly     (verb: GrafVerb; poly: PolyHandle);
PROCEDURE StdRgn      (verb: GrafVerb; rgn: RgnHandle);
PROCEDURE StdBits     (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
                       mode: INTEGER; maskRgn: RgnHandle);
PROCEDURE StdComment  (kind,dataSize: INTEGER; dataHandle: QDHandle);
FUNCTION  StdTxMeas   (byteCount: INTEGER; textBuf: QDPtr; VAR numer,
                       denom: Point; VAR info: FontInfo) : INTEGER;
PROCEDURE StdGetPic   (dataPtr: QDPtr; byteCount: INTEGER);
PROCEDURE StdPutPic   (dataPtr: QDPtr; byteCount: INTEGER);
```

GLOSSARY

bit image: A collection of bits in memory which have a rectilinear representation. The Macintosh screen is a visible bit image.

bitMap: A pointer to a bit image, the row width of that image, and its boundary rectangle.

boundary rectangle: A rectangle defined as part of a bitMap, which encloses the active area of the bit image and imposes a coordinate system on it. Its top left corner is always aligned around the first bit in the bit image.

character style: A set of stylistic variations, such as bold, italic, and underline. The empty set indicates normal text (no stylistic variations).

clipping: Limiting drawing to within the bounds of a particular area.

clipping region: Same as clipRgn.

clipRgn: The region to which an application limits drawing in a grafPort.

coordinate plane: A two-dimensional grid. In QuickDraw, the grid coordinates are integers ranging from -32768 to +32767, and all grid lines are infinitely thin.

cursor: A 16-by-16-bit image that appears on the screen and is controlled by the mouse; called the "pointer" in other Macintosh documentation.

cursor level: A value, initialized to Ø when the system is booted, that keeps track of the number of times the cursor has been hidden.

empty: Containing no bits, as a shape defined by only one point.

font: The complete set of characters of one typeface, such as Helvetica.

frame: To draw a shape by drawing an outline of it.

global coordinate system: The coordinate system based on the top left corner of the bit image being at (Ø,Ø).

grafPort: A complete drawing environment, including such elements as a bitMap, a subset of it in which to draw, a character font, patterns for drawing and erasing, and other pen characteristics.

grafPtr: A pointer to a grafPort.

handle: A pointer to one master pointer to a dynamic, relocatable data structure (such as a region).

hotSpot: The point in a cursor that is aligned with the mouse position.

kern: To stretch part of a character back under the previous character.

local coordinate system: The coordinate system local to a grafPort, imposed by the boundary rectangle defined in its bitMap.

missing symbol: A character to be drawn in case of a request to draw a character that is missing from a particular font.

pattern: An 8-by-8-bit image, used to define a repeating design (such as stripes) or tone (such as gray).

pattern transfer mode: One of eight transfer modes for drawing lines or shapes with a pattern.

picture: A saved sequence of QuickDraw drawing commands (and, optionally, picture comments) that you can play back later with a single procedure call; also, the image resulting from these commands.

picture comments: Data stored in the definition of a picture which does not affect the picture's appearance but may be used to provide additional information about the picture when it's played back.

picture frame: A rectangle, defined as part of a picture, which surrounds the picture and gives a frame of reference for scaling when the picture is drawn.

pixel: The visual representation of a bit on the screen (white if the bit is Ø, black if it's 1).

point: The intersection of a horizontal grid line and a vertical grid line on the coordinate plane, defined by a horizontal and a vertical coordinate.

polygon: A sequence of connected lines, defined by QuickDraw line-drawing commands.

port: Same as grafPort.

portBits: The bitMap of a grafPort.

portBits.bounds: The boundary rectangle of a grafPort's bitMap.

portRect: A rectangle, defined as part of a grafPort, which encloses a subset of the bitMap for use by the grafPort.

region: An arbitrary area or set of areas on the coordinate plane. The outline of a region should be one or more closed loops.

row width: The number of bytes in each row of a bit image.

solid: Filled in with any pattern.

source transfer mode: One of eight transfer modes for drawing text or transferring any bit image between two bitMaps.

style: See character style.

thePort: A global variable that points to the current grafPort.

transfer mode: A specification of which boolean operation QuickDraw should perform when drawing or when transferring a bit image from one bitMap to another.

visRgn: The region of a grafPort, manipulated by the Window Manager, which is actually visible on the screen.

MACINTOSH USER EDUCATION

The Resource Manager: A Programmer's Guide /RMGR/RESOURCE

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:	First Draft (ROM 2.0)	Caroline Rose	2/2/83
	Second Draft (ROM 4)	Caroline Rose	6/21/83
	Third Draft (ROM 7)	Caroline Rose	10/3/83
	Errata added	Caroline Rose	3/8/84

ABSTRACT

Macintosh applications make use of many resources, such as menus, fonts, and icons. These resources are stored in resource files separately from the application code, for flexibility and ease of maintenance. This manual describes resource files and the Resource Manager routines.

Errata:

The low-order bit of the resource attribute byte is no longer available for use by your application; it's now reserved for internal use by the Resource Manager.

There's a new function:

```
FUNCTION SizeResource (theResource: Handle) : INTEGER;
```

Given a handle to a resource, SizeResource returns the size of the resource in bytes. If the resource isn't in memory, the size is read from the resource file. If the given handle isn't a handle to a resource, SizeResource will return -1 and the ResError function will return the error code resNotFound.

TABLE OF CONTENTS

3 About This Manual
4 About the Resource Manager
6 Overview of Resource Files
8 Resource Specification
10 Resource References
13 Using the Resource Manager
15 Resource Manager Routines
15 Initializing the Resource Manager
16 Opening and Closing Resource Files
17 Checking for Errors
18 Setting the Current Resource File
18 Getting Resource Types
19 Getting and Disposing of Resources
22 Getting Resource Information
23 Modifying Resources
28 Advanced Routines
29 Resources within Resources
31 Format of a Resource File
33 Notes for Assembly-Language Programmers
35 Summary of the Resource Manager
37 Summary of the Resource File Format
38 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved. Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Resource Manager, the part of the Macintosh User Interface Toolbox through which an application accesses various resources that it uses, such as menus, fonts, and icons. *** Eventually it will become part of a large manual describing the entire Toolbox. *** It discusses resource files, where resources are stored. Resources form the foundation of every Macintosh application; even the application's code is a resource. In a resource file, the resources used by the application are stored separately from the code for flexibility and ease of maintenance.

- You can use an existing program for creating and editing resource files, or write one of your own. These programs will call Resource Manager routines.

- Usually you'll access resources indirectly through other Toolbox units, such as the Menu Manager and the Font Manager, which in turn call the Resource Manager to do the low-level resource operations. In some cases, you may need to call a Resource Manager file-opening routine and possibly other routines to access resources directly.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Resource Manager and the file system may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic functions of the Finder, which are performed with the help of the Resource Manager. (To the user, the Finder is known as the Desktop Manager.)

- The Operating System error codes.

- The Macintosh file system, as documented *** though probably not up-to-date *** in the Macintosh Operating System Reference Manual. You need to know about this only if you want to understand exactly how resources are implemented internally; you don't have to know it to be able to use the Resource Manager.

If you're going to write your own program to create and edit resource files, you also need to know the exact format of each type of resource. The documentation for the Toolbox unit that deals with a particular type of resource will tell you what you need to know for that resource.

This manual begins with an introduction to the Resource Manager and resources, an overview of resource files, and a discussion of resource specification, all of which offer useful general information. The next

section deals with resource references; you can skip it if you're only going to access resources through other Toolbox units.

Next, a section on using the Resource Manager introduces you to its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Resource Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. A discussion of how resources point to each other is followed by a section giving the exact format of a resource file. *** Also, to be removed eventually: notes for programmers who will use the Resource Manager routines from assembly language. ***

Finally, there's a summary of the Resource Manager data structures and routine calls and a summary of the resource file format, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE RESOURCE MANAGER

Macintosh applications make use of many resources, such as menus, fonts, and icons, which are stored in resource files. For example, an icon resides in a resource file as a 32-by-32 bit image, and a font as a large bit image containing the characters of the font. In some cases the resource consists of descriptive information (such as, for a menu, the menu title, the text of each command in the menu, whether the command is checked with a check mark, and so on). The Resource Manager keeps track of resources in resource files and provides routines that allow applications and other Toolbox units to access them.

There's a resource file associated with each application, containing the resources specific to that application; these resources include the application code itself. There's also a system resource file, which contains standard resources shared by all applications (also called system resources).

The resources used by an application are created and changed separately from the application's code. This separation is the main advantage to having resource files. A change in the title of a menu, for example, won't require any recompilation of code, nor will translation to a foreign language.

The Resource Manager is initialized by the system when it starts up, and the system resource file is opened as part of the initialization. Your application's resource file is opened when the application starts up. When instructed to get a certain resource, the Resource Manager normally looks first in the application's resource file and then, if the search isn't successful, in the system resource file. This makes it easy to share resources among applications and also to override a system resource with one of your own (if you want to use something other than a standard icon in an alert box, for example).

You refer to a resource by passing the Resource Manager a resource specification, which consists of a type and either an ID number or a name. Any resource type is valid, whether one of those reserved by the Toolbox (such as for menus and fonts) or a type created for use by your application. Given a resource specification, the Resource Manager will read the resource into memory and return a handle to it.

(eye)

> The Resource Manager knows nothing about the formats of the individual types of resources. Only the routines in other Toolbox units that call the Resource Manager have this knowledge.

While most access to resources is read-only, certain applications may want to modify resources. You can change the content of a resource or its ID number, name, or other attributes--everything except its type. For example, you can designate whether the resource should be kept in memory or whether, as is normal for large resources, it can be removed from memory and read in again when needed. You can change existing resources, remove resources from the resource file altogether, or add new resources to the file.

Not only can an application's resource file contain resources themselves, but it may also contain references to resources in the system resource file. These references need not be in the application's resource file in order for the system resources to be found, because the system resource file will be searched anyway as part of the normal search process; however, the references do serve other purposes. One is that a reference can have a different name than the system resource itself, thus providing an "alias" for the resource. But more important, these references let the Finder know what resources the application uses, thus ensuring that those resources will accompany the application if you should copy it to a disk that has a different system resource file on it. References to system resources can be added or removed with Resource Manager routines.

Resource files are not limited to applications; anything stored in a file can have its own resources. For example, documents usually have resource files containing references to the system resources they use, such as fonts and icons. As in an application's resource file, these references tell the Finder what resources the document uses. An unusual font used in only one document can be included in the resource file for that document rather than in the system resource file.

(hand)

> Although shared resources are usually stored in the system resource file, you can have other resource files that contain resources shared by two or more applications (or documents, or whatever). In this case, however, the Finder will know nothing about the connection between the shared resources and the files that use them.

A number of resource files may be open at one time; the Resource Manager always searches the files in the reverse of the order that they

were opened. Since the system resource file is opened when the Resource Manager is initialized, it's always searched last. Usually the search starts with the most recently opened resource file, but you can change it to start with a file that was opened earlier. (See Figure 1.)

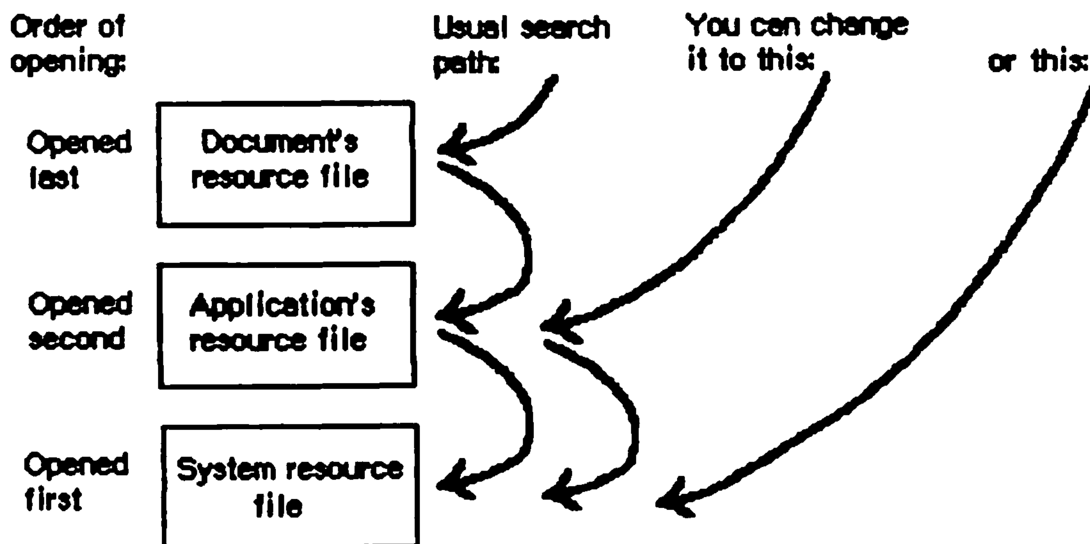


Figure 1. Resource File Searching

OVERVIEW OF RESOURCE FILES

Resources may be put in a resource file with the aid of the Resource Editor, which is documented *** nowhere right now, because it isn't yet available. Meanwhile, you can use the Resource Compiler. You describe the resources in a text file that the Resource Compiler uses to generate the resource file. The exact format of the input file to the Resource Compiler is given in the manual "Putting Together a Macintosh Application". ***

A resource file is not a file in the strictest sense. Although it's functionally like a file in many ways, it's actually just one of two parts, or "forks", of a file. (See Figure 2.) Every file has a resource fork and a data fork (either of which may be empty). The resource fork of an application file contains not only the resources used by the application but also the application code. The code is divided into different segments, each of which is a resource; this allows various parts of the program to be loaded and purged dynamically. The data fork of an application file initially contains nothing, but the application may store data there if desired, by using the Operating System file I/O routines. All data related to resources is stored in the resource fork via the Resource Manager.

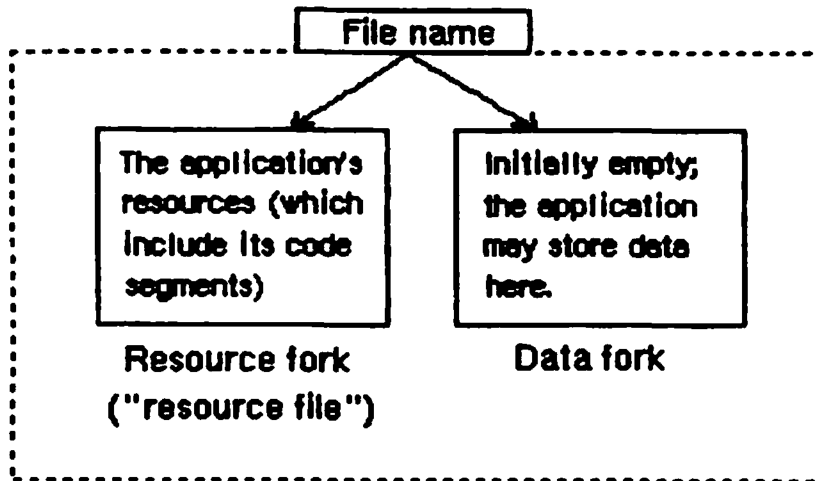


Figure 2. An Application File

As shown in Figure 3, the system resource file has this same structure. The resource fork contains the system resources and the data fork contains the RAM-based Operating System routines. Figure 3 also shows the structure of a file containing a document; the resource fork contains the document's resources and the data fork contains the data that comprises the document.

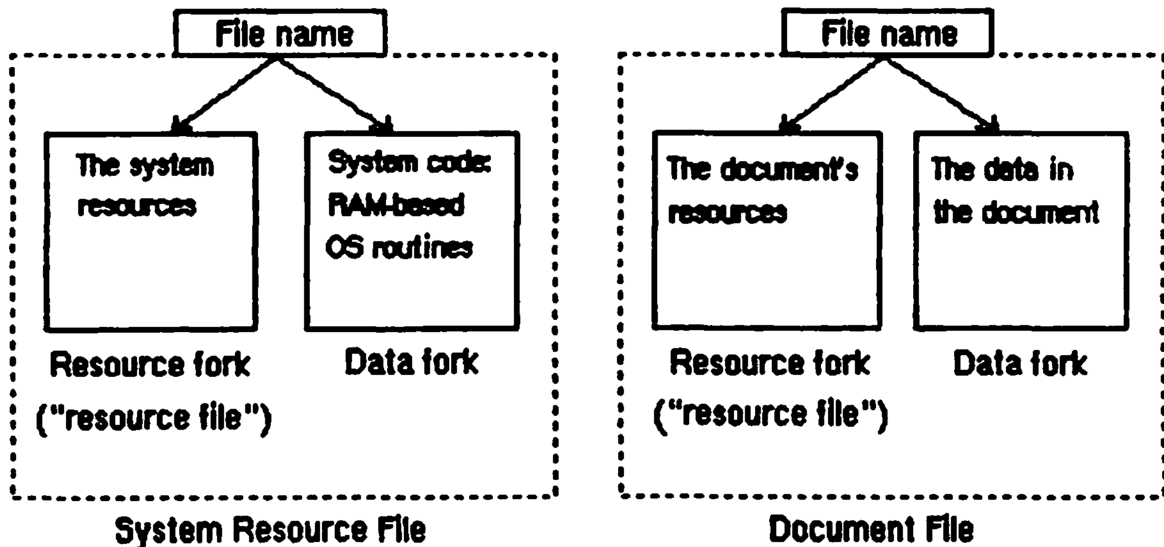


Figure 3. Other Files

To open a resource file, the Resource Manager calls the appropriate Operating System routine and returns the reference number it gets from the Operating System. This is a number greater than 0 by which you can refer to the file when calling other Resource Manager routines. Most of the routines, however, don't have such a parameter; instead, they assume that the current resource file is where they should perform their operation (or begin it, in the case of a search for a resource). The current resource file is the last one that was opened unless you

specify otherwise.

A resource file consists primarily of resource data and a resource map. The resource data consists of the resources themselves (for example, the bit image for an icon or the descriptive information for a menu). The resource map provides the connection between a resource specification and the corresponding resource data. It's like the index of a book; the Resource Manager looks up the resource you specify in the resource map and learns where its resource data is located. The resource map leads to a resource in the same file as the map or provides a reference to a system resource.

The resource map is read into memory when the file is opened and remains there until the file is closed. Notice that although we say the Resource Manager searches resource files, it actually searches the resource maps that were read into memory, and not the resource files on the disk.

Resource data is normally read into memory when needed, though you can specify that it be read in as soon as the resource file is opened. Once read in, the data for a particular resource may or may not be kept in memory, depending on an attribute of that resource that's specified in the resource map. Resources consisting of a relatively large amount of data are usually designated as purgeable, meaning they may be removed from the heap (purged) when space is required by the Memory Manager. Before accessing such a resource through its handle, you can ask the Resource Manager to read the resource into memory again if it was purged.

(hand)

> Programmers concerned about the amount of available memory should be aware that there's a 12-byte overhead in the resource map for every resource and an additional 12-byte overhead for memory management if the resource is read into memory.

To modify a resource, you change the resource data or resource map in memory. The change becomes permanent only at your explicit request, and then only when the application terminates or when you call a routine specifically for updating or closing the resource file.

Each resource file also contains a partial copy of the file's directory entry, written and used by the Finder, and up to 128 bytes of any data the application wishes to store there.

RESOURCE SPECIFICATION

In a resource file, every resource is assigned a type, an ID number, and optionally a name. When calling a Resource Manager routine to access a resource, you specify the resource by passing its type and either its ID number or its name. This section gives some general information about resource specification.

The resource type is a sequence of four characters. Its Pascal data type is:

```
TYPE ResType = PACKED ARRAY [1..4] OF CHAR;
```

The standard resource types recognized by the Macintosh User Interface Toolbox are as follows:

Resource type	Meaning
'CODE'	Application code segment
'WIND'	Window template
'WDEF'	Window definition function
'MENU'	Menu
'MDEF'	Menu definition procedure
'MBAR'	Menu bar
'CNTL'	Control template
'CDEF'	Control definition function
'DLOG'	Dialog template
'ALRT'	Alert template
'DITL'	Item list in a dialog or alert
'ICON'	Icon
'FONT'	Font
'FWID'	Font widths
'CURS'	Cursor
'PICT'	Picture
'PAT '	Pattern (The space is required.)
'PAT#'	Pattern list
'STR '	String (The space is required.)
'DRVR'	Desk accessory or other I/O driver
'KEYC'	Keyboard configuration
'PACK'	Package
'ANYB'	Any bytes

In addition, the type 'DSAT' is reserved for system use.

(eye)

Uppercase and lowercase letters are distinguished in resource types. For example, 'Menu' will not be recognized as the resource type for menus.

Notice that some of the resources listed above are "templates". A template is a list of parameters used to build a Toolbox object; it is not the object itself. For example, a window template contains information specifying the size and location of the window, its title, whether it's visible, and so on. The Window Manager uses this information to build the window in memory and then never accesses the template again.

You can use any four-character sequence (except those listed above) for resource types specific to your application.

Every resource has an ID number, or resource ID. The resource ID must be unique within each resource type, but resources of different types may have the same ID. The standard resources in the system resource

file are usually numbered starting from Ø. The exact range of ID numbers reserved for system resources varies according to resource type. To be safe, if you want the ID numbers of your own resources not to conflict with those of the system resources, you should start numbering from at least 256 (or call a Resource Manager routine that will return an unused resource ID).

(hand)

For assembly-language programmers, the file ResEqu.Text contains predefined constants for the various resource types and for the ID numbers of standard resources.

A resource may optionally have a resource name. Like the resource ID, the resource name must be unique within each type. When comparing resource names, The Resource Manager uses the standard Operating System string comparison routine, which doesn't distinguish between uppercase and lowercase and interprets diacritical marks in foreign names properly.

RESOURCE REFERENCES

The connection between a resource specification and the corresponding resource data is provided by the resource map, via resource references. As illustrated in Figure 4, there are two kinds of resource reference:

- Local references, which are references to resources in this resource file. These point to the resource data in the file and contain a handle to the data if it's in memory.

- System references, which are references to system resources. These provide a resource specification for the resource in the system resource file, which in turn leads to a local reference to the resource in that file.

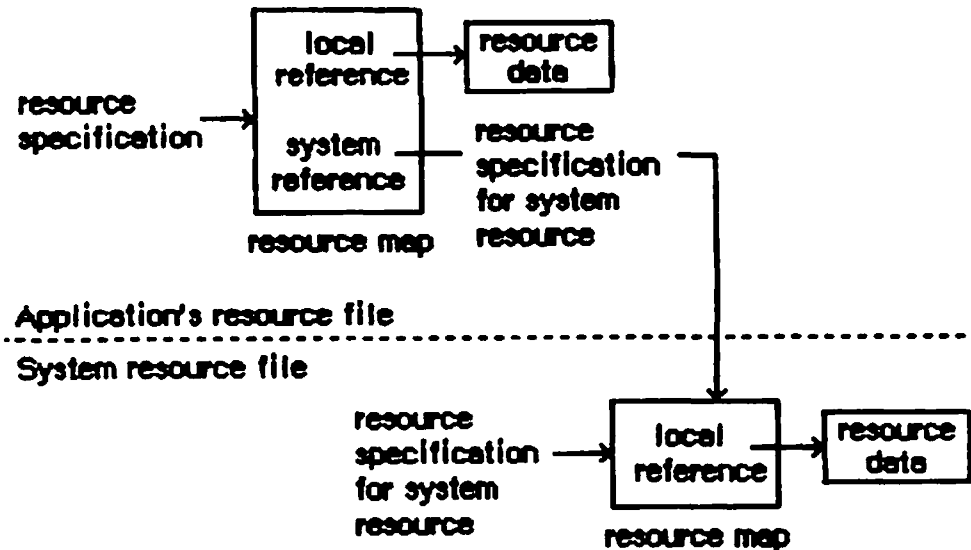


Figure 4. Resource References in Resource Maps

Every resource reference has its own type, ID number, and optional name. In the case of local references, the ID number and name are those of the resource itself. A system reference, on the other hand, may have its own ID number and name, different from those of the resource it refers to in the system resource file.

Suppose you're accessing a resource for the first time. You pass a resource specification to the Resource Manager, which looks for a match among all the references in the resource map; if none is found, it looks at the references in the resource map of the next resource file to be searched. (Remember, it looks in the resource map in memory, not in the file.) Eventually it gets to a local reference to the resource, which tells it where the resource data is in the file. After reading the resource data into memory, the Resource Manager stores a handle to that data in the local reference (again, in the resource map in memory) and returns the handle so you can use it to refer to the resource in subsequent routine calls.

Every resource reference also has certain resource attributes that determine how the resource should be dealt with. In the routine calls for setting or reading them, each attribute is specified by a bit in the low-order byte of a word, as illustrated in Figure 5.

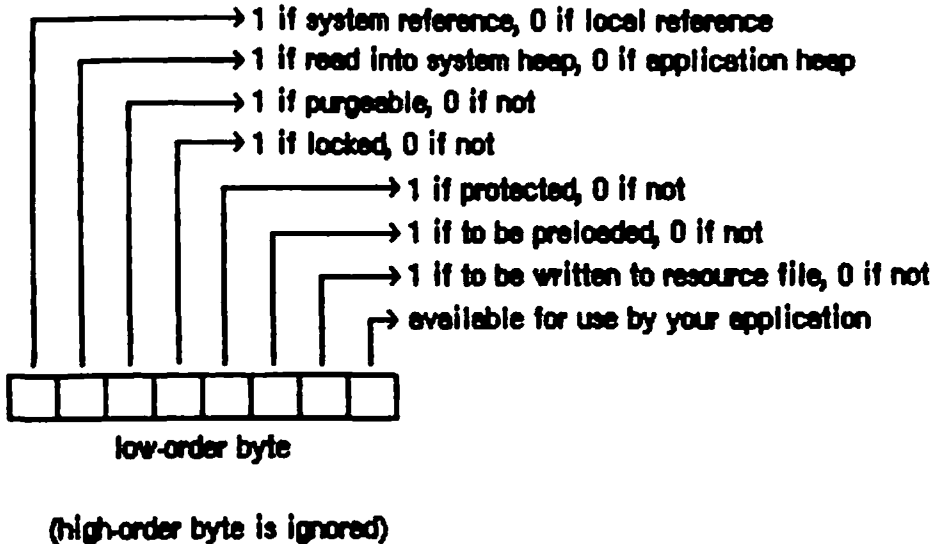


Figure 5. Resource Attributes

The Resource Manager provides a predefined constant for each attribute, in which the bit corresponding to that attribute is set.

```
CONST resSysRef     = 128;  {set if system reference}
      resSysHeap    = 64;   {set if read into system heap}
      resPurgeable  = 32;   {set if purgeable}
      resLocked     = 16;   {set if locked}
      resProtected  = 8;    {set if protected}
      resPreload    = 4;    {set if to be preloaded}
      resChanged    = 2;    {set if to be written to resource file}
      resUser       = 1;    {available for use by your application}
```

(eye)

> Your application should not change the setting of the resSysRef attribute, nor should it set the resChanged attribute directly. (ResChanged is set as a side effect of the procedure you call to tell the Resource Manager that you've changed a resource.)

Normally the resSysHeap attribute is set for all system resources; however, if the resource is too large for the system heap, this attribute will be Ø, and the resource will be read into the application heap.

Since a locked resource is neither relocatable nor purgeable, the resLocked attribute overrides the resPurgeable attribute; when resLocked is set, the resource will not be purgeable regardless of whether resPurgeable is set.

If the resProtected attribute is set, the application can't use Resource Manager routines to do any of the following to the resource: set the ID number or name in the resource reference; remove the resource from the resource file; or remove the system reference to it,

if it's a system resource. The routine that sets the resource attributes may be called, however, to remove the protection or just change some of the other attributes.

The resPreload attribute tells the Resource Manager to read this resource into memory immediately after opening the resource file. This is useful, for example, if you immediately want to draw ten icons stored in the file; rather than read and draw each one individually in turn, you can have all of them read in when the file is opened and just draw all ten.

The resChanged attribute is used only while the resource map is in memory, and must be Ø in the resource file. It tells the Resource Manager whether this resource has been changed.

USING THE RESOURCE MANAGER

This section discusses how the Resource Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Resource Manager initialization happens automatically when the system starts up: the system resource file is opened and its resource map is read into memory. Your application's resource file is opened when the application starts up; you can call CurResFile to get its reference number. You can also call OpenResFile to open any resource file that you specify by name, and CloseResFile to close any resource file. A function named ResError lets you check for errors that may occur during execution of Resource Manager routines.

(hand)

> These are the only routines you need to know about to use the Resource Manager indirectly through other Toolbox units; you can skip to their descriptions in the next section.

Normally when you want to access a resource for the first time, you'll specify it by type and ID number (or type and name) in a call to GetResource (or GetNamedResource). In special situations, you may want to get every resource of each type. There are two routines which, used together, will tell you all the resource types that are in all open resource files: CountTypes and GetIndType. Similarly, CountResources and GetIndResource may be used to get all resources of a particular type.

If you don't specify otherwise, GetResource, GetNamedResource, and GetIndResource read the resource data into memory and return a handle to it. Sometimes, however, you may not need the data to be in memory. You can use a procedure named SetResLoad to tell the Resource Manager not to read the resource data into memory when you get a resource; in this case, the handle returned for the resource will be an empty handle

(a pointer to a NIL master pointer). You can pass the empty handle to routines that operate only on the resource map (such as the routine that sets resource attributes), since the handle is enough for the Resource Manager to tell what resource you're referring to. Should you later want to access the resource data, you can read it into memory with the LoadResource procedure.

Normally the Resource Manager starts looking for a resource in the most recently opened resource file, and searches other open resource files in the reverse of the order that they were opened. In some situations, you may want to change which file is searched first. You can do this with the UseResFile procedure. One such situation might be when you want a resource to be read from the same file as another resource; in this case, you can find out which resource file the other resource was read from by calling the HomeResFile function.

Once you have a handle to a resource, you can call GetResInfo or GetResAttrs to get the information that's stored for that resource in the resource map, or you can access the resource data through the handle (if the resource data is in memory).

Usually you'll just read resources from previously created resource files with the routines described above. You may, however, want to modify existing resources or even create your own resource file. To create your own resource file, call CreateResFile (followed by OpenResFile to open it). The AddResource procedure lets you add resources to a resource file; to be sure a new resource won't override an existing one, you can call the UniqueID function to get an ID number for it. There are a number of procedures for modifying existing resources:

- To remove a resource, call RmveResource.

- To add or remove a reference to a system resource, call AddReference or RmveReference.

- If you've changed the resource data for a resource and want the changed data to be written to the resource file, call ChangedResource; it signals the Resource Manager to write the data out when the resource file is later updated.

- To change the information stored for a resource in the resource map, call SetResInfo or SetResAttrs. If you want the change to be written to the resource file, call ChangedResource. (Remember that ChangedResource will also cause the resource data itself to be written out.)

All these procedures change only the resource map in memory; the changes are written to the resource file when the application terminates (at which time all resource files other than the system resource file are updated and closed) or when one of the following routines is called:

- CloseResFile, which updates the resource file before closing it.

- UpdateResFile, which simply updates the resource file.

- WriteResource, which writes the resource data for a specified resource to the resource file.

RESOURCE MANAGER ROUTINES

This section describes all the Resource Manager procedures and functions. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** for now, see "Using QuickDraw from Assembly Language" in the QuickDraw manual and also "Notes For Assembly-Language Programmers" in this manual ***.

(hand)

> Assembly-language programmers: Except for LoadResource, all Resource Manager routines preserve all registers except AØ and DØ. LoadResource preserves AØ and DØ as well.

Initializing the Resource Manager

Although you don't call these initialization routines (because they're executed automatically for you), it's a good idea to familiarize yourself with what they do.

FUNCTION InitResources : INTEGER;

InitResources is called by the system when it starts up, and should not be called by the application. It initializes the Resource Manager, opens the system resource file, reads the resource map from the file into memory, and returns a reference number for the file.

(hand)

> The application doesn't need the reference number for the system resource file, because every Resource Manager routine that has a reference number as a parameter interprets Ø to mean the system resource file.

PROCEDURE RsrcZoneInit;

RsrcZoneInit is called automatically when your application starts up, to initialize the resource map read from the system resource file; normally you'll have no need to call it directly. It "cleans up" after any resource access that may have been done by a previous application. First it closes all open resource files except the system resource file. Then, for every system resource that was read into the

application heap (that is, whose resSysHeap attribute is Ø), it replaces the handle to that resource in the resource map with NIL. This lets the Resource Manager know that the resource will have to be read in again (since the previous application heap is no longer around).

Opening and Closing Resource Files

PROCEDURE CreateResFile (fileName: Str255);

CreateResFile creates a resource file containing no resource data or copy of the file's directory entry. If there's no file at all with the given name, it also creates an empty data fork for the file. If there's already a resource file with the given name (that is, a resource fork that isn't empty), CreateResFile will do nothing and the ResError function will return an appropriate Operating System error code.

(hand)
Before you can work with the resource file, you need to open it with OpenResFile.

FUNCTION OpenResFile (fileName: Str255) : INTEGER;

OpenResFile opens the resource file having the given name. It reads the resource map from the file into memory and returns a reference number for the file. It also reads in every resource whose resPreload attribute is set. If the resource file is already open, it simply returns the reference number.

(hand)
You don't have to call OpenResFile to open the system resource file or the application's resource file, because they're opened when the system and the application start up, respectively. To get the reference number of the application's resource file, you can call CurResFile after the application starts up (before you open any other resource file).

If the file can't be opened, OpenResFile will return -1 and the ResError function will return an appropriate Operating System error code. For example, an error occurs if there's no resource file with the given name.

PROCEDURE CloseResFile (refNum: INTEGER);

Given the reference number of a resource file, CloseResFile does the following:

- Updates the resource file by calling the UpdateResFile procedure
- For each resource in the resource file, deallocates the memory it occupies by calling the ReleaseResource procedure
- Deallocates the memory occupied by the resource map
- Closes the resource file

If there's no resource file open with the given reference number, CloseResFile will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system resource file, but if you ask to close this file, CloseResFile first closes all other open resource files.

A CloseResFile of every open resource file except the system resource file is done automatically when the application terminates. So you only need to call CloseResFile if you want to close the system resource file, or if you want to close any resource file before the application terminates.

Checking for Errors

```
FUNCTION ResError : INTEGER;
```

Called after one of the various Resource Manager routines that may result in an error condition, ResError identifies the error or returns Ø if no error occurred. If an error occurred at the Operating System level, it returns one of the Operating System error codes, such as those for file I/O errors and the Memory Manager "out of memory" error. (See the Macintosh Operating System Reference Manual for the exact codes.) If an error happened at the Resource Manager level, ResError returns one of the following predefined error codes:

```
CONST resNotFound  = -192;   {resource not found}
      resFNotFound = -193;   {resource file not found}
      addResFailed = -194;   {AddResource failed}
      addRefFailed = -195;   {AddReference failed}
      rmvResFailed = -196;   {RmveResource failed}
      rmvRefFailed = -197;   {RmveReference failed}
```

Each routine description tells which errors may occur for that routine. You can also check for an error after system startup, which calls InitResources, and application startup, which opens the application's resource file.

(hand)
Assembly-language programmers can access the current value of ResError through the global variable resErr.

Setting the Current Resource File

FUNCTION CurResFile : INTEGER

CurResFile returns the reference number of the current resource file. You can call it when the application starts up to get the reference number of its resource file.

(hand)
Assembly-language programmers can access the reference number of the current resource file through the global variable curMap.

FUNCTION HomeResFile (theResource: Handle) : INTEGER;

Given a handle to a resource, HomeResFile returns the reference number of the resource file containing that resource. If the given handle isn't a handle to a resource, HomeResFile will return -1 and the ResError function will return the error code resNotFound.

PROCEDURE UseResFile (refNum: INTEGER);

Given the reference number of a resource file, UseResFile sets the current resource file to that file. If there's no resource file open with the given reference number, UseResFile will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system resource file.

This procedure is useful for changing which resource file is searched first. For example, if you no longer want to override a system resource with one by the same name in your application's resource file, you can call UseResFile(Ø) to make the search begin (and end) in the system resource file.

Getting Resource Types

FUNCTION CountTypes : INTEGER;

CountTypes returns the number of resource types in all open resource files.

PROCEDURE GetIndType (VAR theType: ResType; index: INTEGER);

Given an index ranging from 1 to CountTypes (above), GetIndType returns a resource type in theType. Called repeatedly over the entire range

for the index, it returns all the resource types in all open resource files. If the given index isn't in the range from 1 to CountTypes, GetIndType returns four NUL characters (ASCII code Ø).

Getting and Disposing of Resources

PROCEDURE SetResLoad (load: BOOLEAN);

Normally, the routines that return handles to resources read the resource data into memory if it's not already in memory. SetResLoad(FALSE) affects all those routines so that they will not read the resource data into memory and will return an empty handle. Resources whose resPreload attribute is set will still be read in, however, when a resource file is opened. SetResLoad(TRUE) restores the normal state.

(eye)
> If you call SetResLoad(FALSE), be sure to restore the normal state as soon as possible, because other Toolbox units that call the Resource Manager rely on it.

(hand)
> Assembly-language programmers can access the current SetResLoad state (TRUE or FALSE) through the global variable resLoad.

FUNCTION CountResources (theType: ResType) : INTEGER;

CountResources returns the total number of resources of the given type in all open resource files.

FUNCTION GetIndResource (theType: ResType; index: INTEGER) : Handle;

Given an index ranging from 1 to CountResources(theType), GetIndResource returns a handle to a resource of the given type (see CountResources, above). Called repeatedly over the entire range for the index, it returns handles to all resources of the given type in all open resource files. GetIndResource reads the resource data into memory if it's not already in memory, unless you've called SetResLoad(FALSE).

(eye)
> The handle returned will be an empty handle if you've called SetResLoad(FALSE), or will become empty if the resource data for a purgeable resource is read in but later purged. (You can test for an empty handle with, for example, myHndl^ = NIL.) To read in the data and make the handle no longer be empty, you can call LoadResource.

GetIndResource returns handles for all resources in the most recently opened resource file first, and then for those in the resource files opened before it, in the reverse of the order that they were opened. If you want to find out how many resources of a given type are in a particular resource file, you can do so as follows: Call GetIndResource repeatedly with the index ranging from 1 to the number of resources of that type. Pass each handle returned by GetIndResource to HomeResFile and count all occurrences where the reference number returned is that of the desired file. Be sure to start the index from 1, and to call SetResLoad(FALSE) so the resources won't be read in.

(hand)

> The UseResFile procedure affects which file the Resource Manager searches first when looking for a particular resource but not when getting indexed resources with GetIndResource.

If the given index isn't in the range from 1 to CountResources(theType), GetIndResource returns NIL. It also returns NIL if the resource is to be read into memory but won't fit; in this case, the ResError function will return an appropriate Operating System error code.

FUNCTION GetResource (theType: ResType; theID: INTEGER) : Handle;

GetResource returns a handle to the resource having the given type and ID number, reading the resource data into memory if it's not already in memory and if you haven't called SetResLoad(FALSE) (see the first note above for GetIndResource). GetResource looks in the current resource file and all resource files opened before it, in the reverse of the order that they were opened; the system resource file is searched last. If it doesn't find the resource, GetResource returns NIL. It also returns NIL if the resource is to be read into memory but won't fit; in this case, the ResError function will return an appropriate Operating System error code.

FUNCTION GetNamedResource (theType: ResType; name: Str255) : Handle;

GetNamedResource is the same as GetResource (above) except that you pass a resource name instead of an ID number.

PROCEDURE LoadResource (theResource: Handle);

Given a handle to a resource (returned by GetIndResource, GetResource, or GetNamedResource), LoadResource reads that resource into memory. It does nothing if the resource is already in memory or if the given handle isn't a handle to a resource; in the latter case, the ResError function will return the error code resNotFound. Call this procedure if you want to access the data for a resource through its handle and either you've called SetResLoad(FALSE) or the resource is purgeable.

If you've changed the resource data for a purgeable resource and the resource is purged before being written to the resource file, the changes will be lost; LoadResource will reread the original resource from the resource file. See the descriptions of ChangedResource and SetResPurge for information about how to ensure that changes made to purgeable resources will be written to the resource file.

(hand)
Assembly-language programmers: LoadResource preserves all registers.

PROCEDURE ReleaseResource (theResource: Handle);

Given a handle to a resource, ReleaseResource deallocates the memory occupied by the resource data, if any, and replaces the handle to that resource in the resource map with NIL. (See Figure 6.) The given handle will no longer be recognized as a handle to a resource; if the Resource Manager is subsequently called to get the released resource, a new handle will be allocated. Use this procedure only after you're completely through with a resource.

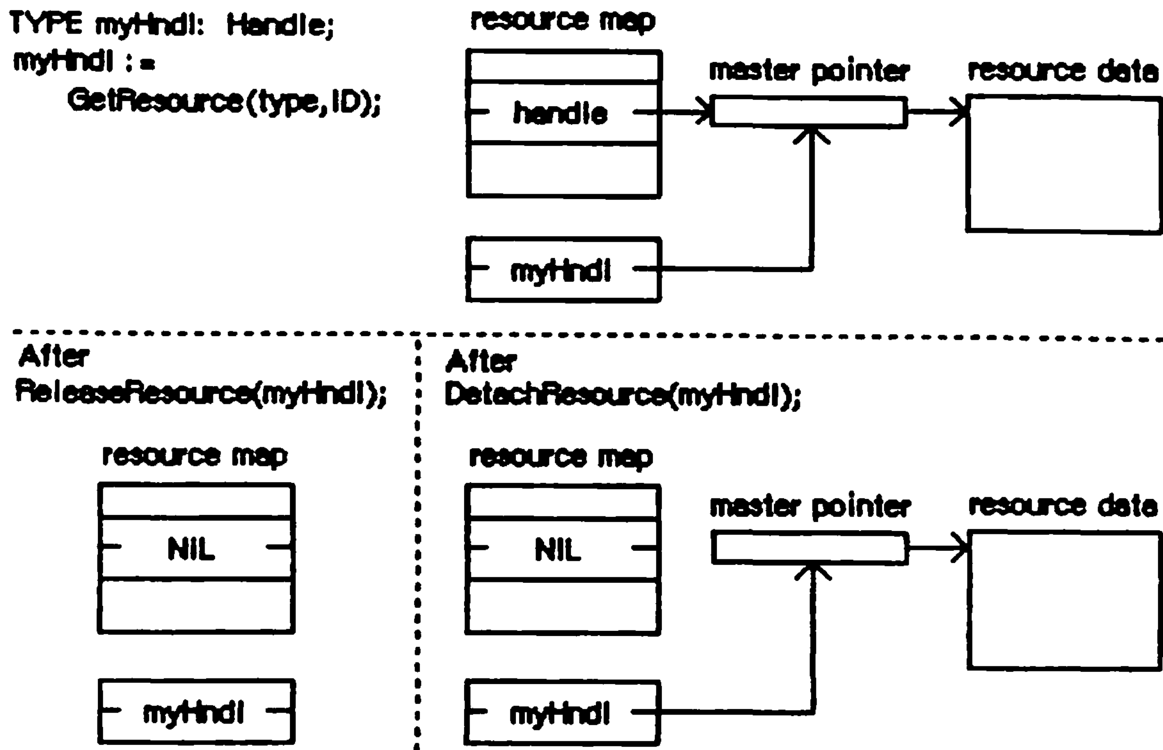


Figure 6. ReleaseResource and DetachResource

If the given handle isn't a handle to a resource, ReleaseResource will do nothing and the ResError function will return the error code resNotFound.

PROCEDURE DetachResource (theResource: Handle);

Given a handle to a resource, DetachResource replaces the handle to that resource in the resource map with NIL. (See Figure 6.) The given handle will no longer be recognized as a handle to a resource; if the Resource Manager is subsequently called to get the detached resource, a new handle will be allocated. DetachResource is useful if you want the resource data to be accessed only by yourself through the given handle and not by the Resource Manager. It's also useful in the unusual case that you don't want a resource to be deallocated when a resource file is closed.

If the given handle isn't a handle to a resource, DetachResource will do nothing and the ResError function will return the error code resNotFound.

Getting Resource Information

FUNCTION UniqueID (theType: ResType) : INTEGER;

UniqueID returns an ID number greater than Ø that isn't currently assigned to any resource of the given type in any open resource file. Using this number when you add a new resource to a resource file ensures that it won't override an existing resource.

PROCEDURE GetResInfo (theResource: Handle; VAR theID: INTEGER; VAR theType: ResType; VAR name: Str255);

Given a handle to a resource, GetResInfo returns the ID number, type, and name of the resource. If the current resource file contains a system reference to the resource, it returns the ID number, type, and name of the system reference, which may be different from those of the resource itself in the system resource file. If the given handle isn't a handle to a resource, GetResInfo will do nothing and the ResError function will return the error code resNotFound.

FUNCTION GetResAttrs (theResource: Handle) : INTEGER;

Given a handle to a resource, GetResAttrs returns the resource attributes for the resource. (Resource attributes are described earlier under "Resource References".) If the current resource file contains a system reference to the resource, GetResAttrs returns the attributes of the system reference, which may be different from those of the resource itself in the system resource file. If the given handle isn't a handle to a resource, GetResAttrs will do nothing and the ResError function will return the error code resNotFound.

Modifying Resources

Except for UpdateResFile and WriteResource, all the routines described below change the resource map in memory and not the resource file itself.

```
PROCEDURE SetResInfo (theResource: Handle; theID: INTEGER; name:
           Str255);
```

Given a handle to a resource, SetResInfo sets the ID number and name of the resource to the given ID number and name. If the current resource file contains a system reference to the resource, SetResInfo sets only the ID number and name of the system reference.

(hand)

> Assembly-language programmers: If you pass NIL for the name parameter, the name will not be changed.

(eye)

> If the resource is a system resource but the current resource file doesn't contain a reference to it, SetResInfo will set the ID number and name in the system resource file itself. This is a dangerous practice, because other applications may already access the resource and may not work properly if the ID number or name is changed.

The change will be written to the resource file when the file is updated if you follow SetResInfo with a call to ChangedResource.

(eye)

> Even if you don't call ChangedResource for this resource, the change may be written to the resource file when the file is updated. If you've **ever** called ChangedResource for **any** resource in the file, or if you've added or removed a resource or a resource reference, the Resource Manager will write out the entire resource map when it updates the file, so all changes made to resource information in the map will become permanent. If you want any of the changes to be temporary, you'll have to restore the original information before the file is updated.

SetResInfo does nothing in the following cases:

- The resProtected attribute for the resource is set.
- The given handle isn't a handle to a resource. The ResError function will return the error code resNotFound.
- The resource map becomes too large to fit in memory (which can happen if a name is passed) or sufficient space for the modified

resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

PROCEDURE SetResAttrs (theResource: Handle; attrs: INTEGER);

Given a handle to a resource, SetResAttrs sets the resource attributes for the resource to attrs. (Resource attributes are described earlier under "Resource References".) If the current resource file contains a system reference to the resource, SetResAttrs sets only the attributes of the system reference. The resProtected attribute takes effect immediately; the others take effect the next time the resource is read in.

(eye)

> Do not use SetResAttrs to set the resChanged attribute; you must call ChangedResource instead. Be sure that the attrs parameter passed to SetResAttrs doesn't change the current setting of this attribute.

The attributes set with SetResAttrs will be written to the resource file when the file is updated if you follow SetResAttrs with a call to ChangedResource. However, even if you don't call ChangedResource for this resource, the change may be written to the resource file when the file is updated. See the last warning for SetResInfo (above).

If the given handle isn't a handle to a resource, SetResAttrs will do nothing and the ResError function will return the error code resNotFound.

PROCEDURE ChangedResource (theResource: Handle);

Call ChangedResource after changing either the information about a resource in the resource map (as described above under SetResInfo and SetResAttrs) or the resource data for a resource, if you want the change to be permanent. Given a handle to a resource, ChangedResource sets the resChanged attribute for the resource. This attribute tells the Resource Manager to do **both** of the following:

- Write the resource data for the resource to the resource file when the file is updated or when WriteResource is called

- Write the entire resource map to the resource file when the file is updated

(eye)

> If you change information in the resource map with SetResInfo or SetResAttrs and then call ChangedResource, remember that not only the resource map but also the resource data will be written out when the resource file is updated.

To change the resource data for a purgeable resource and make the change permanent, you have to take special precautions to ensure that the resource won't be purged while you're changing it. You can make the resource temporarily unpurgeable and then write it out with WriteResource before making it purgeable again. You have to use the Memory Manager routines HNoPurge and HPurge to make the resource unpurgeable and purgeable; SetResAttrs can't be used because it won't take effect immediately. For example:

```
myHndl := GetResource(type,ID);   {or LoadResource(myHndl) if  }
                                  { you've gotten it previously}
HNoPurge(myHndl);                 {make it unpurgeable}
  . . .                           {make the changes here}
  ChangedResource(myHndl);        {mark it changed}
  WriteResource(myHndl);          {write it out}
HPurge(myHndl);                   {make it purgeable again}
```

Or, instead of calling WriteResource to write the data out immediately, you can call SetResPurge(TRUE) before making any changes to purgeable resource data.

ChangedResource does nothing in the following cases:

- The given handle isn't a handle to a resource. The ResError function will return the error code resNotFound.

- Sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

PROCEDURE AddResource (theData: Handle; theType: ResType; theID: INTEGER; name: Str255);

Given a handle to data in memory (not a handle to an existing resource), AddResource adds to the current resource file a local reference that points to the data. It sets the resChanged attribute for the resource, so the data will be written to the resource file when the file is updated or when WriteResource is called. If the given handle is empty, zero-length resource data will be written. AddResource does nothing in the following cases:

- The given handle is NIL or is already a handle to an existing resource. The ResError function will return the error code addResFailed.

- The resource map becomes too large to fit in memory or sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

PROCEDURE RmveResource (theResource: Handle);

Given a handle to a resource in the current resource file, RmveResource removes the local reference to the resource. The resource data will be removed from the resource file when the file is updated.

(hand)
It doesn't deallocate the memory occupied by the resource data; to do that, call the Memory Manager routine DisposeHandle after calling RmveResource.

If the resProtected attribute for the resource is set or if the given handle isn't a handle to a resource in the current resource file, RmveResource will do nothing and the ResError function will return the error code rmvResFailed.

(eye)
It's dangerous to remove a resource from the system resource file, because all system references to it will become invalid.

PROCEDURE AddReference (theResource: Handle; theID: INTEGER; name: Str255);

Given a handle to a system resource, AddReference adds to the current resource file a system reference to the resource, giving it the ID number and name specified by the parameters. It sets the resChanged attribute for the resource, so the reference will be written to the resource file when the file is updated. AddReference does nothing in the following cases:

- The current resource file is the system resource file or already contains a system reference to the specified resource, or the given handle isn't a handle to a system resource. The ResError function will return the error code addRefFailed.

- The resource map becomes too large to fit in memory or sufficient space for the modified resource file can't be reserved on the disk. ResError will return an appropriate Operating System error code.

PROCEDURE RmveReference (theResource: Handle);

Given a handle to a system resource, RmveReference removes the system reference to the resource from the current resource file. (The reference will be removed from the resource file when the file is updated.) In the following cases, RmveReference will do nothing and the ResError function will return the error code rmvRefFailed: the resProtected attribute for the resource is set; there's no system reference to the resource in the current resource file; or the given handle isn't a handle to a system resource.

PROCEDURE UpdateResFile (refNum: INTEGER);

Given the reference number of a resource file, UpdateResFile does the following:

- Changes, adds, or removes resource data in the file as appropriate to match the map. Remember that changed resource data is written out only if you called ChangedResource. If a resource whose data is to be written out has been purged, zero-length resource data will be written.

- Compacts the resource file if necessary, closing up any empty space created when a resource or a resource reference was removed or when a resource was made larger. (If the size of a changed resource is greater than its original size in the resource file, it's written at the end of the file rather than at its original location, leaving empty space at that location. UpdateResFile doesn't close up any empty space created when a resource is made smaller.)

- Writes out the resource map of the resource file, if you ever called ChangedResource for any resource in the file or if you added or removed a resource or a resource reference. All changes to resource information in the map will become permanent as a result of this, so if you want any such changes to be temporary, you must restore the original information before calling UpdateResFile.

If there's no open resource file with the given reference number, UpdateResFile will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system resource file.

The CloseResFile procedure calls UpdateResFile before it closes the resource file, so you only need to call UpdateResFile yourself if you want to update the file without closing it.

PROCEDURE WriteResource (theResource: Handle);

Given a handle to a resource, WriteResource checks the resChanged attribute for that resource and, if it's set (which it will be if you called ChangedResource or AddResource), writes its resource data to the resource file and clears its resChanged attribute. If the resource is purgeable and has been purged, zero-length resource data will be written. WriteResource does nothing if the resProtected attribute for the resource is set or if the given handle isn't a handle to a resource; in the latter case, the ResError function will return the error code resNotFound.

Since the resource file is updated when the application terminates or when you call UpdateResFile (or CloseResFile, which calls UpdateResFile), you only need to call WriteResource if you want to write out just one or a few resources immediately.

PROCEDURE SetResPurge (install: BOOLEAN);

SetResPurge(TRUE) sets a "hook" in the Memory Manager such that before purging data specified by a handle, the Memory Manager will first pass the handle to the Resource Manager. The Resource Manager will determine whether the handle is that of a resource in the application heap and, if so, will call WriteResource to write the resource data for that resource to the resource file if its resChanged attribute is set (see ChangedResource and WriteResource above). SetResPurge(FALSE) restores the normal state, clearing the hook so that the Memory Manager will once again purge without checking with the Resource Manager.

SetResPurge(TRUE) is useful in applications that modify purgeable resources. You still have to make the resources temporarily unpurgeable while making the changes, as shown in the description of ChangedResource, but you can set the purge hook instead of writing the data out immediately with WriteResource. Notice that you won't know exactly when the resources are being written out; most applications will want more control than this. If you wish, you can set your own such hook.

Advanced Routines

The routines described below allow advanced programmers to have even greater control over resource file operations. Just as individual resources have attributes, an entire resource file also has attributes, which these routines manipulate. Like the attributes of individual resources, resource file attributes are specified by bits in the low-order byte of a word. The Resource Manager provides a predefined constant for each attribute, in which the bit corresponding to that attribute is set.

```
CONST mapReadOnly = 128;
      mapCompact  = 64;
      mapChanged  = 32;
```

When the mapReadOnly attribute is set, the Resource Manager will neither write anything to the resource file nor check whether there's sufficient space for the file on the disk when the resource map is modified.

(eye)

> If you set mapReadOnly but then later clear it, the resource file will be written even if there's no room for it on the disk. This would destroy the file.

The mapCompact attribute causes resource file compaction to occur when the file is updated. It's set by the Resource Manager when a resource or a resource reference is removed, or when a resource is made larger and thus has to be written at the end of the resource file. You may want to set mapCompact to force compaction when you've only made resources smaller.

The mapChanged attribute causes the resource map to be written to the resource file when the file is updated. It's set by the Resource Manager when you call ChangedResource or when you add or remove a resource or a resource reference. You can set mapChanged if, for example, you've changed resource attributes only and don't want to call ChangedResource because you don't want the resource data to be written out.

```
FUNCTION GetResFileAttrs (refNum: INTEGER) : INTEGER;
```

Given the reference number of a resource file, GetResFileAttrs returns the resource file attributes for the file. If there's no resource file with the given reference number, GetResFileAttrs will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system reference file.

```
PROCEDURE SetResFileAttrs (refNum: INTEGER; attrs: INTEGER);
```

Given the reference number of a resource file, SetResFileAttrs sets the resource file attributes of the file to attrs. If there's no resource file with the given reference number, SetResFileAttrs will do nothing and the ResError function will return the error code resFNotFound. A refNum of Ø represents the system reference file, but you shouldn't change its resource file attributes.

RESOURCES WITHIN RESOURCES

Resources may point to other resources; this section discusses how this is normally done, for programmers who are interested in background information about resources or who are defining their own resource types.

In a resource file, one resource points to another with the ID number of the other resource. For example, the resource data for a menu includes the ID number of the menu's definition procedure (a separate resource that determines how the menu looks and behaves). To work with the resource data in memory, however, it's faster and more convenient to have a handle to the other resource rather than its ID number. Since a handle occupies two words, the ID number in the resource file is followed by a word containing Ø; these two words together serve as a placeholder for the handle. Once the other resource has been read into memory, these two words can be replaced by a handle to it. (See Figure 7.)

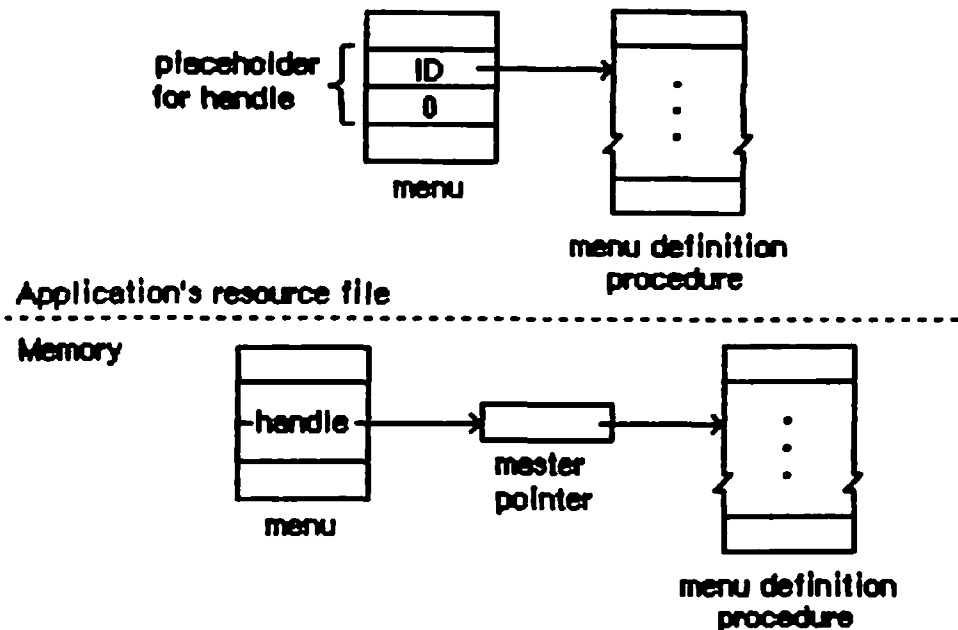


Figure 7. How Resources Point to Resources

(hand)

The practice of using the ID number followed by Ø as a placeholder is simply a convention. If you like, you can set up your own resources to have the ID number followed by a dummy word, or even a word of useful information, or you can put the ID in the second rather than the first word of the placeholder.

In the case of menus, the Menu Manager routine GetMenu calls the Resource Manager to read the menu and the menu definition procedure into memory, and then replaces the placeholder in the menu with the handle to the procedure. There may be other cases where you call the Resource Manager directly and store the handle in the placeholder yourself. It might be useful in these cases to call HomeResFile to learn which resource file the original resource is located in, and then, before getting the resource it points to, call UseResFile to set the current resource file to that file. This will ensure that the resource pointed to is read from that same file (rather than one that was opened after it).

(eye)

If you modify a resource that points to another resource and you make the change permanent by calling ChangedResource, be sure you reverse the process described here, restoring the other resource's ID number in the placeholder.

FORMAT OF A RESOURCE FILE

This section gives the exact format of a resource file, which you need to know if you're writing a program that will create or modify resource files directly. You don't have to know the exact format to be able to use the Resource Manager routines.

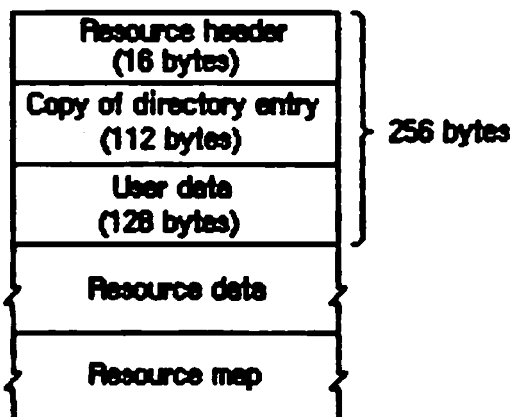


Figure 8. Format of a Resource File

As illustrated in Figure 8, every resource file begins with a resource header. The resource header gives the offsets to and lengths of the resource data and resource map parts of the file, as follows:

Number of bytes	Contents
4 bytes	Offset from beginning of resource file to resource data
4 bytes	Offset from beginning of resource file to resource map
4 bytes	Length of resource data
4 bytes	Length of resource map

(hand)

All offsets and lengths in the resource file are given in bytes.

This is what immediately follows the resource header:

Number of bytes	Contents
112 bytes	Partial copy of directory entry for this file
128 bytes	Available for user data

The directory copy is used by the Finder. The user data may be whatever the you want.

The resource data follows the user data. It consists of the following for each resource in the file:

Number of bytes	Contents
For each resource:	
4 bytes	Length of following resource data
n bytes	Resource data for this resource

To learn exactly what the resource data is for a standard type of resource, see the documentation on the Toolbox unit that deals with that resource type.

After the resource data, the resource map begins as follows:

Number of bytes	Contents
16 bytes	Ø (reserved for copy of resource header)
4 bytes	Ø (reserved for handle to next resource map to be searched)
2 bytes	Ø (reserved for file reference number)
2 bytes	Resource file attributes
2 bytes	Offset from beginning of resource map to type list (see below)
2 bytes	Offset from beginning of resource map to resource name list (see below)

After reading the resource map into memory, the Resource Manager stores the indicated information in the reserved areas at the beginning of the map.

The resource map continues with a type list, reference lists, and a resource name list. The type list contains the following:

Number of bytes	Contents
2 bytes	Number of resource types in the map minus 1
For each type:	
4 bytes	Resource type
2 bytes	Number of resources of this type in the map minus 1
2 bytes	Offset from beginning of type list to reference list for resources of this type

This is followed by the reference list for each type of resource, which contains the resource references for all resources of that type. The reference lists are contiguous and in the same order as the types in the type list. The format of a reference list is as follows:

Number of bytes	Contents
For each reference of this type:	
2 bytes	Resource ID
2 bytes	Offset from beginning of resource name list to length of resource name, or -1 if none
1 byte	Resource attributes
3 bytes	If local reference, offset from beginning of resource data to length of data for this resource If system reference, Ø (ignored)
4 bytes	If local reference, Ø (reserved for handle to resource) If system reference, resource specification for system resource: in high-order word, resource ID; in low-order word, offset from beginning of resource name list to length of resource name, or -1 if none

The resource name list follows the reference list and has this format:

Number of bytes	Contents
For each name:	
1 byte	Length of following resource name
n bytes	Characters of resource name

Figure 9 on the following page shows where the various offsets lead to in a resource file, in general and also specifically for a local reference.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

*** This will be moved to a separate chapter of the final comprehensive manual. For now, see the QuickDraw manual for complete information about how to use the User Interface Toolbox from assembly language. ***

The primary aid to assembly-language programmers is a file named ToolEqu.Text. If you use .INCLUDE to include this file when you assemble your program, all the Resource Manager constants and locations of system globals will be available in symbolic form.

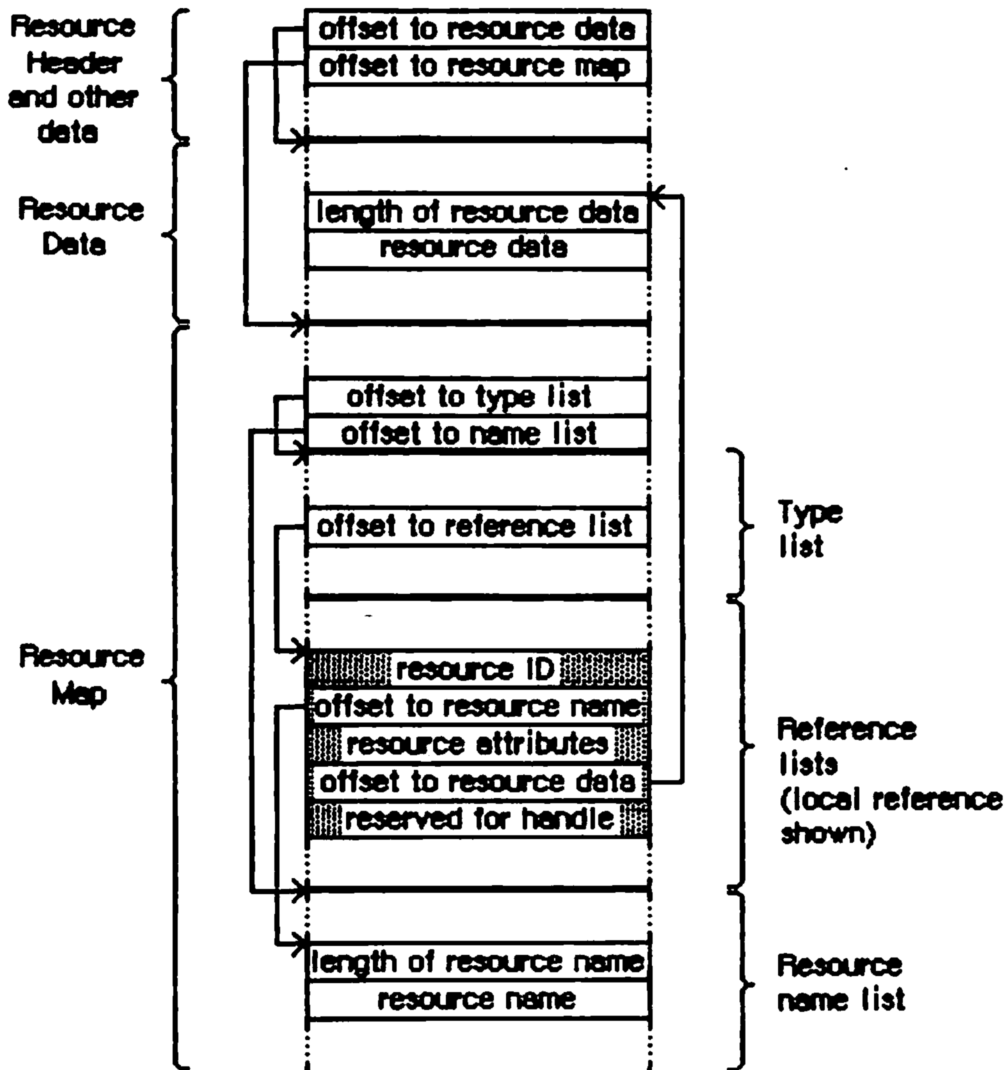


Figure 9. Local Reference in a Resource File

SUMMARY OF THE RESOURCE MANAGER

```
CONST resSysRef     = 128;  {set if system reference}
      resSysHeap    = 64;   {set if read into system heap}
      resPurgeable  = 32;   {set if purgeable}
      resLocked     = 16;   {set if locked}
      resProtected  = 8;    {set if protected}
      resPreload    = 4;    {set if to be preloaded}
      resChanged    = 2;    {set if to be written to resource file}
      resUser       = 1;    {available for use by your application}

      resNotFound  = -192;   {resource not found}
      resFNotFound = -193;   {resource file not found}
      addResFailed = -194;   {AddResource failed}
      addRefFailed = -195;   {AddReference failed}
      rmvResFailed = -196;   {RmveResource failed}
      rmvRefFailed = -197;   {RmveReference failed}

      mapReadOnly = 128;
      mapCompact  = 64;
      mapChanged  = 32;

TYPE ResType = PACKED ARRAY [1..4] OF CHAR;
```

Initializing the Resource Manager

```
FUNCTION  InitResources : INTEGER;
PROCEDURE RsrcZoneInit;
```

Opening and Closing Resource Files

```
PROCEDURE CreateResFile (filename: Str255);
FUNCTION  OpenResFile   (fileName: Str255) : INTEGER;
PROCEDURE CloseResFile  (refNum: INTEGER);
```

Checking for Errors

```
FUNCTION ResError : INTEGER;
```

Setting the Current Resource File

```
FUNCTION  CurResFile : INTEGER;
FUNCTION  HomeResFile (theResource: Handle) : INTEGER;
PROCEDURE UseResFile  (refNum: INTEGER);
```

Getting Resource Types

```
FUNCTION  CountTypes : INTEGER;
PROCEDURE GetIndType  (VAR theType: ResType; index: INTEGER);
```

Getting and Disposing of Resources

```
PROCEDURE SetResLoad       (load: BOOLEAN);
FUNCTION  CountResources   (theType: ResType) : INTEGER;
FUNCTION  GetIndResource   (theType: ResType; index: INTEGER) : Handle;
FUNCTION  GetResource      (theType: ResType; theID: INTEGER) : Handle;
FUNCTION  GetNamedResource (theType: ResType; name: Str255) : Handle;
PROCEDURE LoadResource     (theResource: Handle);
PROCEDURE ReleaseResource  (theResource: Handle);
PROCEDURE DetachResource   (theResource: Handle);
```

Getting Resource Information

```
FUNCTION  UniqueID    (theType: ResType) : INTEGER;
PROCEDURE GetResInfo  (theResource: Handle; VAR theID: INTEGER; VAR
                       theType: ResType; VAR name: Str255);
FUNCTION  GetResAttrs (theResource: Handle) : INTEGER;
```

Modifying Resources

```
PROCEDURE SetResInfo      (theResource: Handle; theID: INTEGER; name:
                           Str255);
PROCEDURE SetResAttrs     (theResource: Handle; attrs: INTEGER);
PROCEDURE ChangedResource (theResource: Handle);
PROCEDURE AddResource     (theData: Handle; theType: ResType; theID:
                           INTEGER; name: Str255);
PROCEDURE RmveResource    (theResource: Handle);
PROCEDURE AddReference    (theResource: Handle; theID: INTEGER; name:
                           Str255);
PROCEDURE RmveReference   (theResource: Handle);
PROCEDURE UpdateResFile   (refNum: INTEGER);
PROCEDURE WriteResource   (theResource: Handle);
PROCEDURE SetResPurge     (install: BOOLEAN);
```

Advanced Routines

```
FUNCTION  GetResFileAttrs (refNum: INTEGER) : INTEGER;
PROCEDURE SetResFileAttrs (refNum: INTEGER; attrs: INTEGER);
```

SUMMARY OF THE RESOURCE FILE FORMAT

(hand)
All offsets and lengths are given in bytes.

Section	Size	Description
Resource Header and other data	4 bytes	Offset to resource data
	4 bytes	Offset to resource map
	4 bytes	Length of resource data
	4 bytes	Length of resource map
	112 bytes	Partial copy of file's directory entry
	128 bytes	User data
Resource Data	For each resource:	
	4 bytes	Length of following resource data
	n bytes	Resource data for this resource
Resource Map	16 bytes	Reserved for copy of resource header
	4 bytes	Reserved for handle to next resource map to be searched
	2 bytes	Reserved for file reference number
	2 bytes	Resource file attributes
	2 bytes	Offset to type list
	2 bytes	Offset to resource name list
Type list	2 bytes	Number of resource types minus 1
	For each type:	
	4 bytes	Resource type
	2 bytes	Number of resources of this type minus 1
	2 bytes	Offset to reference list for this type
Reference lists (one per type, contiguous, same order as in type list)	For each reference of this type:	
	2 bytes	Resource ID
	2 bytes	Offset to length of resource name or -1 if none
	1 byte	Resource attributes
	3 bytes	If local reference, offset to length of resource data If system reference, Ø
	4 bytes	If local, reserved for handle to resource If system, resource specification for system resource: in high-order word, resource ID; in low-order word, offset to length of resource name or -1 if none
Resource name list	For each name:	
	1 byte	Length of following resource name
	n bytes	Characters of resource name

GLOSSARY

current resource file: The last resource file opened, unless you specify otherwise with a Resource Manager routine.

empty handle: A pointer to a NIL master pointer.

local reference: A resource reference to a resource in the same file as the reference. It points to the resource data in the file and contains a handle to the data if it's in memory.

purgeable: Able to be removed from the heap (purged) when space is required by the Memory Manager.

reference number: A number greater than Ø, returned when a file is opened, by which you can refer to that file. In Resource Manager routines that expect a reference number, Ø represents the system resource file.

resource: Data or code stored in a resource file and managed by the Resource Manager.

resource attribute: One of several characteristics, specified by bits in a resource reference, that determine how the resource should be dealt with.

resource data: In a resource file, the data that comprises a resource.

resource file: The resource fork of a file, which contains data used by the application (such as menus, fonts, and icons) and also the application code itself.

resource header: At the beginning of a resource file, data that gives the offsets to and lengths of the resource data and resource map.

resource ID: A number that, together with the resource type, identifies a resource in a resource file. Every resource has an ID number.

resource map: In a resource file, data that is read into memory when the file is opened and that, given a resource specification, leads to the corresponding resource data.

resource name: A string that, together with the resource type, identifies a resource in a resource file. A resource may or may not have a name.

resource reference: In a resource map, a local reference leading to resource data in the same file as the reference, or a system reference leading to data in the system resource file.

resource specification: A resource type and either a resource ID or a resource name.

resource type: The type of a resource in a resource file, designated by a sequence of four characters (such as 'MENU' for a menu).

system reference: In an application's resource file, a resource reference to a system resource. It provides a resource specification for the resource in the system resource file.

system resource: A resource in the system resource file.

system resource file: A resource file containing standard resources, accessed if a requested resource wasn't found in any of the other resource files that were searched.

SAD MACINTOSH ICON
OCT 26, 1983
Patti Kenyon

Here is a example:

Power on the Macintosh, holding down the NMI button on the left side of the computer. The sad Macintosh will appear, with a set of numbers under it. This set of code can be divided into two types of code.

```
OF 000D - Sub Codes
 |
Class Codes
```

Class Codes deals with the initial diagnostic code.

1 = ROM test failed	meaningless
2 = Memtest - Bus subtest	bits set corresponds to suspected bad RAM chips
3 = Memtest - ByteWrite	" "
4 = Memtest - Mod3test	" "
5 = Memtest - Addr uniqueness	" "

Class Code F = exception, only after diagnostics have passed.
This is where the Sub Codes come in.

F = exception	0001 Bus error
	0002 address error
	0003 illegal instruction
	0004 zero divide
	0005 check instruction
	0006 trapv instruction
	0007 privilege violation
	0008 trace
	0009 line 1010
	000A line 1111
	000B other exceptions
	000C nothing
	000D NMI

Another test to see how this works is, remove a RAM chip. Power up the Macintosh.

A new code should appear under the sad Macintosh icon. When I did it, I picked the one closes to the Keyboard connector. The code under the Macintosh was 028000. So number 2's class code tells me that it suspects bad RAM chip. The eight tells me that its the 15th RAM chip.

RAM Chip #		Code under Macintosh
0	=	0001
1	=	0002
2	=	0004
3	=	0008
4	=	0010
5	=	0020

6	=	0040
7	=	0080
8	=	0100
9	=	0200
10	=	0400
11	=	0800
12	=	1000
13	=	2000
14	=	4000
15	=	8000

This is a good example of just one RAM chip being bad, but what is there are multiple RAM chips that are bad? Try taking the 15th and 14th RAM chips out. the Code appears like this. 02C000, we can say since we know the code is in Hex that there are 16 possibilities.

0 0000
1 0001
2 0010
3 0011
4 0100
5 0101
6 0110
7 0111
8 1000
9 1001
A 1010
B 1011
C 1100 meaning the 15th and 14th chip is bad.
D 1101
E 1110
F 1111

This is just a start to understanding what all the codes mean. If we try to keep a list, we should come across all the possibilities.

```
; File SysErr.Text - Macintosh system error equates file.
;_____________________________________________________________
;
;
; All system routines reporting errors include this equate file.
;
; Modification History:
; 16 Feb 83  LAK  Broke out from SysEqu.Text
; 22 Feb 83  LAK  Added equates from SysUtil and Events.
; 10 May 83  MPH  Added new memory manager error codes.
; 07 Jun 83  LAK  Adjusted error codes for file system
; 15 Jun 83  AJH  Added Deep Shit Error Definitions
; 23 Jun 83  LAK  Added initIWMErr for sony driver.
; 18 Jul 83  LAK  Added more memory manager deep shit errors.
; 11 Aug 83  LAK  Added more disk driver error codes.
; 17 Aug 83  LAK  Added file system deep shit error code.
; 18 Aug 83  SC   Added scrap manager error codes.
; 19 Aug 83  LAK  Added stackinheap deep shit code.
; 21 Aug 83  LAK  Added SpdAdjErr, SeekErr, and SectNFErr for disk driver.
;                 Removed BadNybErr; moved clock/pram error codes up by 4.
; 23 Aug 83  LAK  Added NotOpenErr for drivers.
; 27 Aug 83  AJH  added memFull deep shit alert
; 06 Sep 83  AJH  added DSBadLaunch
; 22 Sep 83  BLH  Added Resource Mgr errors
; 10 Oct 83  LAK  Added NoErr equate
; 13 Oct 83  LAK  added NoErr, DSReInsert equates, DSNotThe1
;_____________________________________________________________

; General System Errors (VBL Mgr, Queueing, Etc.)

NoErr       .EQU    0       ; success is absence of errors
QErr        .EQU    -1      ; queue element not found during deletion
VTypErr     .EQU    -2      ; invalid queue element
CorErr      .EQU    -3      ; core routine number out of range
UnimpErr    .EQU    -4      ; unimplemented core routine

; I/O System Errors

ControlErr  .EQU    -17
StatusErr   .EQU    -18
ReadErr     .EQU    -19
WritErr     .EQU    -20
BadUnitErr  .EQU    -21
UnitEmptyErr .EQU   -22
OpenErr     .EQU    -23
ClosErr     .EQU    -24
DRemovErr   .EQU    -25     ; tried to remove an open driver
DInstErr    .EQU    -26     ; DrvrInstall couldn't find driver in resources
AbortErr    .EQU    -27     ; IO call aborted by KillIO
NotOpenErr  .EQU    -28     ; Couldn't rd/wr/ctl/sts cause driver not opened

;  File System error codes:

DirFulErr   .EQU    -33     ; Directory full
DskFulErr   .EQU    -34     ; disk full
NSVErr      .EQU    -35     ; no such volume
IOErr       .EQU    -36     ; I/O error (bummers)
BdNamErr    .EQU    -37     ; there may be no bad names in the final system!
FNOpnErr    .EQU    -38     ; File not open
EOFErr      .EQU    -39     ; End of file
PosErr      .EQU    -40     ; tried to position to before start of file (r/w)
MFulErr     .EQU    -41     ; memory full(open) or file won't fit (load)
TMFOErr     .EQU    -42     ; too many files open
FNFErr      .EQU    -43     ; File not found

WPrErr      .EQU    -44     ; diskette is write protected
FLckdErr    .EQU    -45     ; file is locked
VLckdErr    .EQU    -46     ; volume is locked
```

```
FBsyErr      .EQU    -47     ; File is busy (delete)
DupFNErr     .EQU    -48     ; duplicate filename (rename)
OpWrErr      .EQU    -49     ; file already open with with write permission
ParamErr     .EQU    -50     ; error in user parameter list
RFNumErr     .EQU    -51     ; refnum error
GFPErr       .EQU    -52     ; get file position error
VolOffLinErr .EQU    -53     ; volume not on line error (was Ejected)
PermErr      .EQU    -54     ; permissions error (on file open)
VolOnLinErr  .EQU    -55     ; drive volume already on-line at MountVol
NSDrvErr     .EQU    -56     ; no such drive (tried to mount a bad drive num)
NoMacDskErr  .EQU    -57     ; not a mac diskette (sig bytes are wrong)
ExtFSErr     .EQU    -58     ; volume in question belongs to an external fs
FSDSErr      .EQU    -59     ; file system deep s--t error:
                             ;  during rename the old entry was deleted but could
                             ;   not be restored . . .
BadMDBErr    .EQU    -60     ; bad master directory block
WrPermErr    .EQU    -61     ; write permissions error

; Disk, Serial Ports, Clock Specific Errors

NoDriveErr   .EQU    -64     ; drive not installed
OffLinErr    .EQU    -65     ; r/w requested for an off-line drive

NoNybErr     .EQU    -66     ; couldn't find 5 nybbles in 200 tries
NoAdrMkErr   .EQU    -67     ; couldn't find valid addr mark
DataVerErr   .EQU    -68     ; read verify compare failed
BadCkSmErr   .EQU    -69     ; addr mark checksum didn't check
BadBtSlpErr  .EQU    -70     ; bad addr mark bit slip nibbles
NoDtaMkErr   .EQU    -71     ; couldn't find a data mark header
BadDCkSum    .EQU    -72     ; bad data mark checksum
BadDBtSlp    .EQU    -73     ; bad data mark bit slip nibbles
WrUnderRun   .EQU    -74     ; write underrun occurred

CantStepErr  .EQU    -75     ; step handshake failed
Tk0BadErr    .EQU    -76     ; track 0 detect doesn't change
InitIWMErr   .EQU    -77     ; unable to initialize IWM
TwoSideErr   .EQU    -78     ; tried to read 2nd side on a 1-sided drive
SpdAdjErr    .EQU    -79     ; unable to correctly adjust disk speed
SeekErr      .EQU    -80     ; track number wrong on address mark
SectNFErr    .EQU    -81     ; sector number never found on a track

ClkRdErr     .EQU    -85     ; unable to read same clock value twice
ClkWrErr     .EQU    -86     ; time written did not verify
PRWrErr      .EQU    -87     ; parameter ram written didn't read-verify
PRInitErr    .EQU    -88     ; InitUtil found the parameter ram uninitialized

RcvrErr      .EQU    -89     ; SCC receiver error (framing, parity, OR)
BreakRecd    .EQU    -90     ; Break received (SCC)

; Storage allocator error codes

MemFullErr   .EQU    -108    ; Not enough room in heap zone
NilHandleErr .EQU    -109    ; Handle was NIL in HandleZone or other;
memWZErr     .EQU    -111    ; WhichZone failed (applied to free block);
memPurErr    .EQU    -112    ; trying to purge a locked or non-purgable block;

memAdrErr    .EQU    -110    ; address was odd, or out of range;
memAZErr     .EQU    -113    ; Address in zone check failed;
memPCErr     .EQU    -114    ; Pointer Check failed;
memBCErr     .EQU    -115    ; Block Check failed;
memSCErr     .EQU    -116    ; Size Check failed;

; Resource Manager error codes (other than I/O errors)

ResNotFound      .EQU    -192        ; Resource not found
ResFNotFound     .EQU    -193        ; Resource file not found
AddResFailed     .EQU    -194        ; AddResource failed
AddRefFailed     .EQU    -195        ; AddReference failed
```

```
RmvResFailed    .EQU    -196            ; RmveResource failed
RmvRefFailed    .EQU    -197            ; RmveReference failed

; Scrap Manager error codes

noScrapErr  .EQU    -100    ; No scrap exists error
noTypeErr   .EQU    -102    ; No object of that type in scrap


; Application Error Codes
;
;   errors -1024 to -4095 are reserved for use by the current application


;  Deep Shit Alert ID definitions

DSSysErr      .EQU  32767   ; general system error
DSBusError    .EQU  1       ; bus error
DSAddressErr  .EQU  2       ; address error
DSIllInstErr  .EQU  3       ; illegal instruction error
DSZeroDivErr  .EQU  4       ; zero divide error
DSChkErr      .EQU  5       ; check trap error
DSOvFlowErr   .EQU  6       ; overflow trap error
DSPrivErr     .EQU  7       ; privelege violation error
DSTraceErr    .EQU  8       ; trace mode error
DSLineAErr    .EQU  9       ; line 1010 trap error
DSLineFErr    .EQU  10      ; line 1111 trap error
DSMiscErr     .EQU  11      ; miscellaneous hardware exception error
DSCoreErr     .EQU  12      ; unimplemented core routine error
DSIrqErr      .EQU  13      ; uninstalled interrupt error

DSIOCoreErr   .EQU  14      ; IO Core Error
DSLoadErr     .EQU  15      ; Segment Loader Error
DSFPErr       .EQU  16      ; Floating point error

DSNoPackErr   .EQU  17      ; package 0 not present
DSNoPk1       .EQU  18      ; package 1 not present
DSNoPk2       .EQU  19      ; package 2 not present
DSNoPk3       .EQU  20      ; package 3 not present
DSNoPk4       .EQU  21      ; package 4 not present
DSNoPk5       .EQU  22      ; package 5 not present
DSNoPk6       .EQU  23      ; package 6 not present
DSNoPk7       .EQU  24      ; package 7 not present

DSMemFullErr  .EQU  25      ; out of memory!
DSBadLaunch   .EQU  26      ; can't launch file

DSStknHeap    .EQU  28      ; stack has moved into application heap
DSFSErr       .EQU  27      ; file system map has been trashed
DSReInsert    .EQU  30      ; request user to reinsert off-line volume
DSNotThe1     .EQU  31      ; not the disk I wanted

; Storage allocator trouble codes (deep shit IDs)

MemTrbBase  .EQU    32              ; Memory Manager Trouble Code base.
mtSetLog    .EQU    MemTrbBase      ; Set Logical Size Error.
mtAdjFre    .EQU    MemTrbBase+1    ; Adjust Free Error.
mtAdjCnt    .EQU    MemTrbBase+2    ; Adjust Counters Error.
mtMkeBkf    .EQU    MemTrbBase+3    ; Make Block Free Error.
mtSetSiz    .EQU    MemTrbBase+4    ; Set Size Error.
mtInitMem   .EQU    MemTrbBase+5    ; Initialize Memory Manager Error.
mtBCerr     .EQU    MemTrbBase+6    ;
mtCZerr     .EQU    MemTrbBase+7    ;
mtCZ1err    .EQU    MemTrbBase+8    ;
mtCZ2err    .EQU    MemTrbBase+9    ;
mtCZ3err    .EQU    MemTrbBase+10   ;
mtEqCerr    .EQU    MemTrbBase+11   ;
mtEvCerr    .EQU    MemTrbBase+12   ;
```

```
mtHCerr     .EQU    MemTrbBase+13   ;
mtPCerr     .EQU    MemTrbBase+14   ;
mtSCerr     .EQU    MemTrbBase+15   ;
mtRC1err    .EQU    MemTrbBase+16   ;
mtRC2err    .EQU    MemTrbBase+17   ;
mtSABerr    .EQU    MemTrbBase+18   ;
mtACerr     .EQU    MemTrbBase+19   ;
mtIZCerr    .EQU    MemTrbBase+20   ;
mtPrCerr    .EQU    MemTrbBase+21   ;

; some miscellaneous result codes

EvtNotEnb   .EQU      1     ; event not enabled at PostEvent
NoEvtAvail  .EQU     -1     ; no event available (GetOSEvent,OSEventAvail)
```

The Scrap Manager: A Programmer's Guide /SMGR/SCRAP

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Putting Together a Macintosh Application

Modification History:			
	First Draft (ROM 7)	C. Rose	10/21/83
	Erratum Added	C. Rose	11/16/83

ABSTRACT

The Scrap Manager is a set of simple routines and data types that help Macintosh applications manipulate the Clipboard for cutting and pasting between applications, desk accessories, or an application and a desk accessory. This manual describes the Scrap Manager in detail.

Erratum:

The 'TEXT' type of data in the desk scrap is simply a series of ASCII characters, without a character count or an optional comment. If you want to know the count, you can get it by passing a NIL handle to the GetScrap function.

TABLE OF CONTENTS

3 About This Manual
3 About the Scrap Manager
4 Overview of the Desk Scrap
7 Desk Scrap Data Types
9 Using the Scrap Manager
10 Scrap Manager Routines
10 Getting Scrap Information
11 Keeping the Scrap on the Disk
12 Reading from the Scrap
12 Writing to the Scrap
13 Format of the Desk Scrap
15 Summary of the Scrap Manager
17 Glossary

Copyright (c) 1983 Apple Computer, Inc. All rights reserved. Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Scrap Manager, a new part of the Macintosh User Interface Toolbox in ROM version 7. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox. *** The Scrap Manager supports cutting and pasting between applications, desk accessories, or an application and a desk accessory.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- QuickDraw pictures

- Resources, as discussed in the Resource Manager manual

- The Toolbox Event Manager

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest only to assembly-language programmers is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Scrap Manager, an overview of the scrap that you manipulate with it, and a discussion of the types of data that the scrap may contain.

Next, a section on using the Scrap Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Scrap Manager routines, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions is a section that gives the exact format of the scrap, for those programmers who are interested; you don't have to read this section to be able to use the Scrap Manager routines.

Finally, there's a summary of the Scrap Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE SCRAP MANAGER

The Scrap Manager is a set of simple routines and data types that help Macintosh applications manipulate the <u>desk scrap</u>, which is where data that's cut (or copied) and pasted between applications is stored. An application can also use the desk scrap for storing data that's cut and pasted within the application, but usually it will have its own private scrap for this purpose. The format of the private scrap may be whatever the application likes, since no other application will use it.

From the user's point of view, there's a single place where all cut or copied data resides, and it's called the Clipboard. The Cut command deletes data from a document and places it in the Clipboard; the Copy command copies data into the Clipboard without deleting it from the document. The next Paste command—whether applied to the same document or another, in the same application or another--inserts the contents of the Clipboard at a specified place. An application that offers these editing commands will usually also have a special window for displaying the current Clipboard contents; it may show the Clipboard window at all times or only when requested (via the Show Clipboard and Hide Clipboard commands).

The desk scrap is the vehicle for transferring data not only between two applications but also between an application and a desk accessory, or even between two desk accessories. Desk accessories that display text will commonly allow the text to be cut or copied. The user might, for example, use the Calculator accessory to do a calculation and then copy the result into a document. It's also possible for a desk accessory to allow something to be pasted into it.

(hand)

> The Scrap Manager is optimized for transferring small amounts of data; attempts to transfer very large amounts of data may fail due to lack of memory.

The nature of the data to be transferred varies according to the application. For example, for the Calculator or a word processor the data is text, and for a graphics application it's a picture. The amount of information retained about the data that's transferred also varies. Between two text applications, text can be cut and pasted without any loss of information; however, if the user of a graphics application cuts a picture consisting of text and then pastes it into a document created with a word processor, the text in the picture may not be editable in the word processor, or it may be editable but not look exactly the same as in the graphics application. The Scrap Manager allows for a variety of data types and provides a mechanism whereby applications have control over how much information is retained when data is transferred.

Like any scrap, the desk scrap can be kept on the disk (in the scrap file) if there's not enough room for it in memory. It may remain on the disk throughout the use of the application but must be read back into memory when the application terminates, since the user may then remove that disk and insert another. The Scrap Manager provides routines for writing the desk scrap to the disk and for reading it back into memory.

OVERVIEW OF THE DESK SCRAP

The desk scrap is initially located in the application heap, with a handle to it in low memory. When starting up an application, the Segment Loader temporarily moves the scrap out of the heap into the

stack, reinitializes the heap, and puts the scrap back in the heap. (See Figure 1.) For a short time while it does this, two copies of the scrap exist in the memory allocated for the stack and the heap; for this reason, the desk scrap cannot be bigger than half that amount of memory.

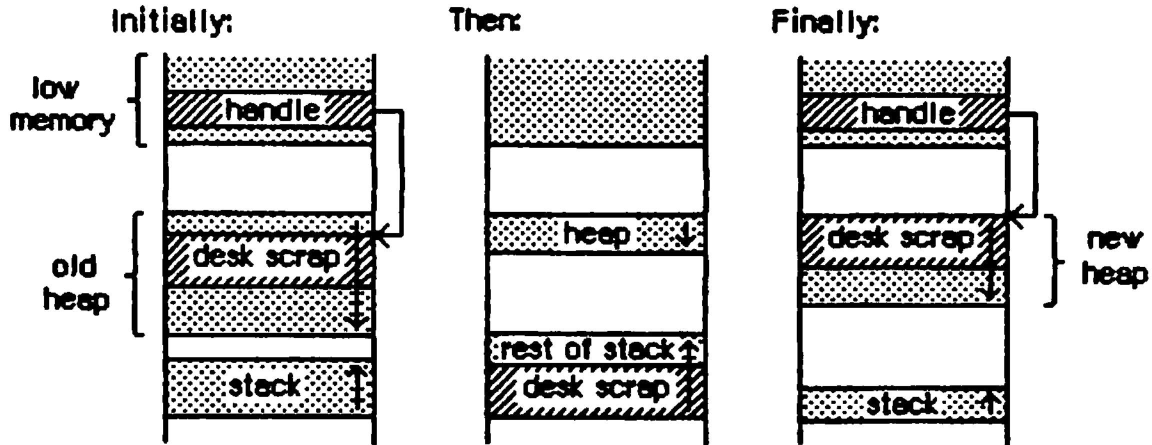


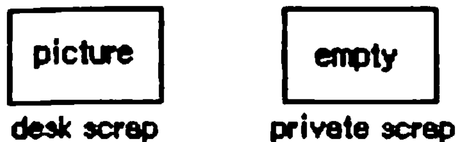
Figure 1. The Desk Scrap at Application Start-up

The application can get the size of the desk scrap by calling a Scrap Manager function named InfoScrap. An application concerned about whether there's room for the desk scrap in memory might be set up so that a small initial segment of the application is loaded in just to check out the scrap size. After a decision is made about whether to keep the scrap in memory or on the disk, the remaining segments can be loaded in as needed.

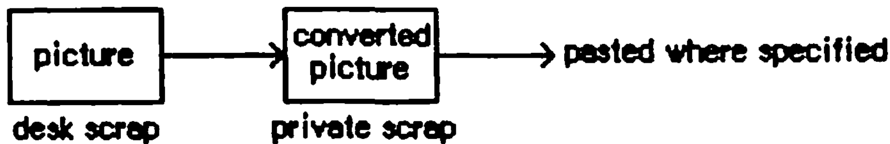
There are certain disadvantages to keeping the desk scrap on the disk. The disk may be write-protected, may not have enough room for the scrap, or may be removed during use of the application. If the application can't write the scrap to the disk, it should put up an alert box informing the user, who may want to abort the application at that point.

The application must use the desk scrap for any Paste command given before the first Cut or Copy command (that is, the first since the application started up or since a desk accessory was deactivated); this requires copying the desk scrap to the private scrap, if any. Clearly the application must keep the contents of the desk scrap intact until the first Cut or Copy command is given. Thereafter it can ignore the desk scrap until a desk accessory is activated or the application is terminated; in either of these cases, it must copy its private scrap to the desk scrap. Thus whatever was last cut or copied within the application will be pasted if a Paste command is then given in a desk accessory or in the next application.

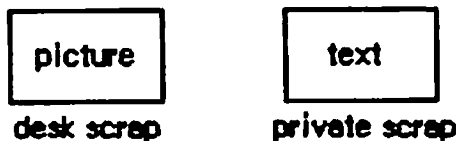
1. User enters word processor after cutting a picture in the previous application.



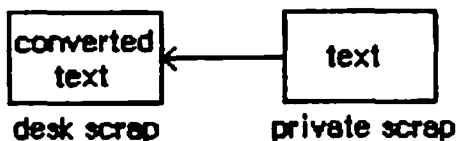
2. User gives Paste command in word processor (without a previous Cut or Copy).



3a. User cuts text in word processor.



3b. User leaves word processor.



OR:

3. User leaves word processor (without a previous Cut or Copy).

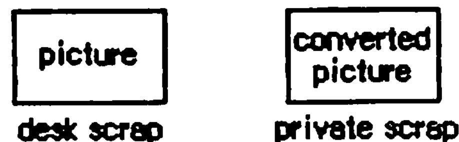


Figure 2. Interaction between Scraps

Figure 2 illustrates how the interaction between the desk scrap and the application's private scrap might occur when the user gives a Paste command in a word processor after cutting a picture in a graphics application. As the picture that was cut gets copied to the private scrap, it's converted to the format of that scrap. If the user leaves the word processor after cutting or copying text, the text first goes into the private scrap and then gets copied to the desk scrap. On the other hand, if the user never gives a Cut or Copy command, the application won't copy the private scrap to the desk scrap, so the original contents of the desk scrap will be retained.

Suppose the word processor in Figure 2 displays the contents of the Clipboard. Normally it will display its private scrap; however, to show the Clipboard contents at any time before step 2, it will have to display the desk scrap instead, or first copy the desk scrap to its private scrap. It can instead simply copy the desk scrap to its private scrap at start-up (step 1), so that showing the Clipboard contents will always mean displaying the private scrap.

A similar scheme to that shown in Figure 2 must be followed when the user reenters an application after using a desk accessory, since the user may have done cutting or copying in the accessory. The application can in fact check whether any such cutting or copying was done, by looking at a count that's returned by InfoScrap. If this count changes during use of the desk accessory, it means the contents

of the desk scrap have changed; the application will have to copy the desk scrap to the private scrap, if any, and update the contents of the Clipboard window, if there is one and if it's visible. If the count returned by InfoScrap hasn't changed, however, the application won't have to take either of these actions.

If the application encounters problems in trying to copy one scrap to another, it should alert the user. The desk scrap may be too large to copy to the private scrap, in which case the user may want to leave the application or just proceed with an empty Clipboard. If the private scrap is too large to copy to the desk scrap, either because it's disk-based and too large to copy into memory or because it exceeds the maximum size allowed for the desk scrap, the user may want to stay in the application and cut or copy something smaller.

DESK SCRAP DATA TYPES

From the user's point of view there can be only one thing in the Clipboard at a time, but internally there may be more than one data item in the desk scrap, each representing the same Clipboard contents in a different form. For example, text cut with a word processor may be stored in the desk scrap both as text and as a QuickDraw picture.

Desk scrap data types are like resource types. As defined in the Resource Manager, their Pascal type is as follows:

```
TYPE ResType = PACKED ARRAY [1..4] OF CHAR;
```

The Scrap Manager recognizes two standard types of data in the desk scrap.

- 'TEXT': a series of ASCII characters, preceded by a long word containing the number of characters and optionally followed by a comment, as described below.
- 'PICT': a QuickDraw picture, which is a saved sequence of drawing commands that can be played back with the DrawPicture command and may include picture comments. (See the QuickDraw manual for details.)

Applications must write at least one of these standard types of data to the desk scrap and must be able to read both types. Most applications will prefer one of these types over the other; for example, a word processor prefers text while a graphics application prefers pictures. An application should at least write its preferred standard type of data to the desk scrap, and ideally will write both types (to pass the most information possible on to the receiving application, which may prefer the other type).

An application reading the desk scrap will look for its preferred data type. If its preferred type isn't there, or if it's there but was written by an application having a different preferred type, some

information may be lost in the transfer process. For example, consider the user of a graphics application who cuts a picture consisting of text and then goes into a word processor and pastes it (as illustrated in Figure 3).

- If the graphics application writes only its preferred data type, picture, to the desk scrap (like application A in Figure 3), the text in the picture will not be editable in the word processor, because it will be seen as just a series of drawing commands and not a sequence of characters.

- On the other hand, if the graphics application takes the trouble of recognizing which characters have been drawn in the picture, and also writes them out to the desk scrap as text (like application B in Figure 3), the word processor will be able to treat them like any text, with editing or whatever. In this case, however, any part of the picture that isn't text will be lost.

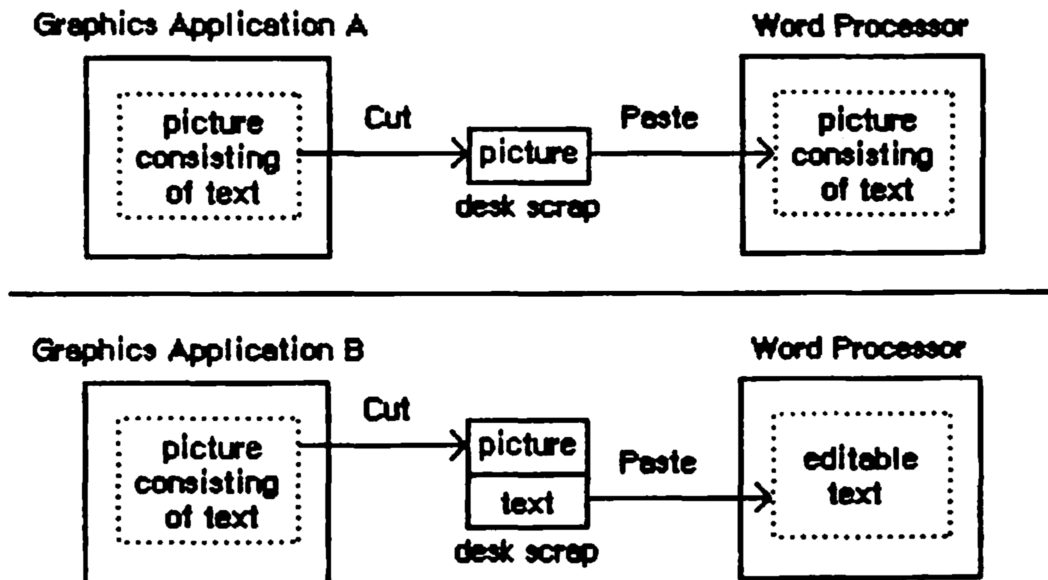


Figure 3. Inter-Application Cutting and Pasting

In addition to the two standard data types, the desk scrap may also contain application-specific types of data. If several applications are to support the transfer of a private type of data, each one will write and read that type--clearly its preferred type--but still must write at least one of the standard types and be able to read both standard types.

(eye)
There should never be more than one of each type of data in the desk scrap at a time.

The order in which data is written to the desk scrap is important: the application should write out the different types in order of preference. For example, if it's a word processor that writes out a

private type of data as well as text and pictures, it should do so in that order.

Since the size of the desk scrap is limited, it may be too costly to write out both an application-specific type of data and one (or both) of the standard types. If so, the comments that can accompany text or pictures might be useful. Instead of creating an application-specific data type, you may be able to encode additional information in these comments. For example, instead of having a data type that consists of text and formatting information combined in an application-specific way, you can encode the formatting information in the text comment. Applications that are to process that information can do so, while others can ignore it.

A text comment follows the last character in the text and must begin with the application ID, a four-character sequence that you choose to uniquely identify your application when you build it. *** (This ID will be discussed further in a future revision of the manual "Putting Together a Macintosh Application".) *** Any data that you like can follow the application ID.

As described in the QuickDraw manual, picture comments may be stored in the definition of a picture with the QuickDraw procedure PicComment. The DrawPicture procedure passes any such comments to a special routine set up by the application for that purpose.

USING THE SCRAP MANAGER

This section discusses how the Scrap Manager routines fit into the general flow of an application program and gives you an idea of which ones you'll need to use. The routines themselves are described in detail in the next section.

The application should inquire as early as possible about the size of the desk scrap to determine whether there will be enough room for itself and the scrap to coexist in the heap; it can do so by calling the InfoScrap function. If there won't be enough room for the desk scrap in the heap, the application should call the UnloadScrap procedure to write the scrap from memory onto the disk. InfoScrap also provides a handle to the desk scrap if it's in memory, its file name on the disk, and a count that's useful for testing whether the contents of the desk scrap have changed during the use of a desk accessory.

If a Paste command is given before the first Cut or Copy command after the application starts up, the application must copy the contents of the desk scrap to its private scrap, if any. It can do this either upon starting up or when the Paste command that needs to use the desk scrap is given. The latter method usually suffices, but applications that support display of the Clipboard will benefit from copying the desk scrap at start-up. The Scrap Manager routine that gets data from the desk scrap is called GetScrap.

When the user gives a command that terminates the application, the application's private scrap will usually have to be copied to the desk scrap. If the desk scrap is on the disk, it must first be read into memory with the LoadScrap function. The application must call ZeroScrap to reinitialize the desk scrap and clear its previous contents, and then PutScrap to put data in the scrap.

(eye)

> Do not copy the private scrap to the desk scrap unless a Cut or Copy command was given that changed the contents of the Clipboard.

The same kind of scrap interaction that occurs at application start-up also applies to returning to the application from a desk accessory (that is, an activate event that activates an application window after deactivating a system window). Similarly, the interaction when an application terminates also applies to accessing a desk accessory from the application (as reported by an activate event that deactivates an application window and activates a system window). Note, however, that a desk accessory shouldn't concern itself with writing or reading the desk scrap from the disk.

Cutting and pasting between two desk accessories follows an analogous scenario. As described in the Desk Manager manual, the way a desk accessory learns it must respond to an editing command is that its control routine receives a message telling it to perform the command; the application needs to call the Desk Manager function SystemEdit to make this happen.

SCRAP MANAGER ROUTINES

This section describes all the Scrap Manager routines. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** for now, see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Getting Scrap Information

FUNCTION InfoScrap : PScrapStuff;

InfoScrap returns a pointer to information about the desk scrap. The PScrapStuff data type is defined as follows:

```
TYPE PScrapStuff = ^ScrapStuff;
     ScrapStuff  = RECORD
                     scrapSize:   LongInt;
                     scrapHandle: Handle;
                     scrapCount:  INTEGER;
                     scrapState:  INTEGER;
                     scrapName:   StringPtr
                   END;
```

ScrapSize is the size of the entire desk scrap in bytes. ScrapHandle is a handle to the scrap if it's in memory, or NIL if not. ScrapCount is a count that changes every time ZeroScrap is called and is useful for testing whether the contents of the desk scrap have changed during the use of a desk accessory (see ZeroScrap under "Writing to the Scrap", below). ScrapState is positive if the desk scrap is in memory or Ø if it's on the disk. ScrapName is a pointer to the name of the scrap file, usually DeskScrap.

Keeping the Scrap on the Disk

FUNCTION UnloadScrap : LongInt;

UnloadScrap writes the desk scrap from memory to the scrap file. If the desk scrap is already on the disk, it does nothing. If no error occurs, UnloadScrap returns Ø; otherwise, it returns an appropriate Operating System error code.

Assembly-language note: The macro you invoke to call UnloadScrap from assembly language is named _UnlodeScrap.

FUNCTION LoadScrap : LongInt;

LoadScrap reads the desk scrap from the scrap file into memory. If the desk scrap is already in memory, it does nothing. If no error occurs, LoadScrap returns Ø; otherwise, it returns an appropriate Operating System error code.

Assembly-language note: The macro you invoke to call LoadScrap from assembly language is named _LodeScrap.

Reading from the Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType; VAR offset:
            LongInt) : LongInt;
```

GetScrap reads the data of type theType from the desk scrap (whether in memory or on the disk), makes a copy of it in memory, and sets up the hDest handle to point to the copy. Usually you'll pass an empty handle in hDest. In the offset parameter, GetScrap returns the location of the data as an offset (in bytes) from the beginning of the desk scrap. If no error occurs, the function result is the length of the data in bytes; otherwise, it's either an appropriate Operating System error code (which will be negative) or the following predefined constant:

```
CONST noTypeErr = -1Ø2;  {there's no data of the requested type}
```

For example, given an empty handle declared as

```
VAR pHndl: PicHandle
```

you can make the following calls:

```
GetScrap(POINTER(ORD(pHndl)),'PICT');
DrawPicture(pHndl);
```

Your application should pass its preferred data type to GetScrap. If it doesn't prefer one data type over any other, it should try getting different types until the offset returned is Ø. An offset of Ø means that data was the first to be written out and so should be the preferred type of the application that wrote it.

If you pass NIL in hDest, GetScrap will not read in the data. This is useful if you want to be sure the data is there before allocating space for its handle, or if you just want to know the size of the data. If there isn't enough room in memory for a copy of the data, as may be the case for a complicated picture, you can customize QuickDraw's picture retrieval so that DrawPicture will read from the picture directly from the scrap file. (QuickDraw also lets you customize how pictures are saved so you can save them in a file; see the QuickDraw manual for details about customizing.)

Writing to the Scrap

```
FUNCTION ZeroScrap : LongInt;
```

ZeroScrap initializes the desk scrap, clearing its contents; you must call it before the first time you call PutScrap (described below). If no error occurs, ZeroScrap returns Ø; otherwise, it returns an

appropriate Operating System error code.

ZeroScrap also changes the scrapCount field of the record of information provided by InfoScrap. This is useful for testing whether the contents of the desk scrap have changed during the use of a desk accessory. The application can save the value of the scrapCount field when one of its windows is deactivated and a system window is activated. Then, each time through its event loop, it can check to see whether the value of the field has changed. If so, it means the desk accessory called ZeroScrap (and, presumably, PutScrap) and thus changed the contents of the desk scrap.

```
FUNCTION PutScrap (length: LongInt; theType: ResType; source: Ptr) :
            LongInt;
```

PutScrap writes the data pointed to by the source parameter to the desk scrap (whether in memory or on the disk). The length parameter indicates the number of bytes to write, and theType is the data type (which should be different from the type of any data already in the desk scrap). If no error occurs, the function result is Ø; otherwise, it's an appropriate Operating System error code.

(eye)

Don't forget to call ZeroScrap (above) to clear the scrap before your first call to PutScrap.

FORMAT OF THE DESK SCRAP

In general, the desk scrap consists of a series of data items that have the following format:

Number of bytes	Contents
4 bytes	Type (a sequence of four characters)
4 bytes	Length of following data in bytes
n bytes	Data; n must be even (if the above length is odd, include an extra byte)

The standard types are 'TEXT' and 'PICT'. You may use any other four-character sequence for types specific to your application.

The format of the data for the 'TEXT' type is as follows:

Number of bytes	Contents
4 bytes	Number of characters in the text
n bytes	The characters in the text
m bytes	Optional comment: the 4-byte application ID followed by any information desired

The data for the 'PICT' type is a QuickDraw picture, which consists of the size of the picture in bytes, the picture frame, and the picture

definition data (which may include picture comments). See the QuickDraw manual for details.

SUMMARY OF THE SCRAP MANAGER

Constants

```
CONST noTypeErr = -1Ø2;  {there's no data of the requested type}
```

Data Structures

```
TYPE PScrapStuff = ^ScrapStuff;
     ScrapStuff  = RECORD
                     scrapSize:   LongInt;
                     scrapHandle: Handle;
                     scrapCount:  INTEGER;
                     scrapState:  INTEGER;
                     scrapName:   StringPtr
                   END;
```

Routines

Getting Scrap Information

```
FUNCTION InfoScrap : PScrapStuff;
```

Keeping the Scrap on the Disk

```
FUNCTION UnloadScrap : LongInt;
FUNCTION LoadScrap :   LongInt;
```

Reading from the Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType; VAR offset: LongInt)
                  : LongInt;
```

Writing to the Scrap

```
FUNCTION ZeroScrap : LongInt;
FUNCTION PutScrap   (length: LongInt; theType: ResType; source: Ptr) :
                     LongInt;
```

Assembly-Language Information

Constants

```
noTypeErr     .EQU    -1Ø2     ;there's no data of the requested type
```

Scrap Information Data Structure

scrapSize	Size of desk scrap in bytes *** (currently named scrapInfo) ***
scrapHandle	Handle to desk scrap in memory
scrapCount	Count changed by ZeroScrap
scrapState	Positive if desk scrap in memory, Ø if on disk
scrapName	Pointer to name of scrap file

Special Macro Names

Routine name	Macro name
LoadScrap	_LodeScrap
UnloadScrap	_UnlodeScrap

GLOSSARY

application ID: A four-character sequence that you choose to identify your application when you you build it.

desk scrap: The place in memory or on the disk where data that's cut (or copied) and pasted between applications is stored.

scrap file: The file containing the desk scrap.

MACINTOSH USER EDUCATION

The Segment Loader: A Programmer's Guide

/SEGLOAD/SEGMENT

See Also: Macintosh Operating System Reference Manual
The Resource Manager: A Programmer's Guide
The Macintosh Finder

Modification History: First Draft (ROM 4) C. Rose 6/24/83

ABSTRACT

This manual describes the Segment Loader of the Macintosh Operating System, which lets you divide your application into several parts and have only some of them in memory at a time.

TABLE OF CONTENTS

3 About This Manual
3 About the Segment Loader
4 Application Parameters
5 Using the Segment Loader
5 Segment Loader Routines
7 Advanced Routines
8 The Jump Table
1Ø Specifying Segments in Your Source File
13 Summary of the Segment Loader
14 Glossary

ABOUT THIS MANUAL

This manual describes the Segment Loader, a new part of the Macintosh Operating System in ROM version 4. *** Eventually it will become part of a large manual describing the entire Operating System and Toolbox. *** The Segment Loader lets you divide your application into several parts and have only some of them in memory at a time.

You should already be familiar with Lisa Pascal, the Macintosh Operating System's Memory Manager, the Finder, and the basic concepts behind the Resource Manager of the Macintosh User Interface Toolbox.

The manual begins with an introduction to the Segment Loader and a description of the parameters that are stored in memory when an application is started up. Next, a section on using the Segment Loader introduces you to its routines and tells how they fit into the flow of your application. This is followed by the detailed descriptions of all Segment Loader routines, their parameters, calling protocol, effects, side effects, and so on.

For advanced programmers, there's a section that discusses the jump table, explaining how the Segment Loader works internally.

Finally, there's a summary of the Segment Loader routine calls, for quick reference, and a glossary of terms defined in this manual.

ABOUT THE SEGMENT LOADER

The Segment Loader allows you to divide the code of an application into several parts or segments. The Finder starts up an application by calling a Segment Loader routine that loads in the main segment (the one containing the main program). Other segments are loaded in automatically when they're needed. Your application can call the Segment Loader to have these other segments removed from memory when they're no longer needed.

The Segment Loader enables you to have programs larger than 32K bytes, the maximum size of a single segment. Also, any code that isn't executed often (such as code for printing hardcopy) need not occupy memory when it isn't being used, but can instead be in a separate segment that's brought in when needed.

This mechanism may remind you of the resources of an application, which the Resource Manager of the User Interface Toolbox reads into memory when necessary. An application's segments are in fact themselves stored as resources; their resource type is 'CODE'. You can use the Resource Compiler to create these resources from your application code. A "loaded" segment has been read into memory by the Resource Manager and locked (so that it's neither relocatable nor purgeable). When a segment is unloaded, it's made relocatable and purgeable.

Every segment has a name. If you do nothing about dividing your program into segments, it will consist of a single segment whose name is blank. Dividing your program into segments means specifying in your source file the beginning of each segment by name. The names are for your use only; they're not kept around after linking.

(eye)
If you do specify segment names, note that normally the main segment should have a blank name. The reason for this is that the intrinsic Pascal routines must be in the same segment as your main program, and the Linker puts those routines in the blank-named segment (so that the right thing will happen if you don't specify any segment names at all).

APPLICATION PARAMETERS

When an application is started up, certain parameters are stored in 32 bytes of memory just above the application's globals, as shown in Figure 1; these are called the application parameters. A5 points to the first of these parameters and may be used with positive offsets to access the others.

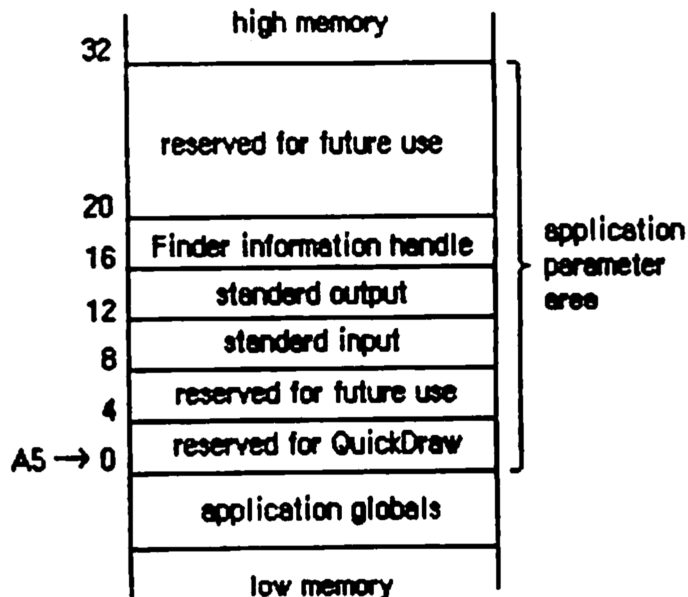


Figure 1. Application Parameters

(hand)
For brevity, we'll say "A5" where we mean "the location pointed to by A5".

The "standard input" and "standard output" parameters indicate the main source of input and destination of output for the Macintosh. They are

usually Ø, meaning the keyboard and the screen, respectively.

The "Finder information handle" is a handle to information that the Finder provides to the application upon starting it up. For example, for a word processor it might be the name of the document to be worked on. *** The exact information will be described here when available. *** Pascal programmers can call the Segment Loader routine GetAppParms to get the Finder information handle.

The other locations in the application parameter area are reserved for future use or for use by QuickDraw.

USING THE SEGMENT LOADER

This section introduces you to the Segment Loader routines and how they fit into the flow of an application program. The routines themselves are described in detail in the next section.

The routine that applications will most commonly use is UnloadSeg, for unloading a particular segment when it's no longer needed. Another useful routine, GetAppParms, lets you get information about your application such as its name and the reference number for its resources. For applications started up in the usual way by the Finder, GetAppParms also gives the Finder information handle that's stored 16 bytes above A5.

The main segment can unload other segments, but it can't get rid of itself; using the Chain routine, however, it can do something close to this. Chain starts up another application without disturbing the application heap. Thus the current application can let another application take over while still keeping its data around in the heap.

The Segment Loader also provides a quick exit to the Finder that doesn't touch the stack, for applications needing it in emergency situations: ExitToShell.

Finally, there are two advanced routines that most applications will never use: Launch and LoadSeg. Launch is called by the Finder to start up an application; it's like Chain but doesn't retain the application heap. LoadSeg is called indirectly (via the jump table, as described later) to load segments when necessary--that is, whenever a routine in an unloaded segment is invoked.

SEGMENT LOADER ROUTINES

This section describes all the Segment Loader routines. Some of the routines are stack-based and so are shown in Pascal; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see QuickDraw manual ***. Other Segment Loader routines are register-based and are described similar to

the way the Operating System routines are described in the current Operating System manual.

PROCEDURE UnloadSeg (routineAddr: Ptr);

UnloadSeg unloads a segment, making it relocatable and purgeable; routineAddr is the address of any routine in the segment. The Segment Loader will reload the segment the next time one of the routines in it is called. It doesn't hurt to call UnloadSeg, because the segment won't actually be purged until the memory it occupies is needed. If you need the unloaded segment again before it's purged, the Segment Loader won't have to access the disk.

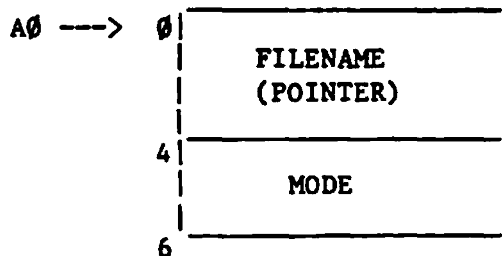
PROCEDURE GetAppParms (VAR apName: Str255; VAR apRefNum: INTEGER; VAR
    apParam: Handle);

GetAppParms returns information about the current application. It returns the application name in apName and the reference number for the application's resources in apRefNum. For applications started up in the usual way by the Finder, it returns the Finder information handle in apParam (as described earlier under "Application Parameters").

(hand)
> For applications started up with the Chain routine (below), the apParam parameter isn't useful.

Chain {register-based}

This routine starts an application up without doing anything to the application heap, so the current application can let another application take over while still keeping its data around in the heap. It configures memory for the sound and video buffers. AØ points to the following:



where FILENAME is a pointer to the application's file name

and MODE tells which sound buffer and video buffer to use (Ø for standard).

The sound and video buffers are constantly scanned by the Macintosh hardware to determine what sounds to emit from its speakers and what to display on its screen. (The video buffer is the bit image corresponding to the display screen.) Two of each type of buffer are available; Figure 2 shows where they're located. If you specify a MODE

value of Ø, you get the standard or "primary" buffers; in this case, the application space begins where shown in Figure 2. Any positive MODE value causes the secondary sound buffer and primary video buffer to be used (which costs 1.5K of memory). Any negative MODE value causes the secondary sound buffer and secondary video buffer to be used (which costs 32K of memory).

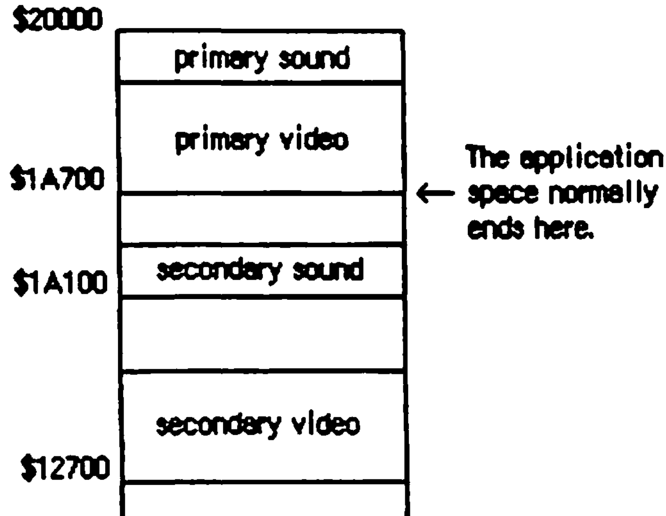


Figure 2. Sound and Video Buffers

Chain closes the resource file for any previous application and opens the resource file for the application being started. It also stores in memory the application parameters designating standard input and standard output. The application is started at its entry point, which causes the main segment to be loaded.

PROCEDURE ExitToShell;

ExitToShell provides an emergency exit for the application, without touching the stack. It simply launches the Finder (starts it up after freeing the storage occupied by the application heap; see Launch below).

Advanced Routines

Launch {register-based}

This routine is called by the Finder to start up an application and will rarely need to be called by an application itself. It's the same as the Chain routine (described above) except that it frees the storage occupied by the application heap and restores the heap to its original size. Also, the Finder provides startup information needed by the application; a handle to the information is located in the system heap

and is copied (as the "Finder information handle") into the application parameter area in memory.

(hand)
Launch preserves a special handle in the application heap which is used for accessing the scrap between applications.

PROCEDURE LoadSeg (segID: INTEGER);

LoadSeg is called indirectly via the jump table (as described in the following section) when the application calls a routine in an unloaded segment. It loads the segment having the given ID number, which was assigned by the Linker. If the segment isn't in memory, LoadSeg calls the Resource Manager to read it in. It changes the jump table entries for all the routines in the segment from the "unloaded" to the "loaded" state and then invokes the routine that was called.

THE JUMP TABLE

This section describes how the Segment Loader works internally, and is included here for advanced programmers; you don't have to know about this to be able to use the common Segment Loader routines.

The loading and unloading of segments is implemented through the application's jump table. Figure 3 shows the location of the jump table in memory for a typical application.

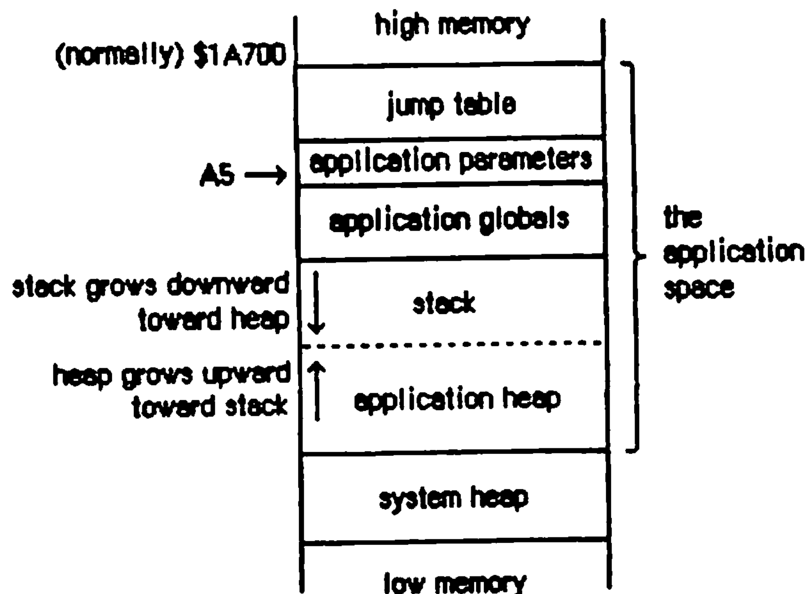


Figure 3. The Application's Space in Memory

When the Linker encounters a call to a routine in another segment, it creates a jump table entry for the routine and addresses the entry with a positive offset from A5. As described below, the jump table entry makes the connections necessary to invoke the routine.

The jump table contains one 8-byte entry for every externally referenced routine in every segment; all the entries for a particular segment are stored contiguously. It refers to segments by ID numbers assigned by the Linker. When an application is started up, its jump table is read in from segment Ø, a special segment created by the Linker for every executable file. Segment Ø contains the following:

Number of bytes	Contents
4 bytes	"Above A5" size; size in bytes from A5 to upper end of application space
4 bytes	"Below A5" size; size in bytes of application globals
4 bytes	Offset of jump table from A5
4 bytes	Length of jump table in bytes
n bytes	Jump table

For most applications, the offset of the jump table from A5 is 32, and the "above A5" size is 32 plus the length of the jump table.

All the jump table entries for a particular segment indicate whether that segment is currently loaded or not, as illustrated in Figure 4.

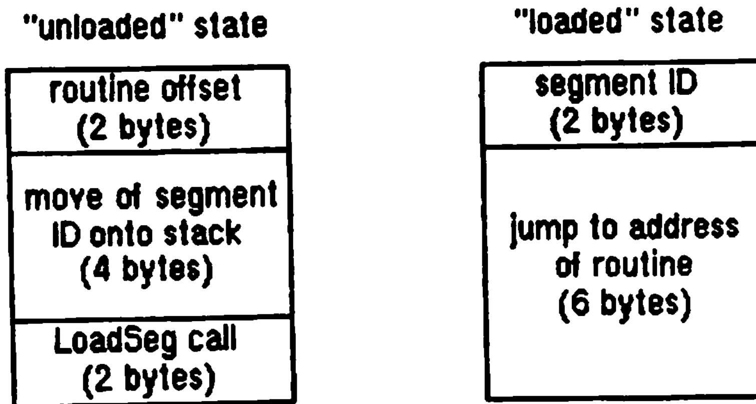


Figure 4. Format of a Jump Table Entry

Initially, of course, the jump table entries are all in the "unloaded" state, which means they contain the following:

Number of bytes	Contents
2 bytes	Offset of this routine from beginning of segment
4 bytes	Instruction that moves the segment ID onto the stack for LoadSeg
2 bytes	Trap that executes LoadSeg

When a call to a routine in an unloaded segment is made, the code in the last six bytes of its jump table entry is executed. This code calls LoadSeg, which loads the segment into memory, transforms all of its jump table entries to the "loaded" state (shown below), and invokes the routine.

Number of bytes	Contents
2 bytes	Segment ID
6 bytes	Instruction that jumps to the address of the routine for which this is an entry

LoadSeg invokes the routine by executing the instruction in the last six bytes of the jump table entry. Subsequent calls to the routine also execute this instruction. If UnloadSeg is called to unload the segment, it restores the jump table entries to their "unloaded" state. Notice that whether the segment is loaded or unloaded, the last six bytes of the jump table entry are executed; the effect depends on the state of the entry at the time.

To be able to set all the jump table entries for a segment to a particular state, LoadSeg and UnloadSeg need to know exactly where all the entries are located. They get this information from the segment header, four bytes at the beginning of the segment which contain the following:

Number of bytes	Contents
2 bytes	Offset of the first routine's entry from the beginning of the jump table
2 bytes	Number of entries for this segment

As described above, segment Ø tells where the beginning of the jump table is located.

SPECIFYING SEGMENTS IN YOUR SOURCE FILE

*** This section will be moved into the next version of the manual entitled "Putting Together a Macintosh Application". ***

You specify the beginning of a segment in your application's source file as follows:

```
{$S segname}
```

where segname is the segment name, a sequence of up to eight characters. Normally you should give the main segment a blank name.

For example, you might structure your program as follows:

```
PROGRAM Shell;

{ The USES statement and your LABEL, CONST, and VAR declarations
  will be here. }

{$S Seg1}

{ The procedures and functions in Seg1 will be here. }

{$S Seg2}

{ The procedures and functions in Seg2 will be here. }

{$S     }

BEGIN

  { The main program will be here. }

END.
```

You can specify the same segment name more than once; the routines will just be accumulated into that segment. To avoid problems when moving routines around in the source file, some programmers follow the practice of putting a segment name specification before every routine.

(eye)
Uppercase and lowercase letters ARE distinguished in segment names. For example, "Seg1" and "SEG1" are not equivalent names.

If you don't specify a segment name before the first routine in your file, the blank segment name will be assumed there.

In assembly language, you specify the beginning of a segment with the following directive:

```
.SEG 'segname'
```

(eye)
This requires version 12.2 of the Lisa Monitor.

You can also specify what segment the routines in a particular file should be in by using the ChangeSeg program. For example, suppose you want to give your main segment a nonblank name (say, "SegMain"); you can't do this without using ChangeSeg, because the Linker puts the intrinsic Pascal routines in the blank-named segment, and they must be in the same segment as your main program. You can use ChangeSeg as shown below to tell the Linker to put the intrinsic Pascal routines, which are in Obj:MacPasLib, in the segment named SegMain.

Prompt	Response
Monitor command line	X {for X(ecute}
What file ?	ChangeSeg <ret>
File to change:	Obj:MacPasLib <ret>
Map all Names ? (Y/N)	Y {for Yes}
New Seg name ?	SegMain <ret>

SUMMARY OF THE SEGMENT LOADER

```
PROCEDURE UnloadSeg    (routineAddr: Ptr);
PROCEDURE GetAppParms  (VAR apName: Str255; VAR apRefNum: INTEGER;
                        VAR apParam: Handle);
```

Chain {register-based}

Input: AØ points to application's file name pointer followed by a word telling which sound and video buffers to use.

Output: The application parameters for standard input and output.

```
PROCEDURE ExitToShell;
```

Advanced Routines

Launch {register-based}

Input: AØ points to application's file name pointer followed by a word telling which sound and video buffers to use.

Output: The application parameters--standard input and output and the Finder information handle.

```
PROCEDURE LoadSeg (segID: INTEGER);
```

GLOSSARY

application parameters: Information stored in 32 bytes of memory just above the application globals when an application is started up.

jump table: A table that contains one entry for every routine in an application and is the means by which the loading and unloading of segments is implemented.

main segment: The segment containing the main program.

segment: One of several parts into which the code of an application may be divided. Not all segments need to be in memory at the same time.

Date: 3 October 1983

To: Mac Developers

Re: Mac serial connector pinout

1 - GND
2 - +5 (may turn into an output handshake line—don't use!)
3 - GND
4 - TXD+
5 - TXD-
6 - +12 (for detecting power on ONLY!)
7 - HSK (CTS or TRxC, depending on 8530 mode)
8 - RXD+
9 - RXD-

For more-or-less RS232, use GND, TXD-, RXD-. The TXD+ and RXD+ signals provide RS422/423 compatibility.

The HSK (handshake) line (an input) connects to both CTS and to TRxC on the 8530. It can be used either for CTS, or for external clock depending on the mode the 8530 is in. As RS232 handshake, it usually connects to pin 20 on a DB-25.

For the Exceedingly Curious...

Signal lines go through a "deglitch" network, which is a "T" consisting of 25-50 ohm resistors and 200 pf to ground.

We use 26LS30 and 26LS32 interface chips between the 8530 SCC and the outside world.

The 8530 Data Carrier Detect lines (*DCDA pin 19 and *DCDB pin 21) are used as mouse inputs and generate interrupts used to detect mouse motion (!!!).

The Mouse Pinout

1 - GND
2 - +5 (Mouse ONLY!)
3 - GND
4 - X2 (to 6522 PB4)
5 - X1 (to 8530 *DCDA)
6 - No connect
7 - *SW (to 6522 PB3)
8 - Y2 (to 6522 PB5)
9 - Y1 (to 8530 *DCDB)

Bob Martin

Asynchronous Serial Driver
04-Feb-83
28-Apr-83 (revised)
16-Jun-83 (revised)
23-Aug-83 (revised)
LAK

changes: async drivers are no longer opened at system initialization time;
CTS and break change status are now optionally posted as events;
added XON/XOFF input flow control.

This ROM-based driver supports full duplex asynchronous mode RS-232/RS-422 communication on the two independent Macintosh serial ports (also called channels). Asynchronous serial mode communication is a term used to describe serial communication (data is sent over one hardware data line) in which time intervals between transmitted characters may be of unequal length: transmission is controlled by a 'start bit' at the beginning and between one and two 'stop bits' at the end of each character. The individual data bits of the character (usually 7 or 8) and the stop and start bits are sent at a constant frequency termed the 'baud rate'. The baud rate, number of data bits per character, inclusion of an optional parity bit, etc., are all options which make asynchronous communication configuration tricky to work with; this serial driver supports most asynchronous-mode options using control calls.

We call the Macintosh hardware chip which controls this serial communication the SCC chip (for Serial Communications Controller); this chip supports the two independent serial ports which we call port A and port B. The connectors for port A and port B are located at the bottom rear of the Macintosh: port A is the port closest to the side of the Macintosh.

Both the input and output refnums associated with a port must be opened; all four serial driver refnums are installed at system initialization time but left unopened. When opened, the serial port associated with the driver is initialized according to the corresponding parameter ram bytes: the parameter ram is initialized to 9600 baud, 8 data, 2 stop, and no parity bits for both ports.

A KillIO call to either input or output refnum will cause all current requests to be aborted and any available buffer bytes to be discarded.

Hardware (CTS) and XON/XOFF output flow control are supported; XON/XOFF input flow control is also supported. A break condition on the line always terminates input requests, but not output requests. Parity errors, overruns, and framing errors optionally terminate input requests. The driver will optionally post events on CTS and break status changes.

The input drivers buffer up to 64 bytes of data with no request pending to avoid overflow.

Refnums for the serial ports are assigned as follows:

```
.AIn     = unit -6  serial port A input
.AOut    = unit -7  serial port A output
```

```
.BIn        = unit -8  serial port B input
.BOut       = unit -9  serial port B output
```

Programming Using the Serial Drivers

The following calls are generally used:

```
Open (".AIn"): refnum1;
Open (".AOut"): refnum2;
Control (refnum1,8,ConfigWord);
Read (refnum1) or Write (refnum2);
Close (refnum1);
Close (refnum2);
```

Control Calls Supported

(Opcode 1 is now used for KillIO).

For operation code 8, the appropriate SCC channel is reset and reinitialized according to the new defaults. The configuration is specified by a word (16 bits) of information in the same format as the clock parameter ram data. Either input or output port driver refnum may be used:

```
A0 ->   (0)  header
        (24) refnum
        (26) $0008
        (28) [V][V][W][W][X][X][Y][Y] [Z][Z][Z][Z][Z][Z][Z][Z]

            VV = 1,2,3,    for 1,1.5,2 stop bits
            WW = 0,1,2,3   for no,odd,no,even parity
            XX = 0,1,2,3,  for 5,7,6,8 data bits
            YY = high byte of baud rate constant, low 2 bits
            ZZZZZZZZ = low byte of baud rate constant
            $CC0A = default (9600 baud, 8 data, 2 stop, no parity bit)
```

YYZZZZZZZZ	baud rate
$17C	300
$0BD	600
$05E	1200
$03E	1800
$02E	2400
$01E	3600
$016	4800
$00E	7200
$00A	9600
$004	19200
$000	57600

Opcode 9 is used to install a new buffer for input buffering: a pointer to the buffer and the buffer length are passed. The async input driver uses

this buffer to store input characters when no input user request is pending; the buffer must be locked down in memory:

```
A0 ->   (0)  header
        (24) refnum
        (26) $0009
        (28) pointer to buffer
        (32) buffer length
        (34) unused
```

Opcode 10 is used to specify handshake options and other miscellaneous controls:

```
A0 ->   (0)  header
        (24) refnum
        (26) $000A
        (28) enable XON/XOFF output handshake if non-zero
        (29) enable CTS output handshake if non-zero
        (30) XON char for software handshake
        (31) XOFF char for software handshake
        (32) errors which cause abort of input requests (1 for abort):
                bit 4 = abort on parity error
                bit 5 = abort on overrun error
                bit 6 = abort on framing error
        (33) status changes which cause events to be posted
                bit 7 = post event on break status change
                bit 5 = post event on CTS change
        (34) enable XON/XOFF input flow control if non-zero (the same
             handshake characters are used for both input and output
             software flow control)
```

Opcode 11 is used to reinitialize the SCC to clear break mode:

```
A0 ->   (0)  header
        (24) refnum
        (26) $000B
```

Opcode 12 is used to set break mode in the SCC channel:

```
A0 ->   (0)  header
        (24) refnum
        (26) $000C
```

<u>Status Calls Supported</u>

For operation code 2, a longword count of the available buffered bytes is returned; either input or output port driver refnum may be used:

```
A0 ->   (0)  header
        (24) refnum
        (26) $0002
        (28) buffered bytes available
```

For operation code 8, three words of status information are returned;

either input or output port driver refnum may be used:

```
A0 ->   (0)  header
        (24) refnum
        (26) $0008
        (28) cumulative errors:
                        bit 8  = soft overrun (local buffer overflow)
                        bit 12 = parity error
                        bit 13 = hard overrun error
                        bit 14 = framing error
        (29) XOFF has been sent to stop input data
        (30) read command pending flag
        (31) write command pending flag
        (32) CTS flag
        (33) XOFF flag
```

<u>Open Routines</u>

The Open routines for the SCC async-mode drivers initialize local variables, allocate buffer storage, install interrupt vectors, and initialize the correct SCC channel according to clock parameter ram values. For input drivers, only the Device Control Entry pointer is noted: SCC initialization is done for output drivers only.

An 'Open' of the RefNum associated with an output port will install interrupt receivers and enable interrupts for both input and output; two 'Open's need to be done for a port to configure input and output DCEs; the Open for the input driver can be done before or after the Open for the output driver.

<u>Miscellaneous Notes</u>

This driver uses four device control blocks: two per port, one input and one output. The input and output "drivers" are closely associated: control and status routines are the same for input and output drivers; the open, close and prime routines differ. The reason for using two device control blocks for one port is simply to support the general full-duplex communication capability of the SCC: both a read and a write request may be executing at the same time for a single port.

Port A now has an added feature: it may be used simultaneously with disk accesses at the highest async baud rate with no worry of overrun. The disk driver now polls SCC port A whenever it must turn interrupts off for longer than 100 microseconds, and then passes any acquired data to the async driver. SCC channel B should be used for output-only connections such as to printers, at low baud rates (a 300 baud modem, for instance), or with protocols which can recover from missed data.

MACINTOSH USER EDUCATION

The Sound Driver: A Programmer's Guide /DEVICE/SOUND

See Also: The Macintosh User Interface Guidelines
Macintosh Operating System Manual
The Device Manager: A Programmer's Guide

Modification History: First Draft (ROM 7) B. Hacker 3/nn/84

*** Preliminary Draft. Not for distribution *** ABSTRACT

The Sound Driver is a set of data types and routines in the Macintosh Operating System for handling sound and music generation in a Macintosh application. This manual describes the Sound Driver in detail.

TABLE OF CONTENTS

3 About This Manual
3 About The Sound Driver
6 Sound Driver Synthesizers
7 Free-Form Synthesizer
8 Square-Wave Synthesizer
9 Four-Tone Synthesizer
11 Using The Sound Driver
12 Advanced Control Routine
14 Summary of the Sound Driver
18 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

The Sound Driver is a set of data structures and routines in the Macintosh Operating System for handling sound and music generation in a Macintosh application. This manual describes the Sound Driver in detail. *** Eventually it will become part of a larger manual describing the entire Toolbox and Operating System. ***

(note)
> This manual describes the Sound Driver in version 7 of the ROM. If you're using a different version, the information presented here may not apply.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager
- devices and device drivers, as described in the Device Manager Manual *** doesn't yet exist ***

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it. *** Currently a Pascal interface to the Sound Driver doesn't exist ***

The manual begins with an introduction to the Sound Driver and what you can do with it. It then steps back a little and looks at the mathematical and physical concepts that form the foundation for the Sound Driver: waveforms, wave frequency, wave amplitude, and wave periods. Once you understand these concepts, read on about how they're translated into sound, music, and speech.

Next, a section on using the Sound Driver describes how you can use Device Manager calls in your application to produce desired sounds. This includes a detailed description of the Sound Driver's control routine—its parameters, calling protocol, effects, and so on.

Finally, there's a summary of the Sound Driver data structures and routine calls, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE SOUND DRIVER

The Sound Driver is a standard Macintosh device driver used to synthesize sound waves. You can use the Sound Driver to generate sound characterized by any kind of waveform by using the three different sound synthesizers in the Sound Driver:

- The four-tone synthesizer is used to make simple harmonic tones, with up to four "voices" producing sound simultaneously; it requires about 50% of the microprocessor's attention during any given time interval.

- The square-wave synthesizer is used to produce less harmonic sounds such as beeps, and requires about 2% of the processor's time.

- The free-form synthesizer is used to make complex music and speech; it requires about 20% of the processor's time.

Figure 1 depicts the waveform of a typical sound wave, and the terms used to describe it. The amplitude is the vertical distance between any given point on the wave and the horizontal line about which the amplitude oscillates; you can think of the amplitude of a wave as its volume level. The wavelength is the horizontal extent of one complete cycle of the wave. Both the amplitude and wavelength can be measured in any unit of distance. The period is the time elapsed during one complete cycle of a wave. The frequency is the reciprocal of the period, or the number of cycles per second (also called Hertz). The phase is some fraction of a wave cycle (measured from a fixed point on the wave).

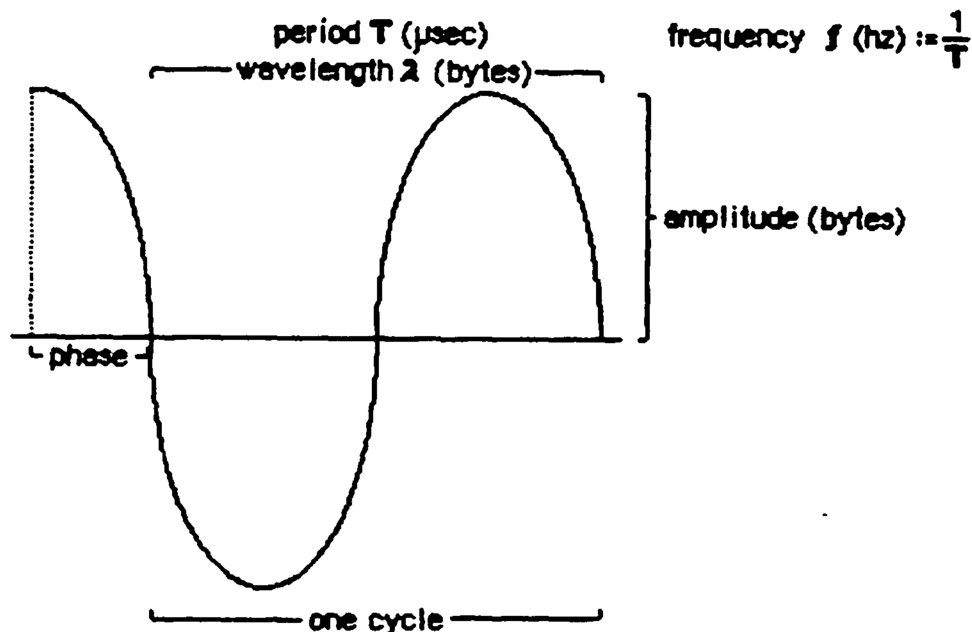
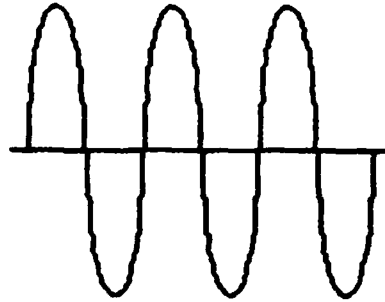


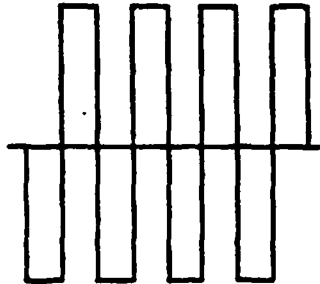
Figure 1. A Waveform

There are many different types of waveforms, three of which are depicted in Figure 2. Sine waves are generated by objects that oscillate periodically at a single frequency (such as a guitar string). Square waves are generated by objects that toggle instantly between two states at a single frequency (such as a doorbell buzzer). Free-form waves are the most common waves of all, and are generated by all

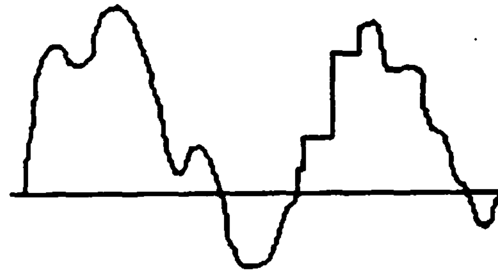
objects that vibrate at rapidly changing frequencies with rapidly changing amplitudes (such as your vocal cords or the instruments of an orchestra all playing at once).



sine wave



square wave



free-form wave

Figure 2. Types of Waveforms

Figure 3 shows the analog representation of a waveform. The Sound Driver represents waveforms digitally, so all waveforms must be converted from their analog representation to a digital representation. The rows of numbers at the bottom of the figure are digital representations of the waveform. The numbers in the upper row are the amplitudes relative to the horizontal zero-amplitude line. The numbers in the lower row all represent the same relative amplitudes, but have been normalized to positive numbers.

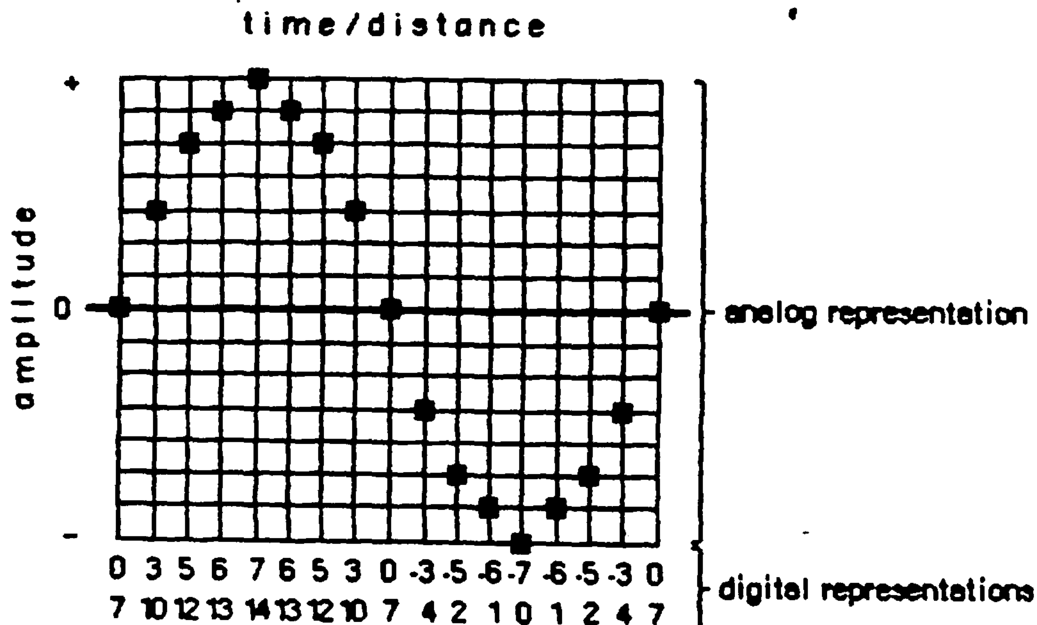


Figure 3. Analog and Digital Representations of a Waveform

A digital representation of a waveform is simply a sequence of wave amplitudes measured at fixed intervals. This sequence of amplitudes is stored in the Sound Driver as a sequence of bytes, each one of which specifies an instantaneous voltage to be sent to the speaker. The bytes are stored in a data structure called a <u>waveform description</u>. Since a sequence of bytes can only represent a group of numbers whose maximum and minimum values differ by less than 256, the amplitudes of your waveforms must be constrained to these same limits.

SOUND DRIVER SYNTHESIZERS

A description of the sound to be generated by a synthesizer is contained in a data structure called a <u>synthesizer buffer</u>. A synthesizer buffer contains the duration, pitch, phase, and waveform of the sound the synthesizer will generate. The exact structure of a synthesizer buffer differs for each type of synthesizer being used.

Free-Form Synthesizer

The free-form synthesizer is used to synthesize complex music and speech. The sound to be produced is represented as a waveform whose complexity and length are limited only by available memory.

A free-form synthesizer buffer consists of one integer and one long integer followed by a waveform description (Figure 4). The waveform description can contain up to 256 bytes. Each amplitude in the waveform description will be generated once; when the end of the

waveform is reached, the synthesizer will stop. The integer must be Ø, to identify the buffer as a free-form buffer. The duration long integer determines the length of time (in 44.93 usec increments) each amplitude in the waveform will be produced. The high-order word of the duration long integer contains the integral part and the low-order word contains the fractional part of the duration. (Binary fractions are described in the Toolbox Utilities manual under Fixed-Point Numbers.)

The time interval specified by the duration long integer can vary between 44.93 usec and 2.95 sec, corresponding to the binary fractions 1.ØØØØ (represented by the four bytes $ØØ Ø1 ØØ ØØ, or the long integer 1) and 65535.9999 (represented by the four bytes $FF FF FF FF, or the long integer 4294967295), respectively.

(note)

As a further example, the time interval 89.86 usec corresponds to the binary fraction 2.ØØØØ, the four bytes $ØØ Ø2 ØØ ØØ, and the long integer 131Ø72. The time interval .Ø115 sec corresponds to the binary fraction 25.5ØØØ, the four bytes $ØØ 19 8Ø ØØ, and the long integer 1671168.

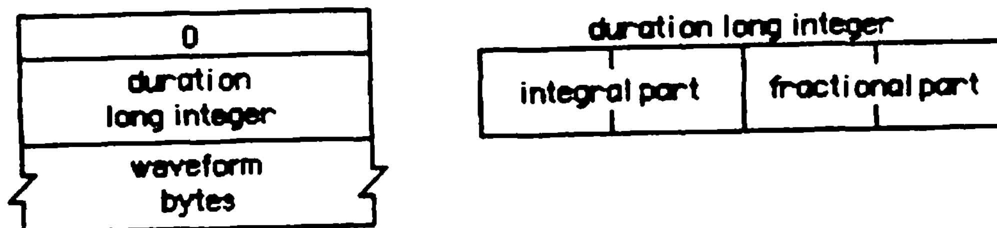


Figure 4. Free-Form Synthesizer Buffer

(note)

Note that the duration long integer specifies a time interval, but it doesn't specify the period of a wave cycle. To determine the time period of a wave cycle in the waveform, use the following relationship:

period = time interval * wavelength

where the wavelength is given in bytes. For example, the period of a wave of 1Ø-byte wavelength with a time interval of 2 usec/byte would be 9ØØ usec (corresponding to 1111 Hz).

Assembly-language note: The address of the free-form buffer currently in use is contained in the system global soundBase.

Square-Wave Synthesizer

The square-wave synthesizer is used to make sounds such as beeps. A square-wave synthesizer buffer consists of a negative integer followed by a sequence of integer triplets (Figure 5). The negative integer identifies the buffer as a square-wave buffer. Each triplet contains the count, amplitude, and duration of a different sound. The square-wave synthesizer doesn't require a waveform description because of the simple form of square waves. You can store as many triplets in a synthesizer buffer as there's room for.

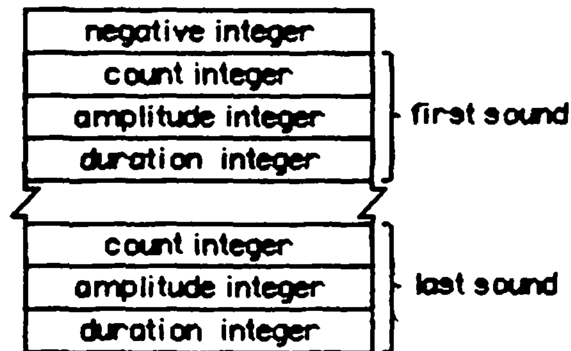


Figure 5. Square-Wave Synthesizer Buffer

Each count integer can range in value from Ø to 65535; the actual frequency the count corresponds to is given by the relationship:

frequency (Hz) = 78336Ø / count

A partial list of count values and corresponding frequencies for notes comprising Ptolemy's diatonic scale (the scale to which pianos are tuned) is given in the summary at the end of this manual.

<u>Assembly-language note</u>: The value of count currently in use is contained in the system global curPitch.

Each amplitude integer can range from Ø to 255. Each duration integer specifies the number of ticks the sound will be generated, ranging from Ø to 255 (corresponding to Ø to 4.25 seconds).

The last sound triplet must be signified by a count integer of Ø. When the square-wave synthesizer is used, the sound specified by each triplet is generated once, and then the synthesizer stops.

Four-Tone Synthesizer

The four-tone synthesizer is used to produce harmonic sounds such as music. It can simultaneously generate four different sounds, each with its own frequency, phase, and waveform.

A four-tone synthesizer buffer consists of an integer and a pointer (Figure 6). The integer can be any positive number, and serves only to identify the buffer as a four-tone buffer. The pointer points to a data structure describing the four tones, called a four-tone record.

Assembly-language note: The address of the four-tone record currently in use is stored in the system global soundPtr.

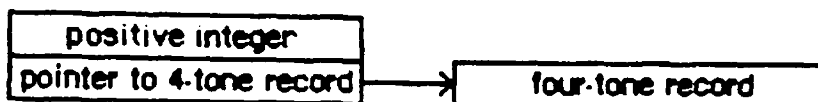


Figure 6. Four-Tone Synthesizer Buffer

A four-tone record consists of a duration integer followed by 12 long integers that contain the rate, phase, and pointers to the waveform descriptions of the four sounds (see Figure 7).

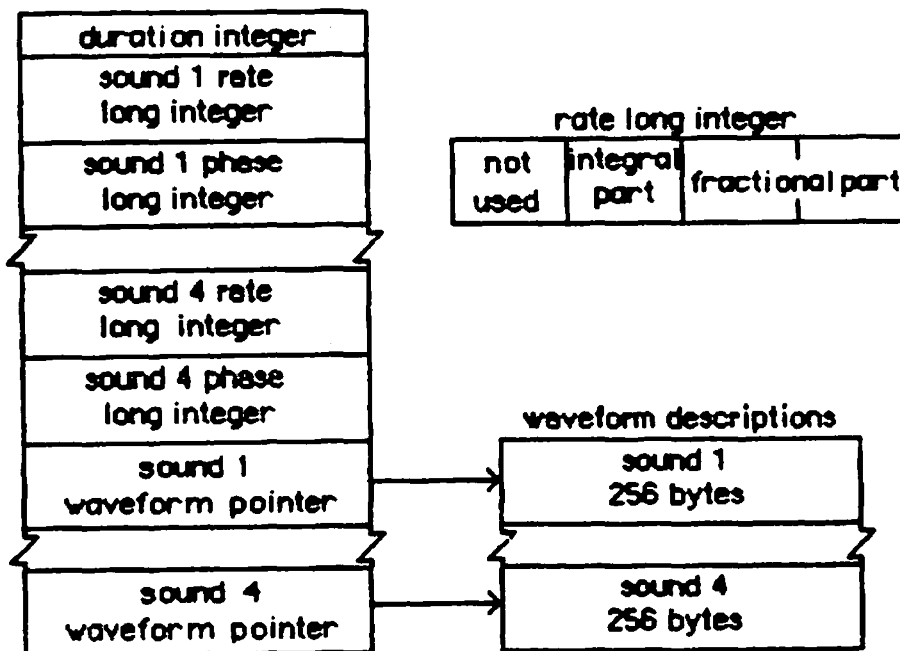


Figure 7. Four-Tone Record

The duration integer indicates the number of ticks that each sound will be generated, from Ø to 255 (Ø to 4.25 seconds). Each phase integer indicates the byte within the waveform description at which the synthesizer should begin producing sound (the first byte is byte number Ø). Each rate long integer determines the speed at which the synthesizer cycles through the waveform. The low-order word of the rate long integer contains the fractional part of the rate, and the low-order byte of the high-order word contains the integral part. (Binary fractions are described in the Toolbox Utilities manual under Fixed-Point Numbers.) The rate long integer can vary between Ø and 16777215.

The waveform description for each tone must contain 256 bytes. The four-tone synthesizer creates sound by starting at the byte in the waveform description specified by the phase, and skipping rate bytes ahead every 44.93 usec; when the time specified by the duration integer has elapsed, the synthesizer stops. The amount of time required to cycle completely through the waveform is 16777216 * 44.93 usec / rate (115Ø2 usec if the rate long integer is 65536--corresponding to about 87 Hz if the waveform contains one wavelength). If the waveform contains one wavelength, the frequency the rate corresponds to is given by

frequency (Hz) = rate / 753.795

The maximum rate of 16777215 corresponds to 44.93 usec, or about 22.3 kHz if the waveform contains one wavelength, and a rate of Ø produces no sound. A partial list of rate values and corresponding frequencies for notes comprising Ptolemy's diatonic scale (the scale to which pianos are tuned) isis given in the summary at the end of this manual.

USING THE SOUND DRIVER

The Sound Driver is a standard Macintosh device driver, and is manipulated via the Device Manager DriverOpen, DriverClose, Write, and Control calls. The Sound Driver doesn't support Read or Status calls.

The Sound Driver is automatically opened when the system starts up. Its driver name is .Sound, and its driver reference number is -4. To close the Sound Driver, call DriverClose(-4); you can reopen it by calling DriverOpen(".Sound").

To use one of the three types of synthesizers to generate sound, use the Memory Manager routines NewHandle and SetHandleSize to allocate heap space for a synthesizer buffer. Then, fill the buffer with values describing the sound, and make an Write call to the Sound Driver. The Write parameters passed must be as follows:

- RefNum must be -4.

- BuffPtr must point to the synthesizer buffer.

- Count must contain the length of the synthesizer buffer, in bytes.

When you use the free-form synthesizer, the amplitudes described by each byte in the waveform description are generated sequentially until the number of bytes specified by the count parameter have been written. When you use the square-wave synthesizer, the sounds described by each sound triplet are generated sequentially until either the end of the buffer has been reached (indicated by a count integer of Ø in the square-wave buffer), or the number of bytes specified by the Write call's count parameter have been written. When you use the four-tone synthesizer, all four sounds are generated for the length of time specified by the duration integer in the four-tone record.

There are three different calls you can make to the Sound Driver's control routine:

- KillIO is a standard control call supported by all drivers. It stops any sound currently being generated, and deletes all asynchronous I/O requests to the Sound Driver that haven't yet been processed.

- SetVolume allows you to change the volume of the sound that passes through the Macintosh speaker. There are eight levels of volume, specified by the three low-order bits in the opParam parameter of the Control call, Ø being low, and 7 high. Applications shouldn't change the speaker volume, as it's really up to the user to choose the normal sound level via the Control Panel desk accessory.

- Advanced Programmers: SetLevel enables you to control the amplitude of the sound generated by the square-wave synthesizer. The amplitude is contained in the opParam parameter of the Control call, and must be in the range Ø to 255. This call is explained in more detail below.

When you call the Sound Driver's control routine, the parameters must contain the following:

- RefNum must be -4.

- OpCode must specify the type of call:

Call	OpCode
KillIO	1
SetVolume	2
SetLevel	3

- OpParam must provide the volume level for a SetVolume call, and the amplitude for a SetLevel call.

(note)
Advanced programmers using low-level Pascal or assembly-language Device Manager routines must pass the above values in a parameter block. In addition, if you're calling the Sound Driver asynchronously, the

ioCompletion parameter must contain either the address of a completion routine or NIL.

Assembly-language note: The current speaker volume level is contained in the system global sdVolume.

Advanced Control Routine

The following paragraphs describe how the Sound Driver uses the Macintosh hardware to produce sound, and how you can intervene in the process to control the square-wave synthesizer. You can skip this section if it doesn't interest you, and you'll still be able to use the Sound Driver as described, except for the SetLevel call.

To generate sound at the amplitude level specified by a square-wave synthesizer buffer, the Sound Driver places the value of the amplitude integer into a 740-byte buffer shared by both the Sound Driver and the disk-motor speed-control circuitry. Then, every 44.93 usec when the video beam wraps from the right edge of the screen to the left, the microprocessor automatically fetches an additional two bytes from this buffer. The high-order byte is sent to the speaker, and the low-order byte to the disk-motor speed-control circuitry.

Assembly-language note: The amplitude level in the 740-byte buffer is contained in the system global soundLevel.

(note)
All the frequencies generated by the Sound Driver are multiples of this 44.93 usec period. The highest frequency the Sound Driver can physically generate corresponds to twice this period, 89.96 usec, or a frequency of 11116 Hz.

You can cause the square-wave synthesizer to start generating sound, and then change the amplitude of the sound being generated any time you wish:

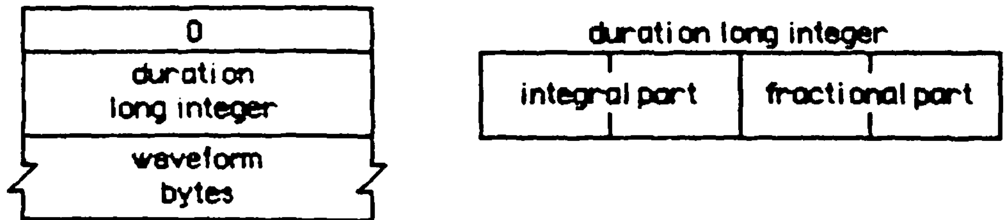
1. Make an asynchronous Write call to the Sound Driver specifying the count, amplitude, and duration of the sound you want generated. The amplitude you specify will be placed in the 740-byte buffer, and the Sound Driver will begin producing sound.

2. Whenever you want to change the sound being generated, make a SetLevel call with the opParam parameter specifying the amplitude of the new sound. The amplitude you specify will be placed in the 740-byte buffer, and the sound will change. You can continue to change the sound until the time specified by the duration integer has elapsed.

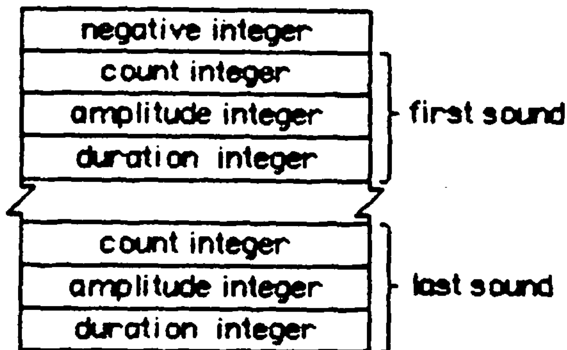
SUMMARY OF THE SOUND DRIVER

Data Structures

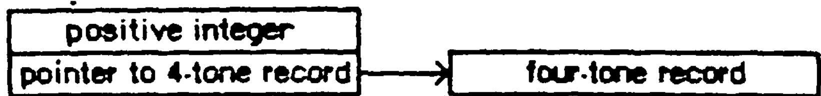
Free-Form Synthesizer Buffer



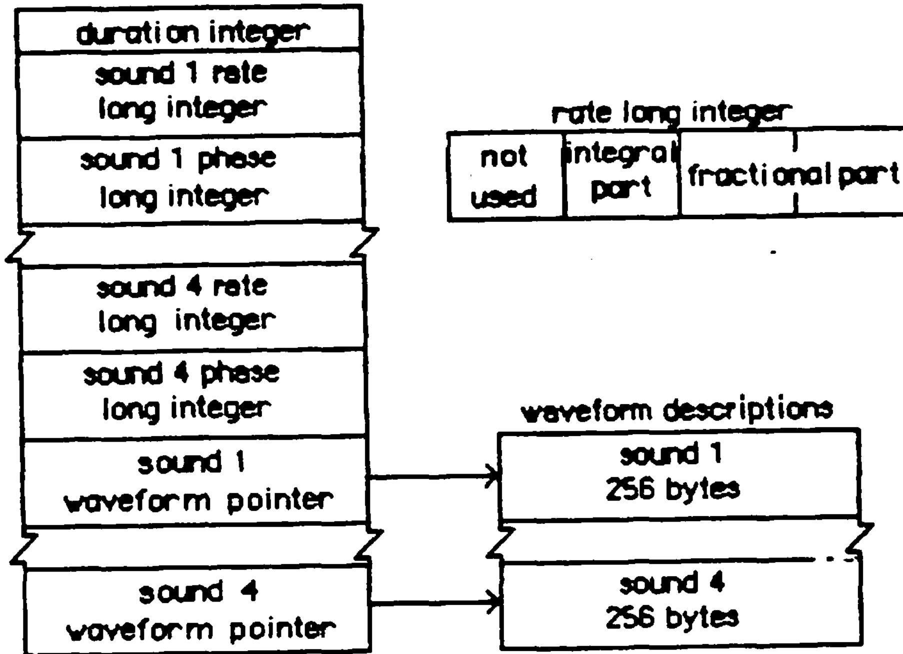
Square-Wave Synthesizer Buffer



Four-Tone Synthesizer Buffer



Four-Tone Record



Sound Driver Control Calls

Call	OpCode
KillIO	1
SetVolume	2
SetLevel	3

Assembly-Language Information

Variables

```
SdVolume        ;speaker volume level
SoundPtr        ;pointer to four-tone record
SoundBase       ;pointer to free-form buffer
SoundLevel      ;amplitude in 740-byte buffer
CurPitch        ;value of count in square-wave synthesizer buffer
```

Sound Driver Values For Notes Comprising Ptolemy's Diatonic Scale

Note (Freqency)	Rate Values for the Four-Tone Synthesizer		Count Values For the Square-Wave Synthesizer	
	Long Word	Long Integer	Word	Integer
3 octaves below middle C				
C (33)	0000 612B	24875	5CBA	23738
Db (35.2)	0000 67A5	26533	56EF	22255
D (37.125)	0000 6D50	27984	526D	21101
Eb (39.6)	0000 749A	29850	4D46	19782
E (41.25)	0000 7976	31094	4A2F	18991
F (44)	0000 818E	33166	458C	17804
Gb (46.9375)	0000 8A35	35381	4131	16689
G (49.5)	0000 91C0	37312	3DD1	15825
Ab (52.8)	0000 9B78	39800	39F4	14836
A (55)	0000 A1F2	41458	37A3	14243
Bb (57.75)	0000 AA0B	43531	34FD	13565
B (61.875)	0000 B631	46641	3174	12660
2 octaves below middle C				
C (66)	0000 C256	49750	2E5D	11869
Db (70.4)	0000 CF4B	53067	2B77	11127
D (74.25)	0000 DAA1	55969	2936	10550
Eb (79.2)	0000 E934	59700	26A3	9891
E (82.5)	0000 F2EC	62188	2517	9495
F (88)	0001 031D	66333	22C6	8902
Gb (93.875)	0001 146A	70762	2099	8345
G (99)	0001 2381	74625	1EE9	7913
Ab (105.6)	0001 36F0	79600	1CFA	7418
A (110)	0001 43E5	82917	1BD1	7121
Bb (115.5)	0001 5417	87063	1A7E	6782
B (123.75)	0001 6C62	93282	18BA	6330
1 octave below middle C				
C (132)	0001 84AC	99500	172F	5935
Db (140.8)	0001 9E96	106134	15BC	5564
D (148.5)	0001 B542	111938	149B	5275
Eb (158.4)	0001 D269	119401	1351	4945
E (165)	0001 E5D8	124376	128C	4748
F (176)	0002 063B	132667	1163	4451
Gb (187.75)	0002 28D5	141525	104C	4172
G (198)	0002 4703	149251	0F74	3956
Ab (211.2)	0002 6DE1	159201	0E7D	3709
A (220)	0002 87CA	165834	0DE9	3561
Bb (231)	0002 A82E	174126	0D3F	3391
B (247.5)	0002 D8C4	186564	0C5D	3165

Middle C

C	(264)	0003 0959	199001	0B97	2967
Db	(281.6)	0003 3D2C	212268	0ADE	2782
D	(297)	0003 6A85	223877	0A4E	2638
Eb	(316.8)	0003 A4D2	238802	09A9	2473
E	(330)	0003 CBB0	248752	0946	2374
F	(352)	0004 0C77	265335	08B1	2225
Gb	(375.5)	0004 51AA	283050	0826	2086
G	(396)	0004 8E06	298502	07BA	1978
Ab	(422.4)	0004 DBC3	318403	073F	1855
A	(440)	0005 0F95	331669	06F4	1780
Bb	(462)	0005 505D	348253	06A0	1696
B	(495)	0005 B188	373128	062F	1583

1 octave above middle C

C	(528)	0006 12B3	398003	05CC	1484
Db	(563.2)	0006 7A59	424537	056F	1391
D	(594)	0006 D50A	447754	0527	1319
Eb	(633.6)	0007 49A4	477604	04D4	1236
E	(660)	0007 9760	497504	04A3	1187
F	(704)	0008 18EF	530671	045[illegible]	1113
Gb	(751)	0008 A354	566100	0413	1043
G	(792)	0009 1C0D	597005	03DD	989
Ab	(844.8)	0009 B786	636806	039F	927
A	(880)	000A 1F2B	663339	037A	890
Bb	(924)	000A A0BA	696506	0350	848
B	(990)	000B 6311	746257	0317	791

2 octaves above middle C

C	(1056)	000C 2567	796007	02E6	742
Db	(1126.4)	000C F4B2	849074	02B7	695
D	(1188)	000D AA14	895508	0293	659
Eb	(1267.2)	000E 9349	955209	026A	618
E	(1320)	000F 2EC1	995009	0251	593
F	(1408)	0010 31DF	1061340	022C	556
Gb	(1502)	0011 46A8	1132200	020A	522
G	(1584)	0012 381B	1194010	01EF	495
Ab	(1689.6)	0013 6F0C	1273610	01D0	464
A	(1760)	0014 3E57	1326680	01BD	445
Bb	(1848)	0015 4175	1393010	01A8	424
B	(1980)	0016 C622	1492510	018C	396

<u>3 octaves above middle C</u>

C	(2112)	ØØ18 4ACF	1592Ø2Ø	Ø173	371
Db	(2252.8)	ØØ19 E965	169815Ø	Ø15C	348
D	(2376)	ØØ1B 5429	179102Ø	Ø14A	33Ø
Eb	(2534.4)	ØØ1D 2692	191Ø42Ø	Ø135	3Ø9
E	(264Ø)	ØØ1E 5D83	199ØØ2Ø	Ø129	297
F	(2816)	ØØ2Ø 63BF	212269Ø	Ø116	278
Gb	(3ØØ4)	ØØ22 8D5Ø	22644ØØ	Ø1Ø5	261
G	(3168)	ØØ24 7Ø36	2388Ø2Ø	ØØF7	247
Ab	(3379.2)	ØØ26 DE18	254722Ø	ØØE8	232
A	(352Ø)	ØØ28 7CAE	265336Ø	ØØDF	223
Bb	(3696)	ØØ2A 82EA	2786Ø3Ø	ØØD4	212
B	(396Ø)	ØØ2D 8C44	2985Ø3Ø	ØØC6	198

GLOSSARY

amplitude: The vertical distance between any given point on a wave and the horizontal line about which the amplitude oscillates.

four-tone record: A data structure describing the four tones produced by a four-tone synthesizer.

four-tone synthesizer: The part of the Sound Driver used to make simple harmonic tones, with up to four "voices" producing sound simultaneously.

free-form synthesizer: The part of the Sound Driver used to make complex music and speech.

frequency: The number of cycles per second (also called Hertz) at which a wave oscillates.

period: The time elapsed during one complete cycle of a wave.

phase: Some fraction of a wave cycle (measured from a fixed point on the wave).

square-wave synthesizer: The part of the Sound Driver used to produce less harmonic sounds such as beeps.

synthesizer buffer: A description of the sound to be generated by a synthesizer.

waveform: The physical shape of a wave.

waveform description: A sequence of bytes describing a waveform.

wavelength: The horizontal extent of one complete cycle of a wave.

MACINTOSH USER EDUCATION

The Structure of a Macintosh Application

/STRUCTURE/STRUCT

See Also: Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

Modification History: First Draft (ROM 7) Caroline Rose 2/8/84

ABSTRACT

This manual describes the overall structure of a Macintosh application program, including its interface with the Finder.

TABLE OF CONTENTS

3 About This Manual
3 Signatures and File Types
4 Finder-Related Resources
5 Version Data
5 Icons and File References
6 Bundles
7 An Example
8 Formats of Finder-Related Resources
8 Opening and Printing Documents from the Finder
11 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved. Distribution in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the overall structure of a Macintosh application program, including its interface with the Finder. *** Right now it describes only the Finder interface; the rest will be filled in later. Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

(hand)
This information in this manual applies to version 7 of the Macintosh ROM and version 1.Ø of the Finder.

You should already be familiar with the following:

- The details of the User Interface Toolbox, the Macintosh Operating System, and the other routines that your application program may call. For a list of all the technical documentation that provides these details, see Inside Macintosh: A Road Map.

- The Finder, which is described in the Macintosh owner's guide.

This manual doesn't cover the steps necessary to create an application's resources or to compile, link, and execute the application program. These are discussed in the manual Putting Together a Macintosh Application.

The manual begins with sections that describe the Finder interface: signatures and file types, used for identification purposes; application resources that provide icon and file information to the Finder; and the mechanism that allows documents to be opened or printed from the Finder.

*** more to come ***

Finally, there's a glossary of terms used in this manual.

SIGNATURES AND FILE TYPES

Every application must have a unique signature by which the Finder can identify it. The signature can be any four-character sequence not being used for another application on any currently mounted volume (except that it can't be one of the standard resource types). To ensure uniqueness on all volumes, your application's signature must be assigned by Macintosh Technical Support.

Signatures work together with file types to enable the user to open or print a document (any file created by an application) from the Finder. When the application creates a file, it sets the file's creator and file type. Normally it sets the creator to its signature and the file type to a four-character sequence that identifies files of that type. When the user asks the Finder to open or print the file, the Finder

starts up the application whose signature is the file's creator and passes the file type to the application along with other identifying information, such as the file name. (More information about this process is given below under "Opening and Printing Documents from the Finder".)

An application may create its own special type or types of files. Like signatures, file types must be assigned by Macintosh Technical Support to ensure uniqueness. When the user chooses Open from an application's File menu, the application will display (via the Standard File Package) the names of all files of a given type or types, regardless of which application created the files. Having a unique file type for your application's special files ensures that only the names of those files will be displayed for opening.

(hand)

> Signatures and file types may be strange, unreadable combinations of characters; they're never seen by end users of Macintosh.

Applications may also create existing types of files. There might, for example, be one that merges two MacWrite documents into a single document. In such cases, the application should use the same file type as the original application uses for those files. It should also specify the original application's signature as the file's creator; that way, when the user asks the Finder to open or print the file, the Finder will call on the original application to perform the operation. To learn the signatures and file types used by existing applications, check with Macintosh Technical Support.

Files that consist only of text—a stream of characters, with Return characters at the ends of paragraphs or short lines--should be given the file type 'TEXT'. This is the type that MacWrite gives to text-only files it creates, for example. If your application uses this file type, its files will be accepted by MacWrite and it in turn will accept MacWrite text-only files (likewise for any other application that deals with 'TEXT' files). Your application can give its own signature as the file's creator if it wants to be called to open or print the file when the user requests this from the Finder.

For files that aren't to be opened or printed from the Finder, as may be the case for certain data files created by the application, the signature should be set to '????' (and the file type to whatever is appropriate).

FINDER-RELATED RESOURCES

To establish the proper interface with the Finder, every application's resource file must specify the signature of the application along with data that provides version information. In addition, there may be resources that provide information about icons and files related to the application. All of these Finder-related resources are described

below, followed by a comprehensive example and (for interested programmers) the exact formats of the resources.

Version Data

Your application's resource file must contain a special resource that has the signature of the application as its resource type. This resource is called the version data of the application. The version data is typically a string that gives the name, version number, and date of the application, but it can in fact be any data at all. The resource ID of the version data is Ø by convention.

As described in detail in Putting Together a Macintosh Application, part of the process of installing an application on the Macintosh is to set the creator of the file that contains the application. You set the creator to the application's signature, and the Finder copies the corresponding version data into a resource file named Desktop. (The Finder doesn't display this file on the Macintosh desktop, to ensure that the user won't tamper with it.)

(hand)
Additional, related resources may be copied into the Desktop file; see "Bundles" below for more information. The Desktop file also contains folder resources, one for each folder on the volume.

Icons and File References

For each application, the Finder needs to know:

- The icon to be displayed for the application on the desktop, if different from the Finder's default icon for applications (see Figure 1).

- If the application creates any files, the icon to be displayed for each type of file it creates, if different from the Finder's default icon for documents.

- What files, if any, must accompany the application when it's transferred to another volume.

Figure 1. The Finder's Default Icons

The Finder learns this information from resources called file references in the application's resource file. Each file reference contains a file type and an ID number, called a local ID, that

identifies the icon to be displayed for that type of file. (The local ID is mapped to an actual resource ID as described under "Bundles" below.) Any file reference may also include the name of a file that must accompany the application when it's transferred to another volume.

The file type for the application itself is 'APPL'. This is the file type in the file reference that designates the application's icon. You also specify it as the application's file type at the same time that you specify its creator--the first time you install the application on the Macintosh.

The ID number in a file reference corresponds not to a single icon but to an icon list in the application's resource file. The icon list consists of two icons: the actual icon to be displayed on the desktop, and a mask consisting of that icon's outline filled with black (see Figure 2). *** For existing types of files, there's currently no way to direct the Finder to use the original application's icon for that file type. ***

Figure 2. Icon and Mask

Bundles

A bundle in the application's resource file groups together all the Finder-related resources. It specifies the following:

- The application's signature and the resource ID of its version data
- A mapping between the local IDs for icon lists (as specified in file references) and the actual resource IDs of the icon lists in the resource file
- Local IDs for the file references themselves and a mapping to their actual resource IDs

The first time you install the application on the Macintosh, you set its "bundle bit", and the Finder copies the version data, bundle, icon lists, and file references from the application's resource file into the Desktop file. *** (The setting of the bundle bit will be covered in the next version of Putting Together a Macintosh Application.) *** If there are any resource ID conflicts between the icon lists and file references in the application's resource file and those in Desktop, the Finder will change those resource IDs in Desktop. The Finder does this same resource copying and ID conflict resolution when you transfer an application to another volume.

(hand)
The local IDs are needed only for use by the Finder.

An Example

Suppose you've written an application named SampWriter. The user can create a unique type of document from it, and you want a distinctive icon for both the application and its documents. The application's signature, as assigned by Macintosh Technical Support, is 'SAMP'; the file type assigned for its documents is 'SAMF'. Furthermore, a file named 'TgFil' should accompany the application when it's transferred to another volume. You would include the following resources in the application's resource file:

Resource	Resource ID	Contents
Version data with resource type 'SAMP'	Ø	The string 'SampWriter Version 1 — 2/1/84'
Icon list	128	The icon for the application The icon's mask
Icon list	129	The icon for documents The icon's mask
File reference	128	File type 'APPL' Local ID Ø for the icon list
File reference	129	File type 'SAMF' Local ID 1 for the icon list File name 'TgFil'
Bundle	128	Signature 'SAMP' Resource ID Ø for the version data For icon lists, the mapping: local ID Ø --> resource ID 128 local ID 1 --> resource ID 129 For file references, the mapping: local ID Ø --> resource ID 128 local ID 1 --> resource ID 129

(hand)
See the manual Putting Together a Macintosh Application for information about how to include these resources in a resource file.

The file references in this example happen to have the same local IDs and resource IDs as the icon lists, but any of these numbers can be different. Different resource IDs can be given to the file references, and the local IDs specified in the mapping for file references can be whatever desired.

Formats of Finder-Related Resources

The resource type for an application's version data is the signature of the application, and the resource ID is Ø by convention. The resource data can be anything at all; typically it's a string giving the name, version number, and date of the application.

The resource type for an icon list is 'ICN#'. The resource data simply consists of the icons, 128 bytes each.

The resource type for a file reference is 'FREF'. The resource data has the format shown below.

Number of bytes	Contents
4 bytes	File type
2 bytes	Local ID for icon list
1 byte	Length of following file name in bytes; Ø if none
n bytes	Optional file name

The resource type for a bundle is 'BNDL'. The resource data has the format shown below. The format is more general than needed for Finder-related purposes because bundles will be used in other ways in the future.

Number of bytes	Contents
4 bytes	Signature of the application
2 bytes	Resource ID of version data
2 bytes	Number of resource types in bundle minus 1
For each resource type:	
4 bytes	Resource type
2 bytes	Number of resources of this type minus 1
For each resource:	
2 bytes	Local ID
2 bytes	Actual resource ID

A bundle used for establishing the Finder interface contains the two resource types 'ICN#' and 'FREF'.

OPENING AND PRINTING DOCUMENTS FROM THE FINDER

When the user selects a document and tries to open or print it from the Finder, the Finder starts up the application whose signature is the document file's creator. An application may be selected along with one or more documents for opening (but not printing); in this case, the Finder starts up that application. If the user selects more than one document for opening without selecting an application, the files must have the same creator. If more than one document is selected for printing, the Finder starts up the application whose signature is the first file's creator (that is, the first one selected if they were selected by Shift-clicking, or the top left one if they were selected

by dragging a rectangle around them).

Any time the Finder starts up an application, it passes along information via the "Finder information handle" in the application parameter area (as described in the Segment Loader manual). Pascal programmers can call the Segment Loader procedure GetAppParms to get the Finder information handle. For example, if applParam is declared as type Handle, the call

```
GetAppParms(applName, applRefNum, applParam)
```

returns the Finder information handle in applParam. The Finder information has the following format:

Number of bytes	Contents
2 bytes	Ø if open, 1 if print
2 bytes	Number of files to open or print (Ø if none)
For each file:	
2 bytes	Volume reference number of volume containing the file
4 bytes	File type
1 byte	File's version number (typically Ø)
1 byte	Ignored
1 byte	Length of following file name in bytes
n bytes	Characters of file name (if n is even, add an extra byte)

The files are listed in order of the appearance of their icons on the desktop, from left to right and top to bottom. The file names don't include a volume prefix. An extra byte is added to any name of even length so that the entry for the next name will begin on a word boundary.

Every application that opens or prints documents should look at this information to determine what to do when the Finder starts it up. If the number of files is Ø, the application should start up with an untitled document on the desktop. If a file or files are specified for opening, it should start up with those documents on the desktop. If only one document can be open at a time but more than one file is specified, the application should open the first one and ignore the rest. If the application doesn't recognize a file's type (which can happen if the user selected the application along with another application's document), it may want to open the file anyway and check its internal structure to see if it's a compatible type. The response to an unacceptable type of file should be an alert box that shows the file name and says that the document can't be opened.

If a file or files are specified for printing, the application should print them in turn, preferably without doing its entire start-up sequence. For example, it may not be necessary to show the menu bar or a document window, and reading the desk scrap into memory is definitely not required. After successfully printing a document, the application should set the file type in the Finder information to Ø. Upon return from the application, the Finder will start up other applications as

necessary to print any remaining files whose type was not set to Ø.
*** The Finder doesn't currently do this, but it may in the future.
***

GLOSSARY

bundle: A resource that maps local IDs of resources to their actual resource IDs; used to provide mappings for file references and icon lists needed by the Finder.

Desktop file: A resource file in which the Finder stores folder resources and the version data, bundle, icons, and file references for each application on the volume.

file reference: A resource that provides the Finder with file and icon information about an application.

file type: A four-character sequence, specified when a file is created, the identifies the type of file.

icon list: A resource consisting of a list of icons.

local ID: A number that refers to an icon list or file reference in an application's resource file and is mapped to an actual resource ID by a bundle.

signature: A four-character sequence that uniquely identifies an application to the Finder.

version data: In an application's resource file, a resource that has the application's signature as its resource type; typically a string that gives the name, version number, and date of the application.

MACINTOSH USER EDUCATION

TextEdit: A Programmer's Guide /TEXT.EDIT/EDIT

See Also: The Macintosh User Interface Guidelines
Macintosh Operating System Manual
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
CoreEdit: A Programmer's Guide

Modification History: First Draft (ROM 7) B. Hacker 9/28/83

ABSTRACT

The TextEdit package of the Macintosh User Interface Toolbox is a set of data types and routines for handling basic text formatting and editing capabilities in a Macintosh application. This manual describes TextEdit in detail.

TABLE OF CONTENTS

3 About This Manual
4 About TextEdit
4 The Editing Environment: Edit Record
5 The Destination and View Rectangles
6 The Selection Range
8 Justification
9 The TERec Data Type
11 Using TextEdit
13 TextEdit Routines
13 Initialization
14 Manipulating Edit Records
14 Editing
17 Selection Range and Justification
17 Mice and Carets
18 Text Display
19 Advanced Routines
20 Notes for Assembly-Language Programmers
21 Summary of TextEdit
23 Glossary

Copyright (c) 1983 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

The TextEdit package of the Macintosh User Interface Toolbox is a set of data types and routines for handling basic text formatting and editing capabilities in a Macintosh application. This manual describes TextEdit in detail.

The Toolbox also includes a more sophisticated text editing package, called CoreEdit. You'll need to use CoreEdit instead of TextEdit if you want fully justified text, recognition of word boundaries during editing ("intelligent cut and paste"), or tabbing. Bear in mind, however, that CoreEdit is not in the Macintosh ROM, and occupies over 6K of your application's available memory instead.

(hand)

> This manual describes the TextEdit that works with version 7 of the ROM. If you're using a different version, the information presented here may not apply.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly points, rectangles, grafPorts, fonts, and character style.
- The ToolBox Event Manager. Some TextEdit routines are called only in response to particular events.
- The Window Manager, particularly update and activate events.

The manual begins with an introduction to TextEdit and what you can do with it. It then discusses the edit record, the primary data structure used by the text editing routines. Learning about this data structure will give you the background you need to understand the routines themselves.

Next, a section on using TextEdit introduces you to its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all text editing procedures and functions--their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions is a section containing notes for programmers who will use TextEdit from assembly language.

Finally, there's a summary of the TextEdit data structures and routine calls, for quick reference, and a glossary of terms used in this manual.

ABOUT TEXTEDIT

TextEdit is a group of compact and efficient routines that provide the basic text editing and formatting capabilities needed in an application. These routines perform operations such as:

- Inserting new text
- Deleting characters that are backspaced over
- Translating mouse activity into text selection
- Moving text within a window
- Deleting selected text and possibly inserting it elsewhere, or copying text without deleting it

Because these routines follow the Macintosh User Interface Guidelines, using them ensures that your application presents a consistent, easy-to-learn interface for end users. In particular, TextEdit supports these standard features:

- Selecting text by clicking and dragging with the mouse, double-clicking to select words instead of characters.
- Inverse highlighting of the current text selection, or display of a blinking vertical bar at the insertion point.
- Word wrap, which prevents words from being split between lines when text is drawn. To TextEdit, a word is any series of printing characters, excluding spaces (ASCII code $20) but including nonbreaking spaces (ASCII code $CA).
- Cutting (or copying) and pasting within an application via the Clipboard *** not currently described as "Clipboard" in the User Interface Guidelines, but will be ***. TextEdit puts text you cut or copy into a string of characters called the scrap.

(hand)

> Cutting and pasting between applications, or between applications and desk accessories, is done with the aid of the Scrap Manager (see the Scrap Manager manual for details).

THE EDITING ENVIRONMENT: EDIT RECORD

To edit text on the screen, the text editing routines need to know where and how to display the text, where to store the text, and other information related to editing. This display, storage, and editing information is contained in an edit record that defines the complete editing environment. The data type of an edit record is called TERec.

You prepare to edit text by passing, to a procedure, a destination rectangle in which to draw the text and a view rectangle in which the text will be visible. The procedure incorporates the rectangles and the drawing environment of the current grafPort into an edit record, and returns a handle to the record:

```
TYPE TEPtr    = ^TERec;
     TEHandle = ^TEPtr;
```

Most of the text editing routines require you to pass this handle as a parameter.

In addition to the two rectangles and a description of the drawing environment, the edit record also contains:

- A handle to the text to be edited
- A pointer to the grafPort
- The current selection range, which determines exactly which characters will be affected by the next editing operation
- The justification of the text, as left, right, or center

The special terms introduced here are described in detail below.

Most programmers won't access any of the fields of an edit record directly, and so don't have to know its exact structure; the necessary access is done with TextEdit routines. Advanced programmers, however, may need to know some of the field names. The structure of an edit record is given below.

The Destination and View Rectangles

The destination rectangle is the rectangle in which the text is drawn. The view rectangle is the rectangle within which the text is actually visible. In other words, the view of the text drawn in the destination rectangle is clipped to the view rectangle (see Figure 1).

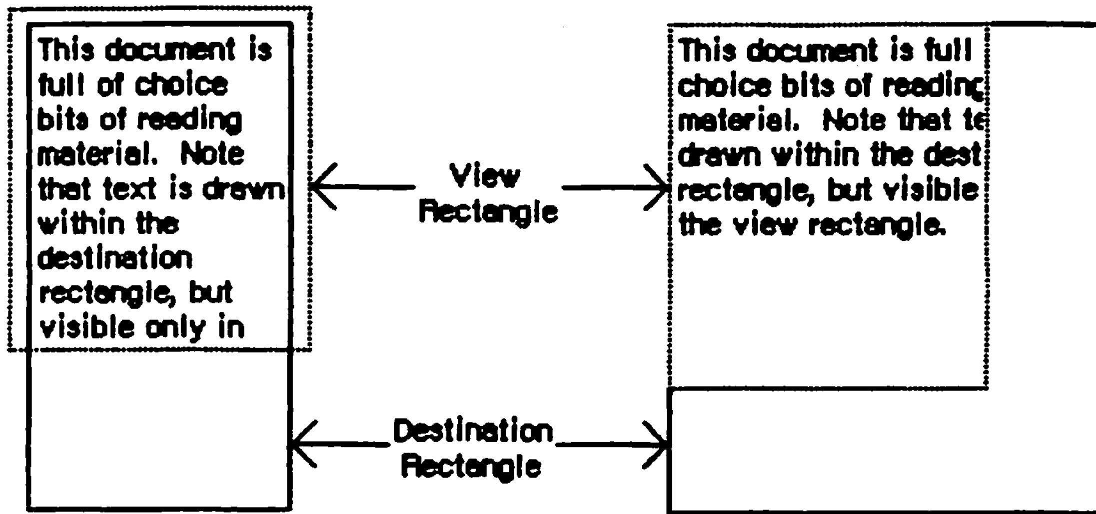


Figure 1. Destination and View Rectangles

You specify both rectangles in the coordinate system of the grafPort. In a document window, the destination rectangle should be inset about four pixels from the left and right edges of the grafPort's portRect (2Ø pixels from the right edge if there's a scroll bar or size box) to ensure that the first and last character in each line is legible.

Edit operations may of course lengthen or shorten the text. If the text becomes too long to be enclosed by the destination rectangle, it's simply drawn beyond the bottom. In other words, you can think of the destination rectangle as bottomless--its sides determine the beginning and end of each line of text, and its top determines the position of the first line.

Normally, at the right edge of the destination rectangle, the text automatically wraps around to the left edge to begin a new line. A new line also begins where explicitly specified by a Return character in the text. Word wrap ensures that words are never split between lines unless they're too long to fit entirely on one line.

The Selection Range

In the text editing environment, a character position is an index into the text, with position Ø corresponding to the first character. The edit record includes fields for character positions that specify the beginning and end of the current selection range, which is the series of characters at which the next editing operation will occur. For example, the procedures that cut or copy from the text of an edit record do so to the current selection range.

The selection range, which is always inversely highlighted, extends from the beginning character position up to but NOT including the end position. Figure 2 shows a selection range defined by the beginning

position 2 and the end position 7; it consists of five characters, those at positions 2 through 6. The end position may be 1 greater than the position of the last character of the text, so that the selection range can include the last character.

Th■■■ection range is inversely highlighted.

Selection range
beginning at position 2
and ending at position 7

The| insertion point is marked with a blinking caret.

Insertion point
at position 3

Figure 2. Selection Range and Insertion Point

If the beginning and end of the selection range are the same, that character position is the text's insertion point, the position where characters will be inserted. By convention, it's usually marked with a caret that blinks (is repeatedly inverted). If, for example, the insertion point is as illustrated in Figure 2 and the inserted characters are " edit", the text will read "The edit insertion point...".

(hand)
We use the word caret here generically, to mean a symbol indicating where something is to be inserted; the specific symbol is a vertical bar. TextEdit does not automatically change the caret to a vertical bar for you. (You must use the QuickDraw procedure SetCursor.)

If you call a procedure to insert characters when there's no insertion point (that is, when there's a selection range of one or more characters), the editing procedure automatically deletes the selection range and replaces it with an insertion point, before inserting the characters.

Justification

TextEdit allows you to specify the justification of the lines of text, that is, their horizontal placement with respect to the left and right edges of the destination rectangle. The different types of justification are illustrated in Figure 3.

- Left justification aligns the text with the left edge of the destination rectangle. This is the default type of justification.
- Center justification centers the text between the left and right edges of the destination rectangle.
- Right justification aligns the text with the right edge of the destination rectangle.

This is an example of left justification. See how the text is aligned with the left edge of the rectangle.

This is an example of right justification. See how the text is aligned with the right edge of the rectangle.

This is an example of center justification. See how the text is centered between the edges of the rectangle.

Figure 3. Justification

(hand)
Trailing and leading spaces on a line are ignored for justification. For example, "Fred" and " Fred " will be aligned identically.

TextEdit has three predefined constants for setting the justification:

```
CONST teJustLeft   = 0;
      teJustCenter = 1;
      teJustRight  = -1;
```

The TERec Data Type

For those who want to know more about the structure of an edit record, some (but not all) of the structure is given here. You can skip this section if you want and still use TextEdit as described above, but some TextEdit features are available only if you change fields in the edit record directly.

(eye)
The fields that are not described exist solely for internal use among the text editing routines; their contents cannot be predicted and must not be changed.

```
  TYPE TERec = RECORD
                 destRect:   Rect;     {destination rectangle}
                 viewRect:   Rect;     {view rectangle}
                 lineHeight: INTEGER;  {line height}
                 firstBL:    INTEGER;  {location of first base line}
                 selStart:   INTEGER;  {start of selection range}
                 selEnd:     INTEGER;  {end of selection range}
                 just:       INTEGER;  {justification}
                 length:     INTEGER;  {length of text}
                 hText:      Handle;   {text to be edited}
                 txFont:     INTEGER;  {text font}
                 txFace:     INTEGER;  {character style}
                 txMode:     INTEGER;  {pen mode}
                 txSize:     INTEGER;  {type size}
                 inPort:     GrafPtr;  {grafPort}
                 crOnly:     INTEGER;  {new line at Return only, if <0}
                 nLines:     INTEGER;  {number of lines}
                 lineStarts: ARRAY [0..32000] OF INTEGER
                                       {positions of line starts}
                 {some fields within the record are for internal use
                  only, and aren't shown here; see the Pascal interface
                  to TextEdit}
               END;
```

Any of the fields in the edit record can be changed, at your discretion. Clearly, some fields (such as length, hText, inPort, and crOnly) might be changed frequently. The lineStarts array should be left unchanged.

The lineHeight field specifies the line height of the text, the number of pixels from the base line of one line to the base line of the next line, as shown in Figure 4. For single-spaced lines, line height is the same as the type size, for double-spaced lines, line height is twice the type size, and so on.

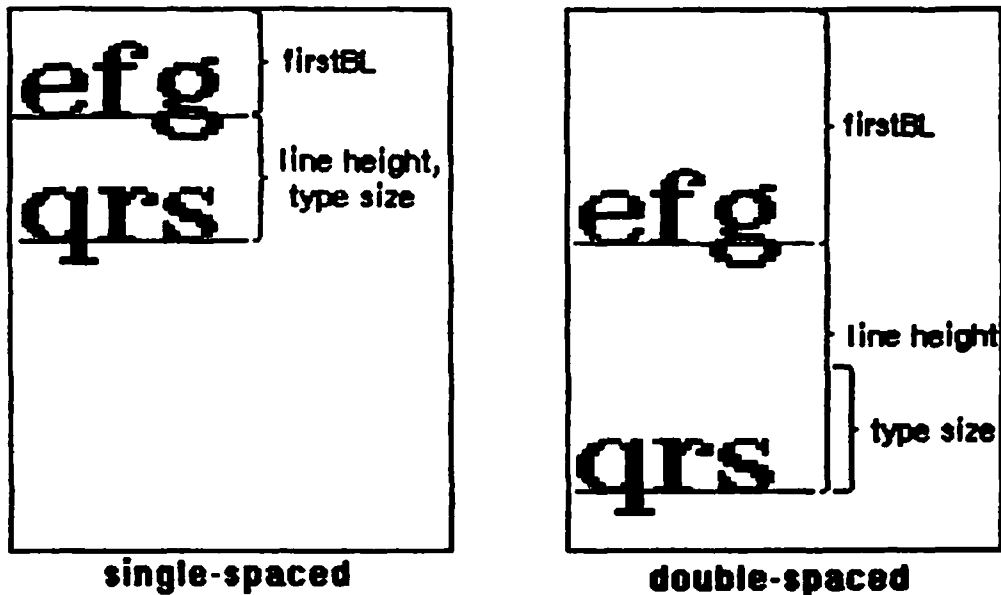


Figure 4. Line Height and FirstBL

The firstBL field specifies the number of pixels from the top of the destination rectangle to the base line of the first line of text. Initially the firstBL field is set for single-spaced lines, but you can change it for any other spacing you want. For example, to change from single to double spacing, use

```
firstBL := firstBL + typeSize
lineHeight := 2 * typeSize
```

where typeSize is the type size of the text.

The hText field is a handle to the text to be edited, and the length field contains the number of characters in the text. You can directly change the text of an edit record by changing these two fields.

The crOnly field specifies whether or not text wraps around at the right edge of the destination rectangle, as shown in Figure 5. If crOnly is zero or positive, text does wrap around. If crOnly is negative, text does not wrap around at the edge of the destination rectangle, and new lines are specified explicitly by Return characters only. This is somewhat faster than wrap around, and is useful in applications such as a programming-language editor, where you don't want a single line of code to be split onto two or more lines.

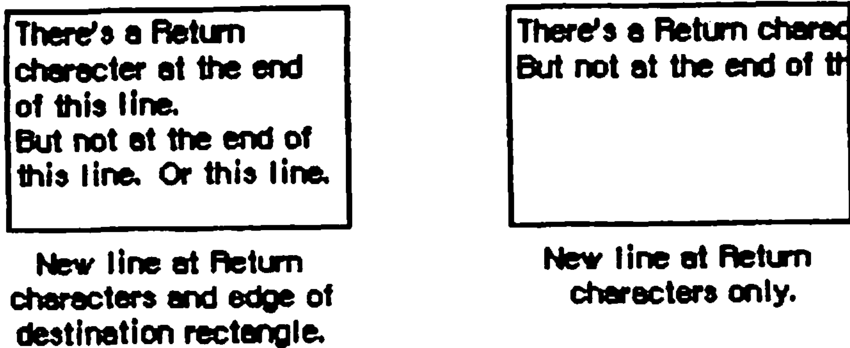


Figure 5. New Lines

The nLines field contains the number of lines in the text. The lineStarts array contains the character position of the first character in each line. It's declared to have 32001 elements to comply with Pascal range checking; it's actually a dynamic data structure having only as many elements as needed.

(hand)
The values of the lineStarts array, selEnd, and selStart are stored internally as unsigned integers. Be aware that negative values passed from Pascal will be interpreted as greater than 32767.

USING TEXTEDIT

This section discusses how the text editing routines fit into the general flow of an application program and gives you an idea of what routines you'll need to use. The routines themselves are described in detail in the next section.

Before using TextEdit, you should initialize QuickDraw, the Font Manager, and the Window Manager, in that order.

The first TextEdit routine to call is the initialization procedure TEInit. Call TENew to allocate an edit record; it returns a handle to the record. Most of the text editing routines require you to pass this handle as a parameter.

To make a blinking caret appear at the insertion point, call the TEIdle procedure as often as possible; if it's not called often enough, the caret will blink irregularly.

Your application's "main loop" should call the Toolbox Event Manager function GetNextEvent to learn whether any events have occurred. Events that pertain to TextEdit need to be handled by TextEdit routines. Whenever a mouse down event occurs within the view rectangle, call the TEClick procedure. TEClick automatically controls

the placement of the selection range and insertion point (including supporting use of the Shift key to make extended selections).

There are several procedures available for editing text. Usually they are called in response to mouse down events and menu selections. The editing procedures are:

- TEKey inserts characters at the insertion point, and deletes characters backspaced over.

- TECut transfers the selection range to the scrap, removing it from the text, and TEPaste inserts the scrap at the insertion point. By calling TECut, changing the insertion point, and then calling TEPaste, you can perform a "cut and paste" operation, moving text from one place to another.

- TECopy copies the selection range to the scrap. By calling TECopy, changing the insertion point, and then calling TEPaste, you can make multiple copies of text.

- TEDelete removes the selection range (without transferring it to the scrap).

- TEInsert inserts text at the insertion point. You can use this to combine two or more documents. TEDelete and TEInsert do not modify the scrap, and consequently are useful for implementing the Undo command (as described in the Macintosh User Interface Guidelines).

After each editing procedure, the text is redrawn from the insertion point to the end of the destination rectangle. You never have to pass the selection range or insertion point to the editing procedures; the procedures simply access that information from the edit record. The editing procedures and TEClick leave the selection range or insertion point where it should be, according to the Macintosh User Interface Guidelines, so you don't have to set it yourself. But, in case you want to, you can modify the selection range directly by using the TESetSelect procedure.

Every time GetNextEvent reports an update event for the text editing window, call TEUpdate (along with the Window Manager procedures BeginUpdate and EndUpdate), to redraw the text.

(hand)

> Advanced programmers: you must call TEUpdate after you change any fields of the edit record if the fields affect the appearance of the text. This ensures that the screen accurately reflects the changed editing environment.

The procedures TEActivate and TEDeactivate must be called each time the Event Manager reports an activate event for the text editing window. TEActivate simply highlights the selection range or displays a caret at the insertion point, and TEDeactivate unhighlights the selection range or removes the caret.

To specify the justification of the text, you can use TESetJust (which requires calling TEUpdate).

If at any time you want to change the text being edited, you can do so by calling TESetText. A common technique (used in dialog boxes, for instance) is to allocate a single edit record for several separate pieces of text where only one may be edited at a time; this saves having to allocate an edit record for each of them.

When you've finished working with the text of an edit record, you can get a handle to the text by calling TEGetText. When you're completely done with an edit record and want to dispose of it, call TEDispose, which removes the text and edit record from the heap.

If you ever want to draw text in any given rectangle (without being able to edit it), use the TextBox procedure.

Advanced programmers may wish to use the TEScroll procedure, to move text within the view rectangle, or TECalText, to recalculate the beginning of each line after changing the text or the destination rectangle.

TEXTEDIT ROUTINES

This section describes all the procedures and functions in TextEdit. They are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see the QuickDraw manual *** and also "Notes for Assembly-Language Programmers" in this manual.

Initialization

PROCEDURE TEInit;

TEInit initializes TextEdit by allocating a handle for the scrap. The scrap is initially empty. Call this procedure once and only once at the beginning of your program.

FUNCTION TENew (destRect,viewRect: Rect) : TEHandle;

TENew allocates a handle for the text, builds and initializes an edit record, and returns a handle to the new edit record. DestRect and viewRect are the destination and view rectangles, respectively. Both rectangles are specified in the current grafPort's coordinates. Call this procedure once for every edit record you want allocated. The edit record incorporates the drawing environment of the grafPort, and is initialized for left-justified, single-spaced text with an insertion point at character position Ø.

Manipulating Edit Records

PROCEDURE TESetText (text: Ptr; length: LongInt; hTE: TEHandle);

TESetText takes the specified text and incorporates it into the edit record specified by hTE. The text parameter points to the text, and the length parameter indicates the number of characters in the text. The selection range is set to an insertion point at the end of the text. TESetText does not affect the text drawn in the destination rectangle, so call TEUpdate (described below) afterwards.

FUNCTION TEGetText (hTE: TEHandle) : CharsHandle;

TEGetText returns a handle to the text of the edit record specified by hTE. The CharsHandle data type is defined as:

```
TYPE CharsHandle = ^CharsPtr;
     CharsPtr    = ^Chars;
     Chars       = PACKED ARRAY [0..32000] OF CHAR;
```

PROCEDURE TEDispose (hTE: TEHandle);

TEDispose deallocates the space allocated for the edit record and text specified by hTE, and returns the memory to the free memory pool. Call this procedure when you're completely through with an edit record.

Editing

PROCEDURE TEKey (key: CHAR; hTE: TEHandle);

TEKey replaces the selection range in the text specified by hTE with the character given by the key parameter, and leaves an insertion point just past the inserted character. If the selection range is an insertion point, TEKey just inserts the character there. If the key parameter contains a Backspace character, the character immediately to the left of the insertion point is deleted. Call TEKey every time the Toolbox Event Manager function GetNextEvent reports a keyboard event that your application decides should be handled by TextEdit.

(eye)

> TEKey blindly inserts every character passed in the key parameter, so it's up to your application to filter out all characters that aren't actual text (such as keys typed in conjunction with modifier or special keys).

PROCEDURE TECut (hTE: TEHandle);

TECut removes the selection range from the text specified by hTE and places it in the scrap. Anything previously in the scrap is deleted. (See Figure 6.) If the selection range is an insertion point, the scrap is emptied.

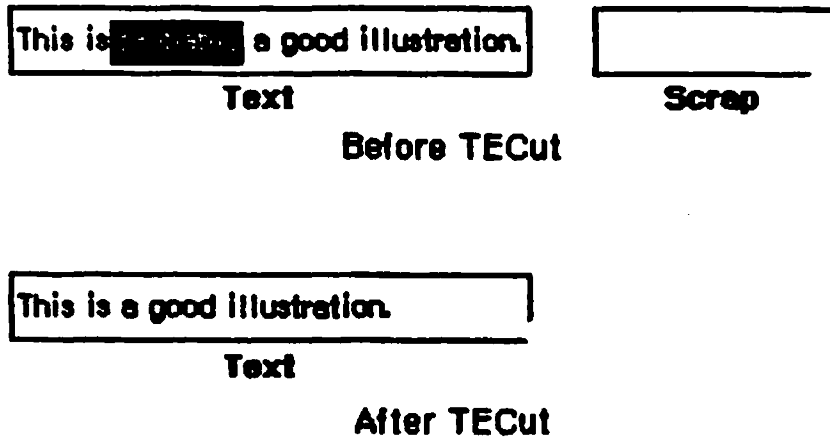


Figure 6. Cutting

PROCEDURE TECopy (hTE: TEHandle);

TECopy copies the selection range from the text specified by hTE into the scrap. Anything previously in the scrap is deleted. The selection range is not deleted. If the selection range is an insertion point, the scrap is emptied.

PROCEDURE TEPaste (hTE: TEHandle);

TEPaste replaces the selection range in the text specified by hTE with the scrap, and leaves an insertion point just past the inserted text. (See Figure 7.) If the scrap is empty, the selection range is deleted. If the selection range is an insertion point, TEPaste just inserts the scrap there.

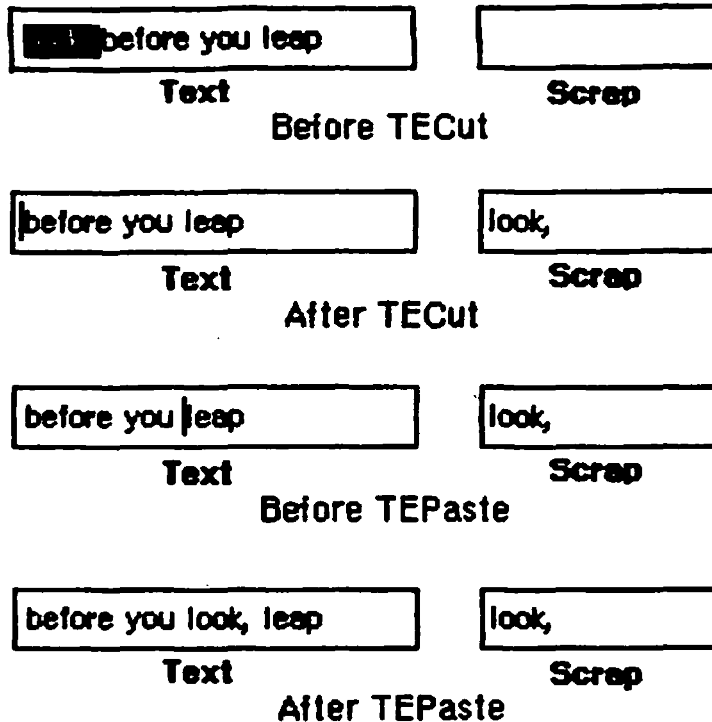


Figure 7. Cutting and Pasting

PROCEDURE TEDelete (hTE: TEHandle);

TEDelete removes the selection range from the text specified by hTE. It's the same as TECut (above) except that it doesn't transfer the selection range to the scrap. If the selection range is an insertion point, nothing happens.

PROCEDURE TEInsert (text: Ptr; length: LongInt; hTE: TEHandle);

TEInsert takes the specified text and inserts it, just before the selection range, into the text indicated by hTE. The text parameter points to the inserted text, and the length parameter indicates the number of characters to be inserted.

(eye)

> Any current selection range is not deleted. This is different from TEKey and TEPaste, and allows your application to support the Undo command (described in the Macintosh User Interface Guidelines) if you want.

Selection Range and Justification

PROCEDURE TESetSelect (selStart,selEnd: LongInt; hTE: TEHandle);

TESetSelect unhighlights the current selection range, and changes it to selStart and selEnd in the text specified by hTE. The new selection range is highlighted. If selStart and selEnd are equal, the selection range is an insertion point, and a caret is displayed.

SelEnd and selStart can range from Ø to 65535. If selEnd is anywhere beyond the last character of the text, the position just past the last character is used.

PROCEDURE TESetJust (j: INTEGER, hTE: TEHandle);

TESetJust changes the justification of the text specified by hTE to j. (See "Justification" under "The Editing Environment: Edit Record".) Call TEUpdate (described below under "Text Display") after TESetJust to cause the text to be redrawn with the new justification.

Mice and Carets

PROCEDURE TEClick (pt: Point; extend: BOOLEAN; hTE: TEHandle);

TEClick controls the placement and highlighting of the selection range as determined by mouse down events. Call TEClick whenever a mouse down event occurs in the view rectangle of the edit record specified by hTE. Pt is the mouse location (in local coordinates) at the time the button was pressed, obtainable from the event record. Pass TRUE for the extend parameter if the Event Manager indicates that the Shift key was held down at the time of the click (for an extended selection range).

(eye)

> Use the QuickDraw procedure GlobalToLocal to convert the global coordinates of the mouse location given in the event record to the local coordinate system for pt.

If the mouse moves, meaning that a drag is occurring, the selection range expands or shrinks accordingly. The current selection range is unhighlighted. In the case of a double click, meaning that word selection has been chosen, the word under the cursor becomes the selection range.

PROCEDURE TEIdle (hTE: TEHandle);

Call TEIdle repeatedly to make a blinking caret appear at the insertion point, if any, in the text specified by hTE. TextEdit observes a minimum blink interval: no matter how often you call TEIdle, the time between blinks will never be less than the minimum interval. You should call this procedure as often as possible to provide a constant frequency of blinking.

(hand)
The initial minimum blink interval setting is 4 ticks (sixtieths of a second). The user can adjust this setting to individual preference with the control panel desk accessory.

PROCEDURE TEActivate (hTE: TEHandle);

TEActivate highlights the selection range in the view rectangle of the edit record specified by hTE. If the selection range is an insertion point, it displays a caret there. This procedure should be called every time the Toolbox Event Manager function GetNextEvent reports that the text editing window has become active.

PROCEDURE TEDeactivate (hTE: TEHandle);

TEDeactivate unhighlights the selection range in the view rectangle of the edit record specified by hTE. If the selection range is an insertion point, it removes the caret. This procedure should be called every time the Toolbox Event Manager function GetNextEvent reports that the text editing window has become inactive.

Text Display

PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);

TEUpdate draws the text specified by hTE within the rectangle specified by rUpdate. The location of the rUpdate rectangle must be given in the coordinates of the grafPort. Call TEUpdate every time the Toolbox Event Manager function GetNextEvent reports an update event--after you call the Window Manager procedure BeginUpdate, and before you call EndUpdate.

Normally you'll use the following when an update event occurs:

```
BeginUpdate(myWindow);
TEUpdate(myWindow^.visRgn^^.rgnBBox, hTE);
EndUpdate(myWindow);
```

Instead of passing rUpdate as shown, you can pass the viewRect, but doing so may result in unnecessary drawing.

PROCEDURE TextBox (text: Ptr; length: LongInt; box: Rect; j: INTEGER);

TextBox draws the specified text in the rectangle indicated by the box parameter, with justification j. (See "Justification" under "The Editing Environment: Edit Record".) The text parameter points to the text, and the length parameter indicates the number of characters to draw. TextBox does not create an edit record, nor can the text that it draws be edited immediately; it's used solely for drawing text. For example:

```
str := 'Planning Procedures';
SetRect(r, 100, 100, 200, 200);
TextBox(@str[1], LENGTH(str), r, teFillCenter);
FrameRect(r);
```

Advanced Routines

These routines are useful only if you're directly accessing the fields of an edit record.

PROCEDURE TEScroll (dh,dv: INTEGER; hTE: TEHandle);

TEScroll moves ("scrolls") the text within the view rectangle of the specified edit record by the number of pixels specified in the dh and dv parameters. The edit record is specified by the hTE parameter. Positive dh and dv values move the text right and down, respectively, and negative values move the text left and up. For example,

```
TEScroll(0, -lnHeight, hTE)
```

scrolls the text up one line (where lnHeight is the value of the lineHeight field in the edit record).

PROCEDURE TECalText (hTE: TEHandle);

TECalText recalculates the beginnings of all lines of text in the edit record specified by hTE, updating elements of the lineStarts array. Call TECalText if you've changed the destination rectangle, the hText field, or anything else that effects the number of characters per line.

(hand)
There really are two ways to specify text to be edited. The easiest, direct method is to use TESetText, which takes an existing edit record, creates a second copy of its text, and makes the edit record point to the copy. An advanced, indirect method is to change the hText field

of the edit record directly, and then call TECalText to recalculate the lineStarts array to match the new text. If you have a lengthy text to edit, use the latter method to save space because it doesn't create a copy.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

Information about how to use the User Interface Toolbox from assembly language is given elsewhere *** (currently, in the QuickDraw manual) ***. This section contains special notes of interest to programmers who will be using TextEdit from assembly language.

If you use .INCLUDE to include a file named ToolEqu.Text when you assemble your program, the TextEdit constants and offsets into the fields of structured types in TextEdit will be available in symbolic form.

There are hooks within TextEdit that allow you insert additional procedures for more sophisticated editing. They require some care because they pass arguments in registers and it's the application's responsibility to save and restore the registers.

Two of the hooks are TEHiHook and TECarHook. If you install a nonzero address in either of these hooks, that address (instead of _InverRect) will be jumped to when a selection range is to be highlighted. The routine called can destroy the contents of the registers AØ, A1, DØ, D1, and D2. A3 will be pointing to a locked edit record, and teSelRect(A3) contains the rectangle enclosing the text being highlighted. For example, the following assembly-language fragment draws underlined selection ranges:

```
UnderHigh
        MOVE.L  (SP)+,AØ                     ;point to rectangle to be
                                             ;  highlighted
        MOVE    top(AØ),-(SP)                ;save existing top coordinate
        MOVE    bottom(AØ),top(AØ)           ;make the top coordinate equal
        SUBQ    #1,top(SP)                   ;  the bottom coordinate - 1
        MOVE.L  AØ,-(SP)                     ;invert the resulting
        _InverRect                           ;   rectangle
        MOVE    (SP)+,teSelRect+top(A3)      ;restore original top coordinate
        RTS
```

Note that the rectangle must be preserved.

TECarHook acts analogously upon insertion points instead of selection ranges. It must be called with teSelRect containing the insertion point rectangle.

*** The explanation of the other hooks is forthcoming. ***

SUMMARY OF TEXTEDIT

```
CONST teJustLeft   = 0;
      teJustCenter = 1;
      teJustRight  = -1;

TYPE CharsHandle = ^CharsPtr;
     CharsPtr    = ^Chars;
     Chars       = PACKED ARRAY [0..32000] OF CHAR;

     TEPtr    = ^TERec;
     TEHandle = ^TEPtr;
     TERec    = RECORD
                  destRect:   Rect;    {destination rectangle}
                  viewRect:   Rect;    {view rectangle}
                  lineHeight: INTEGER; {line height}
                  firstBL:    INTEGER; {position of first base line}
                  selStart:   INTEGER; {start of selection range}
                  selEnd:     INTEGER; {end of selection range}
                  just:       INTEGER; {justification}
                  length:     INTEGER; {length of text}
                  hText:      Handle;  {text to be edited}
                  txFont:     INTEGER; {text font}
                  txFace:     INTEGER; {character style}
                  txMode:     INTEGER; {pen mode}
                  txSize:     INTEGER; {type size}
                  inPort:     GrafPtr; {grafPort}
                  crOnly:     INTEGER  {new line at Return only, if <0}
                  nLines:     INTEGER; {number of lines}
                  lineStarts: ARRAY [0..32000] OF INTEGER;
                                       {positions of lines starts}
                  {more fields for internal use only}
                END;
```

Initialization

```
PROCEDURE TEInit;
FUNCTION  TENew   (destRect,viewRect: Rect) : TEHandle;
```

Manipulating Edit Records

```
PROCEDURE TESetText (text: Ptr; length: LongInt; hTE: TEHandle);
FUNCTION  TEGetText (hTE: TEHandle) : CharsHandle;
PROCEDURE TEDispose (hTE: TEHandle);
```

Editing

```
PROCEDURE TEKey    (key: CHAR; hTE: TEHandle);
PROCEDURE TECut    (hTE: TEHandle);
PROCEDURE TECopy   (hTE: TEHandle);
PROCEDURE TEPaste  (hTE: TEHandle);
PROCEDURE TEDelete (hTE: TEHandle);
PROCEDURE TEInsert (text: Ptr; length: LongInt; hTE: TEHandle);
```

Selection Range and Justification

```
PROCEDURE TESetSelect (selStart,selEnd: LongInt; hTE: TEHandle);
PROCEDURE TESetJust   (j: INTEGER; hTE: TEHandle);
```

Mice and Carets

```
PROCEDURE TEClick      (pt: Point; extend: BOOLEAN; hTE: TEHandle);
PROCEDURE TEIdle       (hTE: TEHandle);
PROCEDURE TEActivate   (hTE: TEHandle);
PROCEDURE TEDeactivate (hTE: TEHandle);
```

Text Display

```
PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);
PROCEDURE TextBox  (text: Ptr; length: LongInt; box: Rect; j: INTEGER);
```

Advanced Routines

```
PROCEDURE TEScroll  (dh,dv: INTEGER; hTE: TEHandle);
PROCEDURE TECalText (hTE: TEHandle);
```

GLOSSARY

caret: A generic term meaning a symbol that indicates where something should be inserted in text. The specific symbol used is a vertical bar.

character position: An index into an array containing text, starting at Ø for the first character.

destination rectangle: In TextEdit, the rectangle in which the text is drawn.

edit record: A complete editing environment, including the text to be edited, the grafPort and rectangle in which to display the text, the arrangement of the text within the rectangle, and other editing and display information.

insertion point: An empty selection range; the character position where text will be inserted (marked with a blinking caret by convention).

justification: The horizontal placement of lines of text relative to the edges of the rectangle in which the text is drawn.

line height: The number of pixels from the base line of one line of text to the base line of the next line of text.

nonbreaking space: The character with ASCII code $CA; drawn as a blank, but interpreted as a nonblank character for the purposes of word wrap.

scrap: A string consisting of the characters most recently cut or copied from text by certain TextEdit routines.

selection range: The series of characters (inversely highlighted), or the character position (marked with a blinking caret), at which the next editing operation will occur.

view rectangle: In TextEdit, the rectangle within which the text is visible.

word: In TextEdit, any series of printing characters, excluding spaces (ASCII code $2Ø) but including nonbreaking spaces (ASCII code $CA).

word wrap: Keeping any series of printing characters intact between lines when text is drawn.

MACINTOSH USER EDUCATION

The Toolbox Utilities: A Programmer's Guide /TOOLUTIL/UTIL

See Also: Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Memory Manager: A Programmer's Guide

Modification History:			
	First Draft	C. Rose	5/16/83
	Second Draft (ROM 7)	C. Rose	1/4/84
	Erratum Added	C. Rose	2/8/84

ABSTRACT

This manual describes the Toolbox Utilities, a set of routines and data types in the User Interface Toolbox that perform generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits.

Erratum:

When the Munger function does a replacement operation, it returns the offset of **the first byte past** where the replacement occurred.

TABLE OF CONTENTS

3 About This Manual
3 Fixed-Point Numbers
4 Toolbox Utility Routines
4 Fixed-Point Arithmetic
4 String Manipulation
5 Byte Manipulation
7 Bit Manipulation
8 Logical Operations
8 Other Operations on Long Integers
9 Graphics Utilities
11 Summary of the Toolbox Utilities
13 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved. Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Toolbox Utilities, a set of routines and data types in the User Interface Toolbox that perform generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits. *** Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

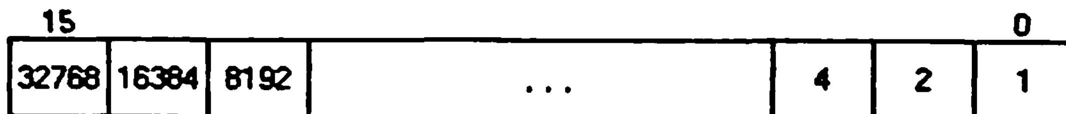
You should already be familiar with Lisa Pascal. Depending on which Toolbox Utilities you're interested in using, you may also need to know about the Macintosh Operating System's Memory Manager, the Resource Manager, and the basic concepts and structures behind QuickDraw.

This manual begins with a discussion of fixed-point numbers. This is followed by the detailed descriptions of all Toolbox Utility procedures and functions, their parameters, calling protocol, effects, side effects, and so on. Finally, there's a summary of the Toolbox Utilities, for quick reference, followed by a glossary of terms used in this manual. *** The glossary has only two entries, but eventually it will be merged with the glossaries from the other Toolbox and Operating System documentation. ***

FIXED-POINT NUMBERS

The Toolbox Utilities include routines for operating on fixed-point numbers. A fixed-point number is a 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word (see Figure 1). Since these numbers occupy the same number of bits as long integers, they could be given the data type LongInt; however, to reflect the different interpretation the bits have as fixed-point numbers, the following data type is defined in the Toolbox Utilities:

```
TYPE Fixed = LongInt;
```



integer (high order)

fraction (low order)

Figure 1. Fixed-Point Numbers

As described in the following section, there are Toolbox Utility routines for converting an integer numerator and denominator into a fixed-point number, multiplying two fixed-point numbers, and rounding a fixed-point number to the nearest integer. You can also use the general-purpose function HiWord (or LoWord) to extract the integer (or fractional) part of a fixed-point number.

TOOLBOX UTILITY ROUTINES

This section describes all the Toolbox Utility procedures and functions. They're presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual ***.

Fixed-Point Arithmetic

See also HiWord and LoWord under "Other Operations on Long Integers" below.

```
FUNCTION FixRatio (numerator,denominator: INTEGER) : Fixed;
```

FixRatio returns the fixed-point number having the given numerator and denominator (either of which may be any signed integer).

```
FUNCTION FixMul (a,b: Fixed) : Fixed;
```

FixMul multiplies the given fixed-point numbers and returns the result.

```
FUNCTION FixRound (x: Fixed) : INTEGER;
```

FixRound rounds the given fixed-point number to the nearest integer and returns the result.

String Manipulation

These routines use the StringHandle data type, which is defined in the Toolbox Utilities as follows:

```
TYPE StringPtr    = ^Str255;
     StringHandle = ^StringPtr;
```

FUNCTION NewString (s: Str255) : StringHandle;

NewString allocates the string specified by s as a relocatable object on the heap and returns a handle to it.

PROCEDURE SetString (h: StringHandle; s: Str255);

SetString sets the string whose handle is passed in h to the string specified by s.

FUNCTION GetString (stringID: INTEGER) : StringHandle;

GetString returns a stringHandle to the string having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('STR ',stringID).

Byte Manipulation

FUNCTION Munger (h: Handle; offset: LongInt; ptr1: Ptr; len1: LongInt;
ptr2: Ptr; len2: LongInt) : LongInt;

*** There's currently no Pascal interface to this routine; declare it as EXTERNAL in your program. ***

Munger manipulates bytes in the string of bytes (the "destination string") to which h is a handle. The offset parameter specifies a byte offset into the destination string. The exact nature of the operation done by Munger depends on the values of the remaining parameters, two pointer/length pairs. In general, (ptr1,len1) defines a substring to be replaced by the second substring (ptr2,len2). If these four parameters are all positive and nonzero, Munger looks for (ptr1,len1) in the destination string, starting from the given offset and ending at the end of the string; the first occurrence it finds is replaced by (ptr2,len2), and the offset at which the replacement occurred is returned. Figure 2 illustrates this; the bytes represent ASCII characters as shown.

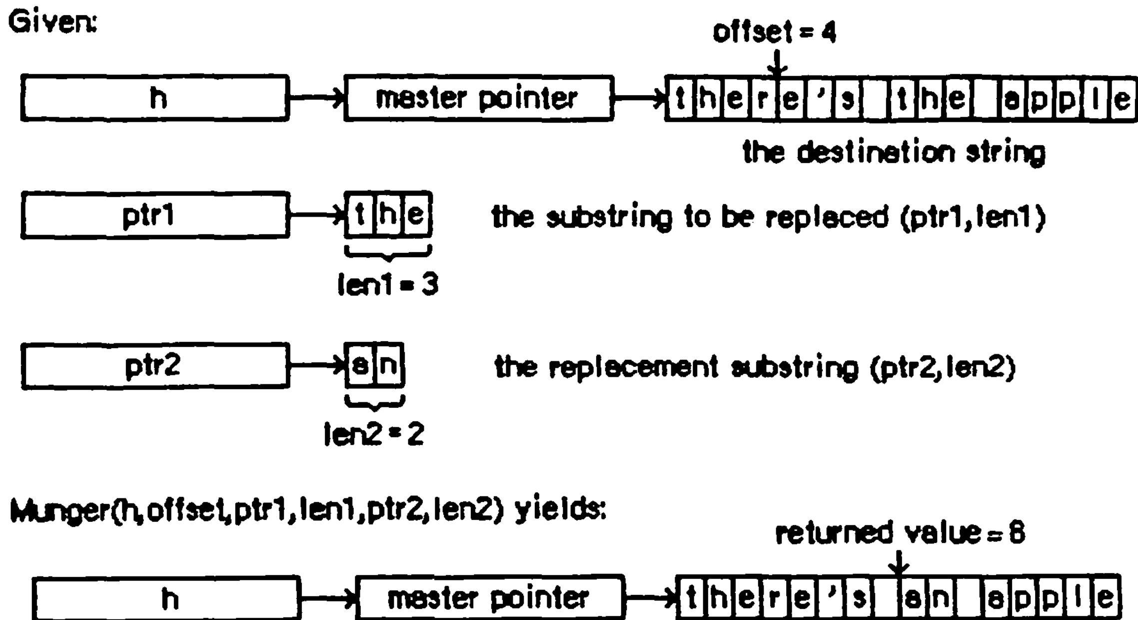


Figure 2. Munger Function

Different operations occur if any of the pointers or lengths is Ø:

- If ptrl is Ø, the substring of length lenl starting at the given offset is replaced by (ptr2,len2). If lenl is negative, the substring from the given offset to the end of the destination string is replaced by (ptr2,len2).

- If lenl is Ø, the substring (ptr2,len2) is simply inserted at the given offset.

- If ptr2 is Ø, the destination string isn't changed; Munger just returns the offset at which it found (ptrl,lenl).

- If len2 is Ø, the replacement substring is empty, so (ptrl,lenl) is deleted rather than replaced.

Munger returns the offset at which the operation occurred--whether replacement, insertion, deletion, or just location of a substring. It returns a negative value if it can't find (ptrl,lenl) in the destination string.

(eye)

Be careful not to specify an offset that's greater than the length of the destination string, or unpredictable things may happen.

Bit Manipulation

These routines manipulate a bit in data pointed to by a given pointer. A bit number indicates which bit; it starts at Ø for the high-order bit of the first byte pointed to and may be any positive long integer specifying an offset from that bit (see Figure 3).

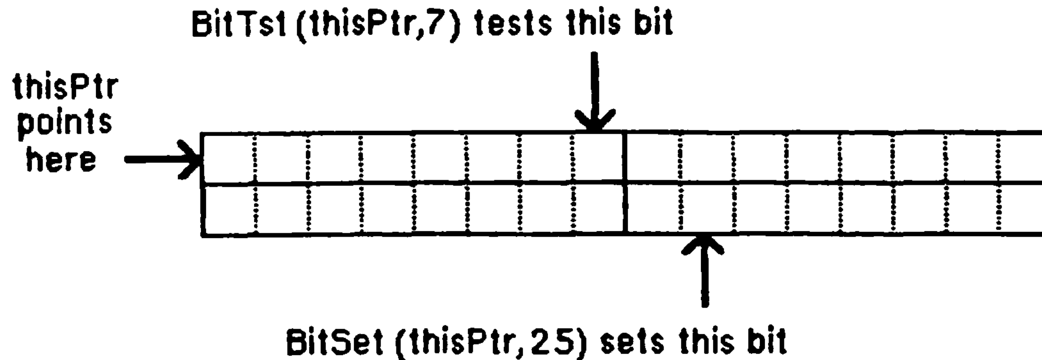


Figure 3. Bit Numbering for Utility Routines

(hand)
Note that this bit numbering is the opposite of the MC68ØØØ bit numbering.

FUNCTION BitTst (bytePtr: Ptr; bitNum: LongInt) : BOOLEAN;

BitTst tests whether a given bit is set and returns TRUE if so or FALSE if not. The bit is specified by bitNum, an offset from the high-order bit of the byte pointed to by bytePtr.

PROCEDURE BitSet (bytePtr: Ptr; bitNum: LongInt);

BitSet sets the bit specified by bitNum, an offset from the high-order bit of the byte pointed to by bytePtr.

PROCEDURE BitClr (bytePtr: Ptr; bitNum: LongInt);

BitSet clears the bit specified by bitNum, an offset from the high-order bit of the byte pointed to by bytePtr.

Logical Operations

FUNCTION BitAnd (long1,long2: LongInt) : LongInt;

BitAnd returns the result of the AND logical operation on the bits comprising the given long integers (long1 AND long2).

FUNCTION BitOr (long1,long2: LongInt) : LongInt;

BitOr returns the result of the OR logical operation on the bits comprising given long integers (long1 OR long2).

FUNCTION BitXor (long1,long2: LongInt) : LongInt;

BitXor returns the result of the XOR logical operation on the bits comprising the given long integers (long1 XOR long2).

FUNCTION BitNot (long: LongInt) : LongInt;

BitXor returns the result of the NOT logical operation on the bits comprising the given long integer.

FUNCTION BitShift (long: LongInt; count: INTEGER) : LongInt;

BitShift logically shifts the bits of the given long integer. Count specifies the direction and extent of the shift, and is taken modulo 31. If count is positive, BitShift shifts that many positions to the left; if count is negative, it shifts to the right. Zeros are shifted into empty positions at either end.

Other Operations on Long Integers

FUNCTION HiWord (x: LongInt) : INTEGER;

HiWord returns the high-order word of the given long integer. One use of this function is to extract the integer part of a fixed-point number.

FUNCTION LoWord (x: LongInt) : INTEGER;

LoWord returns the low-order word of the given long integer. One use of this function is to extract the fractional part of a fixed-point number.

PROCEDURE LongMul (a,b: LongInt; VAR dest: Int64Bit);

LongMul multiplies the given long integers and returns the result in dest, which has the following data type:

```
TYPE Int64Bit = RECORD
                  hiLong:  LongInt;
                  loLong:  LongInt
                END;
```

Graphics Utilities

FUNCTION GetIcon (iconID: INTEGER) : Handle;

GetIcon returns a handle to the icon having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('ICON',iconID).

PROCEDURE PlotIcon (theRect: Rect; theIcon: Handle);

*** There's currently no Pascal interface to this routine; declare it as EXTERNAL in your program. ***

PlotIcon draws the icon whose handle is theIcon in the rectangle theRect, which is in the local coordinates of the current grafPort. It calls the QuickDraw procedure CopyBits and uses the srcCopy transfer mode. (You must have initialized QuickDraw before calling PlotIcon.)

FUNCTION GetPattern (patID: INTEGER) : PatHandle;

GetIcon returns a handle to the pattern having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('PAT ',patID). The PatHandle data type is *** not yet, but soon will be *** defined in the Toolbox Utilities as follows:

```
TYPE PatPtr    = ^Pattern;
     PatHandle = ^PatPtr;
```

FUNCTION GetCursor (cursorID: INTEGER) : CursHandle;

GetIcon returns a handle to the cursor having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('CURS',cursorID). The CursHandle data type is *** not yet, but soon will be *** defined in the Toolbox Utilities as follows:

```
TYPE CursPtr    = ^Cursor;
     CursHandle = ^CursPtr;
```

PROCEDURE ShieldCursor (left,top,right,bottom: INTEGER);

Given the global coordinates of a rectangle, ShieldCursor removes the cursor from the screen if the cursor and the rectangle intersect.

FUNCTION GetPicture (pictureID: INTEGER) : PicHandle;

GetPicture returns a handle to the picture having the given resource ID, reading it from the resource file if necessary. It calls the Resource Manager function GetResource('PICT',pictureID). The PicHandle data type is defined in QuickDraw.

SUMMARY OF THE TOOLBOX UTILITIES

```
TYPE Fixed = LongInt;

     Int64Bit = RECORD
                  hiLong:  LongInt;
                  loLong:  LongInt
                END;

     StringPtr    = ^Str255;
     StringHandle = ^StringPtr;

     CursPtr    = ^Cursor;
     CursHandle = ^CursPtr;

     PatPtr    = ^Pattern;
     PatHandle = ^PatPtr;
```

Fixed-Point Arithmetic

```
FUNCTION FixRatio (numerator,denominator: INTEGER) : Fixed;
FUNCTION FixMul   (a,b: Fixed) : Fixed;
FUNCTION FixRound (x: Fixed) : INTEGER;
```

String Manipulation

```
FUNCTION  NewString (s: Str255) : StringHandle;
PROCEDURE SetString (h: StringHandle; s: Str255);
FUNCTION  GetString (stringID: INTEGER) : StringHandle;
```

Byte Manipulation

```
FUNCTION Munger (h: Handle; offset: LongInt; ptr1: Ptr; len1: LongInt;
                 ptr2: Ptr; len2: LongInt) : LongInt;
```

Bit Manipulation

```
FUNCTION  BitTst (bytePtr: Ptr; bitNum: LongInt) : BOOLEAN;
PROCEDURE BitSet (bytePtr: Ptr; bitNum: LongInt);
PROCEDURE BitClr (bytePtr: Ptr; bitNum: LongInt);
```

Logical Operations

```
FUNCTION BitAnd   (long1,long2: LongInt) : LongInt;
FUNCTION BitOr    (long1,long2: LongInt) : LongInt;
FUNCTION BitXor   (long1,long2: LongInt) : LongInt;
FUNCTION BitNot   (long: LongInt) : LongInt;
FUNCTION BitShift (long: LongInt; count: INTEGER) : LongInt;
```

Other Operations on Long Integers

```
FUNCTION  HiWord  (x: LongInt) : INTEGER;
FUNCTION  LoWord  (x: LongInt) : INTEGER;
PROCEDURE LongMul (a,b: LongInt; VAR dest: Int64Bit);
```

Graphics Utilities

```
FUNCTION  GetIcon      (iconID: INTEGER) : Handle;
PROCEDURE PlotIcon     (theRect: Rect; theIcon: Handle);
FUNCTION  GetPattern   (patID: INTEGER) : PatHandle;
FUNCTION  GetCursor    (cursorID: INTEGER) : CursHandle;
PROCEDURE ShieldCursor (left,top,right,bottom: INTEGER);
FUNCTION  GetPicture   (pictureID: INTEGER) : PicHandle;
```

GLOSSARY

fixed-point number: A 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word.

icon: A 32-by-32 bit image that represents an object, concept, or message.

File: ToolBox Names
Report: TrapList
Selection: Value/Trap: equals A000
through Value/Trap: equals AFFF

Value/	Name:	Fields:
A000	Open	
A001	Close	
A002	Read	
A003	Write	
A004	Control	
A005	Status	
A006	KillIO	
A007	GetVolInfo	
A008	FileCreate	
A009	FileDelete	
A00A	OpenRf	
A00B	Rename	
A00C	GetFileInfo	
A00D	SetFileInfo	
A00E	UnmountVol	
A00F	MountVol	
A010	FileAllocate	
A011	GetEOF	
A012	SetEOF	
A013	FlushVol	
A014	GetVol	
A015	SetVol	
A016	FInitQueue	
A017	Eject	
A018	GetFPos	
A019	InitZone	
A01A	GetZone	
A01B	SetZone	
A01C	FreeMem	
A01D	MaxMem	
A01E	NewPtr	
A01F	DisposePtr	
A020	SetPtrSize	
A021	GetPtrSize	
A022	NWHandle	
A023	DsposeHandle	
A024	SetHandleSize	
A025	GetHandleSize	
A026	HandleZone	
A027	ReAllocHandle	
A028	RecoverHandle	
A029	HLock	
A02A	HUnlock	
A02B	EmptyHandle	
A02C	InitApplZone	
A02D	SetApplLimit	
A02E	BlockMove	
A02F	PostEvent	
A030	OSEventAvail	
A031	GetOSEvent	
A032	FlushEvents	
A033	VInstall	
A034	VRemove	
A035	OffLine	
A036	MoreMasters	
A037	ReadParam	
A038	WriteParam	
A039	ReadDateTime	
A03A	SetDateTime	
A03B	Delay	
A03C	CmpString	
A03D	DrvrInstall	
A03E	DrvrRemove	
A03F	InitUtil	
A040	ResrvMem	
A041	SetFilLock	
A042	RstFilLock	
A043	SetFilType	
A044	SetFPos	
A045	FlushFil	
A046	GetTrapAddress	
A047	SetTrapAddress	
A048	PtrZone	
A049	HPurge	
A04A	HNoPurge	
A04B	SetGrowZone	
A04C	CompactMem	
A04D	PurgeMem	
A04E	AddDrive	
A04F	InstallRDrivers	
AC50	InitCursor	
AC51	SetCursor	
AC52	HideCursor	
AC53	ShowCursor	
AC54	UprString	
AC55	ShieldCursor	
AC56	ObscureCursor	
AC57	SetApplBase	
AC58	BitAnd	
AC59	BitXor	
AC5A	BitNot	
AC5B	BitOr	
AC5C	BitShift	
AC5D	BitTst	
AC5E	BitSet	
AC5F	BitClr	
AC61	Random	
AC62	ForeColor	
AC63	BackColor	
AC64	ColorBit	
AC65	GetPixel	
AC66	StuffHex	
AC67	LongMul	
AC68	FixMul	
AC69	FixRatio	
AC6A	HiWord	
AC6B	LoWord	
AC6C	FixRound	
AC6D	InitPort	
AC6E	InitGraf	
AC6F	OpenPort	
AC70	LocalToGlobal	
AC71	GlobalToLocal	
AC72	GrafDevice	
AC73	SetPort	
AC74	GetPort	
AC75	SetPortBits	
AC76	PortSize	
AC77	MovePortTo	
AC78	SetOrigin	
AC79	SetClip	
AC7A	GetClip	
AC7B	ClipRect	
AC7C	BackPat	
AC7D	ClosePort	
AC7E	AddPt	
AC7F	SubPt	
AC80	SetPt	
AC81	EqualPt	
AC82	StdText	
AC83	DrawChar	
AC84	DrawString	
AC85	DrawText	
AC86	TextWidth	
AC87	TextFont	
AC88	TextFace	
AC89	TextMode	
AC8A	TextSize	
AC8B	GetFontInfo	
AC8C	StringWidth	
AC8D	CharWidth	
AC8E	SpaceExtra	
AC90	StdLine	
AC91	LineTo	

File: ToolBox Names Page 2
Report: TrapList Feb 8, 1984
Selection: Value/Trap: equals A000
through Value/Trap: equals AFFF

Value/	Name:
AC92	Line
AC93	MoveTo
AC94	Moov
AC96	HidePen
AC97	ShowPen
AC98	GetPenState
AC99	SetPenState
AC9A	GetPen
AC9B	PenSize
AC9C	PenMode
AC9D	PenPat
AC9E	PenNormal
ACA0	StdRect
ACA1	FrameRect
ACA2	PaintRect
ACA3	EraseRect
ACA4	InvertRect
ACA5	FillRect
ACA6	EqualRect
ACA7	SetRect
ACA8	OffsetRect
ACA9	InsetRect
ACAA	SectRect
ACAB	UnionRect
ACAC	Pt2Rect
ACAD	PtInRect
ACAE	EmptyRect
ACAF	StdRRect
ACB0	FrameRoundRect
ACB1	PaintRoundRect
ACB2	EraseRoundRect
ACB3	InvertRoundRect
ACB4	FillRoundRect
ACB6	StdOval
ACB7	FrameOval
ACB8	PaintOval
ACB9	EraseOval
ACBA	InvertOval
ACBB	FillOval
ACBC	SlopeFromAngle
ACBD	StdArc
ACBE	FrameArc
ACBF	PaintArc
ACC0	EraseArc
ACC1	InvertArc
ACC2	FillArc
ACC3	PtToAngle
ACC4	AngleFromSlope
ACC5	StdPoly
ACC6	FramePoly
ACC7	PaintPoly
ACC8	ErasePoly
ACC9	InvertPoly
ACCA	FillPoly
ACCB	OpenPoly
ACCC	ClosePoly
ACCD	KillPoly
ACCE	OffsetPoly
ACCF	PackBits
ACD0	UnPackBits
ACD1	StdRgn
ACD2	FrameRgn
ACD3	PaintRgn
ACD4	EraseRgn
ACD5	InvertRgn
ACD6	FillRgn
ACD8	NewRgn
ACD9	DisposeRgn
ACDA	OpenRgn
ACDB	CloseRgn
ACDC	CopyRgn
ACDD	SetEmptyRgn
ACDE	SetRectRgn
ACDF	RectRgn
ACE0	OffsetRgn
ACE1	InsetRgn
ACE2	EmptyRgn
ACE3	EqualRgn
ACE4	SectRgn
ACE5	UnionRgn
ACE6	DiffRgn
ACE7	XOrRgn
ACE8	PtInRgn
ACE9	RectInRg
ACEA	SetStdProcs
ACEB	StdBits
ACEC	CopyBits
ACED	StdTxMeasure
ACEE	StdGetPic
ACEF	ScrollRect
ACF0	StdPutPic
ACF1	StdComment
ACF2	PicComment
ACF3	OpenPicture
ACF4	ClosePicture
ACF5	KillPicture
ACF6	DrawPicture
ACF8	ScalePt
ACF9	MapPt
ACFA	MapRect
ACFB	MapRgn
ACFC	MapPoly
ACFE	InitFonts
ACFF	GetFontName
AD00	GetFNum
AD01	FMSwapFont
AD02	RealFont
AD03	SetFontLock
AD04	DrawGrowIcon
AD05	DragGrayRgn
AD06	NewString
AD07	SetString
AD08	ShowHide
AD09	CalcVis
AD0A	CalcVisBehind
AD0B	ClipAbove
AD0C	PaintOne
AD0D	PaintBehind
AD0E	SaveOld
AD0F	DrawNew
AD10	GetWMgrPort
AD11	CheckUpDate
AD12	InitWindows
AD13	NewWindow
AD14	DisposeWindow
AD15	ShowWindow
AD16	HideWindow
AD17	GetWRefCon
AD18	SetWRefCon
AD19	GetWTitle
AD1A	SetWTitle
AD1B	MoveWindow
AD1C	HiliteWindow
AD1D	SizeWindow
AD1E	TrackGoAway
AD1F	SelectWindow
AD20	BringToFront
AD21	SendBehind
AD22	BeginUpdate
AD23	EndUpdate
AD24	FrontWindow
AD25	DragWindow
AD26	DragTheRgn
AD27	InvalRgn

Fields:

File: ToolBox Names
Report: TrapList
Selection: Value/Trap: equals A000
through Value/Trap: equals AFFF

Value/	Name:
AD28	InvalRect
AD29	ValidRgn
AD2A	ValidRect
AD2B	GrowWindow
AD2C	FindWindow
AD2D	CloseWindow
AD2E	SetWindowPic
AD2F	GetWindowPic
AD30	InitMenus
AD31	NewMenu
AD32	DisposeMenu
AD33	AppendMenu
AD34	ClearMenuBar
AD35	InsertMenu
AD36	DeleteMenu
AD37	DrawMenuBar
AD38	HiliteMenu
AD39	EnableItem
AD3A	DisableItem
AD3B	GetMenuBar
AD3C	SetMenuBar
AD3D	MenuSelect
AD3E	MenuKey
AD3F	GetItemIcon
AD40	SetItemIcon
AD41	GetItemStyle
AD42	SetItemStyle
AD43	GetItemMark
AD44	SetItemMark
AD45	CheckItem
AD46	GetItem
AD47	SetItem
AD48	CalcMenuSize
AD49	GetMHandle
AD4A	SetMenuFlash
AD4B	PlotIcon
AD4C	FlashMenuBar
AD4D	AddResMenu
AD4E	PinRect
AD4F	DeltaPoint
AD50	CountMItems
AD51	InsertResMenu
AD54	NewControl
AD55	DisposeControl
AD56	KillControls
AD57	ShowControl
AD58	HideControl
AD59	MoveControl
AD5A	GetCRefCon
AD5B	SetCRefCon
AD5C	SizeControl
AD5D	HiliteControl
AD5E	GetCTitle
AD5F	SetCTitle
AD60	GetCtlValue
AD61	GetCtlMin
AD62	GetCtlMax
AD63	SetCtlValue
AD64	SetCtlMin
AD65	SetCtlMax
AD66	TestControl
AD67	DragControl
AD68	TrackControl
AD69	DrawControls
AD6A	GetCtlAction
AD6B	SetCtlAction
AD6C	FindControl
AD6E	DeQueue
AD6F	EnQueue
AD70	GetNextEvent
AD71	EventAvail
AD72	GetMouse
AD73	StillDown
AD74	Button
AD75	TickCount
AD76	GetKeys
AD77	WaitMouseUp
AD79	CouldDialog
AD7A	FreeDialog
AD7B	InitDialogs
AD7C	GetNewDialog
AD7D	NewDialog
AD7E	SetIText
AD7F	IsDialogEvent
AD80	DialogSelect
AD81	DrawDialog
AD82	CloseDialog
AD83	DisposeDialog
AD85	Alert
AD86	StopAlert
AD87	NoteAlert
AD88	CautionAlert
AD89	CouldAlert
AD8A	FreeAlert
AD8B	ParamText
AD8C	ErrorSound
AD8D	GetDItem
AD8E	SetDItem
AD8F	SetIText
AD90	GetIText
AD91	ModalDialog
AD92	DetachResouce
AD93	SetResPurge
AD94	CurResFile
AD95	InitResources
AD96	RsrcZoneInit
AD97	OpenResFile
AD98	UseResFile
AD99	UpdateResFile
AD9A	CloseResFile
AD9B	SetResLoad
AD9C	CountResources
AD9D	GetIndResource
AD9E	CountTypes
AD9F	GetIndType
ADA0	GetResource
ADA1	GetNamedResourc
ADA2	LoadResource
ADA3	ReleaseResource
ADA4	HomeResFile
ADA5	SizeRsrc
ADA6	GetResAttrs
ADA7	SetResAttrs
ADA8	GetResInfo
ADA9	SetResInfo
ADAA	ChangedResData
ADAB	AddResource
ADAC	AddReference
ADAD	RmveResource
ADAE	RmveReference
ADAF	ResError
ADB0	WriteResource
ADB1	CreateResFile
ADB2	SystemEvent
ADB3	SystemClick
ADB4	SystemTask
ADB5	SystemMenu
ADB6	OpenDeskAcc
ADB7	CloseDeskAcc
ADB8	GetPattern
ADB9	GetCursor
ADBA	GetString
ADBB	GetIcon
ADBC	GetPicture

Fields:

Value/	Name:	Fields:	
ADBD	GetNewWindow	ADF2	Launch
ADBE	GetNewControl	ADF3	Chain
ADBF	GetMenu	ADF4	ExitToShell
ADC0	GetNewMBar	ADF5	GetAppParms
ADC1	UniqueID	ADF6	GetResFileAttrs
ADC2	SystemEdit	ADF7	SetResFileAttrs
ADC8	SystemBeep	ADF9	InfoScrap
ADC9	SystemError	ADFA	UnloadScrap
ADCA	PutIcon	ADFB	LoadScrap
ADCB	TeGetText	ADFC	ZeroScrap
ADCC	TEInit	ADFD	GetScrap
ADCD	TEDispose	ADFE	PutScrap
ADCE	TextBox		
ADCF	TESetText		
ADD0	TECalText		
ADD1	TESetSelect		
ADD2	TENew		
ADD3	TEUpdate		
ADD4	TEClick		
ADD5	TECopy		
ADD6	TECut		
ADD7	TEDelete		
ADD8	TEActivate		
ADD9	TEDeactivate		
ADDA	TEIdle		
ADDB	TEPaste		
ADDC	TEKey		
ADDD	TEScroll		
ADDE	TEInsert		
ADDF	TESetJust		
ADE0	Munger		
ADE1	HandToHand		
ADE2	PtrToXHand		
ADE3	PtrToHand		
ADE4	HandAndHand		
ADE5	InitPack		
ADE6	InitMath		
ADE7	Pack0		
ADE8	Pack1		
ADE9	Pack2		
ADEA	Pack3		
ADEB	Pack4		
ADEC	Pack5		
ADED	Pack6		
ADEE	Pack7		
ADEF	PtrAndHand		
ADF0	LoadSeg		
ADF1	UnLoadSeg		

File: ToolBox Names Page 1
Report: TrapList Feb 8, 1984
Selection: Value/Trap: equals A000
through Value/Trap: equals FFFF

Name:	Value/	Fields:			
AddDrive	A04E	CopyRgn	ACDC	EraseArc	ACC0
AddPt	AC7E	CouldAlert	AD89	EraseOval	ACB9
AddReference	ADAC	CouldDialog	AD79	ErasePoly	ACC8
AddResMenu	AD4D	CountMItems	AD50	EraseRect	ACA3
AddResource	ADAB	CountResources	AD9C	EraseRgn	ACD4
Alert	AD85	CountTypes	AD9E	EraseRoundRect	ACB2
AngleFromSlope	ACC4	CreateResFile	ADB1	ErrorSound	AD8C
AppendMenu	AD33	CurResFile	AD94	EventAvail	AD71
BackColor	AC63	Delay	A03B	ExitToShell	ADF4
BackPat	AC7C	DeleteMenu	AD36	FileAllocate	A010
BeginUpdate	AD22	DeltaPoint	AD4F	FileCreate	A008
BitAnd	AC58	DeQueue	AD6E	FileDelete	A009
BitClr	AC5F	DetachResouce	AD92	FillArc	ACC2
BitNot	AC5A	DialogSelect	AD80	FillOval	ACBB
BitOr	AC5B	DiffRgn	ACE6	FillPoly	ACCA
BitSet	AC5E	DisableItem	AD3A	FillRect	ACA5
BitShift	AC5C	DisposeControl	AD55	FillRgn	ACD6
BitTst	AC5D	DisposeDialog	AD83	FillRoundRect	ACB4
BitXor	AC59	DisposeMenu	AD32	FindControl	AD6C
BlockMove	A02E	DisposePtr	A01F	FindWindow	AD2C
BringToFront	AD20	DisposeRgn	ACD9	FInitQueue	A016
Button	AD74	DisposeWindow	AD14	FixMul	AC68
CalcMenuSize	AD48	DragControl	AD67	FixRatio	AC69
CalcVis	AD09	DragGrayRgn	AD05	FixRound	AC6C
CalcVisBehind	AD0A	DragTheRgn	AD26	FlashMenuBar	AD4C
CautionAlert	AD88	DragWindow	AD25	FlushEvents	A032
Chain	ADF3	DrawChar	AC83	FlushFil	A045
ChangedResData	ADAA	DrawControls	AD69	FlushVol	A013
CharWidth	AC8D	DrawDialog	AD81	FMSwapFont	AD01
CheckItem	AD45	DrawGrowIcon	AD04	ForeColor	AC62
CheckUpDate	AD11	DrawMenuBar	AD37	FrameArc	ACBE
ClearMenuBar	AD34	DrawNew	AD0F	FrameOval	ACB7
ClipAbove	AD0B	DrawPicture	ACF6	FramePoly	ACC6
ClipRect	AC7B	DrawString	AC84	FrameRect	ACA1
Close	A001	DrawText	AC85	FrameRgn	ACD2
CloseDeskAcc	ADB7	DrvrInstall	A03D	FrameRoundRect	ACB0
CloseDialog	AD82	DrvrRemove	A03E	FreeAlert	AD8A
ClosePicture	ACF4	DsposeHandle	A023	FreeDialog	AD7A
ClosePoly	ACCC	Eject	A017	FreeMem	A01C
ClosePort	AC7D	EmptyHandle	A02B	FrontWindow	AD24
CloseResFile	AD9A	EmptyRect	ACAE	GetAppParms	ADF5
CloseRgn	ACDB	EmptyRgn	ACE2	GetClip	AC7A
CloseWindow	AD2D	EnableItem	AD39	GetCRefCon	AD5A
CmpString	A03C	EndUpdate	AD23	GetCTitle	AD5E
ColorBit	AC64	EnQueue	AD6F	GetCtlAction	AD6A
CompactMem	A04C	EqualPt	AC81	GetCtlMax	AD62
Control	A004	EqualRect	ACA6	GetCtlMin	AD61
CopyBits	ACEC	EqualRgn	ACE3	GetCtlValue	AD60

File: ToolBox Names　　　　　　　　　　　　　Page 2
Report: TrapList　　　　　　　　　　　　　　Feb 8, 1984
Selection: Value/Trap: equals A000
  through Value/Trap: equals FFFF

Name:	Value/ Fields:
GetCursor	ADB9
GetDItem	AD8D
GetEOF	A011
GetFileInfo	A00C
GetFNum	AD00
GetFontInfo	AC8B
GetFontName	ACFF
GetFPos	A018
GetHandleSize	A025
GetIcon	ADBB
GetIndResource	AD9D
GetIndType	AD9F
GetItem	AD46
GetItemIcon	AD3F
GetItemMark	AD43
GetItemStyle	AD41
GetIText	AD90
GetKeys	AD76
GetMenu	ADBF
GetMenuBar	AD3B
GetMHandle	AD49
GetMouse	AD72
GetNamedResourc	ADA1
GetNewControl	ADBE
GetNewDialog	AD7C
GetNewMBar	ADC0
GetNewWindow	ADBD
GetNextEvent	AD70
GetOSEvent	A031
GetPattern	ADB8
GetPen	AC9A
GetPenState	AC98
GetPicture	ADBC
GetPixel	AC65
GetPort	AC74
GetPtrSize	A021
GetResAttrs	ADA6
GetResFileAttrs	ADF6
GetResInfo	ADA8
GetResource	ADA0
GetScrap	ADFD
GetString	ADBA
GetTrapAddress	A046
GetVol	A014
GetVolInfo	A007
GetWindowPic	AD2F
GetWMgrPort	AD10
GetWRefCon	AD17
GetWTitle	AD19
GetZone	A01A
GlobalToLocal	AC71
GrafDevice	AC72
GrowWindow	AD2B
HandAndHand	ADE4
HandleZone	A026
HandToHand	ADE1
HideControl	AD58
HideCursor	AC52
HidePen	AC96
HideWindow	AD16
HiliteControl	AD5D
HiliteMenu	AD38
HiliteWindow	AD1C
HiWord	AC6A
HLock	A029
HNoPurge	A04A
HomeResFile	ADA4
HPurge	A049
HUnlock	A02A
InfoScrap	ADF9
InitApplZone	A02C
InitCursor	AC50
InitDialogs	AD7B
InitFonts	ACFE
InitGraf	AC6E
InitMath	ADE6
InitMenus	AD30
InitPack	ADE5
InitPort	AC6D
InitResources	AD95
InitUtil	A03F
InitWindows	AD12
InitZone	A019
InsertMenu	AD35
InsertResMenu	AD51
InsetRect	ACA9
InsetRgn	ACE1
InstallRDrivers	A04F
InvalRect	AD28
InvalRgn	AD27
InvertArc	ACC1
InvertOval	ACBA
InvertPoly	ACC9
InvertRect	ACA4
InvertRgn	ACD5
InvertRoundRect	ACB3
IsDialogEvent	AD7F
KillControls	AD56
KillIO	A006
KillPicture	ACF5
KillPoly	ACCD
Launch	ADF2
Line	AC92
LineTo	AC91
LoadResource	ADA2
LoadScrap	ADFB
LoadSeg	ADF0
LocalToGlobal	AC70
LongMul	AC67
LoWord	AC6B
MapPoly	ACFC
MapPt	ACF9
MapRect	ACFA
MapRgn	ACFB
MaxMem	A01D
MenuKey	AD3E
MenuSelect	AD3D
ModalDialog	AD91
Moov	AC94
MoreMasters	A036
MountVol	A00F
MoveControl	AD59
MovePortTo	AC77
MoveTo	AC93
MoveWindow	AD1B
Munger	ADE0
NewControl	AD54
NewDialog	AD7D
NewMenu	AD31
NewPtr	A01E
NewRgn	ACD8
NewString	AD06
NewWindow	AD13
NoteAlert	AD87
NWHandle	A022
ObscureCursor	AC56
OffLine	A035
OffsetPoly	ACCE
OffsetRect	ACA8
OffsetRgn	ACE0
Open	A000
OpenDeskAcc	ADB6
OpenPicture	ACF3
OpenPoly	ACCB

File: ToolBox Names
Report: TrapList
Selection: Value/Trap: equals A000
through Value/Trap: equals FFFF

Page 3
Feb 8, 1984

Name:	Value/	Fields:			
OpenPort	AC6F	ReAllocHandle	A027	SetOrigin	AC78
OpenResFile	AD97	RecoverHandle	A028	SetPenState	AC99
OpenRf	A00A	RectInRg	ACE9	SetPort	AC73
OpenRgn	ACDA	RectRgn	ACDF	SetPortBits	AC75
OSEventAvail	A030	ReleaseResource	ADA3	SetPt	AC80
Pack0	ADE7	Rename	A00B	SetPtrSize	A020
Pack1	ADE8	ResError	ADAF	SetRect	ACA7
Pack2	ADE9	ResrvMem	A040	SetRectRgn	ACDE
Pack3	ADEA	RmveReference	ADAE	SetResAttrs	ADA7
Pack4	ADEB	RmveResource	ADAD	SetResFileAttrs	ADF7
Pack5	ADEC	RsrcZoneInit	AD96	SetResInfo	ADA9
Pack6	ADED	RstFilLock	A042	SetResLoad	AD9B
Pack7	ADEE	SaveOld	AD0E	SetResPurge	AD93
PackBits	ACCF	ScalePt	ACF8	SetStdProcs	ACEA
PaintArc	ACBF	ScrollRect	ACEF	SetString	AD07
PaintBehind	AD0D	SectRect	ACAA	SetTrapAddress	A047
PaintOne	AD0C	SectRgn	ACE4	SetVol	A015
PaintOval	ACB8	SelectWindow	AD1F	SetWindowPic	AD2E
PaintPoly	ACC7	SendBehind	AD21	SetWRefCon	AD18
PaintRect	ACA2	SetApplBase	AC57	SetWTitle	AD1A
PaintRgn	ACD3	SetApplLimit	A02D	SetZone	A01B
PaintRoundRect	ACB1	SetClip	AC79	ShieldCursor	AC55
ParamText	AD8B	SetCRefCon	AD5B	ShowControl	AD57
PenMode	AC9C	SetCTitle	AD5F	ShowCursor	AC53
PenNormal	AC9E	SetCtlAction	AD6B	ShowHide	AD08
PenPat	AC9D	SetCtlMax	AD65	ShowPen	AC97
PenSize	AC9B	SetCtlMin	AD64	ShowWindow	AD15
PicComment	ACF2	SetCtlValue	AD63	SizeControl	AD5C
PinRect	AD4E	SetCursor	AC51	SizeRsrc	ADA5
PlotIcon	AD4B	SetDateTime	A03A	SizeWindow	AD1D
PortSize	AC76	SetDItem	AD8E	SlopeFromAngle	ACBC
PostEvent	A02F	SetEmptyRgn	ACDD	SpaceExtra	AC8E
Pt2Rect	ACAC	SetEOF	A012	Status	A005
PtInRect	ACAD	SetFileInfo	A00D	StdArc	ACBD
PtInRgn	ACE8	SetFilLock	A041	StdBits	ACEB
PtrAndHand	ADEF	SetFilType	A043	StdComment	ACF1
PtrToHand	ADE3	SetFontLock	AD03	StdGetPic	ACEE
PtrToXHand	ADE2	SetFPos	A044	StdLine	AC90
PtrZone	A048	SetGrowZone	A04B	StdOval	ACB6
PtToAngle	ACC3	SetHandleSize	A024	StdPoly	ACC5
PurgeMem	A04D	SetItem	AD47	StdPutPic	ACF0
PutIcon	ADCA	SetItemIcon	AD40	StdRect	ACA0
PutScrap	ADFE	SetItemMark	AD44	StdRgn	ACD1
Random	AC61	SetItemStyle	AD42	StdRRect	ACAF
Read	A002	SetIText	AD7E	StdText	AC82
ReadDateTime	A039	SetIText	AD8F	StdTxMeasure	ACED
ReadParam	A037	SetMenuBar	AD3C	StillDown	AD73
RealFont	AD02	SetMenuFlash	AD4A	StopAlert	AD86

Selection: Value/Trap: equals A000
through Value/Trap: equals FFFF

Name:	Value/	Fields:	
StringWidth	AC8C	UprString	AC54
StuffHex	AC66	UseResFile	AD98
SubPt	AC7F	ValidRect	AD2A
SystemBeep	ADC8	ValidRgn	AD29
SystemClick	ADB3	VInstall	A033
SystemEdit	ADC2	VRemove	A034
SystemError	ADC9	WaitMouseUp	AD77
SystemEvent	ADB2	Write	A003
SystemMenu	ADB5	WriteParam	A038
SystemTask	ADB4	WriteResource	ADB0
TEActivate	ADD8	XOrRgn	ACE7
TECalText	ADD0	ZeroScrap	ADFC
TEClick	ADD4		
TECopy	ADD5		
TECut	ADD6		
TEDeactivate	ADD9		
TEDelete	ADD7		
TEDispose	ADCD		
TeGetText	ADCB		
TEIdle	ADDA		
TEInit	ADCC		
TEInsert	ADDE		
TEKey	ADDC		
TENew	ADD2		
TEPaste	ADDB		
TEScroll	ADDD		
TESetJust	ADDF		
TESetSelect	ADD1		
TESetText	ADCF		
TestControl	AD66		
TEUpdate	ADD3		
TextBox	ADCE		
TextFace	AC88		
TextFont	AC87		
TextMode	AC89		
TextSize	AC8A		
TextWidth	AC86		
TickCount	AD75		
TrackControl	AD68		
TrackGoAway	AD1E		
UnionRect	ACAB		
UnionRgn	ACE5		
UniqueID	ADC1		
UnloadScrap	ADFA		
UnLoadSeg	ADF1		
UnmountVol	A00E		
UnPackBits	ACD0		
UpdateResFile	AD99		

ROM 7.0 MacsBug Summary

To use MacsBug, include it on your Mac diskette. The system will say 'MacsBug installed' when the diskette is booted. You may also include the Disassembler in the same manner.

The Mac's modem port should be connected to another computer or terminal running at 9600 baud, no parity. Press the interrupt switch after booting the disk. The mouse should freeze and no error message should appear. On the terminal, a register dump should appear, and an asterisk '*' prompt.

Commands available:

Command	Description		Examples
DM	Display Memory	DM 100 100	DM RA7,-10 20
SM	Set Memory	SM 0 1 2 3 4 5 6	SM 0 'ABCDE'
D#	Display/Set data register #		D0 0000FFFF
A#	Display/Set address register #		A0
SR	Display/Set status register		
PC	Display/Set program counter		
US	Display/Set user stack	(normally A7)	
SS	Display/Set supervisor stack	(normally A7)	
BR	Display/Set break points		BR
	(up to eight)		BR 4DD0 552A
			BR CLEAR
A	Display all address registers		
D	Display all data registers		
TD	Display all registers		
CV	Convert between base 10 and 16		CV $FDEF
	(all arithmetic is 32 bit)		CV &65536

To do hexidecimal addition, use CV $num1,num2
To do hexidecimal subtraction, use CV $num1,-num2
To do hexidecimal negation, use CV $-num1

Command	Description		Examples
G	Go	(continue)	G
		(start at 44D0)	G 44D0
		(continue until PC = 55EA)	G TILL 55EA
T	Trace		T 17
AT	Trace Traps	(traces all traps)	AT
		(trace GetNextEvent)	AT 170
		(trace all Bit Traps)	AT 158 15F
		(trace GetNextEvent in code block at 5000 - 53FF)	AT 170 5000 53FF
		(trace all Bit Traps in code block at 5000 - 53FF)	AT 158 15F 5000 53FF

AB	Break Traps	same as AT, but breaks before executing trap	
HD	Handle Display	lists all allocated handles	HD
AD	Data Break		AD 158 15F 5000 53FF

A simple checksum is calculated for the specified memory range. As each Trap is encountered, the checksum is recalculated. If the checksum differs, the debugger breaks. This call must be made with all four arguments.

AX	Cancel Break	AX

Clears the current AT, AB, AH, AC, or HS command

Disassembler Calls

IL	List lists the next 20 instructions	IL 4DD0 IL
ID	List One lists one instruction	ID 4DD0 ID

Debugger Notation

/	Command Separator	SM PC 4E71 / G
.	Last Address (for DM, SM, IL, ID)	 DM . 100
,	Offset	DM .,100 100
RA#	Address Register	DM RA7,-10 20
RD#	Data Register	SM RA0, RD2

Advanced Debugger Calls

AH	Heap Break	AH 158 15F

A heap check is made as each specified Trap is encountered. If $1A3E8 = 0 then the applzone is checked. (default) If $1A3E8 <> 0 then SysZone is checked. The trap range must be greater than $2E.

An error returns: Bad Heap at A1 A2 where:
A1 = the previous block pointer
A2 = the bad block pointer

HC Heap Check HC

This call checks the heap as described in AH. An error is returned if any of the following conditions are true:

The block size is past the top of memory
The block size is odd
For tree blocks, the next link is negative or past the top of memory
For tree blocks, the previous link is negative or past the top of memory
For rel. blocks, the back pointer is odd
The heap base + back pointer is past the top of memory
The heap base + back pointer do not point to the right master pointer.

HS Heap Scramble HS

If the traps NewPtr, SetPtrSize, NewHandle, SetHandleSize, HandleZone or ReAllocHandle are encountered, the heap is scrambled before executing the trap. It also preforms a heap check before scrambling on all traps > 30.

MR Magic Return MR

This assumes the first word on the stack is a return address generated by a BSR or JSR. It substitutes a break point for the return address. The execution continues until a break occurs. Then, SR is restored.

This is not nestable. All other break point commands are still active.

Known Problems

It is a good idea to initialize DM and IL. DM 0, and ID PC, for example.

DM RA5, as example, intermittently generates an address error. To fix, explicitly type the address in register.

SM PC 4E71, for example, makes the system respond unreliably if a trap or breakpoint was set at that location.

AT, as example, returns a Line 1111 exception. To fix, reboot.

Pascal Program Debug Strategy

Use DM to determine where the program is in memory. Seven letters of each procedure and function will appear in the ASCII columns. (The first letter has its high bit set.) The user program usually starts about 4DD0. The mainline procedures and functions are first, followed by the units and external procedures and functions in the order that you link them. Each procedure or function name suceeds the procedure or function code. To make life easier, link your own units before linking ToolTraps, MemTraps and MacPasLib.

If the program doesn't appear to work at all, find the address of the start of the program. It will be immediately after the name of the last procedure or function.

If you disassemble at that address, you will see a LINK instruction for A6 and a LINK instruction for A5. These address registers are used by Pascal to locate all variables and procedural parameters. Global variables are referenced negatively off A5. Local variables are referenced negatively off A6. Procedural parameters are referenced with a positive offset from A6.

ToolBox calls and other calls to unit-resident procedures and functions are made through JSR *+addr instructions. The table ToolTraps is linked to your program, and contains all of the actual calls themselves.

If you are writing a Pascal program that uses the ToolBox, the first thing you probably do is:

```
InitGraf (@thePort);
```

This assembles into:

```
        LEA     $FF1E(A5),A0
        MOVE.L  A0,A7
        JSR     *+addr
```

You should see this shortly after the LINK instructions. This establishes the beginning of your program.

To find out where this program fails, interrupt the Finder. Set G TILL addr where addr is the beginning of your program. Restart your program. If the program breaks sucessfully at that point, continue to use G TILL to selectly execute your program until it fails more spectacularly, or locks up.

You will find that the 68000 code that the Pascal compiler produces is reasonably readable, and that the compiler produces smart code.

To complement the debugger information, you may want to add debugging code in your program itself. One easy way to do so is to do WRITE s or WRITELN s to the .BOUT port. This information will appear on your debugging terminal. The code fragments required look like:

```
VAR debug : TEXT;

REWRITE (debug, '.BOUT');

WRITELN (debug, chr(10) {linefeed}, 'This is a test', 12345);
```

WRITE and WRITELN support strings, chars, packed array of chars, integers, and booleans.

MACINTOSH PUBLICATIONS

Macintosh User Interface Guidelines	USER.INTERFACE/NEWUIDOC

See Also:

Modification History:		
	First Draft	Hoffman 3/17/82
	Rearranged and Revised	Espinosa 5/11/82
	Total Redesign	Espinosa 5/21/82
	Second Edition Prerelease	Espinosa 7/11/82
	Second Edition	Espinosa 10/11/82

ABSTRACT

One of the major factors in making a system pleasant and easy to use is the system's consistency. This specification's purpose is to set down our agreements about the way programs will interact with users, so that we have a common method for dealing with interface problems, and so that all software written for the Macintosh computer (in-house or by outside vendors) will be consistent with respect to the issues discussed here.

CONFIDENTIAL

CONTENTS

Page	Title	File
		UIDOC
		COVER
		OUTLINE
5	Introduction	INTRO
5	Software Developers' Responsibility	
6	Macintosh's Commitment	
6	About Modes	
8	The Graphic Screen	SCREEN
9	Icons	
11	Accepting User Input	MOUSE
11	The Mouse	
12	Mouse Actions	
13	Double-Clicking	
13	Changing Pointer Shapes	
14	The Keyboard	KEYBOARD
14	Character Keys	
15	Modifier Keys	
15	The COMMAND Key	
15	Special Keys	
16	Typeahead, Auto-repeat, and Audio Feedback	
16	Versions of the Keyboard	
17	The Numeric Keypad	
18	Conceptual Models: Tools and Documents	MODELS
19	Files	
19	Tools	
20	Documents	
21	Resources	
22	The DeskTop Model of Organization	DESKTOP
22	The Desk	
24	Windows	WINDOWS
24	Opening and Closing Windows	
25	The Active Window	
25	Document Windows	
25	Scroll Bars	
27	Multiple Windows	
27	Moving a Window	
28	Changing the Size of a Window	
29	Splitting a Window	
30	Desk Accessories	ACCESSORY
31	Who's on Top?	ONTOP

32 Inside Documents — INSIDE
32 Structure of Documents
33 The Visual Structure
33 Graphics in Documents
34 Appearance of Text — CHARACTERS
35 Typefaces, Typesize and Fonts
36 Typestyles
36 Proportional vs. Monospaced Fonts
37 Standard Fonts

38 Working with Macintosh — WORKING
38 Direct Manipulation: Controls
38 Buttons
39 Check-Boxes
39 Dials
40 Selecting Information — SELECTING
40 The Selection
43 Selection by Command
43 Automatic Scrolling during Selection
44 Extending a Selection
45 Making a Discontiguous Selection

48 Commands — COMMANDS
48 The Menu Bar
48 Of Mice and Menus
49 Notes on General Properties of Menus
51 The Standard Menus
51 The Apple Menu
52 The Edit Menu
53 The File Menu
53 Keyboard-Invoked Commands
55 What Commands Are and Aren't

56 Basic Editing Paradigms — EDITING
56 The Selection
56 The Scrap
57 The Cut and Copy Commands
58 Paste
58 Undo
58 Inserting and Replacing Text
58 Backspacing
59 Cutting and Pasting Between Documents
59 Between Two Documents with the Same Principal Tool
59 Between Two Documents with Different Principal Tools

61 Special Conditions — BOXES
61 Dialog Boxes
62 The Alert Mechanism
63 Alert Boxes
63 How to Phrase an Alert Message
64 Appearance of Alert Boxes

4 User Interface Guidelines

66	Appendix A.	Thou-Shalt-Nots of a Friendly User Interface	FRIENDLY
67	Appendix B.	Pointer Shapes	POINTERS
68	Appendix C.	Hardware Specifications	HARDWARE
7Ø	Appendix D.	Keyboard Layouts and Character Assignments	LAYOUTS
73	Appendix E.	Guide to Icons	
75	Appendix F.	Unresolved Issues	
76	Technical Lexicon		GLOSSARY
85	Index		INDEX

INTRODUCTION

Macintosh is intended to be the first mass-market personal computer. It is designed to appeal to an audience of non-programmers, including people who have traditionally feared and distrusted computers. To achieve this goal, Macintosh must be friendly. The system must, once and for all, dispel any notion that computers are difficult to use. Two key ingredients combine in making a system easy to use: familiarity and consistency.

Familiarity means that the conceptual underpinnings of a system are based on premises or procedures our users already know and employ. Most Macintosh applications are oriented towards common tasks: writing, graphics and paste-up work, ledger sheet arithmetic, chart and graph preparation, and sorting and filing. The actual environment for performing these tasks already exists in people's offices and homes; we mimic that environment to an extent which makes users comfortable with the system. Extensive use of graphics plays an important part in the creation of a familiar and intuitive environment.

Consistency means a uniform way of approaching tasks across applications. For example, when users learn how to insert text into a document, or how to select a column of figures in one application, they should be able to take that knowledge with them into other applications and build upon it. Uniformity and consistency in the user interface reduces frustration and makes a user more amenable to trying new techniques and new software to solve problems.

Consistency and familiarity are by no means orthogonal concepts. Familiar models should be used in a consistent manner to avoid confusion, and consistency should not lead to unfamiliar behavior.

Software Developers' Responsibility

Preservation of a truly consistent working environment requires some deliberate and conscious effort on the part of applications programmers.

If Macintosh is to be successful as a truly mass-market personal computer, software developers must maintain consistency throughout applications by conforming to a common user interface.

(hand)

> It is the responsibility of everyone who writes software for Macintosh to preserve the integrity of the system.

Years of software development, testing, and research have gone into the definition of the Macintosh user interface. The mechanisms outlined in this document have been shown to be well-suited for a variety of applications and tasks. If your application requires approaches not specified in this document, we urge you to build your schemes on top of existing ones and avoid incompatibility at all costs.

Macintosh's Commitment

On many other computers, since little or no user interface aids are built in, each applications programmer invents a new and original interface for each program--which leads to hundreds of different, conflicting, and confusing interfaces.

We hope to avoid this situation on Macintosh by building tools for a versatile, well-tested user interface and placing them in ROM to be used by all applications programs. There's no strict requirement that an applications program must use all or any of the supplied interface tools; but programmers who create their own interface do so at the expense of their own development time, the user's data space, and the entire system's coherency.

Consistency in the user interface is most important in three areas:

- Data selection and editing;

- Command invocation;

- Performance of common system-wide functions.

These are common to all applications. But each application also has its unique requirements, all of which we cannot forsee. To accommodate each application's specific needs, most of the features of the user interface are extensible: a programmer can "customize" the appearance or function of a common interface feature to suit the application.

Macintosh system software is designed to make the implementation of the user interface as simple as possible for the programmer. Most of the recommended user interface features outlined below are implemented with simple calls to the User Interface ToolBox or the Operating System. The substantial documentation available for those packages should serve as an introduction to implementing the user interface described in this document.

About Modes

> "A good man will prefer that mode, by which he can produce the greatest effect."
>
> — Paley, 1794

We adhere to the principles of modeless behavior. Larry Tesler defines a mode as follows:

> A mode of an interactive computer system is a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input.

Modes are most confusing when you're in the wrong one. Unfortunately, this is the most common case. Being in the wrong mode is confusing because it makes future actions contingent upon past ones; it changes the behavior of familiar objects and commands; and it makes habitual actions cause unexpected results.

We advocate avoidance of modes whenever possible. Of course, exceptions must be made, however; there are certain tradeoffs among modality, usefulness, and implementability that must be considered. There are three cases in which modal behavior is generally tolerated:

- Long-term modes with a procedural basis: doing word processing vs. graphics editing, etc. Each application program in Macintosh is a mode.

- Short-term "spring-loaded" modes, in which the user is constantly doing something to perpetuate the mode. Holding down a button or key is the most common example of this kind of mode.

- Alert modes, where the user must rectify an unusual situation before proceeding. Such situations, however, should have been avoided in the first place.

Other modes are acceptable if they meet the following requirements:

- They emulate a familiar real-life model which is itself modal, like picking up different-sized paintbrushes in a graphics editor; or

- They change only the attributes of something, and not its behavior, like the boldface and underline modes of text entry; or

- They block most other normal operations of the system to emphasize the modality, as in error conditions incurable through software ("There's no diskette in the disk drive", for example).

Whatever the modality entails, it must be visible. There must be a clear visual indication of the current mode, and the indication should be near the object being most affected by the mode.

THE GRAPHIC SCREEN

Macintosh distinguishes itself from all other personal computers by its high-resolution graphic screen. While other computers posess similar or greater graphics resolution or ability, no other applies its graphic powers as widely and generally as Macintosh.

Macintosh has a purely graphic display: there is no "text mode" in the machine at all. Text, to Macintosh, is merely a special kind of graphics. Problems of mixing text with graphics go away because they're really the same thing.

Other computers don't do this because of inherent limitations in their processor speed and data path width, and because of a lack of software support of graphics. Not only does Macintosh have a Motorola MC68ØØØ microprocessor (running at a nominal 7 MHz with a 16-bit data path, giving it several times the bandwidth of the Apple II's 65Ø2), but it also has Bill Atkinson's QUICKDRAW graphics package, revolutionary in its speed and ability.

But far more important than raw graphic power is what the software does with it. What Macintosh does can be explained quite simply:

(hand)

> All commands, features, and parameters of the application, and all the user's data, appear as graphic objects on the screen.

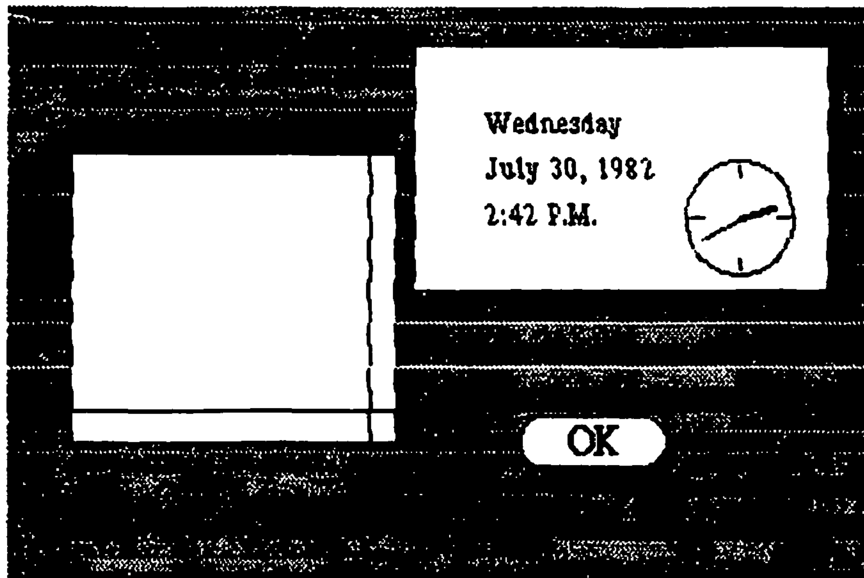


Figure 1. Objects on the Screen

Objects, whenever applicable, resemble the familiar material objects they emulate. Objects that act like pushbuttons "light up" when pressed; objects that act like tab stops look like their counterparts

on a typewriter. Dozens of objects, some emulating everyday objects and some unique to Macintosh, are defined in the User Interface Tool Box.

Objects are designed to look beautiful on the screen. Using the graphic patterns in QuickDraw can give objects a shape and texture beyond simple line graphics. Placing a drop-shadow slightly below and to the right of an object can give it a three-dimensional appearance. The highest aesthetic sensibilities should be used in the design, placement, and animation of objects.

Graphics can distinguish different states of the same object. Many objects on the screen have two states: a "normal" state and a "special" state. Most objects in their normal state are predominantly white, with detail (lettering, symbols, etc.) in black. Inverting the polarity of the object, to make it black with white detail, will highlight the object to represent its special state.

Icons

A fundamental object in Macintosh software is the icon, a small, 32-by-32 square graphic that can be drawn, edited, and moved easily. The Icon Manager has facilities for drawing icons on the screen and setting or resetting bits within them.

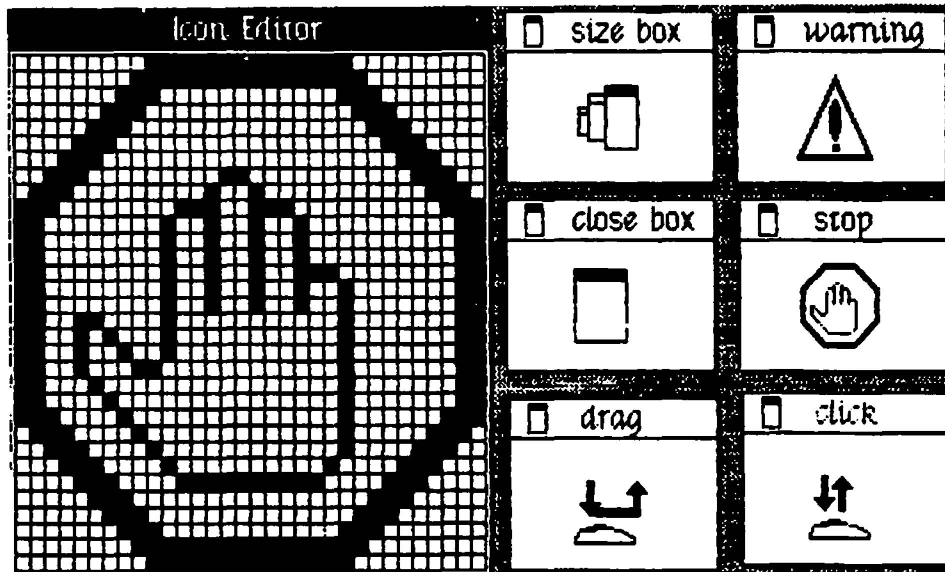


Figure 2. Icons

Icons should be sprinkled liberally over the screen. Wherever an explanation or label is needed, first consider using an icon before using text as the label or explanation. Icons not only contribute to the understandability and attractiveness of the system, they don't need to be translated into foreign languages.

Icons are by no means unique to the software; thy appear on the Macintosh main unit itself, on the shipping materials, unpacking instructions, and in the user manuals. The standard icons used to denote various parts of the Macintosh hardware are shown in the Appendix on icons.

ACCEPTING USER INPUT

All meaningful interaction between a Macintosh and its user takes place via a piece of hardware built in or connected to the main unit. The principal devices for original input to the Macintosh are the mouse and the keyboard; the Macintosh responds to these devices by displaying images on the screen or making sounds with its speaker. No other action of the Macintosh (such as spinning its disk drive, etc.) constitutes a meaningful message to the user.

The Mouse

The mouse is a small device the size of a deck of playing cards, connected to the computer by a long, flexible cable. There is a square button on the top of the mouse. The user holds the mouse and rolls it on a flat, smooth surface.

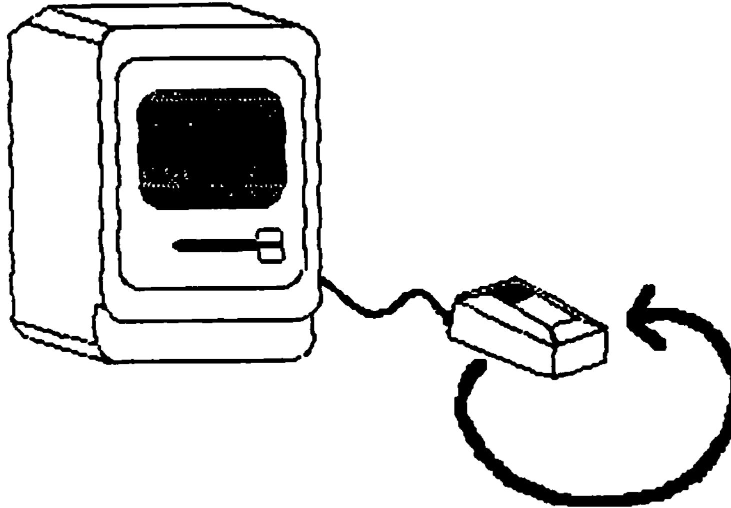


Figure 3. The Mouse and Pointer

A pointer on the screen follows the motion of the mouse. Simply moving the mouse results only in a corresponding movement of the pointer and no other action. Most actions take place when the user positions the focus of the pointer (which should be intuitive, like the point of an arrow or the center of a crosshairs) over an object on the screen and presses the mouse button.

The purpose of the mouse is to allow high-resolution specification of elements on a graphic screen. Many researchers, at Apple and elsewhere, have conducted extensive experimentation with various pointing devices: cursor keys, light pens, graphic tablets, trac balls, etc. We chose the mouse for its ease of use, accuracy, size, and cost. It is compact and lightweight; it resolves to 2ØØ points per inch; it retains its position when not being used; and it requires little muscular strain to position it.

Mouse Actions

The three basic mouse actions are:

- Clicking: Positioning the pointer with the mouse, and briefly pressing and releasing the mouse button without moving the mouse;

- Pressing: Positioning the pointer with the mouse, and pressing and holding the mouse button without moving the mouse; and

- Dragging: Positioning the pointer with the mouse, pressing and holding the mouse button down, moving the mouse to a new position, and releasing the button.

Clicking something with the mouse performs an instantaneous action: selecting a location within the user's document or activating an object.

Pressing an object usually has the same effect as clicking it repeatedly. For example, clicking a scroll arrow causes a document to scroll one line; pressing a scroll arrow causes the document to scroll repeatedly until the mouse button is released.

Dragging can have different effects, depending upon what is under the pointer when the button is pressed. Beginning a drag inside the document frequently results in selection of data. Beginning a drag over an object usually moves that object on the screen. Only certain objects are draggable; large draggable objects have a special area with which the user drags the entire object. Our tests show that users understand dragging an object by a well-marked area rather than by a large, general area.

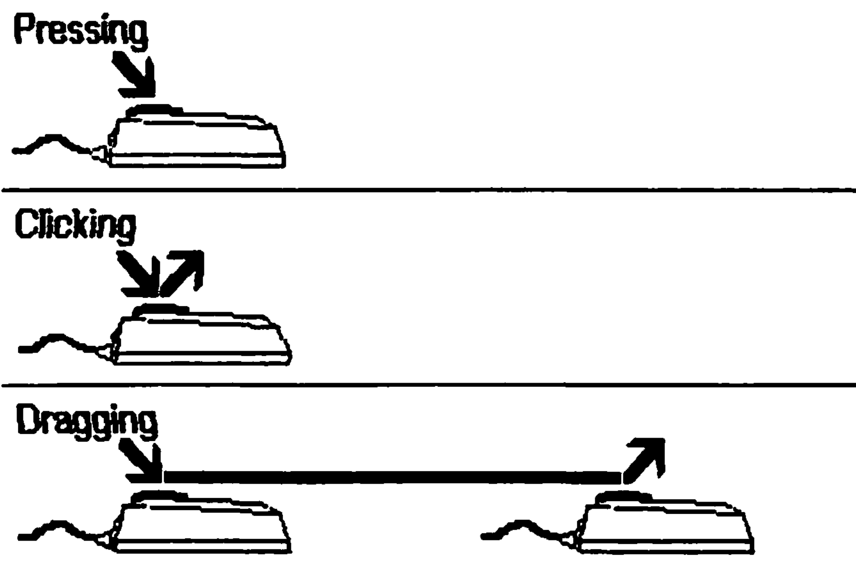


Figure 4. Clicking, Pressing, and Dragging

Dragging is also used to choose an item from a menu, as described below.

(hand)

In general, pushing the mouse button indicates intention, while releasing the button confirms the action.

Dragging an object attaches a flickering outline of the object to the pointer. The outline follows the pointer around the screen while the mouse button is being held down. When the user releases the mouse button, the object moves to the position of the flickering outline, and the outline vanishes.

Every object is restricted to certain boundaries. If the user tries to drag an object out of its natural boundaries, the flickering outline disappears when the pointer travels out of those boundaries. If the user moves the pointer back inside the boundaries with the button still held down, the outline reappears under the pointer and dragging resumes. If, however, the user releases the button while the outline is invisible, the object being dragged does not move; in this way the user can cancel a drag in progress.

Double-Clicking

A variant of clicking involves performing a second click shortly after the end of an initial click. If the downstroke of the second click follows the upstroke of the first by 700 milliseconds or less, the second click should be considered not an independent event, but rather an extension of the first: this action is called "double-clicking". Its most common use is as an optimized means of performing an action that can be performed in another, slower, manner.

(hand)

To allow the software to distinguish efficiently between single clicks and double-clicks on objects that respond to both, a function invoked by double-clicking an object must be an enhancement, superset, or extension of the feature invoked by single-clicking that object.

Changing Pointer Shapes

The pointer may change shape to give feedback on the range of activities that make sense in a particular area of the screen, in a current mode, or both.

1. The results of any mouse action depend on the item under the pointer when the mouse button is pressed. To emphasize the differences among mouse actions, the pointer may assume different appearances in different areas to indicate the mouse's behavior in each area.

2. Although modal behavior is generally discouraged in the Macintosh user interface, sometimes introducing modes makes it simpler to differentiate among the multiplicity of functions of the mouse. For example, in the Graphics Editor, the mouse functions both to draw graphics and to manipulate graphics already drawn. Thus, in this particular application, the mouse is employed in two

different modes. To accent the difference in behavior in these two modes, the pointer may change shape.

The facility to change the pointer appearance to convey modal information is not a unilateral endorsement of modal behavior; see the discussion "About Modes" on page 6 of this document.

The Keyboard

Connected to the Macintosh main unit by a six-foot coil cord is a compact alphanumeric keyboard. The keyboard is used mainly for text and numeric entry.

The keys on the keyboard are arranged in familiar typewriter fashion; there is a utility program with which the user can change the positions of the keys or the characters they generate.

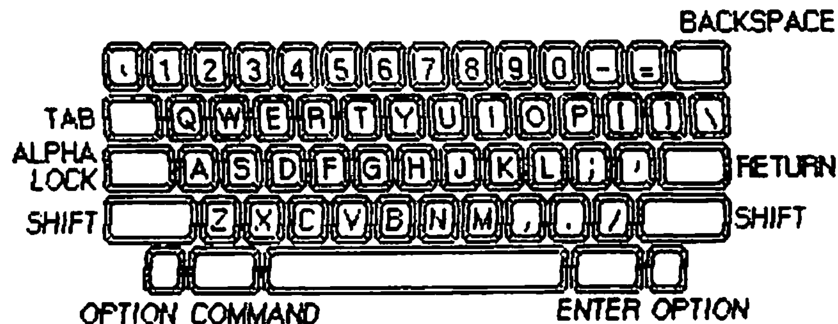


Figure 5. The Macintosh Keyboard

In terms of functionality, the keys are divided into three sets: character keys, modifier keys, and special keys. Character keys enter characters into the computer; modifier keys, in conjunction with character keys, choose among different characters on a key; and special keys give special instructions to the computer.

Character Keys

The alphabetic, numeric, and symbolic keys, and the space bar, enter characters into the computer. Any character key may be associated (and/or labeled) with more than one character; the modifier keys choose among the different characters on each key.

The Basic Editing Paradigms (see that section) define the ways in which characters are typed into a document. All text, whether it be a file name, part of a document, or a search pattern, is typed in and can be edited in exactly the same way.

The keyboard hardware scans the character keys such that it can recognize any two character keys being pressed simultaneously. This feature is called "two-key rollover".

Modifier Keys: SHIFT, CAPS LOCK, OPTION, and COMMAND

Six keys on the keyboard--two labeled SHIFT, two labeled OPTION, one labeled CAPS LOCK, and one labeled COMMAND--change the interpretation of keystrokes or other inputs to the computer. When one of these keys is held down, the behavior of the other keys (and occasionally that of the mouse button) may change. A program can enquire the status of the modifier keys at any time.

The SHIFT and OPTION keys choose among the characters on each character key. SHIFT gives the upper character on two-character keys, or the uppercase letter on alphabetic keys. OPTION gives an alternate character set interpretation, for foreign characters, special symbols, etc. SHIFT and OPTION can be used in combination.

CAPS LOCK latches in the down position when pressed, and releases when pressed again. When down it gives the uppercase letter on alphabetic keys. The operation of CAPS LOCK on alphabetic keys is parallel to that of the SHIFT key, and the CAPS LOCK key has no effect whatsoever on any of the other keys. CAPS LOCK and OPTION can be used in combination on alphabetic keys.

The keyboard hardware can sense any or all of the modifier keys being pressed simultaneously.

The COMMAND key

Pressing a key while holding down the COMMAND key signals that the keypress is not data input, but rather a command invocation (see the section on Commands).

(hand)

> As the OPTION and COMMAND keys are unfamiliar features to users familiar with typewriters, their use should be restricted to expert functions not normally encountered by novice users.

Special keys: ENTER, TAB, RETURN, and BACKSPACE

When the user enters or edits information, the ENTER key confirms that entry. When ENTER is pressed, the computer checks and validates the current entry and allows the user to proceed to a different one. Commonly used to confirm the entry of text, ENTER tells the computer to accept changes made to a field or form (such as a spreadsheet formula or a new file name).

The TAB key is a signal to proceed: it signals movement to the next item in a sequence. TAB often carries the implicit meaning of ENTER before the motion is performed.

The RETURN key is another signal to proceed, but it defines a different type of motion than TAB. A press of the RETURN key signals movement to the leftmost field one step down (just like a carriage return on a typewriter). RETURN also can carry the implicit meaning of ENTER before it performs the movement.

(hand)

> In applications such as the word processor, the TAB and RETURN keys not only perform immediate actions, but store those actions in the text; in such applications the RETURN and TAB keys may be considered character keys.

BACKSPACE is used to delete characters from text, usually in the course of typing that text. The exact use of BACKSPACE is described in the section on the Basic Editing Paradigms.

Typeahead, Auto-Repeat, and Audio Feedback

If the user types at a time when Macintosh is unable to process the keypresses immediately, or the user types more quickly than Macintosh can process, the precocious keystrokes are queued for timely processing. As keystrokes are handled as events through the Operating System's event mechanism, the only limit on the number of characters that can be typed ahead of time is the length of the system's event queue.

Normally, Macintosh "clicks" slightly at every keystroke. This audio feedback in typing is a global preference that the user can change at any time (see the Preferences description, in the section on Desk Accessories).

When the user holds down a key for a certain amount of time, it starts repeating automatically. The delays and the rates of repetition are global preferences that can be changed by the user at any time.

All printable characters, the space bar, the BACKSPACE key, and the RETURN key, inherently have the auto-repeat ability. The auto-repeat ability of each key is a characteristic of the keyboard that the user can change with the same utility program that alters the keyboard layout.

Auto-repeat does not function during typeahead; it only operates when the application is ready to accept keyboard input.

Versions of the Keyboard

There are two physical versions of the keyboard: American and European. The European version has one more key than the American. The key layout on the European version is designed to conform to the ISO standard; the American key layout mimics that of common American office typewriters.

The American keyboard contains 49 character keys (including the space bar and RETURN) that produce all the printable ASCII characters. In

addition, there are the following modifier and special keys: SHIFT, CAPS LOCK, COMMAND, OPTION, ENTER, TAB, and BACKSPACE.

The European keyboard contains 50 character keys; the special and modifier keys are equivalent to those on the American keyboard, but their labels denote their functions symbolically.

(hand)

As the keyboard interface is a general-purpose clocked bidirectional serial port, other devices (such as a music keyboard, etc.) may eventually be attached to this port.

The Numeric Keypad

An optional numeric keypad is offered that connects between the main unit and the standard keyboard. The keypad contains 18 keys that, while labeled similarly to keys on the main keyboard, return different keycodes to the main unit. An application can thus determine the origin of a keystroke. If desired, the keypad keys can be assigned ASCII codes equivalent to their counterparts on the main keyboard.

The character keys on the keypad are labeled with the digits 0 through 9, a decimal point, the four standard arithmetic operators for plus, minus, times, and divide, and a comma. The keypad also contains the special keys ENTER and CLEAR; it has no modifier keys.

The keys on the numeric keypad follow the same rules for typeahead, auto-repeat, and audio feedback as the main keyboard.

Four keys on the numeric keypad are labeled with "field-motion" symbols: small rectangles with arrows exiting them in various directions. Some applications may use these keys to move an object or indicator orthogonally around the screen, and require the user to use the SHIFT key to obtain the four characters (+ * / ,) normally available on those keys.

(hand)

As the numeric keypad is optional equipment, no application shall require it or any keys available on it in order to perform standard functions. Specifically, as the CLEAR key is not available on the main keybaord, a CLEAR function may be implemented with this key only as an optimization of another CLEAR command (such as in a menu).

CONCEPTUAL MODELS

Macintosh, as an appliance computer, has one purpose only: to manipulate information. With it, a user can access, display, interpret, modify, transfer, replicate, and destroy information. Consequently, the central concepts on which the Macintosh system is built deal with things relating to information:

- The container of information, which we call a file;
- The manipulator of information, which we call a tool;
- The presenter and interpreter of information, which we call a window;
- The working environment, which we call the desk top; and
- The information itself, which we call a document.

On the continuum between pure concept and pure object, each of these has its own place. We hope to present our users with only the physical objects that represent these concepts, so that they can grasp the concepts by inference; we will not require them to know the concepts before they encounter any of the objects.

Of the above, files are the most conceptual; we will use the term internally here to mean a generic container of information. As described below, files have many distinct incarnations that the user will encounter.

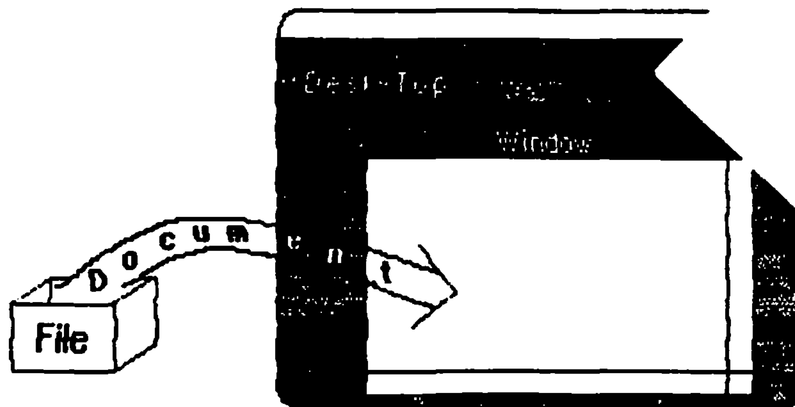


Figure 6. Conceptual Models

The desk top, documents, and windows are the most concrete of the above group: users will see these as objects and not as concepts at all.

Tools are somewhere in the middle: although they have certain distinguishable physical attributes, most of their importance is in the conceptual realm.

Files

A file is a container of information. All the texts, pictures, charts, and address lists that the user puts into Macintosh are stored in files. Files also store information that the user didn't create: information usually more intelligible to the computer than the user.

There are three general classifications for files: those containing documents, those containing tools, and those containing resources. Documents are created by users and can be viewed and modified by users. Tools are created by application programmers; the user can use them but can't modify them. Resources are also created by application programmers, but can be edited by a resource editor to change the way in which a tool communicates with the user (see RESOURCE FILES, below).

Regardless of its contents, a file has many important attributes. Every file has a type that describes its contents and determines which tools can manipulate it; a size that describes how large its contents are; a name by which the user refers to the file; and a label on which the user can put additional information about the file. It also has the dates on which it was created and last modified.

Tools

What we call a "tool" is generally known as an application program: an interactive set of procedures and data structures for manipulating information. Writing, drawing, charting, filing, analysis, and BASIC programming are the fundamental tools Macintosh provides; there are also several other "housekeeping" tools, like using a pocket calculator, note pad, and several other "mini-applications" described later in this document. A tool manifests itself in two ways: it displays a menu bar replete with menus of commands appropriate to that tool; and it places a document window on the desk through which the user can see the information contained in a file.

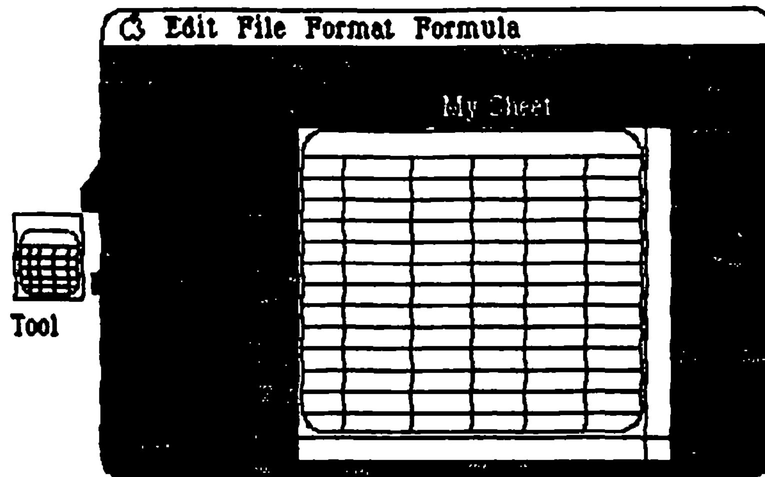


Figure 7. A Typical Tool

Tools, being themselves information (but intelligible to the computer rather than to the user), are stored in files.

(hand)

Only one tool can be in use at any given time.

Documents

Documents are the information that the user has created or wishes to manipulate. Documents can exist inside a file on a diskette or inside the memory of Macintosh. A document comprises a coherent set of different kinds of information.

- Most documents comprise only one kind of information: all text, or one picture, or a series of charts, for example. The user manipulates the information and prints it out as a whole. Every document thus has a principal "type" of information; this type is determined by the tool that formed it.

- A document can comprise more than one kind of information, but it must still form a coherent whole. The user can take information of one kind and add it to a document of another kind. But the document still retains its principal type, and the user can manipulate only the information of that type.

Associated with each kind of document is a principal tool: the tool most appropriate to manipulate that document. The principal tool of any document is usually the tool that created it. Other tools may be able to read and interpret the document; for example, the BASIC language can read word processor documents anticipating the text of a program. Such tools are the document's secondary tools. The distinction is important only when selecting files from the desk.

Resources

Some files contain information that is neither a tool nor the user's information. This information is usually fonts, system programs, configuration information, etc. Although such information may have principal tools (such as a font editor for fonts), it's most commonly used by a tool.

Files containing such information are called resource files. Tools have internal links to the resource files they need; copying a tool file, for example, automatically copies all resource files linked to it. Resource files are usually created by application programmers to accompany tools. The user can edit some resource files by using special resource editors, such as font editors or menu editors.

THE DESK TOP MODEL OF ORGANIZATION

The entire Macintosh working environment is based on familiar and intuitive concepts. The Macintosh screen represents a working surface or a desk top. Papers, writing or drawing utensils, and other common desk accessories have their place on this desk top just as on any other. Whenever possible, the objects on the desk top resemble their real-life counterparts: for example, all papers are white with black lettering.

Figure 8. The Desk Top

The Desk

The desk top metaphor is reinforced by the central tool of the Macintosh system, a tool called the Desk. While most tools manipulate the documents contained in files, the Desk manipulates the files themselves, often regardless of their contents. The basic functions of the Desk are as follows:

- Get, Print, Examine, Delete, or Copy any file or group of files;
- Initialize a diskette;
- Rename or rearrange the files on a diskette;
- Select which diskettes, network diskettes, and peripheral devices to work with.

On the Desk, files are represented by icons, with each file's name as a caption to its icon. The icons can be dragged around the desk asnd positioned in any order or arrangement. Other parts of the system are also represented by icons on the desk: disks and disk drives, printers, etc.

The central purpose of the Desk is to allow the user to manipulate files, and to call up the appropriate tools to work on the documents the files contain. The user invokes a tool from the Desk, and returns to the Desk when finished.

Once using a tool, the user can call up a subset of the standard Desk functions, to choose a new file to work with or to specify a destination for the new work. This subset as presently defined includes selecting disks and files, creating a new file, and renaming and repositioning files.

WINDOWS

Windows are objects on the desk that display information. The information can be a user's document, an error message, or a request for more information. Any number of windows can be present on the desk at any time. As on a real desk, if more windows are placed on the desk than reasonably can fit, the windows "overlap" each other: the windows in front partially or completely obscure those behind them.

Figure 9. Windows

Each window "floats" in its own plane. Think of a number of plates of glass stacked on top of the desk: each plate contains one and only one window, and the plates can be moved to make the windows appear in different places on the desk. Each window can overlap those behind it, and can be overlapped by those in front of it. The frontmost window cannot be overlapped. Even when windows do not overlap, they retain their front-to-back ordering.

Opening and Closing Windows

Windows come up onto the screen in different ways appropriate to the purpose of the window. Some windows are created automatically: for example, when the user wants to work with a document, the tool being used creates a document window in which to present that document.

Many windows have an icon that, when double-clicked, makes the window go away: this icon is called the close box. (This icon is double-clicked, rather than singly-clicked, because of the disturbing consequences of accidentally clicking the icon). The application in control of the window determines what is done with the window when the close box is double-clicked: it can

1. make the window invisible, to be retrieved later; or

2. remove and destroy the window and any information it contained.

If an application wishes not to support closing its window with a close boxe, it should not place the box on the window.

The Active Window

At any given time, one window is of greater importance to the user than any other. Usually, the most important window is presenting the current document; at other times, an error message or information request may be more important. Thus this general rule:

- The most important window at any given time is always frontmost.

Naturally, there must be rules to determine which window is most important at any given time.

- Newly-created windows are usually brought to the front.

- If the user positions the pointer with the mouse inside any window that is not in front, and then clicks the mouse button, that window is brought to the front.

Being in front has more consequences for an window than merely being more visible. The frontmost window is said to be active, and all others inactive.

- A window's active state is visibly distinct from its inactive state; usually, the title or header of the window is highlighted.

- Clicking or dragging inside the active window may perform a useful function; clicking or dragging inside an inactive window merely brings that window to the front.

- All command and data input is handled by the program that is in control of the active window.

Document Windows

Although windows display many kinds of information and requests, the most common appearance of a window is to display the document currently being worked on. Windows displaying documents have parts not usually seen in other windows: scroll bars to move the document under the window; a size box to change the size of the window; and split bars to divide the window into several panels.

Scroll Bars

Scroll bars are used to change the user's view of a document. Only the active window has scroll bars; inactive windows leave black-bordered empty rectangles where their scroll bars will appear when the window is activated.

A scroll bar is a light gray shaft, capped on each end with square boxes labeled with arrows; inside the shaft is a white rectangle. The shaft represents one dimension of the entire document; the white rectangle (called the thumb) represents the portion of the document currently visible inside the window. As the user moves the document under the window, the position of the rectangle in the shaft moves correspondingly.

There are three ways to move the document under the window: by sequential scrolling, by "paging" screenful by screenful through the document, and by directly positioning the thumb.

Clicking a scroll arrow moves the document in the direction of the scroll arrow. For example, when the user clicks the top scroll arrow, the document moves down, bringing the view closer to the top of the document. The thumb moves towards the arrow being clicked.

Each click in a scroll arrow causes movement a distance of one unit in the chosen direction, with the unit of distance being appropriate to the tool: one line for the word processor, one row or column for the spreadsheet, etc. Pressing the scroll arrow causes continuous movement in its direction.

Clicking the mouse anywhere in the gray area of the shaft advances the document by screenfuls. The thumb moves toward the place where the user clicked, and the document moves in the opposite direction; clicking below the thumb, for example, brings the user the next screenful towards the bottom of the document. Pressing in the gray area keeps screenfuls flipping by until the user releases the button or the thumb reaches the pointer.

In both the above schemes the user moves the document incrementally until it is in the proper position under the window; as the document moves, the thumb moves accordingly. The user can also move the document directly to any position simply by moving the thumb to the corresponding position in the shaft. To move the thumb, the user presses on the thumb and drags it along the shaft; a flickering outline of the thumb follows the pointer. When the mouse button is released, the thumb jumps to the position last held by the flickering outline, and the document jumps to the position corresponding to the new position of the thumb.

If the user starts dragging the thumb, and then moves the pointer a certain distance outside the scroll bar, the thumb detaches itself from the pointer and stops following it; if the user releases the mouse button, the thumb returns to its original position and the document remains unmoved. But if the user still holds the mouse button and drags the pointer back into the shaft, the thumb reattaches itself to the pointer and can be dragged as usual.

Multiple Windows

Some tools may be able to keep several windows on the desk at the same time, as part of the same logical document. Different windows can represent:

- Different parts of the same document, such as the beginning and end of a long term paper;
- Different interpretations of the same document, such as the tabular and chart forms of a set of numerical data;
- Different parts of a logical whole, like the listing, execution, and debugging of a BASIC program;
- Separate documents being viewed and/or edited simultaneously.

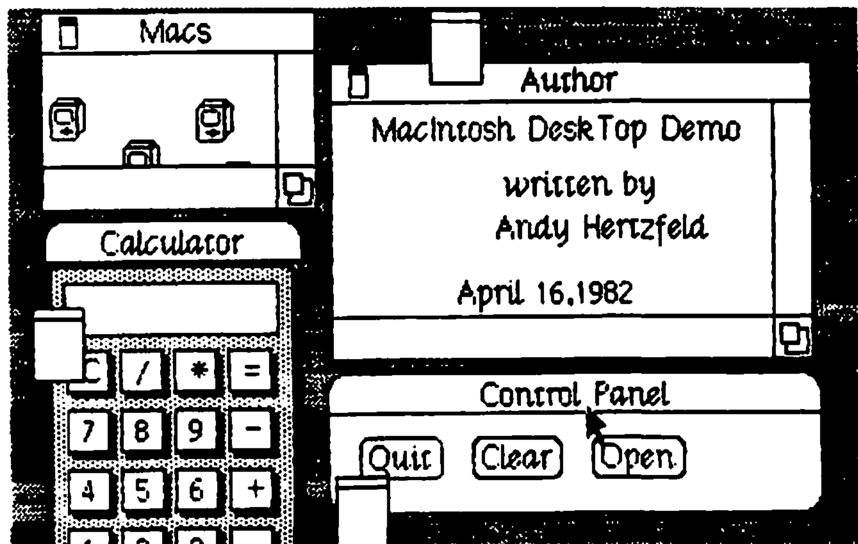


Figure 10. Multiple Windows

Each tool may deal with the meaning and creation of multiple windows in its own way.

There are occasionally better ways to perform the above functions than with multiple windows. Showing different parts of the same document can be done better by splitting the window (see below). Different interpretations of the same document occasionally merit two panes in the same window, rather than two separate windows. The implementation decision can best be made by experimentation and testing on actual users.

Moving a Window

Each tool places windows on the screen wherever it wants them. The user can move a window--to make more room on the desk or to uncover a window it's overlapping--by dragging its title bar. A flickering outline of the window follows the pointer until the user releases the

mouse button. At the release of the button the full window is drawn in its new location.

A window always moves in its own plane; while it's being dragged around, the flickering outline is visible over the windows below it but is hidden under the windows above. Notice that clicking in the title area does not make a window active or bring it to the top.

(hand)
Moving a window does not affect what portion of the document it is displaying.

A window can never be moved off the screen; specifically, it can't be moved such that the visible area of the title bar is less than four pixels square.

Moving a window is fully supported by the Window Manager, and is easily performed with one procedure call; an application program need not care where on the screen its window is placed.

Changing the Size of a Window

If a window has a certain icon in its lower right corner, where the scroll bars come together, the user can change the size of the window--enlarging or reducing it to the desired size. The box that contains the icon is called the size box.

Dragging the size box drags a flickering outline of the window. The outline's top left corner stays fixed, while the bottom right corner follows the pointer. When the mouse button is released, the entire window is redrawn in the size and form of the flickering outline.

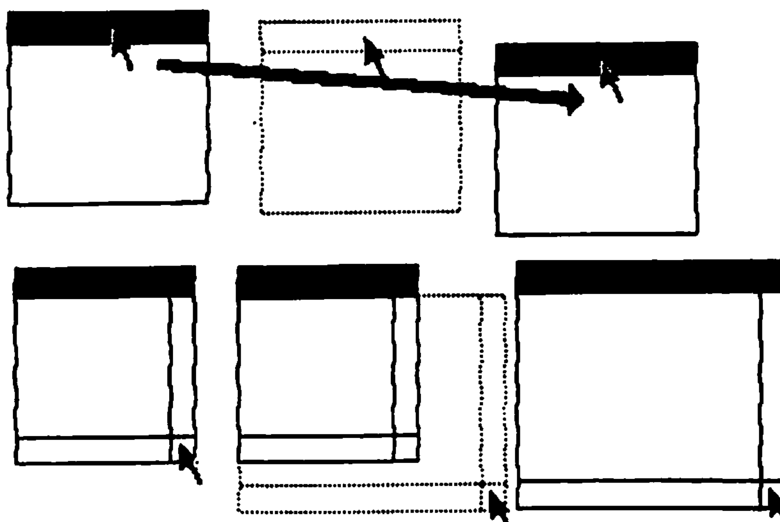


Figure 11. Moving and Sizing a Window

Sometimes it's not appropriate to size a window; some tools may not support this ability. In this case, the size box is empty and clicking

in it produces no effect. If a tool does support sizing a window, however, changing the window's size leaves the document's size unchanged; the window simply displays a larger or smaller portion of the document.

(hand)

> Sizing a window does not affect its contents, or change the position of the top left corner of the window over the document; only the portion of the view that is visible inside the window.

At its maximum size, a window is still small enough that a seven pixel square area of the size box is visible on the screen.

The minimum size window consists of only a title bar the width of the title itself, a horizontal scroll bar (or a blank rectangle of equivalent size), and the size box. If a window is made so small that its title will no longer fit in the title bar, the title is truncated to show as many of its initial characters as possible.

Sizing a window is fully supported by the Window Manager, and is easily performed with one procedure call; an application program need not care about the size of a window.

Splitting a Window

Sometimes it is desirable to be able to see disjoint parts of a document simultaneously. Tools that accommodate such a capability allow the window to be split into independently scrollable panels.

Tools that support split panes place split bars at the top of the vertical scroll bar and at the left of the horizontal one, if present. Pressing a split bar attaches it to the pointer. Dragging the split bar positions it anywhere along the nearby scroll bar; releasing the mouse button drops the split bar at its current position, splits the window at that location, and creates new scroll bars for each panel.

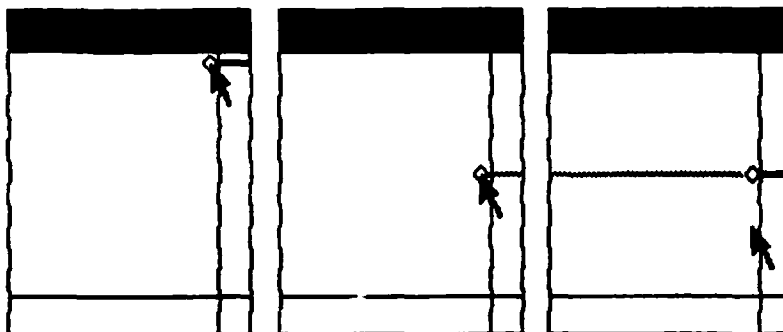


Figure 12. Split Views

The document appears the same, save for the split bar lying across it. But there are now separate scroll bars for each pane; whith these, the user can scroll each pane independently of the other.

Dragging a split bar back to its original position reunites the window in that direction; the left or top view (and its scroll bar) disappears, leaving the right or bottom view.

The number of views in a document does not alter the number of selections per document: i.e., one. The active selection appears highlighted in all views that present it.

Desk Accessories

Macintosh does not allow two tools to be running at once. However, there are several mini-applications that are available while using any tool.

At any time the user can issue a command to call up one of several desk accessories. The basic ones provided include:

Calculator
Alarm Clock
Note Pad
Telegram Form and In-Box (AppleGram)

Accessories are disk-based: only those accessories available on-line can be used. The list of accessories is expanded or reduced according to what's available at any given time. The application can support all accessories in the system with calls to the Desk Manager. On disk, accessories are stored in resource files. More than one accessory can be on the desk at any given time.

Who's on Top?

With a virtual three-dimensional screen it is essential to manage the third dimension so that important items or objects requiring immediate attention are not obscured accidentally. Hence, in order from front to back:

- The pointer
- An alert box
- A dialog box
- The menu bar and all pull-down menus
- The active window
- All other windows
- The desk top

INSIDE DOCUMENTS

(hand)

We strongly subscribe to the doctrine of preservation of visual fidelity, i.e., what you see is what you get.

It's important that a document as seen through a window on the desk closely resemble the same document when committed to paper. The differences (and there will be differences) must be natural and unsurprising. Naturally, the ruler and graph paper used to create a report on Tuesday morning won't be distributed with that report when it's presented that afternoon; printing a document shall not carry the vestiges of the tool that created that document.

Any given tool should be able to manipulate, in some way, everything in the document it presents. Macintosh eventually will have many different tools, and we do not pretend to foresee the needs of all. However, we do provide standard means of manipulating the constituent elements of most documents.

Structure of Documents

In order to discuss the appearance of information inside documents, it is necessary first to digress a bit into the structure of documents.

A document is a collection of information. Each piece of information has its own position in the document, and its own positional relationship to the information around it.

In terms of structure, there are three principal types of documents: texts, free-form documents, and structured documents.

1. Texts consist of a string of information (in this case, characters) that appears two-dimensional but is really linearly ordered. More characters can be inserted anywhere within the text or added onto the end of the text. There is an inherent order to the characters in a text, and definite positions between characters.

2. Free-form documents start completely empty and unstructured, like a blank piece of paper. Information can be placed anywhere within the document; each piece of information has its own position. There may be large, empty spaces in the document that contain no information. There is no inherent ordering among the information in a free-form document. Pictures drawn in the graphics editor are free-form documents.

3. Structured documents have predefined cells to contain information. There is a fixed maximum number of cells per document; no cells can be added, nor can they be removed. Cells are usually arranged in rows and columns; a given cell is a member of one row and one column. There is a definite position between two adjacent cells,

and a position at the corner of a group of four cells. A spreadsheet is a structured document.

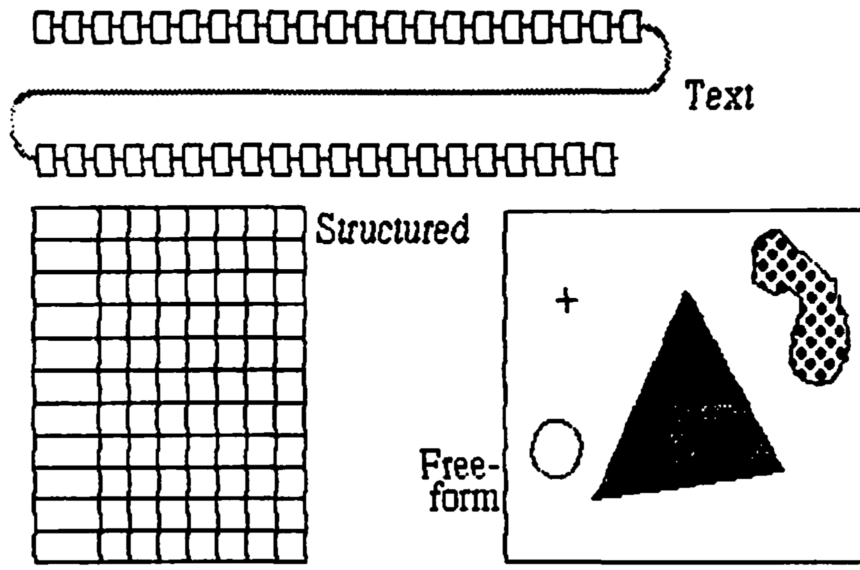


Figure 13. Types of Documents

The type of a document affects many things--mostly how a user selects information inside the document. For example, information in a text can be selected character-by-character, but information in a structured document is selected cell by cell. The exact details of the selection process are described in the section "Selecting Information".

The Visual Structure

The structure can manifest itself visibly inside the document. For example, the rows-and-columns arrangement of a spreadsheet can be clarified by adding graphic grid lines between the cells. These lines are not part of the user's data, but they are part of the document. Such supporting graphics are usually static elements within the document, and cannot be moved or altered. Those that can be altered usually affect only the presentation of the user's data, not the data itself.

At the tool's discretion, the supporting graphics in a document may or may not appear when the document is printed. The grid lines on a spreadsheet might very well appear, while the rulers in a word processor document will probably not be printed.

Graphics in Documents

Not only does Macintosh use graphics to show the structure of a document and to otherwise communicate with its user, it also supports tools to create and manipulate graphic documents. Two such tools are planned: a graphics editor (to design and draw pictures, diagrams, illustrations, signs, etc.), and a charts and graphs package (to do bar

charts, pie charts, hi-lo graphs, etc. from a numerical data base).

Graphic documents are usually free-form: each graphic item in the document has its own position within the document, and there is no inherent relationship among the items (although the tool can define such a relationship). But there's no reason that graphic documents can't be structured. For example, a graphic programming language mught have a text-like or other structure.

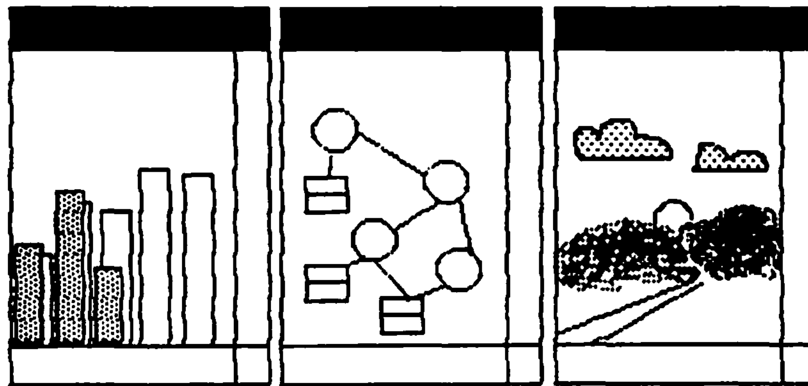


Figure 14. Graphic Documents

Graphics inside documents are produced using the QuickDraw graphics package. The package can draw seven fundamental graphic forms--lines, rectangles, ovals, rounded-corner rectangles, wedges, polygons, and arbitrary regions--either in outline or filled with a solid pattern. It can also place and manipulate images defined bit-by-bit. A tool can give the user the ability to draw anything from simple line drawings to finely textured halftone pictures.

The tool must itself determine how to respond to the mouse and keyboard in creating and manipulating graphics.

Appearance of Text

Most people, even bibliophobes, are confronted with a wide variety of printed matter on daily basis. Our eyes are so accustomed to seeing a myriad of typestyles, typesizes and typefaces used in publications to embellish or emphasize the content, that we no longer take special note. Developing eye-catching and pleasing typefaces has been an art unto itself since Gutenberg. Appropriate and aesthetically embued typesetting has been traditionally the domain of tooled craftsmen. By contrast, the repertoire of currently available computer 'typefaces' is thoroughly devoid of aesthetic nuances and provides but a bleak parody of the printed world.

Macintosh documents can contain characters in a number of different typefaces, typestyles, and typesizes. Type can abut closely or appear loosely packed; parts of some characters (such as the curl of a y) can reach back under or up over adjacent characters; and text can freely intermix with graphics. After all, text is just a specialized form of graphics.

Note that in this context, numbers are considered text: to users, the external appearance of digits is the same as that of other text characters. The following discussion thus pertains to numerical information as well as natural-language text.

- For more information on the aesthetics of type design, see a good typography book; David Gates' Type is recommended. For implementation details on how to place characters on the screen, see the Macintosh User Interface ToolBox manual QuickDraw: A Programmer's Guide.

Typefaces, Typesize, and Fonts

A typeface is a set of typographical characters composed with a coherent "feel" and consistent design. Things that relate characters in a typeface include the thickness of vertical and horizontal lines, the degree and position of curves and swirls, the use of serifs, etc. Typefaces have names, usually historical: Bodoni, Goudy, Tile, etc. The identity of a typeface is independent of its size or any particular typestyle it may conform to (see below).

Typesize in the printing world is measured in points, a point being reasonably close to 1/72 inch. The resolution (in points per inch) of the Macintosh screen is quite close to this, but not close enough to keep accurately to printers' measurements. But we do describe typesize loosely in "points", which have no correlation to the mathematical entity of a point in the QuickDraw graphics package, or to anything else for that matter. In talking about type, we use points as a rough indication of vertical size.

A font is the entire set of characters of a specific typeface and typesize. For example, Helvetica8 refers to a font that contains characters of the typeface named Helvetica at a size of 8 points. In addition to all the uppercase and lowercase letters, numerals and punctuation marks, a font may include mathematical symbols, accented letters or other special characters.

Times Roman 10, **Bold**, *Italic*, Underlined, Outline, Shadow,
Times Roman 14, ***Bold Italic***
Helvetica 10, Bold Outline
Helvetica 14, *Italic Shadow*
Type 10v, Outline Underlined
Cacha 12, ***Bold Italic Underlined Shadow***
Old English 18
Bocklin 36v

Figure 15. Type

Typestyles

Macintosh does not require the use of separate fonts to accommodate different styles of the same typeface. A character of any font may be subjected to a group of transformations that modify its general appearance: such a modification is called a typestyle. There are five fundamental typestyles: bold characters, italic (slanted) characters, outlined characters, underlined characters, and shadowed characters. Any combination of these typestyles can be used, but Macintosh cannot be held accountable for any aesthetic atrocities that may be perpetrated by an insensitive user.

Proportionally Spaced vs. Monospace Fonts

Most printing fonts are proportionally spaced (also known as variable pitch). This means that, for example, the "i" is narrower than an "m"; the "W" wider than the "J".

In a monospace (fixed pitch) font, all characters are of the same width. Monospace fonts are generally less attractive than proportionally spaced fonts. Monospace fonts are sometimes called "typewriter" fonts.

Monospace fonts are appropriate for some applications, such as COBOL coding forms, but generally discouraged in Macintosh. As monospaced fonts are merely a degenerate case of proportional fonts, they can be used just as easily as proportional fonts, when they are needed. It's necessary, for example, for proportional fonts to have monospaced numerals, so that columns of numbers line up neatly when aligned at decimal tab stops.

Standard Fonts

Macintosh uses a distinct system font when presenting its labels, messages, and lists to the user. System-provided text in this font cannot be edited. The Macintosh system font is Cream10; users and tools may not use this font.

There is always a standard font in which all information the user has entered will appear: the user font is Helvetica10, a nice, sans serif, reasonably compact face.

The use of any other fonts depends on the particular tool being used. The word processor, in all probability, will allow the user more multiple font ability than most other tools.

WORKING WITH MACINTOSH

So far, this document has described many things about the Macintosh user interface: how it accepts inpout from the user, how it displays information on its screen, and how the conceptual underpinnings of the system control the structure of interactions. But nothing has been said about how these things work together.

This section describes how input affects output: how Macintosh works. It discusses the methods the user will use to perform actions, select information, and choose commands to operate on that information.

Direct Manipulation: Controls

> "Piaget has hypothesized that infants first learn about causation by realizing that they can directly manipulate objects around them--pull off their blankets, throw their bottles, drop toys... Such direct manipulations, even on the part of infants, involve certain shared features that characterize the notion of direct causation that is so integral a part of our constant everyday functioning in our environment--as when we flip light switches, button our shirts, open doors, etc."
>
> -- Lakoff & Johnson, 1980

Friendly systems act on direct causation--they do what they're told. Performing actions on a system in an indirect fashion (by typing words and pressing RETURN, or by obediently choosing one item from the currently displayed list) reduces the sense of direct manipulation that is basic to the feeling of causation. To give Macintosh users the feeling that they are in control of their machines, many of a tool's features are implemented with controls: graphic objects that, when directly manipulated by the mouse, cause instant action with graphic results.

Three kinds of controls are supported by the Control Manager in the User Interface ToolBox: buttons, check-boxes, and dials.

Buttons

Buttons are small objects, usually inside a window (but occasionally on the desk top), labeled with words or an icon. Clicking or pressing a button performs the instantaneous or continuous action described by the button's label.

Buttons usually perform instantaneous actions, like opening or closing windows, or acknowledging error messages. Occasionally, they can also perform continuous action: the scroll arrows on a scroll bar are continuous-action buttons.

The Control Manager defines one kind of button, an instantaneous or continuous pushbutton, labeled with a verbal title. A tool may include a procedure to define a custom button, which can be linked in to the Control Manager and used just like the standard button.

Check-Boxes

Check-boxes are a variant of buttons. Where buttons perform instantaneous or continuous actions, check-boxes display a state that the user can change. Most commonly seen when filling out a form or setting parameters, check-boxes are small squares that appear either empty or filled in with a check-mark. The boxes are usually adjacent to a word or icon that describes the meaning of the box.

Clicking in a check-box flips its state, from checked to unchecked or vice-versa. Dragging through a field of check-boxes flips the state of the first and assigns the new state to all other boxes dragged through.

A check-box may belong to a group of boxes. If there are no interrelationships among the boxes, they are checked and unchecked as above. But if the boxes are related such that one and only one must be checked at any given time, they work like "radio buttons": clicking in an unmarked box marks that box and unmarks the previously marked box. Such groups should be labeled clearly, "Choose one of the following:". The checked appearance of this kind of box is visually distinct from normal, ungrouped check-boxes.

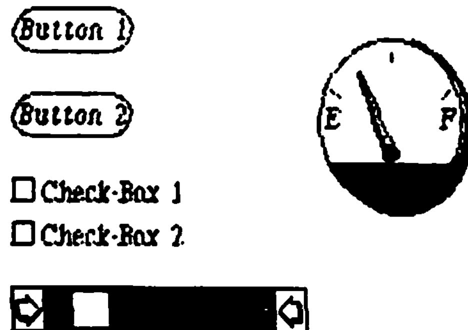


Figure 16. Buttons, Check-Boxes, and Dials

Dials

Dials display the value, magnitude, or position of something in the tool or system, and optionally allow the user to alter that value. Dials are predominantly analog devices, displaying their values graphically and allowing the user to change the value by dragging an indicator; dials may also have a digital display.

The best example of a dial is the shaft of a scroll bar. The indicator of the scroll bar is the thumb; it represents the position of the window over the length of the document. The user can drag the thumb to change that position.

Just as with buttons, there are a few standard dials defined in the ROM, but a programmer can implement a custom dial and link it in with the control mechanism.

Selecting Information

A previous section mentioned that Macintosh has one purpose only: to manipulate information. If this is true, then there is a simple operational paradigm to cover all situations:

(hand)
First select some information, then manipulate it.

This paradigm minimizes modality in basic operations. By selecting the information first, the user is free to select different information without being committed to a certain manipulation.

The following sections describe the two parts of this basic paradigm: how to select information in a document, and how to choose commands to manipulate that information.

The Selection

The selection is the collection of information that will be acted upon by the next command. There is always one and only one active selection in the active window. The selection can be so large as to enclose all the information in the document, or it can be so small as to merely indicate the position between two pieces of information, enclosing nothing at all; the latter selection is called an insertion point. The insertion point indicates the position at which newly entered information will be placed.

Positioning the pointer over the user's information in the active document and pressing the mouse button usually begins a selection. Once the button is pressed, the selection can be completed in two ways:

1. Clicking selects one piece of information or a position between pieces of information.

2. Dragging selects a group of information.

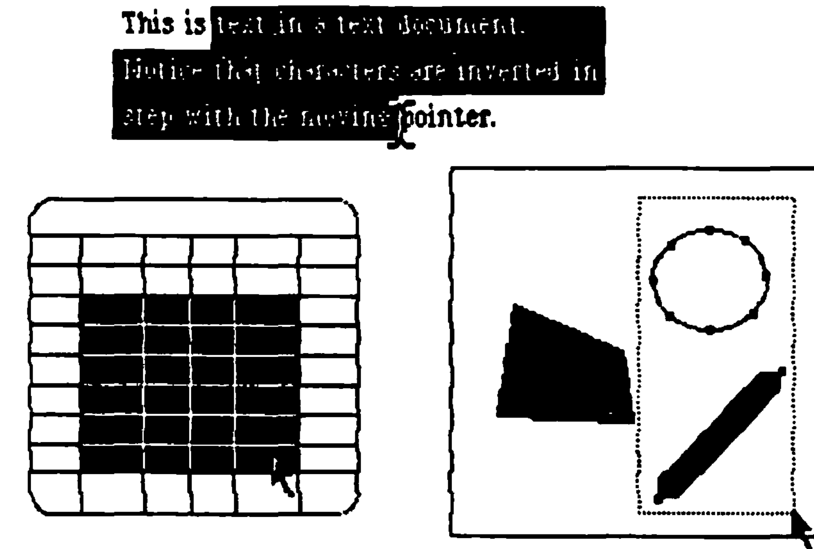


Figure 17. Selecting Information

The exact behavior of clicking and dragging to make the selection depends on the structure of the document.

- Clicking in text selects the position between the two characters nearest the pointer; this position becomes the insertion point. The insertion point in text is represented by a blinking vertical bar.

  Clicking in a structured document selects either the cell under the pointer, the position between two adjacent cells, or the corner of four cells. The latter two selections are insertion points, and are represented by blinking vertical or horizontal bars, or by a blinking cross.

  Clicking in a free-form document selects the item under the pointer. If the pointer is not over a piece of the user's information, clicking either does nothing, or selects a position in the document. This position, the insertion point, is marked by an "anchor" icon.

This is text in a text document.
Notice that characters are inverted in
step with the moving pointer.

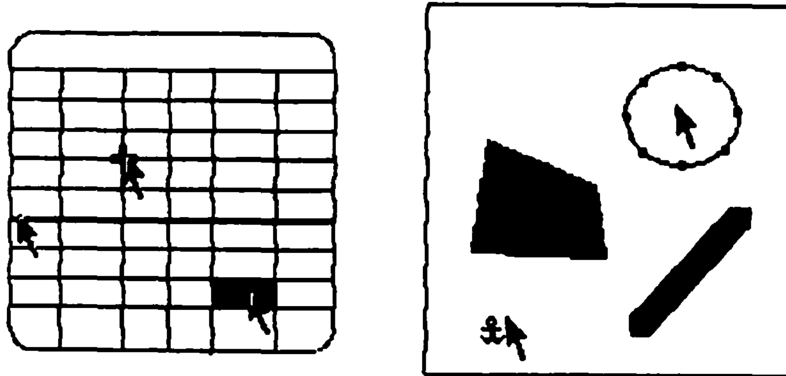


Figure 18. Selection by Clicking

Clicking in editable user information always creates a new selection; the information selected is highlighted and the previous selection is unhighlighted. Highlighted text appears white-on-black; highlighted graphics appear with "knobs".

Dragging through editable user information selects a group of information. It would seem that dragging should select all items dragged over--to select items, press the mouse button, drag across the items, then release--but experience proves that selecting only those items that were dragged over is inefficient. Instead, consider dragging as defining two points: the point where the button was pressed and the point where it was released. Dragging then selects everything between those two points, according to the structure of the document, regardless of the path of the mouse. The objects under the two points are included in the selection, as are all items between those two points.

- Dragging through text selects all characters, in textual order, from the character under the first point to the character under the last point.

  Dragging through a structured document selects all cells in the rectangle whose corners are the cell under the first point and the cell under the last point.

  Dragging through a free-form document selects all items completely enclosed by the rectangle whose corners are the first and last points.

During the dragging, the selection is visible--the items that will be selected are highlighted, in real time, according to the current

position of the pointer. But the selection is not actually confirmed until the mouse button is released. If the user moves the pointer back to the first point and releases the mouse button, the result is the same as a click at that position (see above)

The items between the two points are selected regardless of the relative orientation of the two points. Starting at the end of a sentence and dragging backwards to the beginning operates just as well as starting at the beginning and dragging to the end.

Once the selection is made, the selected items are highlighted and the items in the previous selection are unhighlighted. There is no mechanism for restoring the previous selection.

(hand)

After a selection is made, the pointer becomes invisible so as not to obscure the selection. The pointer reappears the next time the user moves the mouse.

Selection by Command

Some logical groupings of information are more commonly selected than others--columns or rows in a spreadsheet, paragraphs in a word processor, etc. And occasionally it's convenient for the tool to select a piece of information automatically--such as a word or phrase that the user is searching for.

In these cases, the invocation of a command may explicitly or implicitly make a new selection. For example, a tool may have a "Select All" command to select all information in the document; a spelling checker could have a "Select Next Misspelled Word" command, etc.

When any such command is invoked, the tool must scroll the document automatically in order to present as much as possible of the new selection.

Automatic Scrolling During Selection

The only limit on the size of the selection is the size of the document itself; the largest possible selection is the entire document.

But the normal method of selecting as outlined above can't handle selections that extend outside the window. We therefore define a way to scroll the contents of the window during selection:

- If during selection the user drags beyond the borders of the window, the contents of the window will scroll (automatically and continuously) away from that border. New information scrolled into the window becomes selected and is highlighted accordingly. Scrolling stops when the user either releases the mouse button or moves the pointer back into the window: the latter case resumes normal selection.

"Window" in the above paragraph applies to a single panel of a split window; beginning a selection in a panel and moving out of that panel scrolls only that panel.

Extending the Selection

Selection by dragging and automatic scrolling is fine for relatively small selections, but its usefulness deteriorates as the desired selection grows larger. An alternate method can be used to make a large selection: this process is called extending the selection. A selection made in this way is treated the same as any other selection.

Extending the selection merely adds to the current selection. Whereas making a normal selection removes the previous selection, making an extended selection enlarges the previous selection to extend to the newly selected position.

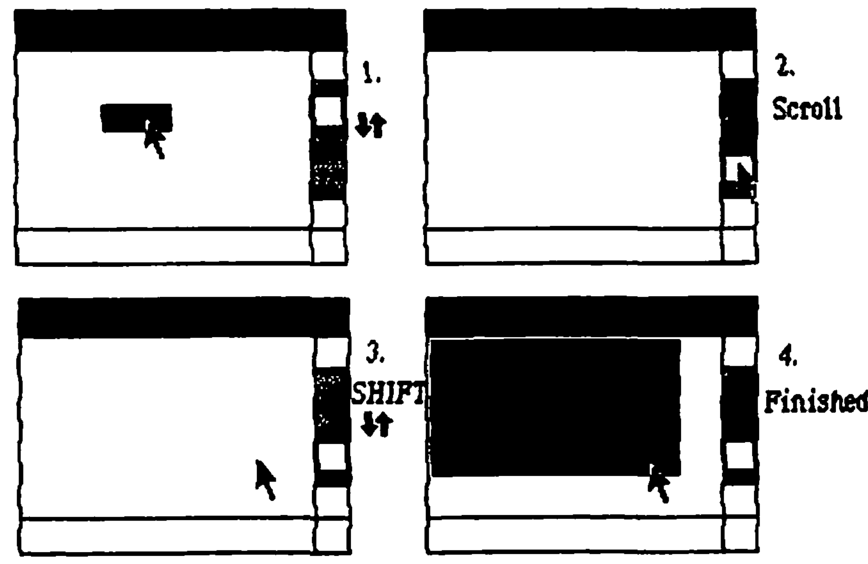


Figure 19. Extending a Selection

An extended selection is made by positioning the pointer, holding down either of the SHIFT keys on the keyboard, then pressing the mouse button. When the mouse button is pressed, all information between the original selection and the current pointer position (inclusive) becomes selected and highlighted. The user can then drag the mouse around and complete the selection as usual. The SHIFT key may be released at any time without affecting the selection.

Extended selections can be made across two panels of a split window.

Making a Discontiguous Selection

Some tools may choose to allow selections that are discontiguous: that comprise one or more unconnected pieces, that have "holes", or both. How a tool deals with operations on such selections is up to its designers; the following is merely an outline of how such selections are made.

(hand)

Discontiguous selection of text is not supported. It causes ambiguity upon insertion.

Making a discontiguous selection is like making an extended selection in that it merely augments the current selection, and also that it is invoked by holding down a keyboard key while pressing the mouse button.

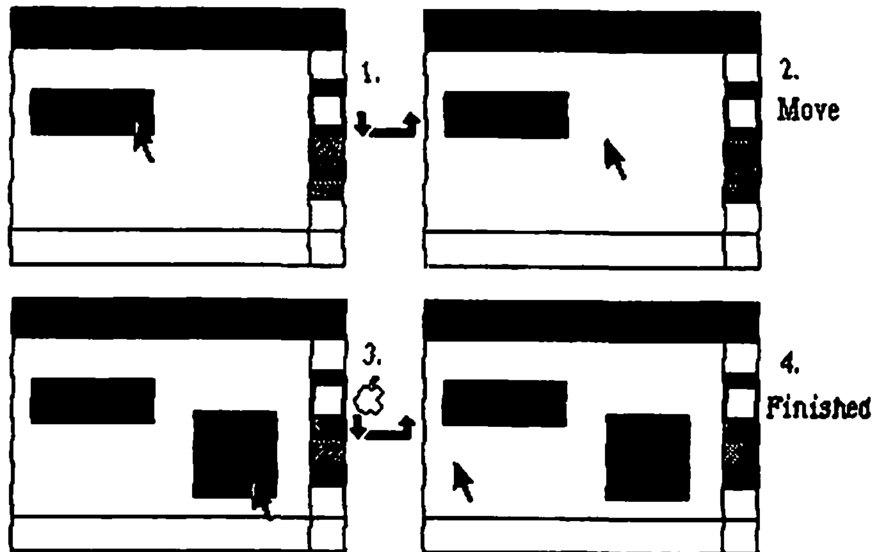


Figure 2Ø. Making a Discontiguous Selection

A discontiguous selection is made by positioning the pointer, holding down the COMMAND key, and pressing the mouse button. It continues like a normal selection: the user drags the mouse to indicate the last point, then releases the mouse button. The COMMAND key may be released at any time without affecting the selection. But the kind of selection that's being made depends upon the posiition of the pointer when the mouse button is pressed:

- If the pointer is not inside the previous selection, the operation is a normal selection that does not remove the previous selection. Both selected areas are highlighted on the screen; they are both considered parts of the selection.

- If the pointer is inside the previous selection, the operation becomes a deselection: the information "selected" becomes deselected and unhighlighted. The remaining information, even if

it contains a hole, is the selection.

With this paradigm, any arbitrary collection of items in the document may be selected. Once again, the selection comprises all highlighted items; there is one and only one selection.

Discontiguous selections can be made in any pane of a split window.

COMMANDS

Once the information to be operated on has been selected, a command to operate on that information can be chosen from lists of commands called menus.

A principal problem with menu-driven systems is that it's difficult for the menu to share the screen with the information being worked on, and especially difficult to show all menus at the same time. Most systems "solve" these problems with modal tree-structured hierarchies of menus, where menus are chosen from a menu of menus, while the user's information has disappeared from the screen. Unfriendly because it segregates information from commands, and confusing because it forces users to "walk" up and down trees of menus, this approach will not work for Macintosh. Instead, taking advantage of Macintosh's ability to overlap things on the screen, we make all menus available at all times (with the user's information still visible) by means of pull-down menus.

The Menu Bar

The menu bar is displayed at the top of the screen. It contains a number of words and phrases: these are the titles of the menus (see below) associated with the current tool. The contents of the menu bar and the corresponding menus are different for each tool. In this sense the tool is said to "own" the menu bar.

There is one and only one menu bar on the screen at any time. Exceptions may be made in special cases: full-screen games may need no menu bar, for example.

(hand)

> The titles in the menu bar, and their corresponding menus, should remain constant throughout the tool. A tool should not change the available menus or put up different menu bars at different times.

Of Mice and Menus

The user positions the pointer over a menu title on the menu bar and presses and holds the mouse button. The title becomes highlighted and a rectangular menu descends from the menu bar under the title; it remains down as long as the mouse button is held down, or until the user moves the pointer away from the menu.

The menu contains a number of items, usually stacked vertically inside the menu; each item names an operation that can be performed. The items may contain words, icons, or both. To invoke a command in the menu, the user drags the pointer down to the menu item (which becomes highlighted), then releases the mouse button. As soon as the mouse button is released, the menu item blinks briefly, the menu disappears, and the command is executed. The menu title in the menu bar remains highlighted until the command has completed execution.

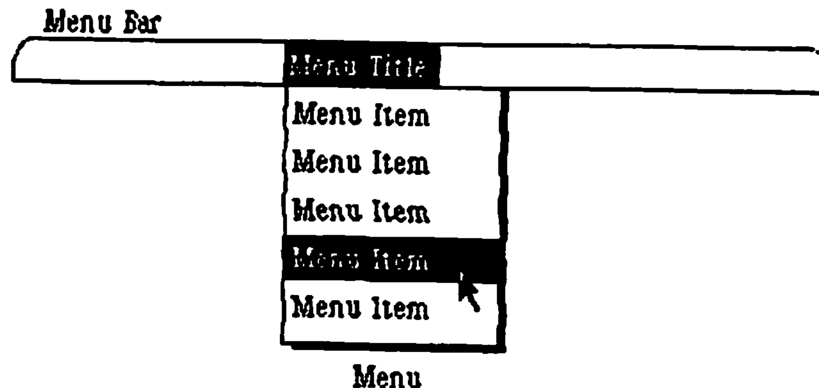


Figure 21. Pull-Down Menus

Because the user chooses a menu item only by pointing the pointer at it, and its command takes effect only when the mouse button is released, if the user drags the mouse outside the menu area (when the menus are showing) and releases the mouse button, no command is selected and no action takes place. Thus there is always recourse should the user have a change of heart after pulling down a menu, and the user is never forced to activate a command.

(hand)

> The menu items, and NOT the menu titles in the menu bar, act upon selections. Users should always be able to peruse the inventory of commands by dragging the pointer across the menu bar without fear of causing something to happen.

The only way to pull down a menu is to press the mouse button while the pointer is in the menu bar. While the user is holding down the mouse button, the pointer does nothing but pull menus down and highlight their items.

If the user tries to perform an operation on a selection that is not currently visible, automatic scrolling occurs to make the selection visible before the operation is performed. The document scrolls until the selection is completely in view or, if the selection is very large, the entire window is filled with the part of the selection nearest to the current position; then the chosen operation is performed.

Notes on General Properties of Menus

Not all menu items are relevant at all times. A menu item that is inapplicable to the current selection is visually distinct from the others (perhaps grayed out) and will not highlight when a user tries to choose it. Repeated attempts by the user to choose an ineffective menu

item warrant explanations from the alert mechanism (see SPECIAL CONDITIONS).

(hand)

A menu in the menu bar can always be pulled down, even if all its menu items are ineffective; in such cases, the menu title is also grayed out. The user should always be able to survey all the available commands, even if they are inoperative.

Commands that may be invoked from the keyboard with the COMMAND key (see below) have a special notation on the right side of the menu. The notation consists at present of an apple symbol and the key that is used with COMMAND to invoke that command.

Menu items are grouped in a menu to emphasize the logical relationships among the groups. Groups are separated by a one-item-high blank space that serves to visually distinguish the groups. This space is not an item and is not highlighted when the pointer moves over it.

Experience shows us that it's easiest for users to choose the second, third, and fourth items in the menu: thy're far enough away from the menu bar to reach them without overshooting, but still not too much of a reach down. We recommend that the most common and safest commands go in these positions.

Also in regards to safety, the commands that cause the greatest effect (such as Quit) should be separated from other, less "dangerous" commands. Similarly, pairs of commands that perform similar functions with slight differences should not be adjacent; a user may choose one accidentally, intending the other, and not notice the subtle difference.

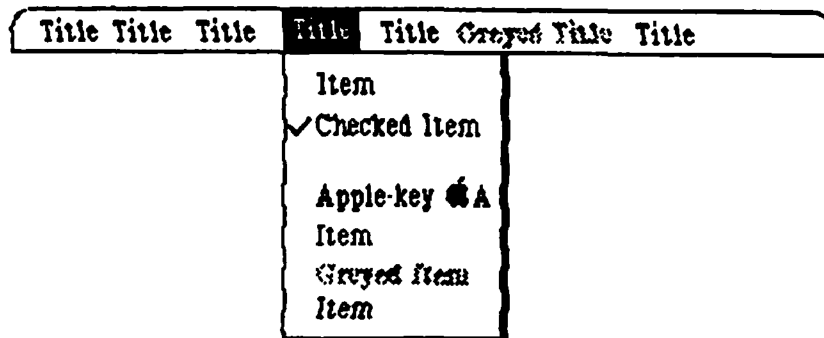


Figure 22. General Properties of Menus

Some commands come in pairs, with only one command of the pair being appropriate at any given time. Most often these pairs control the appearance of something on the desk: one command makes the object visible, and the other command makes it invisible. For example, in the Word Processor, the rulers that set margins and paragraph formatting are normally visible in the window. If the user wishes to remove the rulers, there is a command called "Hide Rulers". When the user invokes this command, the rulers disappear and remain hidden; meanwhile, that command has been replaced with its counterpart, "Show Rulers".

(hand)

> These are not two different commands; they are opposite sides of the same command. The intent of this pairing method is to shorten and simplify menus. The pairing does not make a good indicator of state.

Some status information can be conveniently shown in menus, with the commands that affect that status. If all the information in the selection shares a certain characteristic, and that characteristic can be set with a menu command, that command is marked with a check-mark to show the state of the selection.

Also, in situations where commands in a menu not only perform their function on the selection, but also set a state that controls the interpretation of subsequent input (such as the Bold command), the commands whose states are currently in effect are similarly marked. In this way the menu allows the user not only to change how subsequent input will be interpreted, but also to see the interpretation before changing it.

The Standard Menus

Although the titles on the menu bar are different in each tool, the three menus at the left of the menu bar (the Apple, Edit, and File) remain the same at all times.

The commands and information in these menus pertain to functions common to all Macintosh users: inquiring the state of the current tool and data, invoking global system functions, and loading, saving, and printing documents.

The Apple Menu

```
Apple
Calculator
Alarm Clock
Note Pad
AppleGram
 --------
Tool Information
Document Information
```

Beginning the Apple menu are the names of the desk accessories currently available to the system. Choosing a name activates the corresponding accessory and places it on the desk; double-clicking the close box on the accessory makes it disappear and reactivates the previously active window. The list of available accessories changes with the availability of the accessories themselves.

The "Tool Information" and "Document Information" commands in the Apple menu let the user see information pertaining to the current state of the tool being used (its author, publisher, copyright message, version number, perhaps a hotline number) and the current document (its size, file name, label, creation and modification dates, "home" location or diskette, and any other status information).

These commands, when invoked, present a window that contains the appropriate information; the window remains on the desk top until the user explicitly removes it by double-clicking its close box.

The document information window gives the user the ability to see important but little-used information about the current document, without taking up valuable screen space when the information isn't needed. The tool information is an important tool in the continued support of the customer: should anything go wrong with a tool, the users have a way to refer to the exact version number of the problematic program when seeking help from a dealer or hotline.

In tools that have a global "help" facility, the Help command appears at the bottom of the Apple menu.

The Edit Menu

The Edit menu includes all the editing commands necessary to manipulate pieces of documents.

Edit
Undo {what}

Copy
Cut
Paste

Select Everything

The effects of the four editing commands are more thoroughly discussed in the BASIC EDITING PARADIGMS, below. Briefly, Cut removes the selection from the document, storing it in an intermediate window called the scrap; Paste replaces the current selection with the contents of the scrap; Copy duplicates the selection into the scrap without removing it from the document; and Undo negates the action of the immediately previous command.

Selection commands and other editing functions appropriate to the current tool may also appear in the Edit menu, but the location and order of the first four items must not change.

The File Menu

Although the exact functionality and layout of the File menu has yet to be worked out, our current thinking has it resembling this:

```
File
Quit this tool
  --------
Save this document
Print this document
  --------
Get another document
```

"Save this document" saves the current document into a file; "Get another document" gets a new document from another file; and "Print this document" invokes the printing subsystem of the tool.

"Save" and "Get" allow the user to use a limited subset of the Desk functions in selecting, creating, or naming the file associated with the document.

The "Quit" command is in the Files menu to make sure that users see their opportunity to save their work before quitting. Conversely, in the process of saving their work, they see their opportunity to leave the tool. If the user chooses to Quit before saving the document, the tool should give a gentle yet firm reminder that quitting now will cause the loss of all that information, and request confirmation before actually quitting.

Keyboard-Invoked Commands

The editing paradigms described below allow a user to perform all basic object manipulation--adding, removing, replacing, and moving--using the keyboard to enter text, the mouse to select text, and the commands in the Edit menu to manipulate it.

But this paradigm is likely to generate a lot of hand-waving--the user's hand must move from the keyboard to the mouse, and move the mouse from the document to the menu bar. <u>As an optimization</u> to reduce hand motion, common commands available on the three standard menus may also be invoked from the keyboard, by using the COMMAND key in combination with another key.

(hand)

> When the user holds down the COMMAND key on the keyboard and presses another key, that key is interpreted not as text entry, but as an invocation of a menu command. If the key does not correspond to any implemented command, the alert mechanism is invoked to beep at the first occurrence and give an alert message at any subsequent occurrences.

When one of these command keys is pressed, the menu title of the menu containing the corresponding command highlights while the operation is

being performed, then reverts to normal. The menu itself does not pull down.

The currently defined command keys are as follows:

COMMAND Z — Paste

COMMAND X — Cut

COMMAND C — Copy

COMMAND V — Undo

COMMAND space — Save this document and quit

COMMAND / or ? — Help

In all tools that have a Format or Typestyle menu to change the typestyle while entering text, the following command key aliases are supported:

COMMAND Q — Plain text

COMMAND W — Boldface

COMMAND E — Italic style

COMMAND R — Outline style

COMMAND T — Underlined

COMMAND Y — Shadowed

The commands, just like their counterparts in the menus, are cumulative: pressing COMMAND E while Boldface is already in effect results in bold italic text. The Plain Text command undoes all other styles.

The "OK" and "Cancel" buttons in dialog boxes (see below) also have command aliases:

COMMAND Enter — OK

COMMAND ` or — Cancel

Several emergency commands can be invoked from the keyboard. Note that rebooting the system is not among them.

COMMAND . — Stop current operation

COMMAND 1 — Eject internal diskette

COMMAND 2 — Eject external diskette

(hand)

The command keys are aranged positionally, not mnemonically. The command keys retain their position (not their alphabetical characters) on foreign keyboards.

What Commands Are and Aren't

- Commands, when invoked, operate immediately and return control to the user when completed.
- Commands operate on something visible in the active window, or add or remove a window on the desk.
- Commands that manipulate user information always operate upon the active selection, never upon any nonselected data.
- Commands are either verbs or verb phrases, never nouns with an implied verb.
- Most importantly, commands don't put the tool into an invisible modal state.

BASIC EDITING PARADIGMS

The Macintosh User Interface ToolBox contains a set of core editing routines that standardize the ways the user edits and manipulates text. As long as application programmers use this package properly, every piece of editable text the user sees on the Macintosh screen can be edited using the same quick, consistent methods. The paradigm below supports:

- Inserting, deleting, and replacing text;
- Moving text from one place to another in the same document;
- Carrying information between two similar or dissimilar documents.

The core editing routines also handle font changes, typestyles, and paragraph formatting; these abilities are further discussed in the documentation of those routines.

(hand)

> The following discusses only the operation of Cut, Paste, Copy, Undo, insertion, and replacement on text. The same procedures should operate in a conceptually parallel manner on non-text items, i.e., graphics, spreadsheet cells, etc. It is the responsibility of the designers and programmers to maintain consistency in the editing operations on non-text items.

The Selection

As described in the section on "Inside Documents", there is always one and only one active selection in an active window that contains editable text. A selection takes one of two forms:

1. A selection between two characters that encloses no text: this appears as a blinking vertical bar and is called an insertion point.
2. A selection enclosing one or more characters of text.

The editing commands Cut, Paste, Copy, and Undo, whether invoked from the Edit menu or by the COMMAND key on the keyboard, act upon the selection. Typed characters also affect the selection.

The Scrap

The scrap goes hand in hand with the Edit menu. It is a very special kind of window with a well-defined function: it holds whatever is cut or copied from a document. It sticks around, its contents intact, when the user changes tools.

Every time the user performs a Cut or Copy on the current selection, a copy of the text in the selection replaces the previous contents of the

scrap.

The user can't select the scrap or any information inside it. But the scrap window can be dragged around by its title bar, and can be enlarged or reduced by dragging its size box. In most ways the scrap behaves just like any other window.

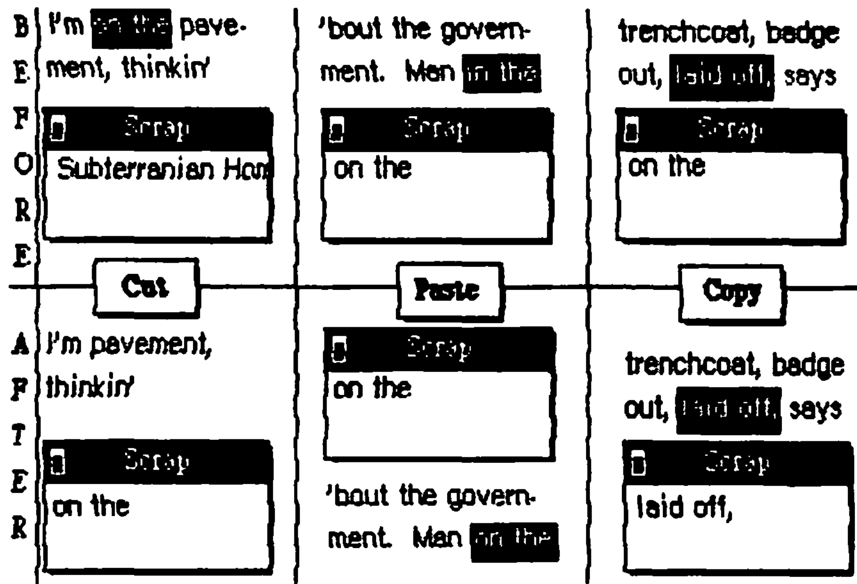


Figure 23. Use of the Scrap

There is only one scrap, which is on the desk for all tools that support Cut and Paste (it's hidden during games and such). If the user doesn't want the scrap to interfere with other things on the screen, the scrap can be shrunk to its smallest size, dragged nearly off the screen, or buried under other documents. Nothing changes the contents of the scrap except Cut, Copy, and Undo.

As the contents of the scrap remain unchanged when applications begin and terminate, the scrap can be used for transferring data among mutually compatible applications (see "Cutting and Pasting between Tools", below).

The Cut and Copy commands

The Cut command removes the current selection from the active document and puts it in the scrap. The selection completely replaces the previous contents of the scrap. The selection in the document is reduced to an insertion point.

If a Cut is attempted when the selection is an insertion point, Cut doesn't light up in the menu when chosen. This prevents people from accidentally cutting twice and losing the scrap.

Perfoming a Copy command puts a copy of the current selection into the scrap, without changing the original selection in the active document. Just as with a Cut, every Copy completely replaces the previous contents of the scrap. Also like Cut, the Copy item won't light up if

the selection is an insertion point.

Paste

The Paste command is the effective antonym of Cut: it replaces the current selection with the contents of the scrap. A Paste leaves the contents of the scrap unaltered; the selection is set to an insertion point at the end of the pasted text. With this, successive invocations of Paste replicate the contents of the scrap at the selected position in the document.

(eye)

> Notice that in a Paste over an existing selection, the contents of the selection do not go into the scrap; they can be recovered only by an immediate invocation of Undo.

Undo

Finally, the Undo command is a one-level negation of the last command. It always applies to all Edit commands; additionally, any larger scope of Undo can be added by the application. If the previous operation was an Undo, it undoes that Undo.

Inserting and Replacing Text

New text can be entered from the keyboard or numeric keypad. Typing new text operates much like a Paste command.

Typed text replaces the current selection. If the current selection is an insertion point, the typed characters appear at the insertion point and the insertion point moves past the characters. If the current selection includes text, the entire selection is automatically reduced to an insertion point, deleting the text; insertion then proceeds as described above.

(hand)

> Notice that if a selection is replaced with an entry from the keyboard, the selection does not go into scrap. Its contents can be recovered only through an immediate invocation of Undo.

Backspacing

Regardless of circumstances and context, if the selection is an insertion point, pressing the BACKSPACE key deletes one character before the insertion point and moves the insertion point to the left of the position previously held by that character. This happens during editing as well as text entry.

Pressing BACKSPACE while the selection contains characters operates much like a Cut, except that the deleted characters go into the backspace buffer (see below) rather than the scrap. The first BACKSPACE deletes the selected text, reducing the selection to an insertion point; subsequent presses of BACKSPACE operate as described

above.

Every press of BACKSPACE stores up the deleted characters in the backspace buffer. Invoking the Undo command reinserts all characters in this buffer back into the document at the insertion point. Performing any other operation, such as typing characters or invoking another command, clears this buffer; the deleted characters are then unrecoverable.

Cutting and Pasting Between Documents

Sometimes the user wants to transfer a portion of one document into another. The documents may have been created with the same tool, or with disparate tools. Macintosh allows this kind of manipulation through the mechanism of Cut/Copy and Paste.

Between Two Documents with the Same Principal Tool

Transferring information from one document to another created by the same application does not pose any difficulty. For example, the user may Copy the return address from 'Letter to Jef', Get 'Letter to Linda' and Paste in the contents of the scrap.

When the user discards a tool and returns to the Desk, the scrap retains not only its contents, but the contextual information pertinent to the tool being used. If the user retrieves that same tool, it can interpret that information, so there is little or no loss of context when carrying something in the scrap from document to document.

Between Documents with Different Principal Tools

Macintosh provides a limited but adequate scheme for transferring information from a document of one type to a document of another type.

Suppose the user wants to transfer a picture of a wolf (previously created using the Graphics Editor) into a Word Processor document named 'Letter to Grandma'. Beginning at the Desk, the user gets the wolf picture, automatically entering the Graphics Editor. There the picture is selected and Cut or Copied into the scrap; then the user returns to the Desk. The picture remains in the scrap.

Now the user calls up the letter to Grandma and enters the Word Processor. Upon selecting a position and attempting to Paste, the Word Processor examines the scrap and determines whether it is palatable. As Graphics Editor pictures are implemented with the QuickDraw picture structure, the Word Processor has no problem interpreting and displaying the picture, and graciously pastes it into the letter. However, in the letter the wolf and the rectangular area around it are selectable only as a single unit; the individual parts of the wolf are not editable. To the Word Processor the wolf is static data.

Each tool may have its own appropriate level of interpretation of the scrap. If the user tries to Paste the scrap in a tool that does not understand it, the tool presents an alert message to inform the user of

the undigestability of the scrap.

SPECIAL CONDITIONS

The <noun>+<verb> syntax is wonderful and clean when the operations are simple and act on only one object. But occasionally a command will require more than one object, or will need additional parameters in order to be most useful to the user. And sometimes a command won't be able to carry out its normal function, or will be be befuddled as to the user's real intent. For these special circumstances we have included two mechanisms: the Dialog Box to garner additional information, and the Alert mechanism to signal error or warning conditions.

Dialog Boxes

Commands in menus normally act upon only one or two objects: the current selection, the scrap, or a default object. If a command needs more information before it can be performed, it presents a Dialog Box to gather the additional information from the user.

A Dialog Box is a rectangle that may contain text, buttons, dials, and icons. It is slightly below the menu bar, a bit narrower than the screen, and as tall as its contents require. It is clearly labelled with the name of the command whose invocation prompted the appearance of the box.

Print the Document
5 copies
8 1/2" by 11" paper
8 1/2" by 14" paper
14" by 11" paper
Stop after printing each page
OK
CANCEL
STOP
PAUSE

Figure 24. A Dialog Box

Some dialog boxes may affect several properties at the same time or show several choices of the same property. In such cases, the choices have check-boxes next to them. The boxes next to properties that are currently in force are checked. Clicking on a check box or the text accompanying it puts a check-mark in the box; this may also cause other boxes to become unchecked.

If the information requested by the dialog box is textual, the user can enter and edit that text just like any other editable text. If the information has a default value (which it should have, if possible), the default text appears selected in the dialog box. If the user starts typing, the selected value will be replaced with what the user types. For boxes with many text items, the first one is selected when the box appears. After editing an item,

- Pressing ENTER, TAB, or RETURN accepts the changes made to the item, and selects the next item in sequence.

- Clicking in another item accepts the changes made to the previous item and selects the newly clicked item.

There are, at the absolute minimum, two buttons in the Dialog Box--"OK" and "Cancel". "OK" enforces the modifications in the properties included in the Dialog Box, removes the Dialog Box from the screen, and performs the command originally issued. "Cancel" dismisses the Dialog Box without effecting any changes.

The "OK" and "Cancel" buttons should always appear in the same relative orientation in the Dialog Box to preserve a consistent feel to the interaction. They should be near the title of the dialog box to remind the user of what command they will perform or cancel. They may be marked with reinforcing icons, e.g., thumbs-up and thumbs-down.

A Dialog Box may include a "Stop" button, marked with an octagonal stop sign, for stopping operations that are in progress, such as printing.

When a command requires some time to execute, its Dialog Box may contain a dial that indicates the level of completion of the task in progress.

The Alert Mechanism

Every user of every application is liable to do something that the application won't understand. From simple typographical errors to slips of the mouse to trying to write on a protected diskette, users will constantly do things an application can't cope with in a normal manner. The Alert mechanism gives applications a way to respond to errors not only in a consistent manner, but in steps according to the severity of the error, the user's level of expertise, and the particular history of the error.

There are three levels of alerts:

1. Note: Probably a minor slip that's signaled by an audible warning.

2. Caution: A condition in which the application can't understand the user's input, and must request that the user change something.

3. Stop: A situation that requires definitive action on the part of the user, such as inserting another diskette.

These are ranked in ascending order of importance. Not only are program errors ranked in this manner, but repeating an error increases its importance: receiving the same Note alert several times, for example, turns it into a Caution, which warrants further explanation and assistance.

Note alerts are signaled by a beep from the speaker; if the speaker volume is turned off, the beep is inaudible. Caution and Stop alerts warrant an alert box (see below).

Alert Boxes

Alert Boxes are similar in appearance to Dialog Boxes. Alert Boxes are intended to give the user warnings and error messages. Before describing Alert Boxes it is worth while mentioning a few words about alert messages in general.

Alert Boxes are displayed to:

- Clarify the system's response to users' actions, (e.g., "This text is not editable"),

- Lead the user through a series of actions required for the completion of certain tasks, (e.g. "Please insert a diskette to be copied to"),

- Inform of a state that might affect users' future activities ("The document is getting too long to hold in memory. You may want to break it up into pieces"),

- Warn the user against doing something irrevocable or dangerous ("You will lose the contents of this diskette if you proceed with initialization. Do you still want to initialize?"), giving an opportunity to cancel the command, and

- Delay while a lengthy operation is being concluded.

How to Phrase an Alert Message

It is important to phrase messages in Alert Boxes so that users are not left guessing the real meaning. Do not use computer jargon. Sometimes it is difficult for the jaded to recognize jargon even as they use it. If you have any doubts of the lucidity of a message, try it on an unsuspecting naive friend.

Use icons whenever possible. Graphics can better describe some error situations than words, and familiar icons help users distinguish their alternatives better. The thumbs-up icon should always lead to the safest route out of a situation.

Generally, it is better to be polite than abrupt, even if it means lengthening the message. The role of the Alert Box is to be helpful and make constructive suggestions, not to give out orders. But its focus is to help the user solve the problem, not to give an interesting

but academic description of the problem itself.

Under no circumstances should an Alert message refer the user to external documentation for further clarification. It should provide a complete encapsulation of the information needed by the user to take appropriate action.

(hand)

> The best way to make an Alert message understandable is to think carefully through the error condition itself. Can the application handle this without an error? Is the error specific enough so that the user can fix the situation? What are the recommended solutions? Can the exact item causing the error be displayed in the alert message?

Be as specific as you can when signaling an error condition.

Appearance of Alert Boxes

An Alert Box is a rectangle just a little narrower than the screen and of variable height. It may contain text, icons, dials and buttons. It appears in a slightly lower position from where Dialog Boxes appear, to emphasize that the alert message is more important.

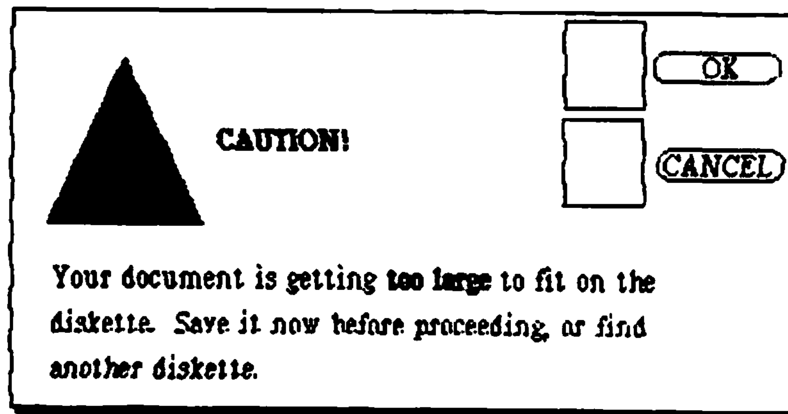


Figure 25. An Alert Box

All Alert Boxes have a "Cancel" button that dismisses the box. Alert Boxes that require confirmation to perform an action have an additional "OK" button. Some Alert Boxes may include a "Stop" button to allow the user to interrupt an ongoing operation. As in Dialog Boxes, the relative orientation of these buttons should remain the same from box to box.

If there are a small but finite number of ways to solve the problem, the box may contain descriptions of those ways, each marked by

check-boxes. The user checks the desired solution and presses the "OK" button.

Alert Boxes that require immediate attention contain a stop sign in the upper-left corner of the box to emphasize the severity of the warning.

Alert Boxes that inform the user about a process' status may display dials to indicate the level of completion of a task, much as in Dialog Boxes.

APPENDIX A: THOU-SHALT-NOTS OF A FRIENDLY USER INTERFACE

Here are six things to avoid when designing a friendly user interface.

1. Assigning more than one consequence to the same action.

2. Giving the user several ways to perform the same function. Generally, it is much easier for users to learn a task when there is only one obvious way of accomplishing it. Too many alternatives in an unfamiliar environment may paralyze the user.

3. Overloading an application with too many esoteric features. Before introducing another nifty feature, ask yourself how the feature will affect the overall complexity of the application, and how many users will benefit from the feature.

(hand)

> Featurism is the single major contributor to system complexity and user intimidation.

4. Changing the state of the world while the user is not looking. One way to make a user comfortable with a system is to create an environment that is predictable and consistent. For example, if the contents of a menu change from one invocation to another, the user comes to think that the machine has a mind of its own, and feels that control of it will always be elusive.

5. Cluttering the screen. A cluttered and busy screen is frequently a symptom of an application design that is not carefully thought out. Reevaluating the reasons for different features (always keeping the end user in mind) will generally result in a simpler, more elegant program and visually more streamlined interface.

6. Overenthusiastic use of modes. It is highly desirable, if not always possible, to allow the user to go from one activity to another without feeling trapped in a mode. For an eloquent discussion of modes, the reader is referred to "The Smalltalk Environment", an article by Larry Tesler in the August, 1981 issue of BYTE magazine.

APPENDIX B: POINTER SHAPES

Certain pointer shapes have been standardized to imply that specific actions will occur when the mouse button is pushed.

(I-beam)

Text selection

(Hollow Cross)

Selection in a structured document

(Plus sign)

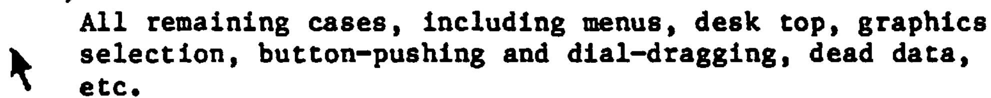
Drawing graphics

(Hourglass)

Long operation in progress (sometimes associated with a dial in a dialog box)

(Arrow)

All remaining cases, including menus, desk top, graphics selection, button-pushing and dial-dragging, dead data, etc.



APPENDIX C: THE PHYSICAL BOX

The following summarizes Macintosh's salient hardware features.

Physical box:

- A main unit with a built-in 9" CRT and a built-in minifloppy drive;

- A detached keyboard;

- A mouse.

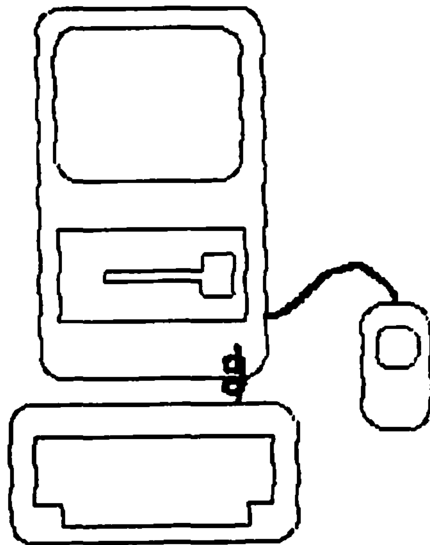


Figure 26. Macintosh

Memory capacity:

- 131,Ø72 bytes (128K) of user and program memory, 21,888 bytes (21 3/8 K) of which are dedicated to the video display;

- 65,536 bytes (64K) permament (ROM) storage;

- 86Ø,16Ø bytes (84ØK) storage on the built-in disk drive.

Microprocessor:

- Sixteen-bit Motorola MC68ØØØ with eight 32-bit data registers, seven 32-bit address registers, and two stack pointers.

- 56 instructions in 14 addressing modes; microprocessor runs at 8 million cycles per second (8MHz).

Display:

- 512 dots wide, 342 dots tall, black and white dots on a square grid. Dots displayed at 8Ø dots per inch on a 9" screen.

This is the only configuration of Macintosh. There are no other memory sizes, no different ROMs, no other video displays. The consistency of the Macintosh user interface is based on the consistency of the hardware: as every Macintosh ever sold is guaranteed to contain the above, every application program written for this configuration will run on 1ØØ% of the installed base.

The only options available are:

- A second 84ØK floppy disk drive;

- An 18-key numeric keypad;

- A dot-matrix or letter-quality printer;

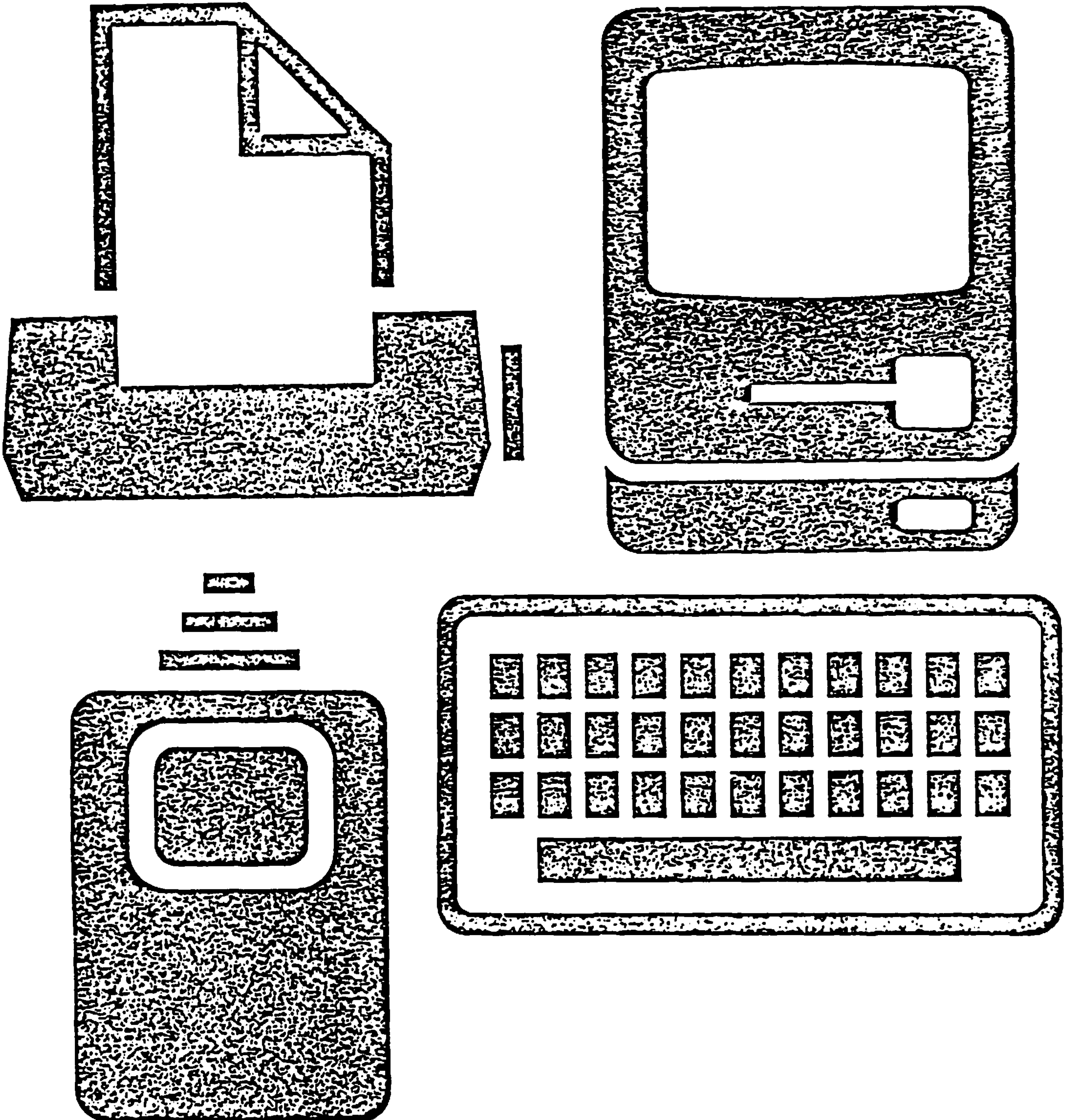
- Connection to a RS-232, RS-422, or network communication device.

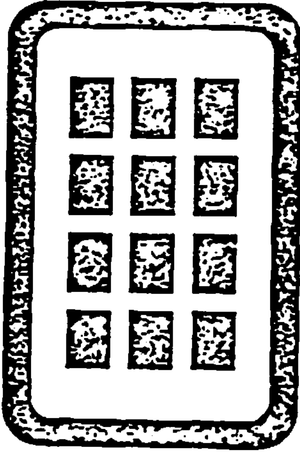
APPENDIX D: KEYBOARD LAYOUTS AND CHARACTER ASSIGGNMENTS

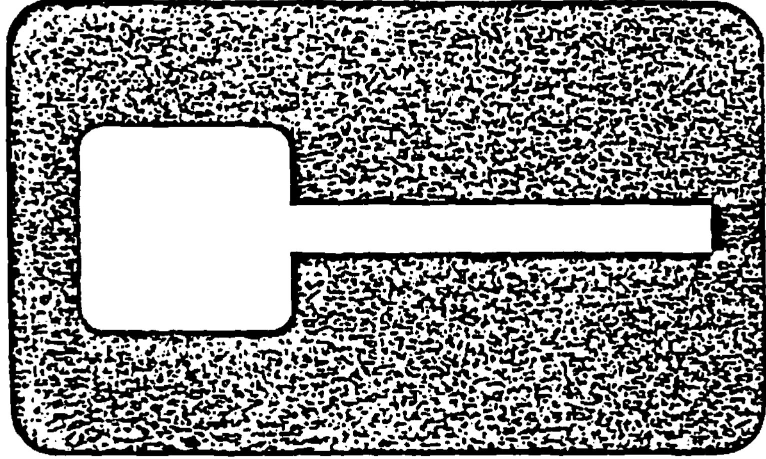
Here are the keyboard layouts and ASCII character assignments for the standard character sets in Macintosh:

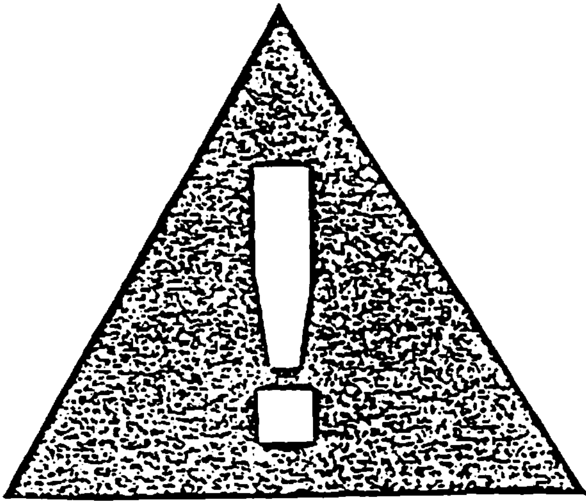
APPENDIX E: GUIDE TO ICONS

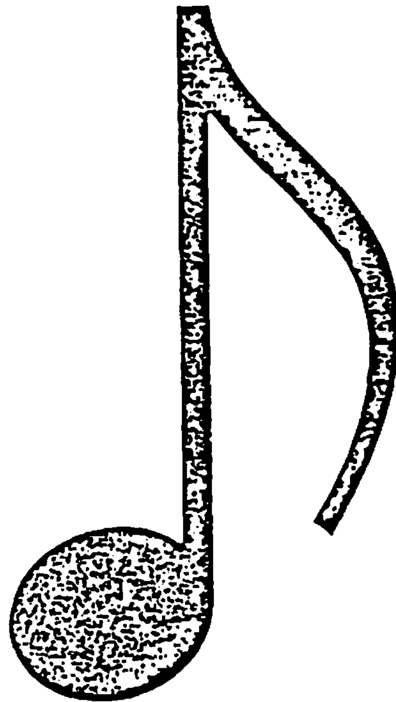
Here are the standard icons as used on our packing materials, on the back of the Macintosh itself, and appearing in Macintosh software:

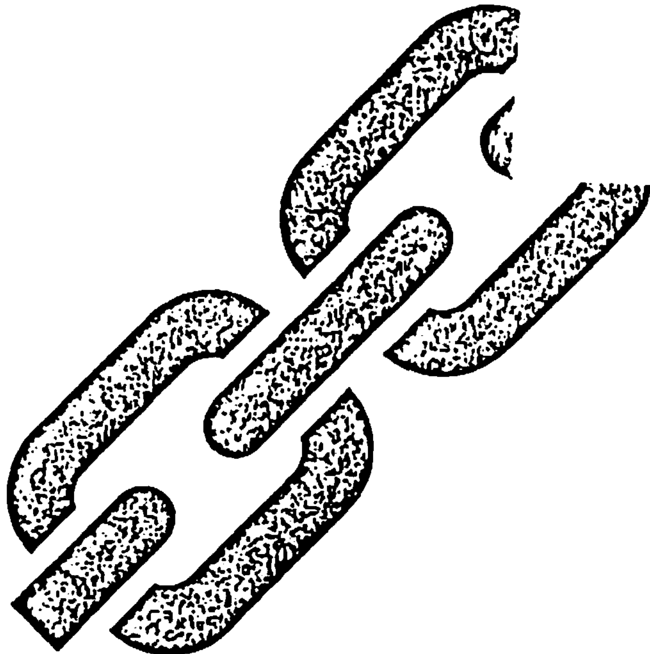












APPENDIX F: UNRESOLVED ISSUES

- What does the Close box do in the main document window for a tool? Does it put away the document, unload the tool, and return the user to the Desk? As Larry's tests show that users occasionally hit the Close box when intending to drag the title bar (or pull down a menu), is it proper for such a commonly-misused icon to perform such a time-consuming task?

- When inactive windows in Lisa are dragged, they are brought to the top afterward. We don't do this.

- Do Show Scrap/Hide Scrap exist? Where? And is the scrap called the Clipping?

- How do Macintosh command-key assignments differ from those on Lisa, and will we have a real Apple key rather than the word COMMAND?

- Do Randy's Core Editing or Word Processor routines support backspace-by-word, or unbackspace?

- There's a clash between the use of the stop sign as a warning icon in Dialog and Alert Boxes and its use as an icon on the interrupt button in the same place.

- COMMAND-Click and SHIFT-click, and their conflict in the Graphics Editor, is unresolved.

- The 1/4"-grey-around-the-edges was dropped in this draft. It is superfluous, hard to code, and adds little to the illusion.

TECHNICAL LEXICON

These terms are defined here in their technical meaning and relationship to one another. Users will never encounter some of the terms mentioned here; neither will they read the descriptions as phrased here. For a users'-eye-view of Macintosh terminology, please see the glossaries in the Macintosh User Style Guide and in the Macintosh Introduction manual.

Term	Definition
Active Selection	(Noun) See Selection, Active
Active Window	(Noun) See Window, Active
Alarm Clock	(Noun) A desk accessory that displays the current date and time, as well as allowing the user to set an alarm date and time and an alarm message. Usage: Same as Desk Accessory
Alert Box	(Noun) A window containing warnings and cautions, which appears when a tool encounters an unsolvable error or a dangerous situation. An alert box always contains two buttons, labeled OK and Cancel. See Also: Alert Message Usage: Present an A.B.
Alert Message	(Noun) An audible or visible message or warning generated by the computer to signal input errors, problems interpreting data, or situations threatening the safety of the user's data. See Also: Alert box
Automatic Scrolling	(Noun) See Scrolling, Automatic
Back	(Noun) The position or orientation of objects on the desk furthest from and least visible to the user; objects in front overlap and obscure objects in the back. See Also: Front Window Behind Usage: Send to the b. In b. of another
Behind	(Adverb) In the position or orientation towards the back. An object on the desk is behind all the objects that are in front of it.
Button	(Noun) A control that causes an action when clicked or pressed. Buttons highlight when pressed. Usage: Press Click

Button, Mouse	(Noun) See Mouse Button.
Calculator	(Noun) A desk accessory that emulates a four-function desk calculator. Calculation results can be cut and pasted between the calculator and the user's document. Usage: Same as Desk Accessory
Cancel button	(Noun) A button that, when pressed, cancels a proposed action or action in progress. The cancel button is labeled "Cancel" and is marked with a thumbs-down icon. See Also: OK button Usage: Same as button
Check Box	(Noun) A control in the shape of a square box, which may or may not have a check mark in it. Clicking in a check box toggles its state, and may affect the state of related check boxes. Usage: Check Click
Choose	(Verb) To pick a menu item from a menu. Usage: Choose a command Choose a menu item
Click	(Verb) To position the pointer and briefly press and release the mouse button without moving the mouse. See Also: Drag Double-Click Usage: Click an object Click the mouse button
Close	(Verb) To remove the window from a document; you close a window to reduce it to an icon that represents the document. Usage: Close a window (never close a file)
Close Box	(Noun) The box on the left side of the title bar of a document window that, when clicked, closes the window. The close box contains an icon of a document that "winks" when ckicked. Usage: Click the close box
Closed	(Adjective) The state of a window when the document it contains is not visible. Documents whose windows have been closed are represented by icons.
Command	(Noun) A word (usually appearing as a menu item) that describes an action that a Macintosh tool can perform; or the action itself. Usage: Choose a command from a menu The command takes effect

Term	Definition
Control	(Noun) An object on the screen that causes an action when clicked or dragged; buttons, dials, and scroll bars are the most common controls. Usage: Use only when necessary.
Control Panel	(Noun) A desk accessory full of controls. With it, the user can change the speaker volume, the keyboard repeat speed and delay, system paranoia level, etc. Usage: Same as Desk Accessory
Desk	(Noun) The tool that deals with coying, moving, creating, deleting, and changing the names of files. Also refers to the smaller version used within applications. Usage: On the desk (?)
Desk Accessories	(Noun) Mini-tools generally available at all times. A pocket calculator, note pad, telegram form, alarm clock, and the control panel are the currently imagined desk accessories. Usage: Get a D.A. Use the D.A.
Desktop	(Noun) The metaphor for the Macintosh working environment. See Also: Desk
Dial	(Noun) A control that acts as a pseudo-analog output and/or input device. See Also: Scroll Bar Usage: Adjust a dial
Dialog Box	(Noun) A window opened by a tool that requests the user for entry or confirmation of information. A dialog box is presented when a chosen command needs more information in order to take effect. See Also: Alert Box Usage: Present a d.b. Close the d.b.
Discontiguous Selection	(Noun) See Selection, Discontiguous
Disk	(Noun) Any kind of rotating magnetic storage device. See Also: Diskette Disk Drive Usage: Save on a d. Get from a d.
Disk Drive	(Noun) The mechanism that stores and retrieves the information on a disk. See Also: Diskette

Term	Definition
Diskette	(Noun) A thin, plastic disk. See Also: Disk Drive Usage: Insert the d. Eject the d. On the d.
Document	(Noun) A collection of information intelligible to a user. See Also: File Window Tool Usage: Get a d. Save a d. Scroll a d.
Document Panel	(Noun) The pane of a document window that presents the document itself, as opposed to status panes, formula panes, etc. See Also: Panel Usage: Avoid if possible.
Document Window	(Noun) A window that displays a document. Document windows usually come equipped with a title bar, one or two scroll bars, a size box, and a close box. Usage: Use only when "window" is ambiguous.
Double-Click	(Verb) To click the mouse button again shortly after a previous click. Double-clicking an object enhances or expands the action normally caused by singly clicking that object. Usage: D.C. an object D.C. the mouse button
Drag	(Verb) To press and hold the mouse button while moving the mouse. Dragging either selects items (when done inside the window) or drags a flickering outline of an object (outside the window). See Also: Click Select Choose Size Window Split a Window Usage: D. an object D. the mouse D. out a rectangle D. across the text
Enter	(Verb) To insert or add information into the computer, usually by typing on the keyboard. Entries are usually terminated by a press of the ENTER key. Usage: E. the name
Extend (the Selection)	(Verb) To make the active selection larger by holding down the COMMAND key while making another selection. The two selections and all items in between become the new selection. See Also: Select Selection Usage: Extend the Selection Make an extended selection
File	(Noun) A storage container for information. See Also: Document Tool Window Resource File

	Usage: Delete a f. Copy a f. Move a f. Rename a f.
File	(Verb) To put a document into a file, or get a document from a file.
File Name	(Noun) The name attached to a file by its creator.
Font	(Noun) A set of characters of the same typeface and size. See Also: Typestyle Usage: Appears in the f.
Front	(Noun) The position or orientation of objects on the desk that are closest and most visible to the user; the active window is always in front of any other windows. See Also: Back Behind Usage: Bring to the f. In f. of others Frontmost
Highlight	(Verb) To emphasize something by making it visually distinct from its normal appearance; by inverting it, underlining it, making it blink, or appear in boldface, etc. See Also: Invert Select Front Window Usage: H. the text Title bar is highlighted
Icon	(Noun) "1. An image; representation. 2. A similie or symbol." (AHD) A graphic representation of a material object, a concept, or a message. Icons may be objects on the desk. Usage: Click an i. Drag an i. Labeled with an i.
Inactive Selection	(Noun) See Selection, Inactive
Inactive Window	(Noun) See Window, Inactive
Insertion Point	(Noun) A selection enclosing nothing; indicates the position between two items in a document, or an absolute position in that document. Indicates the point at which newly inserted items will be placed. See Also: Select Usage: Make an I.P. At the I.P.
Invert	(Verb) To invert the black-and-white polarity of an image; inverting is the most common form of highlighting. Usage: Inversely highlighted

Item
(Noun) A single piece of information in a document. Each character in a text, each shape or line in a picture, and each cell in a spreadsheet is an item.
See Also: Select Drag Extend (the Selection)
Usage: Between two items Click an i. Drag over items

Key
(Noun) A button on the keyboard. Character keys are typed; modifier keys are held; special keys are pressed.
Usage: Press a k. Hold down a k.

Keyboard
(Noun) The device used for entering text and numeric data. The keyboard has 48 character keys, 6 modifier keys, and 4 special keys.
See Also: Press Type Hold
Usage: Type on the k.

Menu
(Noun) A rectangular list of menu items, which is pulled down from the menu bar; the user chooses a menu item by pressing on a menu title, dragging through the menu, and releasing on a menu item.
See Also: Command
Usage: Choose from a m. Pull down a m.

Menu Bar
(Noun) The horizontal strip at the top of the screen that contains the menu titles.

Menu Item
(Noun) One item in a menu. A menu item may contain words, an icon, or both. Menu items usually describe commands. A menu item is highlighted when the pointer is over it.
See Also: Choose
Usage: Choose a m.i.

Menu Title
(Noun) A word or phrase in the menu bar that designates one menu. Pressing on the menu title pulls down ite menu; dragging through the menu highlights menu items.
Usage: Press on the m.t.

Mouse
(Noun) A small device the size of a deck of cards that rolls around on your desk. Moving the mouse causes corresponding motion of the pointer on the screen.
See Also: Mouse button Drag
Usage: Move the m. Drag the m.

Mouse Button
(Noun) A rectangular button on the top of the mouse. Pressing the button initiates some action at the position of the pointer;

MACINTOSH USER EDUCATION

The Vertical Retrace Manager: A Programmer's Guide /VRMGR/TASK

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Inside Macintosh: A Road Map

Modification History: First Draft (ROM 7) B. Hacker 3/dd/84

*** Review Draft. Not for distribution ***

ABSTRACT

This manual describes the Vertical Retrace Manager, the part of the Macintosh Operating System that schedules and performs recurrent tasks during vertical retrace interrupts.

TABLE OF CONTENTS

3 About This Manual
3 About the Vertical Retrace Manager
5 Using the Vertical Retrace Manager
6 Vertical Retrace Manager Routines
7 Summary of the Vertical Retrace Manager
9 Glossary

Copyright (c) 1984 Apple Computer, Inc. All rights reserved.

Distribution of this draft in limited quantities does not constitute publication.

ABOUT THIS MANUAL

This manual describes the Vertical Retrace Manager, the part of the Macintosh Operating System that schedules and performs recurrent tasks during vertical retrace interrupts. *** Eventually it will become part of a larger manual describing the entire Toolbox and Operating System. ***

(eye)

> This manual describes version 7 of the ROM. If you're using a different version, the Vertical Retrace Manager may not work as discussed here.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the basic concepts of

- the Macintosh Operating System's Memory Manager
- interrupts, as described in the Macintosh Operating System's Device Manager manual

The manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

This manual begins with an introduction to the Vertical Retrace Manager and what you can do with it. It then introduces the routines of the Vertical Retrace Manager and tells how they fit into the flow of your application. This is followed by detailed descriptions of the routines used to install and remove recurrent tasks, their parameters, calling protocol, effects, side effects, and so on.

Finally, there's a summary of the Vertical Retrace Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE VERTICAL RETRACE MANAGER

The Macintosh video circuitry generates a _vertical retrace interrupt_ (also known as the vertical blanking or VBL interrupt) sixty times a second while the beam of the display tube returns from the bottom of the screen to the top to display the next frame. The Operating System uses this interrupt as a convenient time to perform the following recurrent tasks:

1. increment the system global ticks (every interrupt)
2. check whether the stack and heap have collided (every interrupt)

3. handle cursor movement (every interrupt)

4. check the state of the mouse button (every other interrupt).

5. handle repeating keystrokes (every 32 interrupts)

These tasks must execute at regular intervals based on the "heartbeat" of the Macintosh, and shouldn't be changed.

An application can add any number of its own tasks to be executed by the Vertical Retrace Manager. Application tasks can perform whatever actions you desire (as long as memory is neither allocated or released), and can be set to execute at any frequency (up to once per vertical retrace interrupt). For example, a task within an electronic-mail application might check every 1/10 second to see if it has received any messages.

Information describing each application task is contained in the vertical retrace queue, the structure of which is shown in Figure 1.

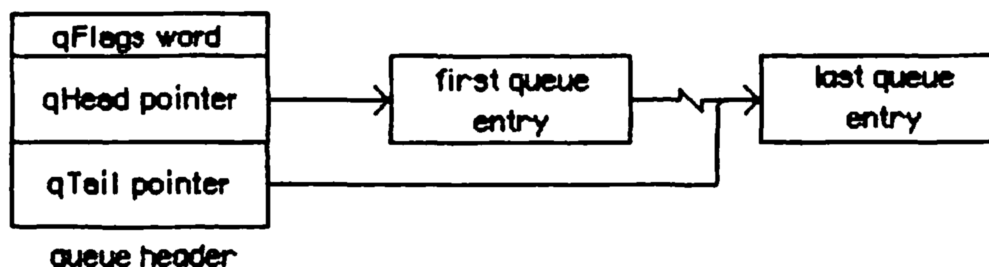


Figure 1. Vertical Retrace Queue

QHead points to the first entry in the queue, and qTail points to the last entry in the queue. Bit 6 of qFlags is set whenever the Vertical Retrace Manager is executing.

Assembly-language note: You can refer to the vertical retrace queue by using the system global vblQueue, which points to the qFlags word.

Each entry in the vertical retrace queue contains information about each task:

```
TYPE VBLCntrlBlk = RECORD
                      vblLink:  Ptr;
                      vblType:  INTEGER;
                      vblAddr:  ProcPtr;
                      vblCount: INTEGER;
                      vblPhase: INTEGER;
                   END;

     VBLCBPtr = ^VBLCntrlBlk;
```

VBLLink points to the next entry in the queue, and vblType indicates the queue type, which should always be the value of the predefined constant vType. VBLAddr contains the address of the task. VBLCount specifies the number of vertical retrace interrupts between successive calls to the task. This value is decremented each interrupt until it reaches zero, at which point the task is called. The task must reset vblCount, or its entry will be removed from the queue after it has been executed. VBLPhase contains an integer (smaller than vblCount) used to bias vblCount when the task is first added to the queue. This ensures that two or more routines added to the queue at the same time with the same vblCount value will be out of phase with each other, and won't be called during the same interrupt.

USING THE VERTICAL RETRACE MANAGER

This section discusses how the Vertical Retrace Manager routines fit into the general flow of an application program. The routines themselves are described in detail in the next section.

The Vertical Retrace Manager is automatically initialized each time the system is started up. To add an application task to the vertical retrace queue, call VInstall. When your application no longer wants a task to be executed, it can remove the task from the vertical retrace queue by calling VRemove. Application tasks shouldn't call VRemove--either the application should call VRemove, or the task should simply not reset vblCount.

An application task cannot call routines that cause memory to be allocated or released. This severely limits the actions of tasks, and you might prefer using the Desk Manager routine SystemTask to perform periodic actions. Or, since the very first thing the Vertical Retrace Manager does during a vertical retrace interrupt is increment the system global ticks, your application could poll ticks and perform periodic actions whenever it changes.

Assembly-language note: Application tasks may use registers DØ through D3 and AØ through A3, and must save and restore any additional registers used. They must exit with an RTS.

VERTICAL RETRACE MANAGER ROUTINES

FUNCTION VInstall (vblBlockPtr: VBLCBPtr) : OSErr;

VInstall adds the task described by vblBlockPtr to the vertical retrace queue.

Result codes	noErr	No error
	vTypErr	VBLType field isn't vType

FUNCTION VRemove (vblBlockPtr: VBLCBPtr) : OSErr;

VInstall removes the task described by vblBlockPtr from the vertical retrace queue.

Result codes	noErr	No error
	vTypErr	VBLType field isn't vType

SUMMARY OF THE VERTICAL RETRACE MANAGER

Constants

```
CONST vType = 1; vertical retrace queue entry type
```

Data Structures

```
TYPE VBLCntrlBlk = RECORD
                     vblLink:  Ptr;
                     vblType:  INTEGER;
                     vblAddr:  ProcPtr;
                     vblCount: INTEGER;
                     vblPhase: INTEGER;
                   END;

     VBLCBPtr = ^VBLCntrlBlk;
```

Routines

```
FUNCTION VInstall (vblBlockPtr: VBLCBPtr) : OSErr;
FUNCTION VRemove  (vblBlockPtr: VBLCBPtr) : OSErr;
```

Assembly-Language Information

Constants

```
vType       .EQU      1       ;vertical retrace queue entry type
```

Vertical Retrace Queue Entry

vblLink	Next queue entry
vblType	Always vType
vblAddr	Location of application task
vblCount	Number of interrupts between task calls
vblPhase	Bias for vblCount

System Globals

Name	Size	Contents
vblQueue	4 bytes	Pointer to the vertical retrace queue

Result Codes

Name	Value	Meaning
vTypErr	-2	VBLType isn't vType

GLOSSARY

vertical retrace interrupt: The interrupt that occurs sixty times a second while the beam of the display tube returns from the bottom of the screen to the top to display the next frame.

vertical retrace queue: A list of the application tasks to be executed during the vertical retrace interrupt.

MACINTOSH USER EDUCATION

The Window Manager: A Programmer's Guide /WMGR/WINDOW

See Also: Macintosh User Interface Guidelines
Macintosh Operating System Reference Manual
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Desk Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
Toolbox Utilities: A Programmer's Guide

Modification History:			
	First Draft	P. Stanton-Wyman	8/16/82
	Interim Release	C. Rose	9/30/82
	Second Draft	C. Rose	10/8/82
	Revised	C. Rose	11/2/82
	Third Draft (ROM 2.1)	C. Rose	3/1/83
	Fourth Draft (ROM 7)	C. Rose	8/25/83

ABSTRACT

Windows play an important part in the Macintosh world, since all information presented by an application appears in windows. The Window Manager provides routines for creating and manipulating windows. This manual describes those routines along with related concepts and data types.

Summary of significant changes and additions since last version:

- Changes have been made to the predefined window definition IDs (page 8) and the window classes (page 12). An rDocProc type of window no longer requires the corner-rounding in the refCon field.

- DrawDocGrow has been replaced by DrawGrowIcon (page 23).

- A close box or size box appears in active windows only. (See FindWindow, page 23, and window definition function, page 35.)

- The discussions of DragWindow, GrowWindow, and SizeWindow have been clarified, and examples have been added for InvalRect and ValidRect (page 25).

- PinRect and DragGrayRgn (formerly DragTheRgn) are now described as Window Manager routines rather than Toolbox Utilities (page 30).

- InsertWindow and DeleteWindow have been removed.

TABLE OF CONTENTS

3 About This Manual
4 About the Window Manager
6 Windows and GrafPorts
6 Window Regions
8 Windows and Resources
9 Window Records
10 Window Pointers
11 The WindowRecord Data Type
13 How a Window is Drawn
15 Making a Window Active: Activate Events
16 Using the Window Manager
17 Window Manager Routines
18 Initialization and Allocation
20 Window Display
23 Mouse Location
24 Window Movement and Sizing
27 Update Region Maintenance
29 Miscellaneous Utilities
31 Low-Level Routines
32 Format of a Window Template
33 Defining Your Own Windows
35 Format of a Window Definition Function
35 The Draw Window Frame Routine
37 The Hit Routine
37 The Routine to Calculate Regions
38 The "New Window" Routine
38 The Dispose Routine
38 The Grow Routine
38 The Draw Size Box Routine
39 Notes for Assembly-Language Programmers
40 Summary of the Window Manager
43 Glossary

ABOUT THIS MANUAL

This manual describes the Window Manager, a major component of the Macintosh User Interface Toolbox. *** Eventually it will become part of a large manual describing the entire Toolbox. *** The Window Manager allows you to create, manipulate, and dispose of windows in a way that's consistent with the Macintosh User Interface Guidelines.

(hand)

> This manual describes version 7 of the ROM. If you're using a different version, the Window Manager may not work as discussed here.

Like all documentation about Toolbox units, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- The basic concepts and structures behind QuickDraw, particularly points, rectangles, regions, grafPorts, and pictures. You don't have to know the QuickDraw routines in order to use the Window Manager, though you'll be using QuickDraw to actually draw inside a window.

- Resources, as discussed in the Resource Manager manual.

- The Toolbox Event Manager. Some Window Manager routines are called only in response to certain events.

- A Macintosh application that uses windows, as an illustration of the window concepts presented here.

The manual begins with an introduction to the Window Manager and what you can do with it. It then discusses the basics about windows: the relationship between windows and grafPorts; the various regions of a window; and the relationship between windows and resources. Following this is a discussion of the window record, where the Window Manager keeps all the information it needs about a window. There are also sections on what happens when a window is drawn and when a window becomes active or inactive.

Next, a section on using the Window Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Window Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not interest all readers. The exact format of the resource data that defines a window is given, followed by special information for programmers who want to define their own windows and for those who will use the Window Manager routines from assembly language.

Finally, there's a summary of the Window Manager data structures and routine calls, for quick reference, and a glossary of terms used in this manual.

ABOUT THE WINDOW MANAGER

The Window Manager is a tool for dealing with windows on the Macintosh screen. The screen represents a working surface or desktop; graphic objects appear on the desktop and can be manipulated with the mouse. A window is an object on the desktop that presents information, such as a document or a message. Windows can be any size or shape, and there can be one or many of them, depending on aesthetics and available memory.

Some types of window are predefined for you. One of these is the standard document window, as illustrated in Figure 1. Every document window has a title bar containing a title that's centered and in the system font and system font size. In addition, a particular document window may or may not have a close box or a size box; both of these are supported by the Window Manager. There may also be scroll bars along the bottom and/or right edge of a document window. Scroll bars are controls, and are discussed in the Control Manager manual.

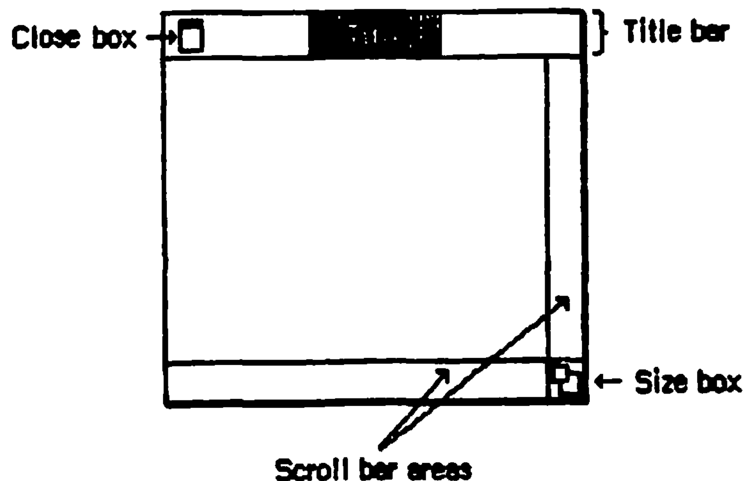


Figure 1. An Active Document Window

Your application can easily create predefined types of window such as document windows, or it can define its own types of window. Some windows may be created indirectly for you when you use other parts of the Toolbox; an example is the window the Dialog Manager creates to display an alert box. Windows created either directly or indirectly by an application are collectively called application windows. There's also a class of windows called system windows, which are not created as the result of something done by the application. Desk accessories are displayed in system windows.

The document window shown in Figure 1 above is the frontmost (active) window, the one that will be acted on when the user types, gives commands, or whatever is appropriate to the application being used. Its title is highlighted so it can be distinguished from other, inactive windows that may be on the screen. *** Method of highlighting will change. *** Since a close box or size box will perform its special function only in an active window, neither box appears at all in an inactive window (see Figure 2).

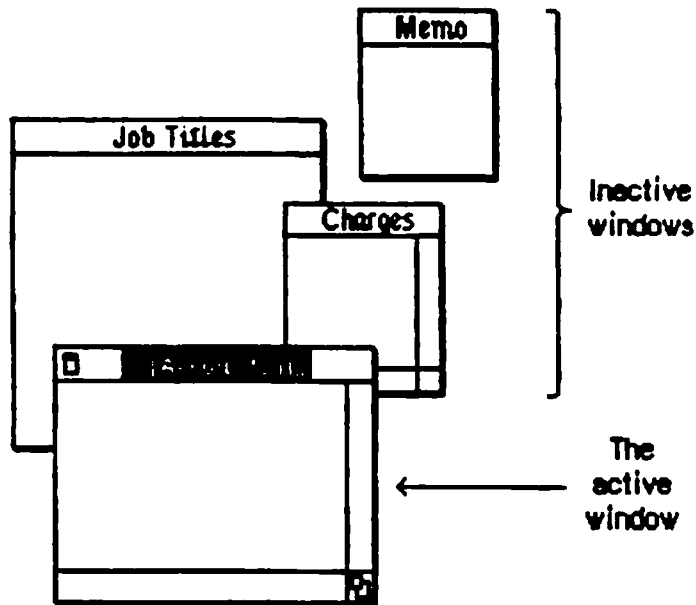


Figure 2. Overlapping Document Windows

(hand)

> If a document window has neither a size box nor scroll bars, the lines delimiting those areas aren't drawn, as in the Memo window in Figure 2.

An important function of the Window Manager is to keep track of overlapping windows. You can move windows to different places on the screen, change their plane (their front-to-back ordering), or change their size, all without concern for how the various windows overlap. The Window Manager makes sure that any newly exposed areas are redrawn, and that the application can draw into any window without running over onto windows in front of it.

Finally, the Window Manager makes it easy for you to set up your application so that mouse actions cause these standard responses inside a document window, or similar responses inside other windows:

- Clicking anywhere in an inactive window makes it the active window by bringing it to the front and highlighting its title.

- Clicking inside the close box of the active window makes the

window close (so it no longer presents information) or disappear altogether.

- Dragging anywhere inside the title bar of a window (except in the close box, if any) pulls an outline of the window across the screen, and releasing the mouse button moves the window to the new location. If the window isn't the active window, it becomes the active window unless the COMMAND key was also held down *** (key name may change) ***. A window can never be moved completely off the screen; by convention, it can't be moved such that the visible area of the title bar is less than four pixels square.

- Dragging inside the size box of the active window changes the size of the window.

WINDOWS AND GRAFPORTS

It's easy for applications to use windows: to the application, a window is a grafPort that it can draw into like any other with QuickDraw routines. When you create a window, you specify a rectangle that becomes the portRect of the grafPort in which the window contents will be drawn. The bitMap for this grafPort, its pen pattern, and other characteristics are the same as the default values set by the OpenPort routine in QuickDraw, except for the character font, which is set to the application font. These characteristics will apply whenever the application draws in the window, and they can easily be changed with QuickDraw routines (SetPort to make the grafPort the current port, and other routines as appropriate).

There is, however, more to a window than just the grafPort that the application draws in. For example, in a document window, the title bar and outline of the window are drawn by the Window Manager, not by the application. The part of a window that the Window Manager draws is called the <u>window frame</u>, since it usually surrounds the rest of the window. The Window Manager draws window frames in a grafPort that has the entire screen as its portRect and is called the <u>Window Manager port</u>.

WINDOW REGIONS

Every window has the following two regions:

- The <u>content region</u>: the area that the application draws in.

- The <u>structure region</u>: the entire window; its complete "structure" (the content region plus the window frame).

The content region lies within the rectangle you specify when you create the window (that is, the portRect of the window's grafPort); for a document window, it's the entire portRect. This is where information

is presented by the application and where the size box and scroll bars of a document window are located. Clicking in this region of an inactive window makes it the active window.

A window may also have any of the regions listed below. By convention, clicking or dragging in one of these regions causes the indicated action.

- A go-away region within the window frame. Clicking in this region of the active window makes the window close or disappear.

- A drag region within the window frame. Dragging in this region pulls an outline of the window across the screen, moves the window to a new location, and makes it the active window unless the COMMAND key was held down.

- A grow region, usually within the content region. Dragging in this region of the active window changes the size of the window. In a document window, the grow region is in the content region, but in windows of your own design it may be in either the content region or the window frame.

Figure 3 illustrates the various regions of a document window and its window frame.

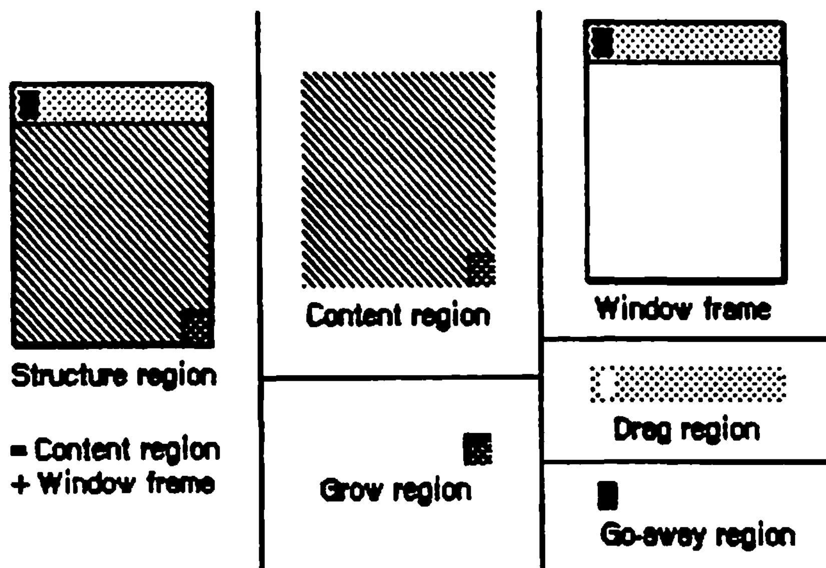


Figure 3. Document Window Regions and Frame

An example of a window that has no drag region is the window that displays an alert box. On the other hand, you can design a window whose drag region is the entire structure region and whose content region is empty; such a window might present a fixed picture rather than information that's to be manipulated.

Another important window region is the update region. The Window Manager keeps track of all areas of the content region that have to be redrawn, and accumulates them into the update region. For example, if

you bring to the front a window that was overlapped by another window, the Window Manager adds the formerly overlapped (now exposed) area of the front window's content region to its update region. The update region is maintained for the most part by the Window Manager. If a window's content region includes a size box or scroll bars, however, the application has to manipulate the update region itself to ensure that they get redrawn properly; the Window Manager provides update region maintenance routines for this purpose.

WINDOWS AND RESOURCES

The general appearance and behavior of a window is determined by a routine called its window definition function, which is stored as a resource in a resource file. Most applications will use the predefined window definition functions provided in the system resource file; some will write their own window definition functions (as described later in the section "Defining Your Own Windows").

When you create a window, you specify the type of window with a window definition ID. The window definition ID tells the Window Manager the resource ID of the definition function for this type of window, and also provides some other information. (The details are discussed in the section on defining your own windows; you don't have to know them to use the predefined window types.) The Window Manager calls the Resource Manager to read the window definition function from the resource file into memory. Later, when it needs to perform certain basic operations such as drawing the window frame, the Window Manager calls the window definition function.

You can use one of the following constants as a window definition ID to refer to a predefined type of window:

```
CONST documentProc = 0;   {standard document window}
      dBoxProc     = 1;   {alert box or modal dialog box}
      dBoxZero     = 2;   {like dBoxProc but with no shadow}
      mdBoxProc    = 3;   {modeless dialog box}  *** forthcoming ***
      rDocProc     = 16;  {desk accessory window}
```

- The dBoxProc type of window resembles an alert box or a "modal" dialog box (the kind that requires the user to respond before doing any other work on the desktop). It's a rectangular window with no go-away region, drag region, or grow region and with a two-pixel-thick "shadow".

- The dBoxZero type of window is just like the dBoxProc type except that it has no shadow.

- The mdBoxProc type of window looks like a "modeless" dialog box, the kind that lets the user work elsewhere on the desktop before responding. *** Its exact appearance has yet to be determined. ***

- The rDocProc type of window is the window used for desk accessories. It's like a document window with no grow region, with rounded corners, and with a method of highlighting that inverts the entire title bar.

Rounded-corner windows are drawn by the QuickDraw routine FrameRoundRect, which requires that the diameters of curvature be passed in its ovalWidth and ovalHeight parameters. For an rDocProc type of window, the diameters of curvature are both 16. *** A way to specify different diameters via the window definition ID is forthcoming. ***

To create a particular window, the Window Manager needs to know not only the window definition ID but also other information specific to this window, such as its title (if any), its location, and its plane. You can supply all the needed information in parameters to a Window Manager routine or, better yet, you can store it as a single resource in a resource file and pass the resource ID instead. This type of resource, which is called a window template, simplifies the process of creating a number of windows of the same type. More important, it allows you to isolate specific window descriptions from your application code. Translation of window titles into a foreign language, for example, would require only a change to the resource file.

(hand)

> You can create window templates and store them in resource files with the aid of the Resource Editor *** eventually (for now, the Resource Compiler) ***. The Resource Editor relieves you of having to know the exact format of a window template, but for interested programmers this information is given in the section "Format of a Window Template".

WINDOW RECORDS

The Window Manager keeps all the information it requires for its operations on a particular window in a window record. The window record contains the following:

- The grafPort for the window
- A handle to the window definition function
- A handle to the window's title, if any
- A handle to a list of controls, if any, in the window
- A pointer to the next window in the window list, which is a list of all windows ordered according to their front-to-back positions on the desktop

- The window class, which tells whether the window is a system window, a dialog or alert window, or a window created directly by the application

The handle to the window's title has a data type that you may want to use yourself elsewhere; it's defined in the Window Manager as follows:

```
TYPE StringPtr = ^Str255;
     StrHandle = ^StringPtr;
```

The window record also contains an indication of whether the window is currently visible. This means only that the window is drawn in its plane, not necessarily that you can see it on the screen. If, for example, it's completely overlapped by another window, it's still "visible" even though it can't be seen in its current location.

The reference value field of the window record is a 32-bit field that the application may store into and access for any purpose. For example, it might contain a handle to data associated with the window, such as a TextEdit edit record.

Finally, a window record may contain a handle to a QuickDraw picture of the window contents. The application can swap out the code and data that draw the window contents if desired, and instead use this picture. For more information, see "How a Window is Drawn".

The data type for a window record is called WindowRecord. A window record is a dynamic data structure and is referred to by a pointer, as discussed further under "Window Pointers" below. You can store into and access most of the fields of a window record with Window Manager routines, so normally you don't have to know the exact field names. Occasionally--particularly if you define your own type of window--you may need to know the exact structure; it's given below under "The WindowRecord Data Type".

Window Pointers

There are two types of pointer through which windows can be accessed: WindowPtr and WindowPeek. Most users will only need to use WindowPtr.

The Window Manager defines the following type of window pointer:

```
TYPE WindowPtr = GrafPtr;
```

It can do this because the first thing stored in a window record is the window's grafPort. This type of pointer can be used to access fields of the grafPort or can be passed to QuickDraw routines that expect pointers to grafPorts as parameters. The application might call such routines to draw into the window, and the Window Manager itself calls them to perform many of its operations. The Window Manager gets the additional information it needs from the rest of the window record beyond the grafPort.

In some cases, however, a more direct way of accessing the window record may be necessary or desirable. For this reason, the Window Manager also defines the following type of window pointer:

```
TYPE WindowPeek = ^WindowRecord;
```

Programmers who want to access WindowRecord fields directly must use this type of pointer (which derives its name from the fact that it lets you "peek" at the additional information about the window). A WindowPeek pointer is also used wherever the Window Manager will not be calling QuickDraw routines and will benefit from a more direct means of getting to the data stored in the window record.

A simple Pascal operation lets you switch from one type of window pointer to the other. For example, if wPtr is of type WindowPtr and wPeek is of type WindowPeek, you can convert from one type to the other as follows:

```
wPeek := POINTER(ORD(wPtr));  {convert from WindowPtr to WindowPeek}

wPtr := POINTER(ORD(wPeek));  {convert from WindowPeek to WindowPtr}
```

(hand)

From assembly language, of course, there's no type checking on pointers, and the two types of pointer are equal.

The WindowRecord Data Type

For those who want to know more about the data structure of a window record or who will be defining their own types of window, the exact data structure is given here.

```
TYPE WindowRecord = RECORD
                      port:          GrafPort;
                      windowKind:    INTEGER;
                      visible:       BOOLEAN;
                      hilited:       BOOLEAN;
                      goAwayFlag:    BOOLEAN;
                      spareFlag:     BOOLEAN;
                      strucRgn:      RgnHandle;
                      contRgn:       RgnHandle;
                      updateRgn:     RgnHandle;
                      windowDefProc: Handle;
                      dataHandle:    Handle;
                      titleHandle:   StringHandle;
                      titleWidth:    INTEGER;
                      controlList:   Handle;
                      nextWindow:    WindowPeek;
                      windowPic:     PicHandle;
                      refCon:        LongInt
                    END;
```

The port is the window's grafPort.

WindowKind identifies the window class. If negative, it means the window is a system window. It may also be one of the following predefined constants:

```
dialogKind = 2;     {dialog or alert window}
userKind   = 8;     {window created directly by the application}
```

WindowKind values 1 through 7 are reserved for system use. UserKind is stored in this field when a window is created directly by the application (rather than indirectly through the Dialog Manager, as for dialogKind); for such windows the application can in fact set the window class to any value greater than 8 if desired.

When visible is TRUE, the window is currently visible.

Hilited and goAwayFlag are checked by the window definition function when it draws the window frame, to determine whether the window should be highlighted and whether it should have a go-away region. For a document window, this means that if hilited is TRUE, the title of the window is highlighted, and if goAwayFlag is also TRUE, a close box appears in the highlighted title bar.

SpareFlag is reserved for future use.

StrucRgn, contRgn, and updateRgn are region handles, as defined in QuickDraw, to the structure region, content region, and update region of the window. These regions are all in global coordinates.

The windowDefProc field contains a handle to the window definition function for this type of window. This handle is returned by the Resource Manager after it reads the definition function from the resource file into memory. From the window definition ID that you provide when you create the window, the Window Manager can tell what resource ID to pass on to the Resource Manager.

(hand)

> The high-order byte of the windowDefProc field contains some additional information that the Window Manager gets from the window definition ID; for details, see the section "Defining Your Own Windows". Also note that if you write your own window definition function and won't be sharing it with other applications, you can put it in with the application code and just store a handle to it in the windowDefProc field.

DataHandle is reserved for use by the window definition function. If the window is one of your own definition, your window definition function may use this field to store and access any information it wishes. If only four or fewer bytes of information are needed, your definition function can store it directly in the dataHandle field rather than use a handle.

TitleHandle is a stringHandle to the window's title, if any.

TitleWidth is the width, in pixels, of the window's title in the system font and system font size. This width is determined by the Window Manager and is normally of no concern to the application.

ControlList is a handle to a list of controls, if any, in the window. The Control Manager is responsible for maintaining this list.

NextWindow is a pointer to the next window in the window list, that is, the window behind this window. If this window is the furthest back (with no windows between it and the desktop), nextWindow is NIL.

WindowPic is a handle to a QuickDraw picture of the window contents, or NIL if the application will draw the window contents in response to an update event, as described under "How a Window is Drawn", below.

RefCon is the window's reference value field, which the application may store into and access for any purpose.

(hand)
> Notice that the go-away, drag, and grow regions are not included in the window record. Although these are conceptually regions, they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are, and it can do so with great flexibility.

HOW A WINDOW IS DRAWN

When a window is drawn or redrawn, the following two-step process usually takes place: the Window Manager draws the window frame and the application draws the window contents.

To perform the first step of this process, the Window Manager calls the window definition function with a request that the window frame be drawn. It manipulates regions of the Window Manager port as necessary before calling the window definition function, to ensure that only what should and must be drawn is actually drawn on the screen. Depending on a parameter passed to the routine that created the window, the window definition function may or may not draw a go-away region in the window frame (a close box in the title bar, for a document window).

Usually the second step is that the Window Manager generates an update event to get the application to draw the window contents. It does this by accumulating in the update region the areas of the window's content region that need updating. The Toolbox Event Manager periodically checks to see if there's any window whose update region is not empty; if it finds one, it reports (via the GetNextEvent routine) that an update event has occurred, and passes along the window pointer in the event message. (If it finds more than one such window, it issues an

update event for the frontmost one, so that update events are reported in front-to-back order.) The application should respond as follows:

1. Call BeginUpdate, a routine that temporarily replaces the visRgn of the window's grafPort with the intersection of the visRgn and the update region.

2. Draw the window contents, entirely or in part. Normally it's more convenient to draw the entire content region, but it suffices to draw only the update region. In either case, since the visRgn is limited to where it intersects the update region, only the parts of the window that require updating will actually be drawn on the screen.

3. Call EndUpdate, which restores the normal visRgn and sets the update region to the empty region.

Figure 4 illustrates the effect of BeginUpdate and EndUpdate on the visRgn and update region of a window that's redrawn after being brought to the front.

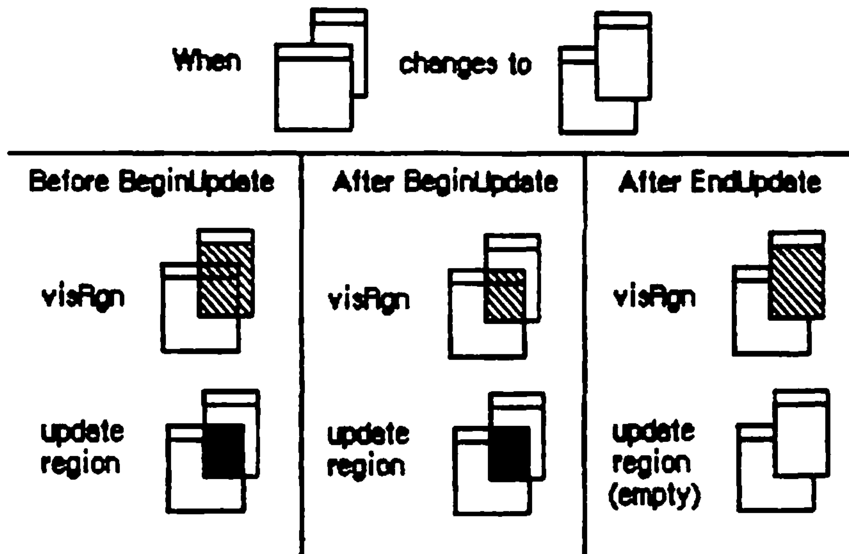


Figure 4. Updating Window Contents

(hand)

Although unlikely, it's possible that a desk accessory may not be set up to handle update events, so the application may receive an update event for a system window. For this reason, it's a good idea to check whether the window to be updated is one that was created by your application; if it's not, just ignore it.

The Window Manager allows an alternative to the update event mechanism that may be useful for simple windows: a handle to a QuickDraw picture may be stored in the window record. If this is done, the Window Manager doesn't generate an update event to get the application to draw

the window contents; instead, it calls the QuickDraw routine DrawPicture to draw the picture whose handle is stored in the window record (and it does all the necessary region manipulation). If the amount of storage occupied by the picture is less than the size of the code and data necessary to draw the window contents, and the application can swap out that code and data, this drawing method is more economical (and probably faster) than the usual updating process.

MAKING A WINDOW ACTIVE: ACTIVATE EVENTS

A number of Window Manager routines change the state of a window from inactive to active or from active to inactive. For each such change, the Window Manager generates an activate event, passing along the window pointer in the event message and, in the modifiers field of the event record, bits that indicate the following:

- Whether this window has become active (bit Ø=1) or inactive (bit Ø=Ø).

- Whether the class of the active window is changing from an application window to a system window or vice versa. (If so, bit 1=1; if no such change is happening, bit 1=Ø.)

When the Toolbox Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event on to the application (via the GetNextEvent routine). Activate events have a higher priority than any other type of event.

Usually when one window becomes active another becomes inactive, and vice versa, so activate events are most commonly generated in pairs. When this happens, the Window Manager generates first the event for the window becoming inactive, and then the event for the window becoming active. Sometimes only a single activate event is generated, such as when there's only one window in the window list. When the active window is permanently disposed of, no activate event is generated to report that it's inactive, because the window no longer exists at all.

Activate events for dialog and alert windows are handled by the Dialog Manager. In response to activate events for windows created directly by your application, you might take actions such as the following:

- In a document window containing a size box or scroll bars, erase the size box icon or scroll bars when the window becomes inactive and redraw them when it becomes active.

- In a window that contains text being edited, remove the highlighting or blinking vertical bar from the text when the window becomes inactive and restore it when the window becomes active.

- Enable or disable a menu or certain menu items as appropriate to match what the user can do when the window becomes active or

inactive.

(hand)
Like update events, activate events for system windows may be passed to your application because a desk accessory wasn't set up to handle them. Although this will rarely happen, it's a good idea to check whether the window to which the activate event applies is one that was created by your application, and ignore it if not.

USING THE WINDOW MANAGER

This section discusses how the Window Manager routines fit into the general flow of an application program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

To use the Window Manager, you must have previously called InitGraf to initialize QuickDraw and InitFonts to initialize the Font Manager. The first Window Manager routine to call is the initialization routine InitWindows, which draws the desktop and the (empty) menu bar.

Where appropriate in your program, use NewWindow or GetNewWindow to create any windows you need; these functions return a window pointer, which you can then use to refer to the window. NewWindow takes descriptive information about the window from its parameters, whereas GetNewWindow gets the information from window templates in a resource file. You can supply a pointer to the storage for the window record or let it be allocated by the routine creating the window; when you no longer need a window, call CloseWindow if you supplied the storage, or DisposeWindow if not.

When the Event Manager reports that an update event has occurred, call BeginUpdate, draw the update region or the entire content region, and call EndUpdate (see "How a Window is Drawn", above). You can also use InvalRect or InvalRgn to prepare a window for updating, and ValidRect or ValidRgn to temporarily protect portions of the window from updating.

When drawing the contents of a window that contains a size box in its content region, you'll draw the size box if the window is active or just the lines delimiting the size box and scroll bar areas if it's inactive. The FrontWindow function tells you which is the active window; the DrawGrowIcon procedure helps you draw the size box or delimiting lines. You'll also call the latter procedure when an activate event occurs that makes the window active or inactive.

(hand)
To be safe, it's a good idea to check that an update or activate event received by your application applies to one of its own windows and not a system window.

When a mouse down event occurs, call the FindWindow function to find out which part of which window the mouse button was pressed in.

- If it was pressed in the content region of an inactive window, make that window the active window by calling SelectWindow.

- If it was pressed in the grow region of the active window, call GrowWindow to pull around an image that shows the window's size will change, and then SizeWindow to actually change the size.

- If it pressed in the drag region of any window, call DragWindow, which will pull an outline of the window across the screen, move the window to a new location, and, if the window is inactive, make it the active window (unless the COMMAND key was held down).

- If it was pressed in the go-away region of the active window, call TrackGoAway to handle highlighting of the go-away region and to determine whether the mouse is inside the region when the button is released. Then do whatever is appropriate as a response to this mouse action in the particular application. For example, call CloseWindow or DisposeWindow if you want the window to go away permanently, or HideWindow if you want it to disappear temporarily.

(hand)

> If the mouse button was pressed in the content region of an active window (but not in the grow region), call the Control Manager routine FindControl if the window contains controls. If it was pressed in a system window, call the Desk Manager routine SystemClick. See the Control Manager and Desk Manager manuals for details.

The procedure that simply moves a window without pulling around an outline of it, MoveWindow, can be called at any time, as can SizeWindow--though the application should not surprise the user by taking these actions unexpectedly. There are also routines for changing the title of a window, placing a window behind another window, and making a window visible or invisible. Call these Window Manager routines wherever needed in your program.

WINDOW MANAGER ROUTINES

This section describes first the Window Manager procedures and functions that are used in most applications, and then the low-level routines for use by software developers who have their own ideas about what to do with windows. The routines are presented in their Pascal form; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see "Using QuickDraw from Assembly Language" in the QuickDraw manual and also "Notes for Assembly-Language Programmers" in this manual. ***

Initialization and Allocation

```
PROCEDURE InitWindows;
```

InitWindows initializes the Window Manager. It creates the Window Manager port; you can get a pointer to this port with the GetWMgrPort procedure (below). InitWindows draws the gray desktop with rounded corners, and a white menu bar with a black line underneath. Call this procedure once before all other Window Manager routines.

```
PROCEDURE GetWMgrPort (VAR wPort: GrafPtr);
```

GetWMgrPort returns in wPort a pointer to the Window Manager port.

(hand)
> Assembly-language programmers can access this pointer through the global variable wMgrPort.

```
FUNCTION NewWindow (wStorage: Ptr; boundsRect: Rect; title: Str255;
           visible: BOOLEAN; procID: INTEGER; behind: WindowPtr;
           goAwayFlag: BOOLEAN; refCon: LongInt) : WindowPtr;
```

NewWindow creates a window as specified by its parameters, adds it to the window list, and returns a windowPtr to the new window. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

WStorage is a pointer to the storage to use for the window record. For example, if you've declared the variable wRecord of type WindowRecord, you can pass @wRecord as the first parameter to NewWindow. If you pass NIL for wStorage, the window record will be allocated on the heap; this is not recommended except for programs that have an unusually large amount of memory available or have been set up to dispose of windows dynamically.

BoundsRect, a rectangle given in global coordinates, determines the window's size and location. It becomes the portRect of the window's grafPort; note, however, that the portRect is in local coordinates.

(hand)
> The bitMap, pen pattern, and other characteristics of the window's grafPort are the same as the default values set by the OpenPort routine in QuickDraw, except for the character font, which is set to the application font rather than the system font.

Title is the window's title, which appears centered and in the system font and system font size in the title bar of a document window. If the title of a document window is longer than will fit in the title

bar, only as much of the beginning of the title as will fit is displayed.

If the visible parameter is TRUE, NewWindow draws the window. First it calls the window definition function to draw the window frame; if goAwayFlag is also TRUE and the window is frontmost (as specified by the behind parameter, below), it draws a go-away region in the frame. Then it generates an update event for the entire window contents.

ProcID is the window definition ID, which leads to the window definition function for this type of window. The window definition IDs for the predefined types of windows are listed in the "Windows and Resources" section. Window definition IDs for windows of your own design are discussed in the section "Defining Your Own Windows".

The behind parameter determines the window's plane. The new window is inserted in back of the window pointed to by this parameter. To put the new window behind all other windows, use behind=NIL. To place it in front of all other windows, use behind=POINTER(-1); in this case, NewWindow will unhighlight the previously active window, highlight the window being created, and generate appropriate activate events.

RefCon is the window's reference value, set and used only by the application.

NewWindow also sets the window class in the window record to indicate that the window was created directly by the application.

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

Like NewWindow (above), GetNewWindow creates a window as specified by its parameters, adds it to the window list, and returns a windowPtr to the new window. The only difference between the two functions is that instead of having the parameters boundsRect, title, visible, procID, goAwayFlag, and refCon, GetNewWindow has a single windowID parameter, where windowID is the resource ID of a window template that supplies the same information as those parameters. The wStorage and behind parameters of GetNewWindow have the same meaning as in NewWindow.

```
PROCEDURE CloseWindow (theWindow: WindowPtr);
```

CloseWindow removes the given window from the screen and deletes it from the window list. It returns to the heap the storage used by all data structures associated with the window, but does not dispose of the window record itself. Call this procedure when you're done with a window if you supplied NewWindow or GetNewWindow a pointer to the window storage (in the wStorage parameter) when you created the window.

Any update events for the window are discarded. If the window was the frontmost window and there was another window behind it, the latter window is highlighted and an appropriate activate event is generated.

PROCEDURE DisposeWindow (theWindow: WindowPtr);

DisposeWindow removes the given window from the screen, deletes it from the window list, and disposes of the window record. It returns to the heap all data structures associated with the window. Call this procedure when you're done with a window if you let the window record be allocated on the heap when you created the window (by passing NIL as the wStorage parameter to NewWindow or GetNewWindow).

Any update events for the window are discarded. If the window was the frontmost window and there was another window behind it, the latter window is highlighted and an appropriate activate event is generated.

(hand)

The macro you invoke to call this routine from assembly language is named _DisposWindow.

Window Display

These procedures affect the appearance or plane of a window but not its size or location.

PROCEDURE SetWTitle (theWindow: WindowPtr; title: Str255);

SetWTitle changes the title of theWindow to the given title, performing any necessary redrawing of the window frame. If the new title of a document window is longer than will fit in the title bar, only as much of the beginning of the title as will fit is displayed.

(hand)

In a document window, the title is centered in the title bar if it fits. If it doesn't fit, it's left-justified (against the close box, if any, leaving a small amount of space between the close box and the beginning of the title).

PROCEDURE GetWTitle (theWindow: WindowPtr; VAR title: Str255);

GetWTitle returns the title of theWindow.

PROCEDURE SelectWindow (theWindow: WindowPtr);

SelectWindow makes theWindow the active window as follows: it unhighlights the previously active window, brings theWindow in front of all other windows, highlights theWindow, and generates the appropriate activate events. Call this procedure if there's a mouse down event in the content region of an inactive window.

PROCEDURE HideWindow (theWindow: WindowPtr);

HideWindow makes theWindow invisible if it isn't already invisible and has no effect if it is already invisible. If theWindow is the frontmost window and there's a window behind it, HideWindow also unhighlights theWindow, brings the window behind it to the front, highlights that window, and generates appropriate activate events (see Figure 5).

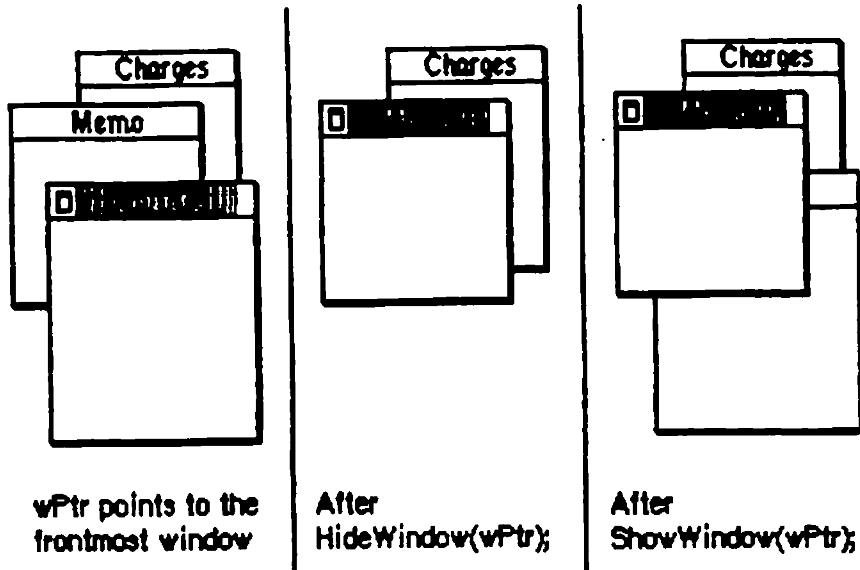


Figure 5. Hiding and Showing Document Windows

PROCEDURE ShowWindow (theWindow: WindowPtr);

ShowWindow makes theWindow visible if it's not already visible and has no effect if it is already visible. It does not change the front-to-back ordering of the windows. Remember that if you previously hid the frontmost window with HideWindow, HideWindow will have brought the window behind it to the front; so if you then do a ShowWindow of the window you hid, it will no longer be frontmost (see Figure 5 above).

(hand)
Although it's inadvisable, you can create a situation where the frontmost window is invisible. If you do a ShowWindow of such a window, it will highlight the window if it's not already highlighted and will generate an activate event to force this window from inactive to active.

PROCEDURE ShowHide (theWindow: WindowPtr; showFlag: BOOLEAN);

If showFlag is FALSE, ShowHide makes theWindow invisible if it's not already invisible and has no effect if it is already invisible. If

showFlag is TRUE, ShowHide makes theWindow visible if it's not already visible and has no effect if it is already visible. Unlike HideWindow and ShowWindow, ShowHide never changes the highlighting or front-to-back ordering of windows or generates activate events.

(eye)

Use this procedure carefully, and only in special circumstances where you need more control than allowed by HideWindow and ShowWindow.

PROCEDURE HiliteWindow (theWindow: WindowPtr; fHilite: BOOLEAN);

If fHilite is TRUE, this procedure highlights theWindow if it's not already highlighted and has no effect if it is highlighted. If fHilite is FALSE, HiliteWindow unhighlights theWindow if it is highlighted and has no effect if it's not highlighted. The exact way a window is highlighted depends on its window definition function; a document window's definition function highlights only the window's title.

Normally your application won't have to call this procedure, since it should call SelectWindow to make a window active, and SelectWindow takes care of the necessary highlighting changes. Highlighting a window that isn't the active window is contrary to the Macintosh User Interface Guidelines.

PROCEDURE BringToFront (theWindow: WindowPtr);

BringToFront brings theWindow to the front of all other windows and redraws the window as necessary. Normally your application won't have to call this procedure, since it should call SelectWindow to make a window active, and SelectWindow takes care of bringing the window to the front. If you do call BringToFront, however, remember to call HiliteWindow to make the necessary highlighting changes.

PROCEDURE SendBehind (theWindow: WindowPtr; behindWindow: WindowPtr);

SendBehind sends theWindow behind behindWindow, redrawing any exposed windows. If behindWindow is NIL, it sends theWindow behind all other windows. If theWindow is the active window, it unhighlights theWindow, highlights the new active window, and generates the appropriate activate events.

(eye)

Do not use SendBehind to deactivate a previously active window. Calling SelectWindow to make a window active takes care of deactivating the previously active window.

```
FUNCTION FrontWindow : WindowPtr;
```

FrontWindow returns a pointer to the first visible window in the window list (that is, the active window).

```
PROCEDURE DrawGrowIcon (theWindow: WindowPtr);
```

Call this procedure in response to an update or activate event involving a window that contains a size box in its content region. If theWindow is active (highlighted), DrawGrowIcon draws the size box; otherwise, it draws whatever is appropriate to show that the window temporarily cannot be sized. The exact appearance and location of what's drawn depend on the window definition function. For an active document window, DrawGrowIcon draws the size box icon in the bottom right corner of the portRect of the window's grafPort, along with the lines delimiting the size box and scroll bar areas (16 pixels in from the right edge and bottom of the portRect). It doesn't erase the scroll bar areas, so if the window doesn't contain scroll bars you should erase those areas yourself after the window's size changes. For an inactive document window, DrawDocGrow draws only the delimiting lines (again, without erasing anything).

Mouse Location

```
FUNCTION FindWindow (thePt: Point; VAR whichWindow: WindowPtr) :
            INTEGER;
```

When a mouse down event occurs, the application should call FindWindow with thePt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). FindWindow tells which part of which window, if any, the mouse button was pressed in. If it was pressed in a window, whichWindow is set to the window pointer; otherwise, it's set to NIL. The integer returned by FindWindow is one of the following predefined constants:

```
inDesk      = 0;  {none of the following}
inMenuBar   = 1;  {in the menu bar}
inSysWindow = 2;  {in a system window}
inContent   = 3;  {in the content region (except grow, if active)}
inDrag      = 4;  {in the drag region}
inGrow      = 5;  {in the grow region (active window only)}
inGoAway    = 6;  {in the go-away region (active window only)}
```

Usually inDesk means that the mouse button was pressed on the desktop, outside the menu bar or any windows; however, it may also mean that the mouse button was pressed inside a window frame but not in the drag region or go-away region of the window. Usually one of the last four values is returned for windows created by the application.

(eye)
If the window is a document window that doesn't contain a size box, the application should treat inGrow the same as inContent; if it's a document window that has no close box, FindWindow will never return inGoAway for that window.

FUNCTION TrackGoAway (theWindow: WindowPtr; thePt: Point) : BOOLEAN;

When there's a mouse down event in the go-away region of theWindow, the application should call TrackGoAway with thePt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). TrackGoAway keeps control until the mouse button is released, highlighting the go-away region as long as the mouse position remains inside it, and restoring the region to normal when the mouse moves outside it. The exact way a window's go-away region is highlighted depends on its window definition function; the highlighting of a document window's close box is illustrated in Figure 6. *** This method of highlighting may change. *** When the mouse button is released, TrackGoAway leaves the go-away region in its normal state and returns TRUE if the mouse is inside the go-away region or FALSE if it's outside the region.

Normal close box

Highlighted close box

Figure 6. A Document Window's Close Box

Window Movement and Sizing

PROCEDURE MoveWindow (theWindow: WindowPtr; hGlobal,vGlobal: INTEGER;
front: BOOLEAN);

MoveWindow moves theWindow to another part of the screen, without affecting its size or plane. The top left corner of the portRect of the window's grafPort is moved to the screen point indicated by the global coordinates hGlobal and vGlobal. If the front parameter is TRUE and theWindow isn't the active window, MoveWindow makes it the active window by calling SelectWindow(theWindow).

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

When there's a mouse down event in the drag region of theWindow, the application should call DragWindow with startPt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). DragWindow pulls a gray outline of theWindow around, following the path of the mouse until the button is released. When the mouse button is released, DragWindow moves theWindow to the location to which it was dragged (by calling MoveWindow). If theWindow is not the active window and the COMMAND key was not being held down, DragWindow makes it the active window (by passing TRUE for the front parameter when calling MoveWindow).

If the mouse button is released when the mouse position is outside the limits of boundsRect, a rectangle given in global coordinates, DragWindow returns without moving theWindow or making it the active window. Typically boundsRect will be (4,24,508,338), which is four pixels in from the menu bar and from the other edges of the screen; this ensures that there won't be less than a four-pixel-square area of the title bar visible on the screen.

FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point; sizeRect: Rect) : LongInt;

When there's a mouse down event in the grow region of theWindow, the application should call GrowWindow with startPt equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). GrowWindow pulls a grow image of the window around, following the path of the mouse until the button is released. The grow image for a document window is a gray outline of the entire window and also the lines delimiting the title bar, size box, and scroll bar areas; Figure 7 illustrates this for a document window containing a size box and scroll bars, but the grow image would be the same even if the window contained no size box, one scroll bar, or no scroll bars. In general, the grow image is defined in the window definition function and is whatever is appropriate to show that the window's size will change.

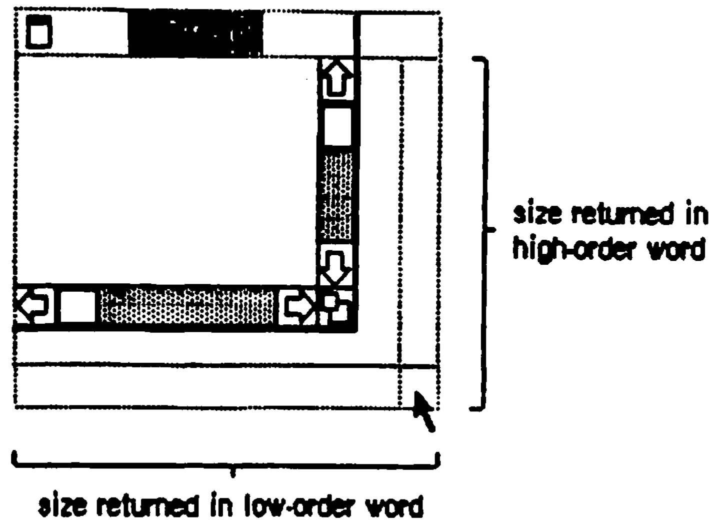


Figure 7. GrowWindow Operation on a Document Window

The application should subsequently call SizeWindow (see below) to change the portRect of the window's grafPort to the new one outlined by the grow image. The sizeRect parameter specifies limits, in pixels, on the vertical and horizontal measurements of what will be the new portRect. SizeRect.top is the minimum vertical measurement, sizeRect.left is the minimum horizontal measurement, sizeRect.bottom is the maximum vertical measurement, and sizeRect.right is the maximum horizontal measurement.

GrowWindow returns the actual size for the new portRect as outlined by the grow image when the mouse button is released. The high-order word of the LongInt is the vertical measurement in pixels and the low-order word is the horizontal measurement. A return value of Ø indicates that the size is the same as that of the current portRect.

(hand)

> The Toolbox Utility function HiWord takes a long integer as a parameter and returns an integer equal to its high-order word; the function LoWord returns the low-order word.

```
PROCEDURE SizeWindow (theWindow: WindowPtr; w,h: INTEGER; fUpdate:
          BOOLEAN);
```

SizeWindow enlarges or shrinks the portRect of theWindow's grafPort to the width and height specified by w and h, or does nothing if w and h are Ø. The window's position on the screen does not change. The new window frame is drawn; if the width of a document window changes, the title is again centered in the title bar, or is truncated at its end if it no longer fits. If fUpdate is TRUE, SizeWindow accumulates any newly created area of the content region into the update region (see Figure 8); normally this is what you'll want. If you pass FALSE for

fUpdate, you're responsible for the update region maintenance yourself. For more information, see InvalRect and ValidRect below.

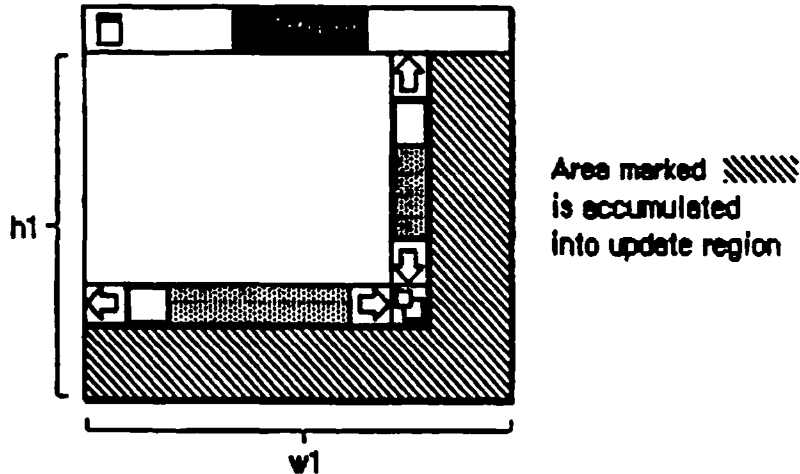


Figure 8. SizeWindow Operation on a Document Window

(eye)

You should change the window's size only when the user has done something specific to make it change.

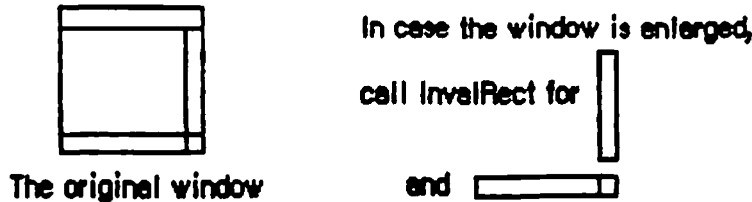
Update Region Maintenance

PROCEDURE InvalRect (badRect: Rect);

InvalRect tells the Window Manager that a rectangle within a window has changed and must be updated (that is, accumulated into the window's update region). The specified rectangle, which is given in local coordinates, lies within the content region of the window whose grafPort is the current port.

Normally the Window Manager keeps track of what has to be updated and so there's no need to use this procedure. One case where it's useful is when you're calling SizeWindow (described above) for a document window that contains a size box or scroll bars. Suppose you're going to call SizeWindow with fUpdate=TRUE. If the window is enlarged as shown in Figure 8 above, you'll want not only the newly created part of the content region to be updated, but also the two rectangular areas containing the (former) size box and scroll bars; before calling SizeWindow, you can call InvalRect twice to accumulate those areas into the update region. In case the window is made smaller, you'll want the new size box and scroll bar areas to be updated, and so can similarly call InvalRect for those areas after calling SizeWindow. See Figure 9 for an illustration of this type of update region maintenance.

Before SizeWindow with fUpdate=TRUE:



After SizeWindow:

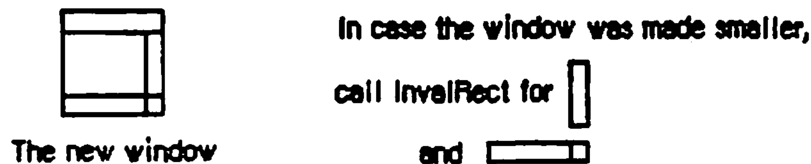


Figure 9. Update Region Maintenance with InvalRect

As another example, suppose the application uses a QuickDraw routine to scroll up text in a document window and wants to show new text added at the bottom of the window. It can use a drawing routine to draw the window, or it can instead cause only the added text to be redrawn, by accumulating that area into the update region with InvalRect.

PROCEDURE InvalRgn (badRgn: RgnHandle);

InvalRgn is the same as InvalRect (see above) but for a region that has changed rather than a rectangle.

PROCEDURE ValidRect (goodRect: Rect);

ValidRect tells the Window Manager that the application has already drawn a rectangle within a window and to cancel any updates accumulated for that area (that is, remove goodRect from the window's update region). The specified rectangle, which is given in local coordinates, lies within the content region of the window whose grafPort is the current port. Using ValidRect results in better performance and less redundant redrawing in the window.

For example, suppose you've called SizeWindow (described above) with fUpdate=TRUE for a document window that contains a size box or scroll bars. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling SizeWindow, you can redraw the size box or scroll bars immediately and then call ValidRect for the areas they occupy in case they were in fact accumulated into the update region; this will avoid redundant drawing.

PROCEDURE ValidRgn (goodRgn: RgnHandle);

ValidRgn is the same as ValidRect (see above) but for a region that has been drawn rather than a rectangle.

PROCEDURE BeginUpdate (theWindow: WindowPtr);

When an update event occurs for theWindow, call BeginUpdate to replace the visRgn of the window's grafPort with the intersection of the visRgn and the update region. You would then usually draw the entire content region, though it suffices to draw only the update region; in either case, only the parts of the window that require updating will actually be drawn on the screen. Every call to BeginUpdate must be balanced by a call to EndUpdate (see below, and see "How a Window is Drawn").

PROCEDURE EndUpdate (theWindow: WindowPtr);

Call EndUpdate to restore the normal visRgn of theWindow's grafPort, which was changed by BeginUpdate as described above. EndUpdate also sets theWindow's update region to the empty region.

Miscellaneous Utilities

PROCEDURE SetWRefCon (theWindow: WindowPtr; data: LongInt);

SetWRefCon changes the reference value associated with theWindow to the given data.

FUNCTION GetWRefCon (theWindow: WindowPtr) : LongInt;

GetWRefCon returns the reference value associated with theWindow.

PROCEDURE SetWindowPic (theWindow: WindowPtr; pic: PicHandle);

SetWindowPic stores the given picture handle in the window record for theWindow, so that when theWindow's contents are to be drawn the Window Manager will draw this picture rather than generate an update event.

FUNCTION GetWindowPic (theWindow: WindowPtr) : PicHandle;

GetWindowPic returns the handle to the picture that draws theWindow's contents, previously stored with SetWindowPic (above).

FUNCTION PinRect (theRect: Rect; thePt: Point) : LongInt;

PinRect "pins" thePt inside theRect: The high-order word of the value returned is the vertical coordinate of thePt or, if thePt lies to the left or the right of theRect, the vertical coordinate of the left or right edge of theRect, respectively. The low-order word of the value returned is the horizontal coordinate of thePt or, if thePt lies above or below theRect, the horizontal coordinate of the top or bottom of theRect.

```
FUNCTION DragGrayRgn (theRgn: RgnHandle; startPt: Point;
            limitRect,slopRect: Rect; axis: INTEGER; actionProc:
            ProcPtr) : LongInt;
```

Called when the mouse button is down inside theRgn, DragGrayRgn pulls a gray outline of the region around, following the path of the mouse until the button is released. DragWindow calls this function before actually moving the window, and the Control Manager routine DragControl similarly calls it for controls. You can call it yourself to pull around the outline of any region, and then use the information it returns to determine where to move the region.

The startPt parameter is assumed to be the point where the mouse button was originally pressed, in the local coordinates of the current grafPort. The high-order word of the value returned by DragGrayRgn contains the vertical coordinate of the ending mouse point minus that of the original point; the low-order word contains the difference between the horizontal coordinates.

LimitRect and slopRect should also be in the local coordinates of the current grafPort. LimitRect limits the travel of the region's outline; DragGrayRgn will never move the mouse position outside this rectangle. If the mouse button is released outside limitRect, DragGrayRgn returns -32768 (hexadecimal 8000). SlopRect allows the user some "slop" in moving the mouse; it should completely enclose limitRect. DragGrayRgn's behavior while tracking the mouse depends on the position of the mouse with respect to these two rectangles.

- When the mouse is inside limitRect, the region's outline follows it normally. If the mouse button is released there, the region should be moved to the mouse position.

- When the mouse is outside limitRect but inside slopRect, the outline "pins" at the edge of limitRect. If the mouse button is released there, the region should be moved to this "pinned" location.

- When the mouse is outside slopRect, the outline disappears from the screen, but DragGrayRgn continues to follow the mouse; if it moves back into slopRect, the outline reappears. If the mouse button is released outside slopRect, the region should not be moved from its original position.

The axis parameter allows you to constrain the outline's motion to only one axis:

Axis parameter	Meaning
0	No constraint
1	Horizontal motion only
2	Vertical motion only

If an axis constraint is in effect, the outline will follow the mouse's movements along the specified axis only, ignoring motion along the other axis. With or without an axis constraint, the mouse must still be inside the slop rectangle for the outline to appear at all.

The actionProc parameter is a pointer to a procedure that defines some action to be performed repeatedly for as long as the user holds down the mouse button; the procedure should have no parameters. If actionProc is NIL, DragGrayRgn simply retains control until the mouse button is released, performing no action while the mouse button is down.

(hand)
Assembly-language programmers who want the region's outline to be drawn in a pattern other than gray can store the pattern in the low-memory global dragPattern and call the above function at the entry point DragTheRgn.

Low-Level Routines

These low-level routines are not normally used by an application but may be of interest to advanced programmers.

```
FUNCTION CheckUpdate (VAR theEvent: EventRecord) : BOOLEAN;
```

CheckUpdate is called by the Toolbox Event Manager. From the front to the back in the window list, it looks for a visible window that needs updating (that is, whose update region is not empty). If it finds one whose window record contains a picture handle, it draws the picture (doing all the necessary region manipulation) and looks for the next visible window that needs updating. If it ever finds one whose window record doesn't contain a picture handle, it stores an update event for that window in theEvent and returns TRUE. If it never finds such a window, it returns FALSE.

```
PROCEDURE ClipAbove (window: WindowPeek);
```

ClipAbove sets the clipRgn of the Window Manager port to be the grayRgn (that is, the desktop) intersected with the current clipRgn, minus the structure regions of all the windows above the given window.

PROCEDURE PaintOne (window: WindowPeek; clobbered: RgnHandle);

PaintOne paints the given window, clipped to the clobbered region and all windows above it. If some content is exposed, PaintOne erases it and adds it to the update region. If the window is NIL, it's painted gray (it's the desktop). This procedure generates update events as appropriate.

PROCEDURE PaintBehind (startWindow: WindowPeek; clobbered: RgnHandle);

PaintBehind calls PaintOne (above) to paint startWindow and all the windows behind startWindow, clipped to the clobbered region.

PROCEDURE SaveOld (window: WindowPeek);

SaveOld saves the given window's current structure region and content region for the DrawNew operation (see below). It must be followed by a call to DrawNew. Note that SaveOld and DrawNew are NOT nestable.

PROCEDURE DrawNew (window: WindowPeek; update: BOOLEAN);

DrawWindow is called after SaveOld (above). It updates the area clobbered := (oldStruct XOR newStruct) UNION (oldContent XOR newContent). If update is TRUE, updates are accumulated.

PROCEDURE CalcVis (window: WindowPeek);

CalcVis calculates the visRgn of the given window by starting with its content region and subtracting the structure region of each window in front of it.

PROCEDURE CalcVisBehind (startWindow: WindowPeek; clobbered: RgnHandle);

CalcVisBehind calculates the visRgns of startWindow and all windows behind startWindow that intersect with the clobbered region. It's called after PaintBehind (see above).

(hand)
The macro you invoke to call this routine from assembly language is named _CalcVBehind.

FORMAT OF A WINDOW TEMPLATE

As described above, the GetNewWindow function takes the resource ID of a window template as a parameter, and gets from that template the same information that the NewWindow function gets from six of its

parameters. The resource type for a window template is 'WIND', and the resource data has the following format:

Number of bytes	Contents
8 bytes	Same as boundsRect parameter to NewWindow
2 bytes	Same as procID parameter to NewWindow
2 bytes	Same as visible parameter to NewWindow
2 bytes	Same as goAwayFlag parameter to NewWindow
4 bytes	Same as refCon parameter to NewWindow
n bytes	Same as title parameter to NewWindow (1-byte length in bytes, followed by the characters of the title)

DEFINING YOUR OWN WINDOWS

Certain types of window, such as the standard document window, are predefined for you. However, you may want to define your own type of window--maybe a round or hexagon-shaped window, or even a window shaped like an apple. QuickDraw and the Window Manager make it possible for you to do this.

(hand)
For the convenience of the application's end user, remember to conform to the Macintosh User Interface Guidelines for windows as much as possible.

To define your own type of window, you must write a window definition function. Usually you'll store the definition function in a resource file. When you create a window, you provide a window definition ID, which leads to the window definition function. The window definition function contains routines that define the window by performing basic operations such as drawing the window frame. When the Window Manager needs to perform one of these operations, it calls the window definition function with a parameter that identifies the operation, and the window definition function in turn takes the appropriate action.

The window definition ID contains the resource ID of the window definition function in its upper 12 bits and a variation code in its lower four bits. The variation code allows a single window definition function to implement several related types of window as "variations on a theme". For example, the dBoxProc type of window is a variation of the standard document window; both use the window definition function whose resource ID is Ø, but the document window has a variation code of Ø while the dBoxProc window has a variation code of 1.

For a given resource ID and variation code, then, the window definition ID is:

16 * resource ID + variation code

The resource ID numbers 0 through 8 are reserved for predefined window definition functions in the system resource file. Unless you want to override one of the predefined functions, the resource ID you choose for your own window definition function should be greater than 8.

The resource type for window definition functions is 'WDEF'. The Dialog Manager calls the Resource Manager to access the resource of type 'WDEF' that has the given resource ID. The Resource Manager reads the window definition function into memory and returns a handle to it. The Window Manager stores the handle in the windowDefProc field of the window record and stores the variation code in the high-order byte of that field. Later, when it needs to call the window definition function, it passes the variation code as a parameter. Figure 10 illustrates this process.

Window definition ID supplied when window is created:

Resource Manager call made by Window Manager:

defHandle = GetResource ('WDEF', resourceID);

Field in window record:

passed to window definition function

Figure 10. Window Definition Handling

(hand)

> If you won't be sharing your window definition function with other applications, you may want to store it in with the application code rather than as a separate resource. When creating the window, you would give the window definition ID of any standard type of window and specify that the window not be made visible. Then you would replace the contents of the windowDefProc field with the handle (and variation code, if any) for your window definition function.

The resource data for a window definition function is simply the assembled code of the function, which may be written in Pascal or assembly language; the only requirement is that its entry point must be at the beginning.

Format of a Window Definition Function

You may choose any name you wish for the window definition function. Here's how you would declare one named MyOwnWindow:

```
FUNCTION MyOwnWindow (varCode: INTEGER; window: WindowPtr;
          message: WindowMessage; param: LongInt) : LongInt;
```

VarCode is the variation code, as described above.

The window parameter indicates the window that the operation will affect. If the window definition function needs to use a WindowPeek type of pointer more than a WindowPtr, you can simply specify WindowPeek instead of WindowPtr in the function declaration.

The message parameter identifies the operation.

```
TYPE WindowMessage = (wDraw, wHit, wCalcRgns, wNew, wDispose,
      wGrow, wDrawGIcon);
```

Message	Operation
wDraw	Draw the window frame
wHit	Tell what region the mouse button was pressed in
wCalcRgns	Calculate the strucRgn and contRgn
wNew	Do any special window initialization
wDispose	Take any special actions when the window is disposed of
wGrow	Draw the window's grow image
wDrawGIcon	Draw the window's size box in its content region

As described below in the explanations of the routines that perform these operations, the value passed for param, the last parameter of the window definition function, depends on the operation. Where it's not mentioned below, this parameter is ignored. Similarly, the window definition function is expected to return a value only where indicated; in other cases, the function should return 0.

(hand)

"Routine" here does not necessarily mean a procedure or function. While it's a good idea to set these up as subprograms inside the window definition function, you're not required to do so.

The Draw Window Frame Routine

When the window definition function receives a wDraw message, it should draw the window frame in the current grafPort, which will be the Window Manager port. (For details on drawing, see the QuickDraw manual.)

(eye)

Do not change the visRgn or clipRgn of the Window Manager port, or overlapping windows may not be handled properly.

This routine should make certain checks to determine exactly what it should do. If the visible field in the window record is FALSE, the routine should do nothing; otherwise, it should examine the value of param received by the window definition function, as described below.

If param is Ø, the routine should draw the entire window frame. If the hilited field in the window record is TRUE, the window frame should be highlighted in whatever way is appropriate to show that this is the active window. If goAwayFlag in the window record is also TRUE, the highlighted window frame should include a go-away region; this is useful when you want to define a window such that a particular window of that type may or may not have a go-away region, depending on the situation.

Special action should be taken if the value of param is wInGoAway (a predefined constant, equal to 4, which is one of those returned by the hit routine as described below). If param is wInGoAway, the routine should do nothing but "toggle" the state of the window's go-away region from normal to highlighted or, if it's already highlighted, from highlighted to normal. The highlighting should be whatever is appropriate to show that the mouse button has been pressed inside the region. Simple inverse highlighting may be used or, as in document windows, the appearance of the region may change considerably. In the latter case, the routine should use a "mask" consisting of the normal state of the region Xor'ed with its highlighted state (where Xor stands for the logical operation "exclusive or"). When such a mask is itself Xor'ed with either state of the region, the result is the other state; Figure 11 illustrates this.

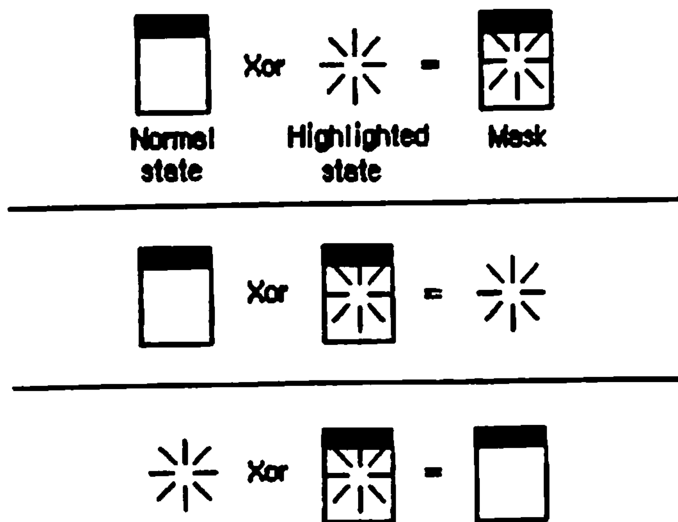


Figure 11. Toggling the Go-Away Region

Typically the window frame will include the window's title, which should be in the system font and system font size for consistency with the Macintosh User Interface Guidelines. The Window Manager port will already be set to use the system font and system font size.

(hand)
Nothing drawn outside the window's structure region will be visible.

The Hit Routine

When the window definition function receives a wHit message, it also receives as its param value the point where the mouse button was pressed. This point is given in global coordinates, with the vertical coordinate in the high-order word of the LongInt and the horizontal coordinate in the low-order word. The window definition function should determine where the mouse button "hit" and should return one of these predefined constants:

```
wNoHit     = 0;  {none of the following}
wInContent = 1;  {in the content region (except grow, if active)}
wInDrag    = 2;  {in the drag region}
wInGrow    = 3;  {in the grow region (active window only)}
wInGoAway  = 4;  {in the go-away region (active window only)}
```

Usually, wNoHit means the given point isn't anywhere within the window, but this is not necessarily so. For example, the document window's hit routine returns wNoHit if the point is in the window frame but not in the title bar.

The constants wInGrow and wInGoAway should be returned only if the window is active, since the size box and go-away region won't be drawn if the window is inactive. In an inactive document window, if the mouse button is pressed where the close box would be if the window were active, the hit routine returns wInDrag.

Of the regions that may have been hit, only the content region necessarily has the structure of a region and is included in the window record. The hit routine can determine in any way it likes whether the drag, grow, or go-away region has been hit. It can, for example, simply compare the coordinates of the given point to the coordinates of the points that delimit a particular region. Or the application can use the formal region data structure if desired, and point at it through the dataHandle field of the window record.

The Routine to Calculate Regions

The routine executed in response to a wCalcRgns message should calculate the window's structure region and content region based on the current grafPort's portRect. These regions, whose handles are in the strucRgn and contRgn fields, are in global coordinates. The Window Manager will request this operation only if the window is visible.

(hand)
When you calculate regions for your own type of window, do not alter the clipRgn or the visRgn of the window's grafPort. The Window Manager and QuickDraw take care of

this for you. Altering the clipRgn or visRgn may result in damage to other windows.

The New Window Routine

A wNew message tells the window definition function to execute a "new window" routine that does any special initialization which may be required when the window is created. For example, if the content region is unusually shaped, it might allocate space for the region and store the region handle in the dataHandle field of the window record. The "new window" routine for a document window does nothing.

The Dispose Routine

The routine executed in response to a wDispose message should take any special actions that may be required when the window is disposed of (with the Window Manager routine CloseWindow or DisposeWindow). It might, for example, deallocate space that was allocated by the "new window" routine. The dispose routine for a document window does nothing.

The Grow Routine

When the window definition function receives a wGrow message, it also receives a pointer to a rectangle as its param value. The rectangle is in global coordinates and is usually aligned at its top left corner with the portRect of the window's grafPort. The grow routine should draw a grow image of the window to fit the given rectangle (that is, whatever is appropriate to show that the window's size will change, such as an outline of the content region). The Window Manager requests this operation repeatedly as the user drags inside the grow region. The grow routine should draw in the current grafPort, which will be the Window Manager port, and should use the grafPort's current pen pattern and pen mode, which are set up (as gray and notPatXor) to conform to the Macintosh User Interface Guidelines.

The grow routine for a document window draws a gray outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

The Draw Size Box Routine

Thw wDrawGIcon message tells the window definition function to draw the size box ("grow icon") in the content region of the window if the window is active (highlighted) or, if the window is inactive, whatever is appropriate to show that it temporarily can't be sized. For active document windows, this routine draws the size box icon in the bottom right corner of the portRect of the window's grafPort, along with the lines delimiting the size box and scroll bar areas; for inactive windows, it draws just the delimiting lines.

(hand)

If the size box is located in the window frame rather than the content region, this routine should do nothing.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

*** This will be moved into a separate chapter of the final comprehensive manual. For now, see the QuickDraw manual for complete information about how to use the User Interface Toolbox from assembly language. ***

The primary aid to assembly-language programmers is a file named ToolEqu.Text. If you use .INCLUDE to include this file when you assemble your program, all the Window Manager constants, locations of system globals, and offsets into the fields of structured types will be available in symbolic form.

SUMMARY OF THE WINDOW MANAGER

```
CONST documentProc = 0;    {standard document window}
      dBoxProc     = 1;    {alert box or modal dialog box}
      dBoxZero     = 2;    {like dBoxProc but with no shadow}
      mdBoxProc    = 3;    {modeless dialog box}  *** forthcoming ***
      rDocProc     = 16;   {desk accessory window}

      dialogKind  = 2;   {dialog or alert window}
      userKind    = 8;   {window created directly by the application}

      inDesk      = 0;   {none of the following}
      inMenuBar   = 1;   {in the menu bar}
      inSysWindow = 2;   {in a system window}
      inContent   = 3;   {in the content region (except grow, if active)}
      inDrag      = 4;   {in the drag region}
      inGrow      = 5;   {in the grow region (active window only)}
      inGoAway    = 6;   {in the go-away region (active window only)}

      wNoHit      = 0;   {none of the following}
      wInContent  = 1;   {in the content region (except grow, if active)}
      wInDrag     = 2;   {in the drag region}
      wInGrow     = 3;   {in the grow region (active window only)}
      wInGoAway   = 4;   {in the go-away region (active window only)}

TYPE StringPtr    = ^Str255;
     StringHandle = ^StringPtr;

     WindowPtr    =  GrafPtr;
     WindowPeek   =  ^WindowRecord;

     WindowRecord =  RECORD
                       port:          GrafPort;
                       windowKind:    INTEGER;
                       visible:       BOOLEAN;
                       hilited:       BOOLEAN;
                       goAwayFlag:    BOOLEAN;
                       spareFlag:     BOOLEAN;
                       strucRgn:      RgnHandle;
                       contRgn:       RgnHandle;
                       updateRgn:     RgnHandle;
                       windowDefProc: Handle;
                       dataHandle:    Handle;
                       titleHandle:   StringHandle;
                       titleWidth:    INTEGER;
                       controlList:   Handle;
                       nextWindow:    WindowPeek;
                       windowPic:     PicHandle;
                       refCon:        LongInt
                     END;

     WindowMessage = (wDraw, wHit, wCalcRgns, wNew, wDispose, wGrow,
                      wDrawGIcon);
```

Initialization and Allocation

```
PROCEDURE InitWindows;
PROCEDURE GetWMgrPort   (VAR wPort: GrafPtr);
FUNCTION  NewWindow     (wStorage: Ptr; boundsRect: Rect; title: Str255;
                         visible: BOOLEAN; procID: INTEGER; behind:
                         WindowPtr; goAwayFlag: BOOLEAN; refCon: LongInt)
                         : WindowPtr;
FUNCTION  GetNewWindow  (windowID: INTEGER; wStorage: Ptr; behind:
                         WindowPtr) : WindowPtr;
PROCEDURE CloseWindow   (theWindow: WindowPtr);
PROCEDURE DisposeWindow (theWindow: WindowPtr);
```

Window Display

```
PROCEDURE SetWTitle     (theWindow: WindowPtr; title: Str255);
PROCEDURE GetWTitle     (theWindow: WindowPtr; VAR title: Str255);
PROCEDURE SelectWindow  (theWindow: WindowPtr);
PROCEDURE HideWindow    (theWindow: WindowPtr);
PROCEDURE ShowWindow    (theWindow: WindowPtr);
PROCEDURE ShowHide      (theWindow: WindowPtr; showFlag: BOOLEAN);
PROCEDURE HiliteWindow  (theWindow: WindowPtr; fHilite: BOOLEAN);
PROCEDURE BringToFront  (theWindow: WindowPtr);
PROCEDURE SendBehind    (theWindow: WindowPtr; behindWindow: WindowPtr);
FUNCTION  FrontWindow :  WindowPtr;
PROCEDURE DrawGrowIcon  (theWindow: WindowPtr);
```

Mouse Location

```
FUNCTION FindWindow  (thePt: Point; VAR whichWindow: WindowPtr)
                      : INTEGER;
FUNCTION TrackGoAway (theWindow: WindowPtr; thePt: Point) : BOOLEAN;
```

Window Movement and Sizing

```
PROCEDURE MoveWindow (theWindow: WindowPtr; hGlobal,vGlobal: INTEGER;
                      front: BOOLEAN);
PROCEDURE DragWindow (theWindow: WindowPtr; startPt: Point; boundsRect:
                      Rect);
FUNCTION  GrowWindow (theWindow: WindowPtr; startPt: Point; sizeRect:
                      Rect) : LongInt;
PROCEDURE SizeWindow (theWindow: WindowPtr; w,h: INTEGER; fUpdate:
                      BOOLEAN);
```

Update Region Maintenance

```
PROCEDURE InvalRect   (badRect: Rect);
PROCEDURE InvalRgn    (badRgn: RgnHandle);
PROCEDURE ValidRect   (goodRect: Rect);
PROCEDURE ValidRgn    (goodRgn: RgnHandle);
PROCEDURE BeginUpdate (theWindow: WindowPtr);
PROCEDURE EndUpdate   (theWindow: WindowPtr);
```

Miscellaneous Utilities

```
PROCEDURE SetWRefCon    (theWindow: WindowPtr; data: LongInt);
FUNCTION  GetWRefCon    (theWindow: WindowPtr) : LongInt;
PROCEDURE SetWindowPic  (theWindow: WindowPtr; pic: PicHandle);
FUNCTION  GetWindowPic  (theWindow: WindowPtr) : PicHandle;
FUNCTION  PinRect       (theRect: Rect; thePt: Point) : LongInt;
FUNCTION  DragGrayRgn   (theRgn: RgnHandle; startPt: Point; limitRect,
                         slopRect: Rect; axis: INTEGER; actionProc:
                         ProcPtr) : LongInt;
```

Low-Level Routines

```
FUNCTION  CheckUpdate   (VAR theEvent: EventRecord) : BOOLEAN;
PROCEDURE ClipAbove     (window: WindowPeek);
PROCEDURE PaintOne      (window: WindowPeek; clobbered: RgnHandle);
PROCEDURE PaintBehind   (startWindow: WindowPeek; clobbered: RgnHandle);
PROCEDURE SaveOld       (window: WindowPeek);
PROCEDURE DrawNew       (window: WindowPeek; update: BOOLEAN);
PROCEDURE CalcVis       (window: WindowPeek);
PROCEDURE CalcVisBehind (startWindow: WindowPeek; clobbered: RgnHandle);
```

GLOSSARY

activate event: An event generated by the Window Manager when a window changes from active to inactive or vice versa.

active window: The frontmost window on the desktop.

application window: A window created as the result of something done by the application, either directly or indirectly (as through the Dialog Manager).

content region: The area of a window that the application draws in.

desktop: The screen as a surface for doing work in Macintosh.

document window: A standard Macintosh window for presenting a document.

drag region: A region in the window frame. Dragging inside this region moves the window to a new location and makes it the active window unless the COMMAND key was down.

go-away region: A region in the window frame. Clicking inside this region of the active window makes the window close or disappear.

grow image: The image pulled around when dragging inside the grow region occurs; whatever is appropriate to show that the window's size will change.

grow region: A window region, usually within the content region, where dragging changes the size of an active window.

inactive window: Any window that isn't the frontmost window on the desktop.

modal dialog: A dialog that requires the user to respond before doing any other work on the desktop.

modeless dialog: A dialog that allows the user to work elsewhere on the desktop before responding.

plane: The front-to-back position of a window on the desktop.

reference value: In a window record, a 32-bit field that the application may store into and access for any purpose.

structure region: An entire window; its complete "structure".

system window: Any window that isn't created as the result of something done by the application. Desk accessories are displayed in system windows.

update event: An event generated by the Window Manager when the update region of a window is to be drawn.

update region: A window region consisting of all areas of the content region that have to be redrawn.

variation code: A number that distinguishes closely related types of windows and is passed as part of a window definition ID when a window is created.

visible window: A window that's drawn in its plane on the desktop (but may be completely overlapped by another window).

window: An object on the desktop that presents information, such as a document or a message.

window class: An indication of whether a window is a system window, a dialog or alert window, or a window created directly by the application.

window definition function: A function called by the Window Manager when it needs to perform basic operations on a particular type of window, such as drawing the window frame.

window definition ID: A number passed to window-creation routines to indicate the type of window. It consists of the window definition function's resource ID and a variation code.

window frame: The structure region minus the content region.

window list: A list of all windows ordered according to their front-to-back positions on the desktop.

Window Manager port: A grafPort that has the entire screen as its portRect and is used by the Window Manager to draw window frames.

window record: The internal representation of a window, where the Window Manager stores all the information it needs for its operations on that window.

window template: A resource that contains information from which the Window Manager can create a window.