Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is", and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Copyright

This manual and the software (computer programs) described in it are copyrighted by Apple or by Apple's software suppliers, with all rights reserved. Under the copyright laws, this manual or the programs may not be copied, in whole or part, without the written consent of Apple, except in the normal use of the software or to make a backup copy. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given or loaned to another person. Under the law, copying includes translating into another language.

You may use the software on any computer owned by you but extra copies cannot be made for this purpose. For some products, a multi-use license may be purchased to allow the software to be used on more than one computer owned by the purchaser, including a shared-disk system. (Contact your authorized Apple dealer for information on multi-use licenses.)

Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple Dealer.

© Apple Computer, Inc. 1982
20525 Mariani Avenue
Cupertino, California 95014

Apple and the Apple logo are registered trademarks of Apple Computer, Inc.

Simultaneously published in the U.S.A and Canada.

Reorder Apple Product #A3L0027-A

Apple III

SOS Reference Manual

Volume 1: How SOS Works

Acknowledgements

Knuth, *Fundamental Algorithms: The Art of Computer Programming,* Vol. I, 2/e.© 1981. Reproduction of book cover. Reprinted with permission.

Writer: Don Reed

Contributions and assistance: Bob Etheredge, Tom Root, Bob Martin, Dick Huston, Steve Smith, Dirk van Nouhuys, Ralph Bean, Jeff Aronoff, Bryan Stearns, Russ Daniels, Lynn Marsh, and Dorothy Pearson

Contents

Volume 1: How SOS Works

Volume 2: The SOS Calls

Figures and Tables

Volume 1: How SOS Works

3 Devices 39

4 Files 49

5 File Organization on Block Devices 75

6 Events and Resources 103

Volume 2: The SOS Calls

Preface

For your convenience and ease of reference, this manual is divided into two volumes. Volume 1: How SOS Works describes the operating system of the Apple III. Volume 2: The SOS Calls defines the individual SOS calls. Notice that the sequence of chapter numbers in Volume 1 continues unchanged into Volume 2.

Scope of this Manual

This manual describes SOS (pronounced "sauce"), the Sophisticated Operating System of the Apple III. With the information in this manual you'll be able to write assembly-language programs that use the full power of the Apple III.

However, this manual is not a course in assembly-language programming. It assumes that you can program in assembly language and know the architecture of the 6502 microprocessor upon which the Apple III is based; it will explain how the architecture of the Apple III processor goes beyond that of the standard 6502. If you need more information on 6502 assembly-language programming, refer to one of the books listed in the bibliography of this manual.

The companion volume to this manual, the *Apple III SOS Device Driver Writer's Guide*, contains the information you may need about the interface hardware of the Apple III, and tells how to create device drivers to use that hardware. If you wish to create custom interface software or hardware for the Apple III, read the present manual before turning to the *Apple III SOS Device Driver Writer's Guide.*

Using this Manual

Before you begin with this manual, you should prepare yourself by reading the following:

- *the Apple III Owner's Guide* introduces you to some of the fundamental features of the Apple III—features that you will be exploring more deeply in this manual;
- *the Apple III Standard Device Drivers Manual* describes the workings of the Apple III's video screen, keyboard, graphics, and communications interfaces;
- *the Apple III Pascal Program Preparation Tools* manual explains the use of the Apple III Pascal Assembler, which is the only assembler that works with SOS.

You should also finish reading this preface, to learn about the notation and examples used in this manual.

About the Examples

Included in this manual are many sample programs and code fragments. These are intended as demonstrations *only*. In order to illustrate their concepts as well as possible, they are written to be clear and concise, without necessarily being efficient or comprehensive.

Notation and Symbols

Some special symbols and numeric notations are used throughout this manual.

Numeric Notation

We assume that you are familiar with the hexadecimal (hex) numbering system. All hexadecimal numbers in the text and tables of this manual are preceded by a dollar sign ($). Any number in the text, a table, or illustration that is not preceded by a dollar sign is a decimal number.

Program listings from the Apple III Pascal Assembler, however, do not prefix hex numbers with dollar signs. In such listings, you can distinguish decimal numbers from hex by the fact that decimal numbers end with a decimal point (.). You can distinguish hex numbers from labels by the fact that hex numbers always begin with a digit from 0 to 9, and labels always begin with a letter.

Type	Notation in Text	Notation in Listings
Decimal	255	255.
Hexadecimal	$3A5	3A5
Hexadecimal	$BAD1	0BAD1
Label	BAD1	BAD1

Table 0-1. Numeric Notation

Additional notations are introduced in Chapter 1.

Special Symbols

Four special symbols are used in this manual to emphasize information about helpful or unusual features of the system.

☞ This symbol precedes a paragraph that contains especially useful information.

👁 Watch out! This symbol precedes a paragraph that warns you to be careful.

✋ Stop! This symbol precedes a paragraph warning you that you are about to destroy data or harm hardware.

v1.2 This symbol precedes a paragraph that is specific to versions 1.1, 1.2, and 1.3 of SOS. Note especially that, although the symbol indicates version 1.2, it is also applicable to versions 1.1 and 1.3.

1

The Abstract Machine

1.1 About Operating Systems

An *operating system* is the traffic controller of a computer system. A well-designed operating system increases the power and usefulness of a computer in three important ways. First, an operating system establishes an *abstract machine* that is defined by its concepts and models, rather than by the physical attributes of particular hardware. Second, it acts as a *resource manager*, to ease the programming task. Finally, it provides a *common foundation for software*.

If you are an experienced programmer of small computers, such as the Apple II, but you have never written large programs for a machine with an operating system, you should pay particular attention to this section.

1.1.1 An Abstract Machine

The low-level programming language of a computer is determined not only by its central processor, but by its operating system as well. The operating system is thus an essential part of the programming environment: knowing how it works lets you write programs that use the full power of the machine.

Most importantly, the combination of hardware and operating system software creates an abstract machine that is neither the hardware nor the operating system, but a synthesis of both. This is the machine you program.

The major advantage of the abstract-machine concept is that a program written for the abstract machine is not bound by the current configuration of the hardware. The operating system can compensate for expansions, enhancements, or changes in hardware, making these changes invisible to the programs. Thus programs properly written for an abstract machine need not be modified to respond to changes or improvements in the hardware.

1.1.2 A Resource Manager

An operating system also controls the flow of information into, out of, and within the computer. It provides standard ways to store and retrieve information on storage devices, communicate with and control input/output devices, and allocate memory to programs and data. It also provides certain "housekeeping" functions, such as reading and setting the system clock.

The operating system saves you work. You don't have to write your own procedures for disk-access, communications, or memory-management: the operating system performs such functions for you.

1.1.3 A Common Foundation for Software

An operating system also provides a common base on which to build integrated applications. This, above all, promotes compatibility between programs and data. If two programs use the same file structure and the same memory-management techniques, it's much easier to make the programs work with each other and share data. If all mass storage devices support a common file structure, it is much easier for a program to expand its capacity by substituting a larger device.

Any service provided by SOS is provided *only* by SOS. The continued correct operation of your program under future versions can be assured only if you use the services provided and make no attempt to circumvent SOS.

1.2 Overview of the Apple III

The Apple III/SOS Abstract Machine has six principal parts (see Figure 1-1):

- An interpreter, which is the program executed at boot time;
- The operating system, SOS;
- Memory;
- A set of files, for the storage and transfer of information;
- A set of devices and drivers, for the communication of information; and
- The 6502 instruction set, with extended addressing capabilities.

All of these rest on a base created by the hardware of the machine.

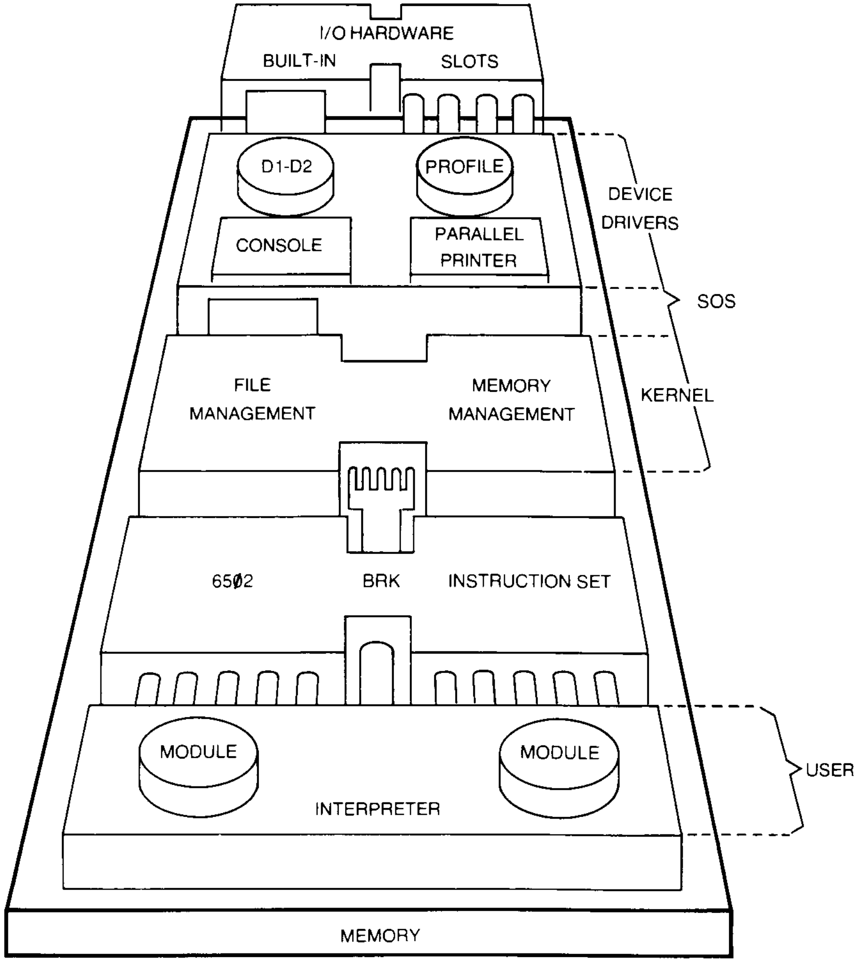


Figure 1-1. The Apple III/SOS Abstract Machine

The rest of this section describes these parts in brief.

1.2.1 The Interpreter

An *interpreter* is an assembly-language program that starts automatically when SOS boots. Interpreters include the Business BASIC and Pascal language interpreters, as well as the application program Apple Writer III.

Only one interpreter can reside in the system at a time. An interpreter is loaded each time the system is booted; the system cannot operate without an interpreter. In addition, language interpreters such as Pascal and BASIC allow separate assembly-language routines, called *modules*, to be loaded and executed.

An interpreter consists of 6502 assembly-language code, including SOS calls. The construction and execution of interpreters and modules is described in Chapter 7.

1.2.2 SOS

SOS is the operating system of the Apple III. It provides a standard interface between the interpreter and the computer's hardware. An interpreter communicates with SOS by making subroutine-like calls to SOS. SOS returns the results of each call to the interpreter. *SOS calls* are of four types:

- *File management calls* read, write, create, and delete files.
- *Device management calls* read the status of a device or control the device.
- *Utility management calls* provide access to the system clock, joystick, and event fence.
- *Memory management calls* allocate and deallocate memory for the interpreter.

SOS also controls all asynchronous operations of the computer, through the mechanisms of interrupts and events, as described in Chapter 6. An interrupt from a device is detected by SOS and handled, under the control of SOS, by an interrupt handler in that device's driver. An event is detected by a device driver and handled, under the control of SOS, by an event-handler subroutine in the interpreter.

SOS is always resident in the system and is loaded from the boot disk's SOS.KERNEL and SOS.DRIVER files when the system is booted. The SOS.KERNEL file contains that part of the operating system that must always be present for the Apple III to function and which does not change from machine to machine: file management, memory management, utility management. Some device management functions, such as translating file calls into calls to device drivers, are also in the SOS kernel. The Disk III driver is included in the SOS kernel because the Apple III system always has a built-in Disk III.

The SOS.DRIVER file includes other device management functions. This file, which is also loaded at boot time, contains the drivers you can reconfigure or remove. The device drivers provide a way for a specific device to support the general concept of a file. For example, you can write a program to send output to the driver .PRINTER . The program contains no information about individual printers: it merely tells SOS to print so many bytes on the printer represented by .PRINTER . The driver .PRINTER translates the SOS calls into the control codes for the specific printer it is written for. To use a different printer, you need only configure a different .PRINTER driver into the operating system.

You can find more information about the standard device drivers that control the text and graphics displays, the keyboard, and the communications ports in the *Apple III Standard Device Drivers Manual*; information about other drivers is in the manuals for their devices; information about creating your own device drivers is in the *Apple III SOS Device Driver Writer's Guide.*

1.2.3 Memory

Although the standard addressing space of the 6502 microprocessor is 64K bytes, the Apple III machine architecture and SOS provide efficient access to a maximum of 512K bytes of memory through the use of two enhanced addressing modes. These modes are described in Chapter 2.

Current hardware supports up to 256K bytes.

Several SOS calls create a memory management and allocation system. An interpreter can cause SOS to find an unused segment of memory, and return that segment's size and location. SOS keeps track of all allocated segments, so that a program that uses only SOS-allocated segments cannot accidentally destroy programs or data used by other parts of the system.

The memory management system also allows an interpreter to acquire additional memory. This means that an interpreter need not be restricted to the use of a specific area of memory, so that the interpreter will run without modification on machines of different memory sizes: the only difference will be in performance.

SOS acts as a memory bookkeeper, keeping track of memory allocated to the interpreter, its modules, and the operating system. This bookkeeper notes whether memory allocation ever violates the rules (that is, whether the same memory space is ever allocated to two programs at the same time); but it does not halt a program that breaks the rules, so the programmer must exercise care. An executing program has access to all memory within its own module. Any time it requests additional space, it should release it as soon as it is not needed.

1.2.4 Files

Files are the principal means of data storage in the Apple III. A file is simply a standardized means by which information is organized and accessed on a peripheral device. All programs and data (even the operating system itself) are stored in files. All devices are represented as files.

The way a file is used is independent of the way the hardware actually accesses that file. Files can be either on random-access devices (such as disk drives) or on sequential-access devices (such as communications interfaces); files on the Apple III's built-in disk drive are accessed in exactly the same manner as files on a large remote hard-disk drive. SOS lets you perform simple operations on files (such as read, write, rename) that are actually complex operations on the devices that store your information.

SOS uses a hierarchical structure of directories and subdirectories to expedite file access. As described in the *Apple III Owner's Guide,* related files can be grouped together in directories and subdirectories, and special naming conventions make it easier to specify groups of files.

1.2.5 Devices

The Apple III can support a variety of peripheral devices. Some of these devices are built into the Apple III itself; others must be plugged into peripheral interface connectors inside the Apple III.

SOS supports operations on two types of devices: block devices and character devices. Block devices read and write blocks of 512 bytes in random-access fashion; character devices read and write single bytes in sequential-access fashion: both support the concept of a file to which you read and write single bytes. SOS defines the ways in which you can control and read the status of both kinds of devices.

1.2.6 The 6502 Instruction Set

The 6502 is the processor in both the Apple II and the Apple III, but in the Apple III its power is extended in two ways:

- Additional hardware gives it two enhanced addressing modes, allowing it to address efficiently far more than 64K bytes of memory.
- The BRK instruction is used to execute SOS calls. SOS calls can be thought of as an extension of the 6502 instruction set: that is, a set of 4-byte 6502 instructions that are emulated in software by the operating system.

2

Programs and Memory

This chapter describes the methods an interpreter uses to obtain and manipulate memory. The actual writing and construction of an interpreter is described in Chapter 7.

2.1 Addressing Modes

Since the 6502's address bus is only 16 bits wide, it can directly address only 64K bytes. This is not enough memory for many of the applications the Apple III is intended for, so the Apple III/SOS system has been designed with new addressing techniques to allow you to efficiently access up to 512K bytes of memory.

The Apple III's memory is subdivided into banks of 32K bytes each. The architecture of SOS can support up to 16 such banks, or a system with 512K bytes.

The current Apple III hardware supports up to eight banks, or 256K bytes.

Certain regions of memory are reserved for use by SOS and its device drivers; the rest is available for use by an interpreter and its data.

Two methods are used to specify locations in the Apple III's memory:

- *bank-switched* addressing, which specifies locations with a bank-plus-address form; and
- *enhanced indirect* addressing, which specifies locations with a three-byte pointer form.

2.1.1 Bank-Switched Memory Addressing

The bank-switched method is the standard memory-addressing technique used to execute interpreter code; it can also be used for data access. In bank-switched addressing (see Figure 2-1), the 6502's addressing space is filled by two banks at a time.

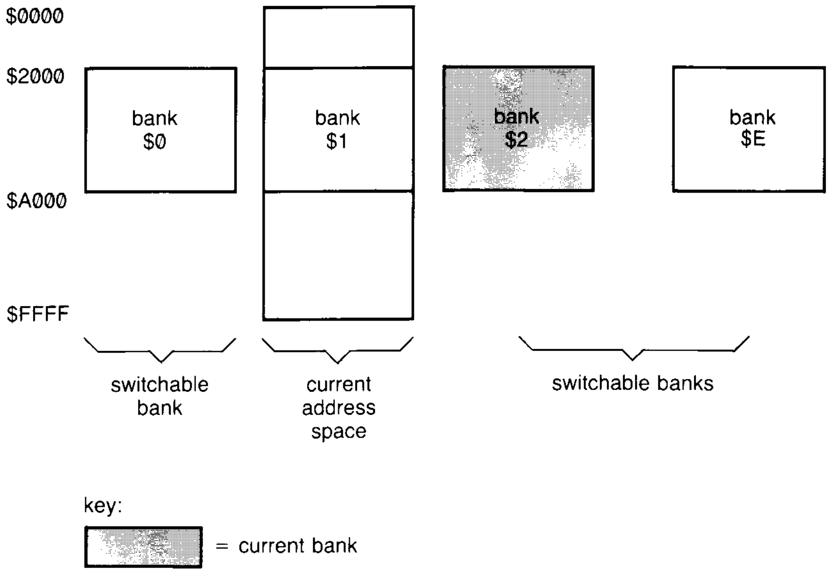


Figure 2-1. Bank-Switched Memory Addressing

One bank (called the "SOS bank", or S-bank) is always present. This unswitched bank occupies locations $0000 through $1FFF and locations $A000 through $FFFF in the standard 6502 addressing space. The larger region contains SOS. The smaller region contains data areas used by SOS, as well as the interpreter's zero page and stack page, described in section 2.2.1.

Locations $2000 through $9FFF are occupied by one of up to 15 switchable banks, numbered $0 through $E. Normally, the highest bank in the system (bank $2 for a 128K system, bank $6 for a 256K system, bank $E for a 512K system) is switched into this space: this bank contains the interpreter. But the interpreter can cause any of the other banks to be switched in, either to execute code or to access data. To switch another bank into the address space (see Figure 2-2), the interpreter changes the contents of the *bank register* (memory location $FFEF), as explained in section 2.4.1.

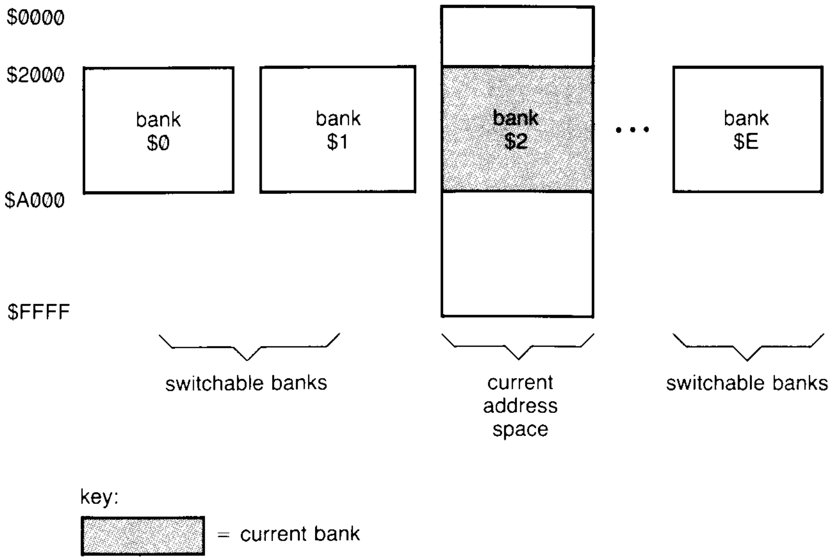


Figure 2-2. Switching in Another Bank

Locations within the S-bank or the currently selected bank may be specified by a two-byte address, notated here as four hexadecimal digits:

$nnnn	$0000 to $1FFF	S-Bank Address
	$2000 to $9FFF	Current-Bank Address
	$A000 to $FFFF	S-Bank Address

where each *n* is a hexadecimal digit. This address uniquely identifies any location within the current address space.

Locations in bank-switched memory (all banks but the S-bank) are specified by their four-digit address, plus the number of the bank they reside in. The addresses of these locations are in the form:

$b:nnnn	$0:2000 to $0:9FFF	Bank-Switched
	$1:2000 to $1:9FFF	Addresses
	.	
	.	
	.	
	$E:2000 to $E:9FFF	

where *$b* is a hexadecimal digit from $0 to $E, and each *n* is a hexadecimal digit.

Addresses in the current bank can be specified with or without the bank number: that is, in current-bank form or in bank-switched form. The addresses $E:2000 and $2000 are equivalent if bank $E is switched in.

Note that bank-switched address specifications such as $0:FFDF and $2:01FF are not standard: these addresses, being in S-bank space and unaffected by bank-switching, are normally specified without the bank number.

Address	**Specifies**
$0:2000	First location in bank 0
$2:9FFF	Last location in bank 2
$F:32A4	Invalid: there is no bank $F.
$1:B700	Non-standard: use S-bank specification $B700

Table 2-1. Addresses in Bank-Switched Notation

2.1.2 Enhanced Indirect Addressing

The second memory-addressing method, *enhanced indirect addressing*, uses a three-byte *extended address* to access each memory location. This method lets a program in one bank access data in other banks. Enhanced indirect addressing lets any 6502 instruction that allows indirect (-X or -Y) addressing to access data within any *pair* of adjacent memory banks. (For example, banks $0 and $1, and banks $1 and $2, constitute bank pairs.) This addressing method is considerably more efficient than bank-switching, since the bank register need not be altered in order to access data in other banks.

Enhanced indirect addressing is used for data access only. Programs cannot execute in the memory space defined by this method.

An extended address specification consists of a two-byte address and one extension byte, or *X-byte*, which has no relation to the 6502's X *register*. The address is in standard 6502 form (low byte followed by high byte), and may be from $0000 to $FFFF, with some restrictions explained later. The X-byte is of the form shown in Figure 2-3.

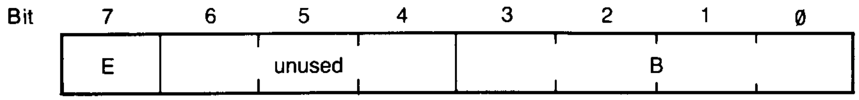


Figure 2-3. X-byte Format

Bit 7 of the X-byte is the enhanced-addressing bit, or *E-bit*; bits 0 through 3 are the bank-pair field, or *B field*. If the E-bit is 0, normal indirect addressing takes place, using the S-bank and current bank. If the E-bit is 1, enhanced indirect addressing (see Figure 2-4) takes place, and the B field determines which of several bank pairs are mapped into the address space.

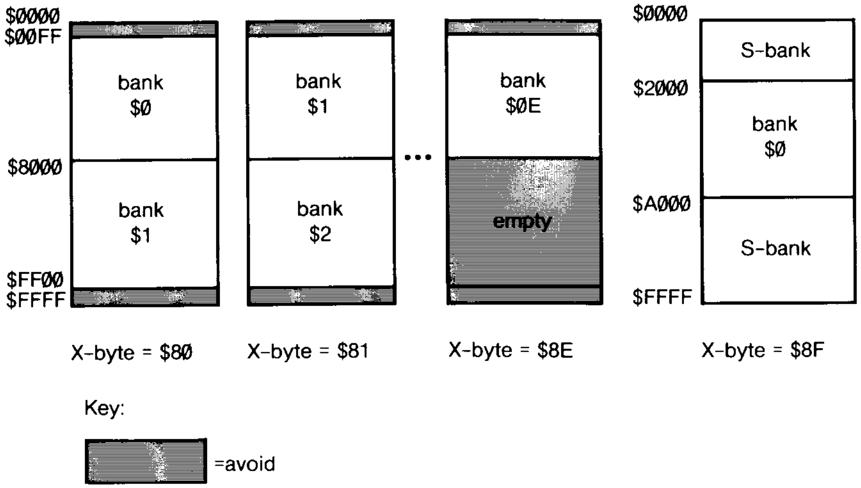


Figure 2-4. Enhanced Indirect Addressing

The X-byte selects one of up to 16 pairs of banks to fill the 64K memory space, and the two-byte address selects a specific location within the bank pair. Extended addresses have this form:

$8x:nnnn	$80:0100 to $80:FFFF	Banks 0 and 1
	$81:0100 to $81:FFFF	Banks 1 and 2
	.	
	.	
	.	
	$8m:0100 to $8m:7FFF	Bank m
	$8F:0000 to $8F:FFFF	S-bank and Bank 0

where *x* and each *n* are hexadecimal digits, and *m* is the number of the highest switchable bank.

Extended address notation differs from bank-switched address notation in the number of digits before the colon. An extended address begins with a two-digit X-byte, whose first digit is always $8; a bank-switched address begins with a one-digit bank number.

The X-byte can range from $80 (banks 0 and 1) to $8m (bank m), where *m* is the number of the highest bank: $2 for a 128K system; $6 for a 256K system; or $E for a 512K system. The highest bank pair is not really a pair: it ends at $8m:7FFF, and higher addresses will produce undefined results. The X-byte has a singular value, $8F, which pairs the S-bank with bank $0 (see hand paragraph below).

Note that the addresses $8n:0000 to $8n:00FF are not accessible via enhanced indirect addressing. Any reference to these addresses will give you a location on the currently selected zero page. To address these locations ($8n:0000 to $8n:00FF) you can use the equivalent address in the next-lower bank pair: that is, $8(n-1):8000 to $8(n-1):80FF. (See fourth example below). This trick does not work for the addresses $80:0000 to $80:00FF: for these addresses, you can use the equivalent addresses $8F:2000 to $8F:20FF (see hand, below).

In addition, the addresses $8n:FF00 through $8n:FFFF should generally be avoided, as indexing these addresses by the value in the Y-register may cause a carry and produce an address in the range $8n:0000 through $8n:00FF—this address is on the zero page. The locations $8n:FF00 through $8n:FFFF may be addressed with the equivalent addresses in the next-higher bank pair: that is, $8(n+1):7F00 through $8(n+1):7FFF.

The invalid and risky regions are shown in color in Figure 2-4.

Address	**Specifies**
$80:8000	First location in bank $1
$81:7FFF	Last location in bank $1
$03:2215	Not an extended address: X-byte ignored
$81:002E	Invalid: use $80:802E
$81:FF2E	Risky: use $82:7F2E

Table 2-2. Extended Addresses

The X-byte $8F is unique: it causes the S-bank and bank $0 to be switched into the 6502's address space in their standard bank-switched arrangement. Bank $0 is mapped to the locations $8F:2000 to $8F:9FFF, so no part of it conflicts with the zero page. The X-byte $8F is used primarily by graphics device drivers to access the graphics area at the bottom of bank $0. (See the eye paragraph in section 2.4.2.2.)

2.2 Execution Environments

An Apple III program's *execution environment* defines the state of the machine while that program is running. The two major programs, SOS and your interpreter, run in different environments; assembly-language modules run in an environment much like the interpreter environment; and device drivers run in part of the SOS environment.

The environment defines the location of the program being executed, the location and type of memory that program can access, the processor speed, and the kinds of interrupts the program can handle. (Interrupts are explained in Chapter 6 and in the *Apple III SOS Device Driver Writer's Guide*.) The environment also determines whether and how one program can communicate with another. The environment also specifies which zero page and stack the executing program will use, as explained in the next section.

2.2.1 *Zero Page and Stack*

The 6502 microprocessor reserves the first two pages in memory for special access. The *zero page* (locations $0000 through $00FF) is used by several 6502 addressing modes for indirect addressing and to save execution time and code space.

But the zero page has only 256 locations, and if both the interpreter and SOS are trying to save data in that page, it quickly fills up. The Apple III resolves this contention by allocating separate zero pages to the interpreter ($1A00 through $1AFF) and SOS ($1800 through $18FF). Thus when an interpreter accesses a zero-page location (by executing an instruction followed by a one-byte address), it's accessing an area of memory completely separate from the zero-page storage of SOS.

Similarly, page one (locations $0100 through $01FF) is used as a 256-byte push-down stack for temporary data storage and subroutine and interrupt control. Programs that call many nested subroutines and save many temporary values on the stack can quickly fill it up. Again, the Apple III resolves this contention by allocating separate stacks to the interpreter ($1B00 through $1BFF) and to SOS ($0100 through $01FF).

Each zero page and stack is accessible from other environments as a different page in memory. The SOS kernel, for example, can access locations in the interpreter's zero page by using the addresses $1A00 through $1AFF.

An interpreter should access only its own zero page and stack. An interpreter that writes into the SOS zero page or stack will generally come to an untimely and untidy end.

2.2.2 *The Interpreter Environment*

The interpreter is in the highest switchable bank of memory (bank $n): for a 128K system, this would be bank $2; for a 256K system, bank $6; for a 512K system, bank $E. Figure 2-5 shows the interpreter placement in memory.

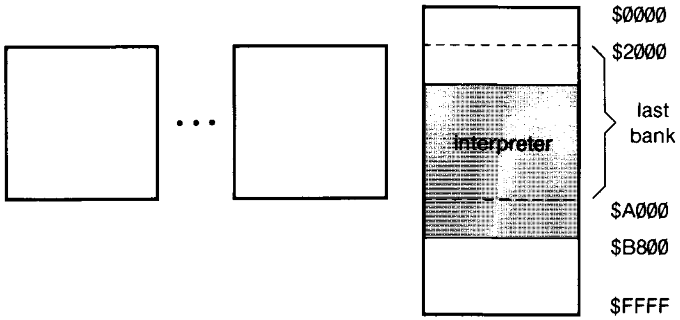


Figure 2-5. Interpreter Memory Placement

v1.2 An interpreter shorter than 6K bytes is located entirely in locations $A000 through $B7FF of the S-bank. An interpreter longer than 6K ($1800) bytes begins in the highest bank (the first byte is between $n:$2000 and $n:$9FFF), and ends in the S-bank (the last byte is at location $B7FF). For example, an interpreter that is 10K ($2800) bytes long in a 128K system would reside from $2:9400 to $B7FF.

Although the maximum size of an interpreter is 38K ($9800) bytes, we recommend that interpreters be restricted to 32K ($8000) bytes, for compatibility with future versions of SOS. A longer interpreter can be split up into a main unit and one or more separately-loaded modules.

An interpreter runs at a nominal 2 MHz clock rate. In practice, execution speed is approximately 1.4 MHz if the Apple III's video display is on; turning off the video display (using the .CONSOLE driver's CTRL-5 command) raises execution speed to 1.8 MHz. (The remaining 0.2 MHz is consumed by memory refresh.) An interpreter must be fully interruptable, so no timing loop in an interpreter will be reliable, except to provide a guaranteed minimum time.

The interpreter's zero and stack pages, always accessible by normal zero-page and stack operations, can also be addressed as pages $1A and $1B. Page $16 is used as the extension page for enhanced indirect addressing (see section 2.1.2).

Environment Attribute	**Setting**
IRQ Interrupts	Enabled
NMI Interrupts	Enabled or Disabled
Processor Speed	Full speed
Zero Page	Page $1A
Stack Page	Page $1B
Extend Page	Page $16
Bank	Highest

Table 2-3. Interpreter Environment

Of the above environment attributes, only the bank register (location $FFEF) should be changed by an interpreter. Adherence to this rule is essential for correct system operation.

An assembly-language module operates in the same environment as the interpreter, except that it may reside in a different bank (see section 7.4). An assembly-language module must share the interpreter's zero page and stack.

2.2.3 *SOS Kernel Environment*

The SOS kernel (SOS without its device drivers) resides in the upper regions of S-bank memory, and uses the lower areas of the S-bank for data and buffer storage (see Figure 2-6).

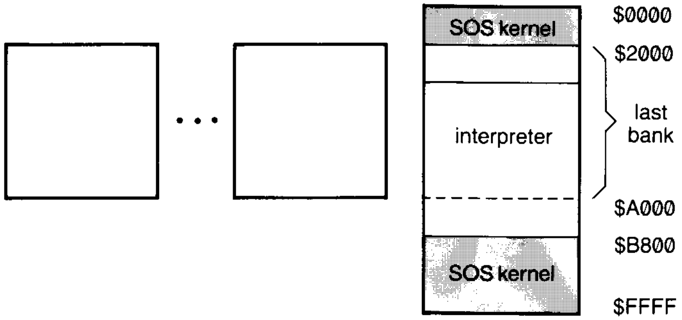


Figure 2-6. SOS Kernel Memory Placement

The SOS kernel uses no bank-switched memory.

SOS uses its own zero page and stack (pages $18 and $01, respectively). It can be interrupted by both IRQ and NMI interrupts.

Environment Attribute	Setting
IRQ Interrupts	Enabled
NMI Interrupts	Enabled
Processor Speed	Full speed
Zero Page	Page $18
Stack Page	Page $01
Extend Page	Page $14
Bank	S-bank

Table 2-4. SOS Kernel Environment

2.2.4 SOS Device Driver Environment

Device drivers are placed directly below the interpreter (that is, in memory locations with smaller addresses), in the highest-numbered bank in the system (see Figure 2-7). Any drivers that do not fit into that bank are placed in the next lower bank, beginning at $9FFF and moving down to lower-numbered addresses.

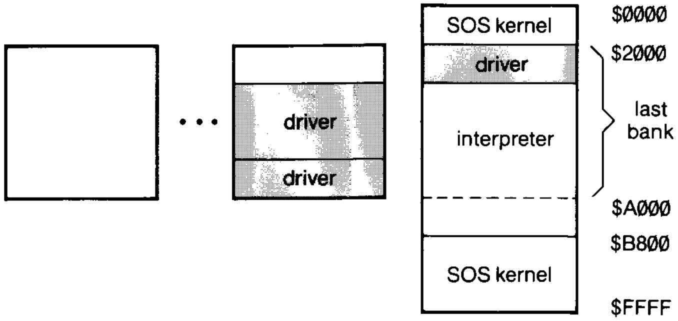


Figure 2-7. SOS Device Driver Memory Placement

Drivers share the SOS zero page and stack. A driver must reserve space within itself for all buffers that it uses: it cannot claim any memory outside itself.

Environment Attribute	Setting
IRQ Interrupts	Enabled or Disabled
NMI Interrupts	Enabled or Disabled
Processor Speed	Full Speed or Fixed 1 MHZ
Zero Page	Page $18
Stack Page	Page $01
Extend Page	Page $14
Bank	Interpreter's or Lower

Table 2-5. SOS Device Driver Environment

A device driver can alter the execution speed; it can disable interrupts for up to 500 microseconds to run timing loops: for more information, see the *Apple III SOS Device Driver Writer's Guide.*

2.2.5 Environment Summary

The environment determines what actions a program can perform and what other programs it can communicate with. The following table summarizes the capabilities of each environment.

Function	Interpreter*	Kernel	Driver
Can perform a SOS call	Yes	No	No
Can call SOS subroutines	No	Yes	Yes
Can be interrupted	Yes	Yes	Yes**
Can respond to IRQ	No	Yes	Yes
Can respond to NMI	No	Yes	No
Can disable interrupts	No	Yes	Yes
Can detect and queue an event	No	Yes	Yes
Can respond to an event***	Yes	No	No
Can access interpreter memory	Yes	Yes	Yes
Can access free memory	Yes	Yes	Yes

* An assembly-language module runs in the same environment as its interpreter.

** A device driver can contain a special section, called an *interrupt handler*, designed specifically to handle IRQ interrupts.

*** Events, or software interrupts, are defined in Chapter 6.

Table 2-6. Environment Summary

2.3 Segment Address Notation

When an interpreter is loaded into memory, it occupies part of the S-bank and part of the highest-numbered bank. The region below the interpreter is occupied by the device drivers; the region below the drivers is free memory, as shown in Figure 2-8.

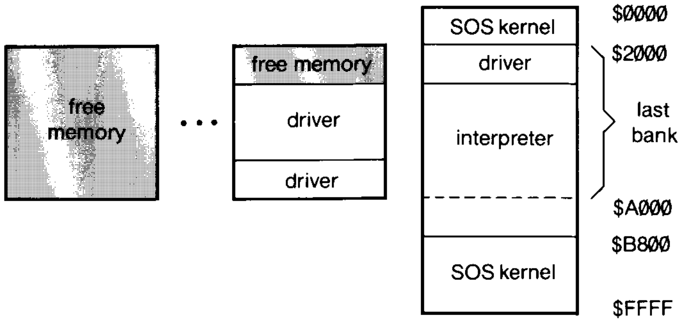


Figure 2-8. Free Memory

The interpreter has access to its own space. If it needs more memory, it can gain access to free memory by using the SOS memory calls. These calls use *segment address notation*, to define segments of memory for allocation (see Figure 2-9). Segment address notation resembles bank-switched address notation, except that it defines addresses of segments, not bytes, of memory in either the S-bank or a switchable bank. A *page* is a group of 256 contiguous bytes with a common high address byte. A *segment* is a set of contiguous pages. The lowest page in a segment is called the *base*; the highest page is called the *limit*. Each bank of memory contains 128 pages, numbered $20 through $9F.

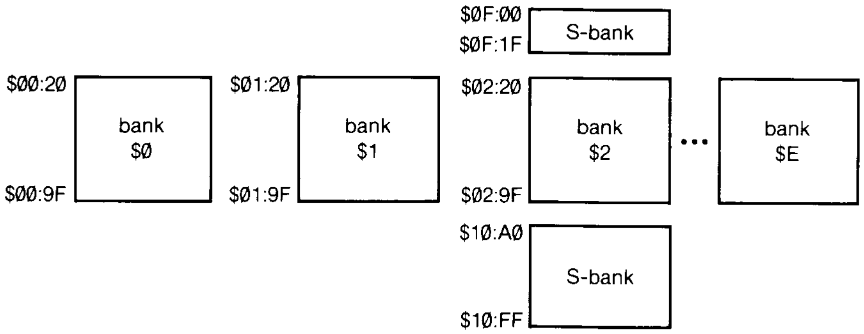


Figure 2-9. Segment Address Notation

Each page of memory has a corresponding *segment address*, which is very similar to that page's starting address in bank-switched memory. The format is:

$bb:pp	$00:20 to $00:9F $01:20 to $01:9F . . . $0E:20 to $0E:9F	Segment Addresses

where *bb* is the bank number (one byte) and *pp* is the page number (one byte) in that bank. Notice that for segment addresses in bank-switched memory the page part of the segment address is always between $20 and $9F.

Segment Address	Specifies
$01:30	Page beginning at $01:3000
$04:62	Page beginning at $04:6200
$00:9F	Page beginning at $00:9F00

Table 2-7. Addresses in Segment Notation

A segment address specifies an entire page, not just the first location in that page. A base segment address and a limit segment address together specify a segment.

Segment addresses can also specify pages in S-bank memory: the format then is slightly different. For segments in the lower part of the S-bank, the bank part of the segment address is always $0F; for segment addresses in the upper part of the S-bank, the bank part of the segment address is always $10. In either case, the page part (as above) is the same as the high byte of the memory address.

$bb:pp	$0F:00 to $0F:1F $10:A0 to $10:FF	Segment Addresses

Segment Address	Specifies
$0F:14	Page beginning at $1400
$0F:02	Page beginning at $0200
$10:B8	Page beginning at $B800

Table 2-8. Addresses in Segment Notation, S-Bank

Before segment addresses can be used by an interpreter, they must be converted into bank-switched or extended addresses. These conversions are explained in section 2.4.3. The SOS memory calls that use segment addresses are explained below.

2.3.1 Memory Calls

Interpreters use these SOS calls to allocate and release memory. The name of each call below is followed by its parameters (in boldface). The input parameters are directly-passed values. The output parameters are all directly-passed results. The SOS call mechanism is explained in Chapter 8; the individual calls are described fully in Chapter 11 of Volume 2.

REQUEST_SEG

[**base**, **limit**, **seg_id**: value; **seg_num**: result]

This call requests the allocation of the contiguous region of memory bounded by the base and limit segment addresses. A new segment is allocated if and only if no other segment currently occupies any part of the requested region of memory. If a segment is allocated, an entry for it is made in the segment table.

FIND_SEG

[**search_mode**, **seg_id**, **pages**: value; **pages**, **base**, **limit**, **seg_num**: result]

This call searches memory from the highest memory address down, until the first free space of length **pages** that meets the search restrictions in **search_mode** is found. If such a space is found, this free space is allocated to the caller as a segment (as in REQUEST_SEG): both the segment number and the location in memory of the segment are returned. If a segment with the specified size is not found, then the size of the largest free segment which meets the given criterion will be returned in **pages**. In this case, however, error SEGRQDN will be returned, indicating that the segment was not created.

CHANGE_SEG

[**seg_num**, **change_mode**, **pages**: value; **pages**: result]

This call changes either the base or limit segment address of the specified segment by adding or releasing the number of pages specified by the **pages** parameter. If the requested boundary change overlaps an adjacent segment or the end of the memory, then the change request is denied, error SEGRQDN is returned, and the maximum allowable page count is returned in the **pages** parameter.

GET_SEG_INFO

[**seg_num**: value; **base**, **limit**, **pages**, **seg_id**: result]

This call returns the beginning and ending locations, size in pages, and identification code of the segment specified by **seg_num**.

GET_SEG_NUM

[**seg_address**; value; **seg_num**: result]

This call returns the segment number of the segment, if any, that contains the segment address.

RELEASE_SEG

[**seg_num**: value]

This call releases the memory occupied by segment **seg_num** by removing the segment from the segment table. The memory space formerly occupied by segment **seg_num** can now be allocated to another program. If **seg_num** equals zero, then all non-system segments (those with segment identification codes greater than $0F) will be released.

2.4 Memory Access Techniques

The Apple III augments the eleven addressing modes of the 6502 in two ways: bank-switching and enhanced indirect addressing. Bank-switched addressing is used for executing code segments residing in bank-switched memory. Enhanced indirect addressing is used for access to data in memory. These techniques give your programs efficient access to all of memory.

In addition, SOS uses segment address notation to allocate free memory for programs. Segment address notation is reserved for the SOS memory management calls, which the interpreter uses to obtain and release memory.

This section discusses the most common modes of access to program and data storage areas in the Apple III. It shows how the memory addressing methods introduced in section 2.1 and 2.3 are used in performing various operations, and how these methods can be used in a program. It also presents sample algorithms that convert the address of a location from one form to another.

2.4.1 Subroutine and Module Addressing

The 6502's JMP and JSR instructions affect the flow of control within an interpreter. As the interpreter resides in the S-bank and the highest switchable bank, the destination for these instructions is specified in S-bank or current-bank notation. The JSR and JMP instructions should be used in the normal 6502 absolute addressing mode. Here are three examples of such instructions.

```
AA40| 4C 3A85        JMP   853A    ; Jump to location $853A
                                   ; in interpreter

8B80| 20 5022        JSR   2250    ; Jump to subroutine at
                                   ; location $2250

23BB| 4C 52B6        JMP   0B652   ; Jump to location $B652,
                                   ; in the S-bank
```

All assembly-language listings in this manual were made with the Apple III Pascal Assembler. This is the only assembler supported for the Apple III.

If an interpreter wishes to transfer control to a module residing in another bank, the normal addressing mode will not work: the interpreter must switch in the proper bank before performing the JMP or JSR.

Bank-switching can be performed only by code residing in S-bank (that is, unswitched) memory. An interpreter that performs bank-switching should use a single dispatching routine, located between locations $A000 and $B7FF in the S-bank, for all bank-switching.

The interpreter switches in a given bank by storing the number of the bank in the bank register (location $FFEF). Once this is done, the JMP or JSR instruction can be executed normally. Here's a valid jump:

```
0000| FFEF     BREG    .EQU   0FFEF    ; Define bank register

A050| A9 01            LDA    #01      ; Jump to location $1:326B
A052| 8D EFFF          STA    BREG
A055| 4C 6B32          JMP    326B
```

Here's a jump into oblivion:

```
0000| FFEF     BREG    .EQU   0FFEF    ; Define bank register

8B40| A9 02            LDA    #00      ;This program will crash,
8B42| 8D EFFF          STA    BREG     ; as it is not located
8B45| 4C 4440          JMP    4044     ; in the S-bank.
```

The module, once switched-in, can use current-bank addresses to jump around inside itself, and can JMP or RTS back to the part of the interpreter in S-bank memory, without bank-switching. The interpreter must, however, switch the highest bank back in before any interpreter code below S-bank memory can be executed. To do this the interpreter must save its own bank number before calling the module. The interpreter can read the contents of the bank register to find the number of its bank, then call a module and, upon returning, restore the proper bank. The following subroutine demonstrates how an interpreter would call a module located at $1:3300.

```
0000| FFEF     BREG    .EQU   0FFEF    ; Define bank register

A700| AD EFFF          LDA    BREG     ; Get the current bank
A703| 48               PHA             ; Save it on the stack
A704| A9 01            LDA    #01      ; Switch in
A706| 8D EFFF          STA    BREG     ;   bank $1
A709| 20 0033          JSR    3300     ; Call the module
A70C| 68               PLA             ; Upon return, restore
A70D| 8D EFFF          STA    BREG     ;   the bank number.
A710| 60               RTS             ; Return to main code.
```

Only the lower four bits of the bank register contain the current bank number; the upper four bits should be zero.

2.4.2 *Data Access*

An interpreter can access data in three places:

- In the interpreter's zero page;
- In a table within the interpreter itself;
- In a segment allocated from free memory.

Data can be accessed in locations $0000 through $00FF, the interpreter's zero page, by instructions in absolute, zero-page, or zero-page indexed mode. For example,

```
6BA7| A5 54            LDA    54       ; Value on zero page

747F| 8D E300          STA    00E3     ; Also on zero page
```

To access data in a table within itself, the interpreter must use the absolute address of the table (in current-bank or S-bank notation) in absolute or indexed addressing mode.

```
7075| CD 9BAB          CMP    0AB9B     ; Compare location $AB9B
                                        ;   to accumulator

585D| BD 5022          LDA    2250,X    ; Load accumulator from
                                        ;   byte $2250 + X
```

Data in free memory can be accessed by an interpreter in two ways: by bank-switching or by enhanced indirect addressing. All data used by an interpreter must be stored in SOS-allocated segments (see section 11.1 of Volume 2). To begin storing data in free memory, an interpreter must first request a segment of free memory from SOS, using a REQUEST_SEG or FIND_SEG call. SOS will return a segment address, which the interpreter can change into an address more suitable for data access. Conversion algorithms are described in section 2.4.3.

2.4.2.1 *Bank-Switched Addressing*

Bank-switching for data access operates just like bank-switching for module execution (described in section 2.4.1). To perform an operation on location $*b:nnnn*, store $*b* in the bank register and perform the operation on absolute location $*nnnn*. For instance,

```
0000| FFEF      BREG. .EQU   0FFEF     ; Define bank register

0000|                 .ORG   0A3AA     ; Code starts here

A3AA| AD EFFF         LDA    BREG      ; Save current bank register
A3AD| 48              PHA

A3AE| A0 00           LDY    #00       ; Perform a loop to
A3B0| 8C EFFF         STY    BREG      ; zero all locations
A3B3| 98              TYA              ; from $0:9800 to
A3B4| 99 0098   LOOP  STA    9800,Y    ; $0:98FF.
A3B7| C8              INY              ;
A3B8| D0 FB           BNE    LOOP      ;

A3BA| 68              PLA              ; Store bank register
A3BB| 8D EFFF         STA    BREG      ;
```

Just as in module execution, the code to perform bank-switched data access must reside in the part of the interpreter that is located in S-bank memory, and you must remember to restore the original contents of the bank register before returning to the main part of the interpreter.

2.4.2.2 *Enhanced Indirect Addressing*

Enhanced indirect addressing allows an interpreter to access any location in bank-switched memory without having to switch in the proper bank and then switch back. Any 6502 instruction that supports indirect-X or indirect-Y addressing (ADC, AND, CMP, EOR, LDA, ORA, SBC, STA) can use enhanced indirect addressing.

To perform a normal (not enhanced) indirect operation on location $*hilo*, you store $*lo* in a location $*nn* on zero page, and store $*hi* in the following location. You must also store $00 in location $nn+1 of the X-page: the $00 turns off extended addressing. Then you perform the operation in an indirect mode on location $*nn*. The two bytes at $*nn* are a pointer: you can increment, decrement, and test them to move the pointer through your data structure.

Enhanced indirect addressing merely adds one step to this process. To perform an enhanced indirect addressing operation, in the interpreter environment, on location $*xx:hilo*, you store $*lo* in $*nn*, $*hi* in $nn+1, and $*xx* in location $*16nn*+1. Then perform the operation in an indirect mode on location $*nn*. The location $*16nn*+1 is the *extension byte*, or *X-byte*, of the pointer.

Enhanced indirect addressing takes effect whenever you execute an indirect-mode instruction and bit 7 of the pointer's extension byte (X-byte) is 1: that is, whenever the extension byte is between $80 and $8F. If you wish to perform normal indirect operations, using bank-switched addressing rather than enhanced indirect addressing, you should store your pointer in bank-switched form in the zero page, and set its extension byte to $00, which will make sure bit 7 is 0. For instance,

```
61EE| A9 89     LDA    #89      ; Perform a LDA $82:3289 :
61F0| 85 57     STA    57       ; To set up, first put
61F2| A9 32     LDA    #32      ;    $lohi in zero page
61F4| 85 58     STA    58       ;    locations $57 and $58;

61F6| A9 82     LDA    #82      ; then put $xx into
61F8| 8D 5816   STA    1658     ;    location $1658.

61FB| A0 00     LDY    #00      ; Index by 0.
61FD| B1 57     LDA    (57),Y   ; Perform the operation.
```

Once the three bytes are stored, you can manipulate them almost as easily as a two-byte pointer, and you can use one pointer to access data in all 15 switchable banks (a total of 480K). This makes it easy to handle large data structures.

Remember that enhanced indirect addressing is different from bank-switched addressing. For a description of the two methods, see section 2.1.

If you are using the enhanced indirect-Y addressing mode and are using the Y-register to index from an extended address, we strongly recommend that you avoid using addresses $8n:FF00 through $8n:FFFF. Adding a Y value to one of these addresses may cause a carry and create an address in the range $8n:0000 through $8n:00FF, which will access a location on the zero page. If you keep your pointer below $8n:FF00 whenever you are using a non-zero Y register in the enhanced indirect-Y addressing mode, you will avoid this problem.

2.4.3 *Address Conversion*

Most interpreters deal mainly with addresses in segment and extended form: bank-switched addresses are used only when an interpreter must execute code in a different bank. But bank-switched addresses are a convenient intermediate form between segment and extended addresses: they can be readily converted to either of the other forms.

The following algorithms describe the basic conversions between addresses in segment, bank-switched, and extended forms.

2.4.3.1 *Segment to Bank-Switched*

A segment address specifies a page in bank-switched memory. When you convert a segment address to a bank-switched address, the result is the address of the first byte in that page.

To convert a segment address $*bb:pp* to a bank-switched address $*B:NNNN*,

```
if   (bb = 0F) or (bb = 10)
   then B := 0
    else B := bb;
NNNN := pp00
```

For example, the following segment and bank-switched addresses are equivalent.

Segment		**Bank-Switched**		
$04:63	=	$(4):(6300)	=	$4:6300
$07:89	=	$(7):(8900)	=	$7:8900
$10:1F	=	$(0):(1F00)	=	$0:1F00

The bank part, *bb*, of the segment address is converted to $0 if it indicates the S-bank, or truncated if it indicates any other bank. It then becomes the bank part of the bank-switched result. The page part, *pp*, of the segment address becomes the high part of the bank-switched address, and the low part is set to $00.

2.4.3.2 *Segment to Extended*

When converting to extended form, you must be careful to make sure that the result is in the valid range of extended addresses. You must also handle the special cases of S-bank segment addresses and the segment address $00:20.

To convert a segment address $*bb:pp* into an extended address $*XX:NNNN*,

```
if  (   (bb = $00)                     {zero bank}
     or (bb = $0F)                     {low S-bank}
     or (bb = $10)  )                  {high S-bank}
 then
   begin
     XX      := $8F  ;
     NNNN    := pp00
   end
 else                                  {general case}
   begin
     XX      := $80+bb-1   ;
     NNNN    := pp00+$6000
   end;
```

For example, the following segment and extended addresses are equivalent:

Segment		Extended		
$09:2A	=	$(80+9-1):(2A00+6000)	=	$88:8A00
$02:94	=	$(80+2-1):(9400+6000)	=	$81:FF00
$0F:1E	=	$(8F):(1E00)	=	$8F:1E00

If the segment address specifies a page in S-bank memory, the *bb* part is ignored, and the *pp* part is converted to the address of the beginning of a page in the S-bank/bank 0 pair of the enhanced indirect addressing space.

If the segment address is in bank-switched memory, the *bb* part is converted to the *xx* byte that selects a bank pair with the specified bank in the top half of the pair. The *pp* part is then converted to the address of the beginning of the proper page in that bank pair.

2.4.3.3 Extended to Bank-Switched

When changing an extended address to bank-switched form, you must handle the special case of an S-bank extended address. You must also determine whether the extended address points to a location within the upper or lower bank in its bank pair.

To convert an extended address $*xx*:*nnnn* to a bank-switched address $*B:NNNN*,

```
if   (xx = $8F) then
  begin
    B       := $0   ;
    NNNN    := nnnn
  end
else
    if   (nnnn < $8000) then
      begin
        B       := xx-$80       ;
        NNNN    := nnnn+$2000
      end
    else
      begin
        B       := xx-$80+1    ;
        NNNN    := nnnn-$6000
      end;
```

For example, the following extended and bank-switched addresses are equivalent:

Extended		Bank-switched		
$86:4365	=	$(86-80):(4365+2000	=	$6:6365
$82:EFB4	=	$(82-7F):(EFB4-6000)	=	$3:8FB4
$8F:2000	=	$($0):(2000)	=	$0:2000

If the extended address refers to a location in the S-bank, the bank part of the bank-switched address is set to $0 and the address part is used directly.

If the extended address refers to bank-switched memory, then the *xx* part specifies a bank pair. If the address part is less than $8000, the extended address refers to a location in the lower bank in the pair; otherwise, it refers to a location in the upper bank. The bank part is set to the bank number, and the address part is adjusted to the proper location within the specified bank.

2.4.4 *Pointer Manipulation*

Most data structures you use are accessed by three-byte pointers in extended-address form. The preceding section described how to create an extended-address pointer from a segment address; this section describes how to increment and test such a pointer.

These algorithms are designed for ease of explanation, not for efficiency. They work, but are not intended to be incorporated verbatim into real applications.

2.4.4.1 *Incrementing a Pointer*

An increment operation defines successive values of a pointer, and thus traces a path through successive locations in memory (see Figure 2-10). This path covers all switchable banks, but omits the S-bank. The path traced by the algorithm below begins at the first location in bank 0, extended address $8F:2000. It continues through the first page in this bank, then proceeds to the second page in the same bank with the extended address $80:0100. This path is chosen to avoid the invalid address range $80:0000 to $80:00FF.

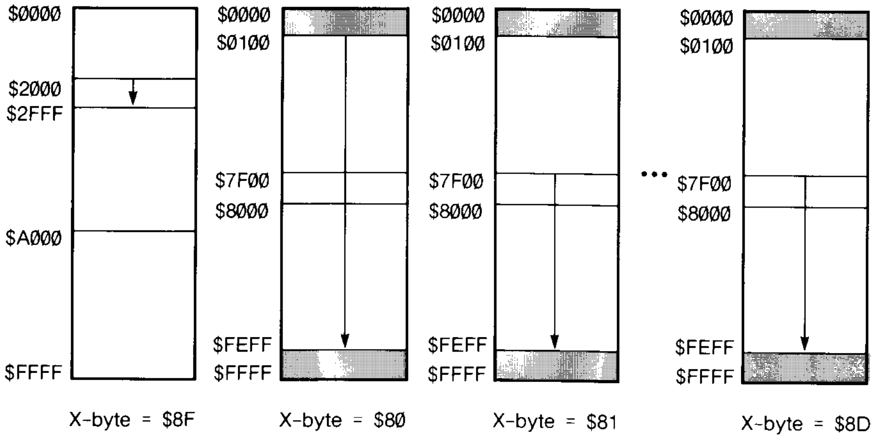


Figure 2-10. Increment Path

The path then continues through the last location in bank 1, extended address $80:FFFF. The path switches to the next bank pair and continues with the first location in bank 2, $81:8000. The path continues in this manner to the last location in the last bank in memory, at which point it terminates.

The following algorithm increments an extended address $*xx:nnnn*.

```
repeat
  nnnn  :=  nnnn  +1  ;                 {Move to next location.          }
  if (xx = $8F) and (nnnn > $20FF)
    then begin
      xx := $80 ;                       {If beyond location $8F:2100,    }
      nnnn : = nnnn-$2000               {move to location $80:0100       }
    end;
  if ( nnnn > $FEFF )                   {If near end of a bank pair,     }
    then begin
      nnnn  := nnnn-$8000 ;             {switch to middle                }
      xx     := xx + 1                  {of next bank pair.              }
    end;
  until xx > $8D;                       {If no next pair, then stop.     }
```

Notice how this algorithm switches from one bank to the next when its address part reaches $FF00. This is to prevent the pointer from ever taking a value between $8n:FF00 and $8n:FFFF, which can cause problems when used in an instruction in the indirect-Y addressing mode.

2.4.4.2 *Comparing Two Pointers*

Two pointers can be considered equal under three conditions. When you compare two pointers for equality, you must test all three conditions.

You can reduce the number of tests by comparing the two extension bytes first, then ordering the two numbers according to their extension bytes if they are unequal.

The following algorithm compares $*xx:nnnn* to $*XX:NNNN* for equality, assuming that *xx* <=*XX*.

```
if    (   (  (xx = XX   )                      and (nnnn = NNNN          )  )    {1}
       or (  (xx = XX-1)  and (XX <> $8F)  and (nnnn = NNNN + $8000)  )      {2}
       or (  (xx = $00  )  and (XX =  $8F)  and (nnnn = NNNN - $2000)  )   ) {3}
then equal   : =   true
```

The three conditions are as follows:

{1} The two pointers are expressed identically;

{2} The two pointers are expressed in terms of adjacent bank pairs;

{3} The first pointer is expressed in bank-switched form, and the second is expressed in extended form.

Note that without the preliminary sorting of the two pointers according to their extension bytes, two more cases (a total of 8 more byte comparisons) are necessary to test for equality.

2.4.5 Summary of Address Storage

Addresses in the three forms given above are stored in memory in these ways:

- *S-bank* and *current bank* addresses are stored in normal 6502 style: as two consecutive bytes, low byte followed by high byte. Heed the warnings on bank-switched addressing given in section 2.4.1.
- *Segment addresses* point to pages and are stored as two consecutive bytes, bank part followed by page part.
- *Extended addresses* are stored in the zero page and X-page. The address is stored in the zero page as two consecutive bytes, low byte followed by high byte. The X-byte is stored in the X-page (page $0F:16, in the interpreter environment) at the byte position parallel to the high byte of the address in zero page. An extended address is referred to by the location of the low byte of the address part: for instance, the pointer at location $0050 has its low part at $0050, high part at $0051, and X-byte at $1651 (in the interpreter environment).

Devices

3.1 Devices and Drivers

A *device* is a part of the Apple III, or a piece of external equipment, that can transfer information into or out of the Apple III. Devices include the keyboard and screen, disk drives, and printers.

Devices provide the foundation upon which the SOS file system is constructed. In general, your program will talk to devices only through the SOS file system.

3.1.1 Block and Character Devices

SOS recognizes two kinds of devices: *character devices* and *block devices*. A character device reads or writes a stream of characters, one character at a time: it can neither skip characters nor go back to a previous character. A character device is usually used to get information to and from the outside world: it can be an input device, an output device, or an input/output device. The console (screen and keyboard), serial interface, and printer are all character devices.

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

3.1.2 Physical Devices and Logical Devices

A *physical device* is a physically distinct piece of hardware: if an external device, it usually has its own box. A *logical device* is what SOS and the interpreter regard as a device: it has a name. For example, the keyboard and the screen are separate physical devices; but SOS regards them as one logical device—the console. On the other hand, if a disk drive contained two disks, each could be a separate logical device.

3.1.3 Device Drivers and Driver Modules

Programs called *device drivers* provide the communication link between the SOS kernel and input/output devices: they take the streams of characters coming from SOS and convert them to physical actions of the device, or convert device actions into streams of characters for SOS to process. Device drivers for the standard Apple III devices are included in the SOS.DRIVER file: you can change or delete these, or add new ones, by using the System Configuration Program (SCP) option on the Utilities disk, as explained in the *Apple III Owner's Guide* and the *Apple III Standard Device Drivers Manual*.

The Disk III driver is included in the SOS.KERNEL file. It cannot be removed or changed by the user, except to specify the number of drives in the system.

Each logical device connected to the system has its own device driver: SOS can access the logical device through its driver. Related device drivers, such as drivers for separate logical devices on one physical device, can be grouped into a *driver module*. The drivers in a module can share code or system resources, such as interrupt lines. A driver module must be configured into the system as a package: unneeded drivers cannot be deleted from it. Each driver in the module is named separately.

The SOS kernel and the interpreter only deal with logical devices and their drivers. Whether the logical device is one physical device, several physical devices, or part of a physical device, is academic to the interpreter writer: it is only necessary to know that all three cases are possible. Similarly, SOS and the interpreter communicate with a device driver in precisely the same way whether or not the driver is part of a driver module.

3.1.4 Device Names

A logical device and its driver are both identified by a *device name*. If a driver module has several drivers, each has a different device name, by which it can be separately addressed. The driver module itself has no name, as it is never addressed as such. (The SCP refers to a module by the name of the first driver in it.)

A device name is up to 15 characters long: the first is a period; the second is a letter; the rest can be either letters or digits, in any combination (see Figure 3-1).

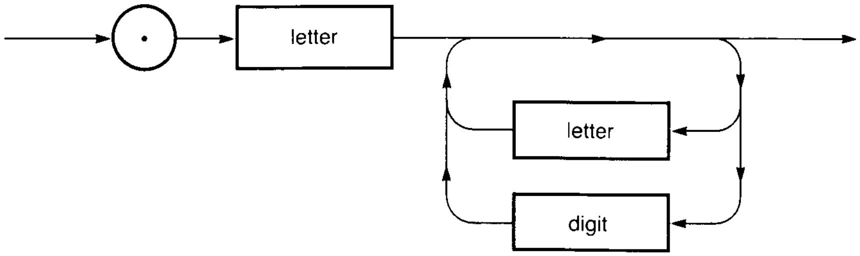


Figure 3-1. Device Name Syntax

Some legal device names are

.D1
.PRINTER
.BLOCKDEVICE

Some illegal device names are

PRINTER	(the first character is not a period)
.BLOCK.DEVICE	(*only* the first character can be a period)
.BLOCK DEVICE	(a device name cannot contain a space)
.BLOCK/DEVICE	(a device name cannot contain a /)

A logical block device also has a *volume name*, discussed in section 4.1.3.2, which is the name of the medium (for example, a flexible disk) in the device. In general, the volume name, rather than the device name, should be used for communicating with the device.

3.2 The SOS Device System

Since SOS accesses all devices through their drivers, the devices can be organized as a single-level tree, as illustrated by Figure 3-2:

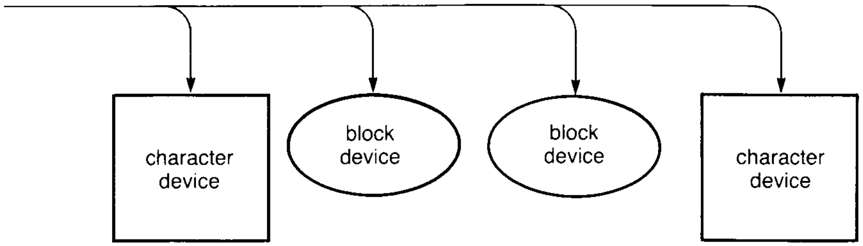


Figure 3-2. The SOS Device System

This system of devices underlies the system of files that will be developed in the next chapter.

3.3 Device Information

Certain information about a logical device and its driver is stored in the driver's *Device Information Block* (DIB), which is broken into the *DIB header* and the *DIB configuration block*. The header contains information that SOS uses to distinguish between block and character devices and between devices in each class. It can be read by the GET_DEV_NUM and D_INFO calls, but cannot be changed. The configuration block contains data that can be changed by the SCP, such as the baud rate of a device. The size and contents of the configuration block differ for each device. Some information in the DIB header can be used only by SOS; the information that can be read by the interpreter is described below.

dev_name and **dev_num**

A device name is up to 15 characters long: the first is a period; the second is a letter; the rest can be either letters or digits, in any combination. The device name can be changed only by the SCP.

Linked with every device name is one and only one device number. Access to information in the DIB is usually gained via the device number, which can be obtained from the device name through the GET__DEV__NUM call. Access to data stored or transmitted by a device is gained via the device name by accessing a similarly-named file, as explained in Chapter 4.

slot_num and **unit_num**

A device can use an interface card plugged into one of the four peripheral interface connectors (called slots) inside the Apple III: such devices have a slot number, which indicates which of the four slots the card is plugged into. A device that does not use an interface card has a slot number of zero.

Related device drivers can be grouped into a driver module: each such driver has a unit number that indicates the placement of that driver, and its device, in its group. Each driver in a driver module has a separate DIB, but the drivers may share code. For example, the formatter drivers on the Utilities disk have separate DIBs but share the same code: they can be called separately via their unit numbers.

The SOS unit number has nothing to do with the logical unit number that the Apple III Pascal System assigns to devices.

For more information about the internal operation of devices, see the *Apple III SOS Device Driver Writer's Guide*.

dev_type and **sub_type**

Apple assigns two identifiers to each device indicating the device's functions. The device type lets you determine whether a given device is a printer, a communications interface, a storage device, a graphics device, or whatever; the device subtype distinguishes between devices of the same type (to separate letter-quality printers from line printers, for example).

An interpreter that wishes to communicate with a certain type of device, but does not know the name or number of a device of that type, can examine these identifiers to find a suitable device.

manuf_id and **version_num**

Apple assigns two identifiers to each device and device driver: one to identify the manufacturer of the device and driver, and one to indicate their version number. An interpreter can use these identifiers to ensure compatibility with different versions of the same device.

total_blocks

This field indicates the total number of blocks on a block device.

If you wish a **dev_type**, **sub_type**, **manuf_id**, or **version_num** to be assigned to a device and driver, contact the Apple Computer PCS Division Product Support Department. This will ensure that the identifiers of each device and driver are unique and are available to interpreter-writers.

3.4 Operations on Devices

An interpreter can perform these operations on any device:

- Find the device number associated with a given device name, using a GET__DEV__NUM call, or find the device name associated with a given device number, using a D__INFO call;
- Obtain the slot number, unit number, device type, device subtype, manufacturer's identification, and version number of a device, using a D__INFO call.

An interpreter can perform these operations on a character device:

- Receive device status information, using a D__STATUS call;
- Send device control information, using a D__CONTROL call.

Using the System Configuration Program, you can

- Add a new device to the system;
- Remove a device from the system;
- Alter the configuration block of a device;
- Change the name, device type or subtype, or slot number of a device.

See the *Apple III Standard Device Drivers Manual*, for information on device and control requests for specific devices, and the *Apple III SOS Device Driver Writer's Guide* for a complete specification on the SOS/driver interface.

3.5 Device Calls

The calls summarized below all operate on devices directly. The name of each call below is followed by its parameters (shown in boldface). The input parameters are directly-passed values and pointers to tables. The output parameters are all directly-passed results. The first list is of required parameters; the second, present only for D_INFO, is of optional parameters. The SOS call mechanism is explained in Chapter 8; the individual calls are described fully in Chapter 12 of Volume 2.

D_STATUS

[**dev_num**, **status_code**: value; **status_list**: pointer]

This call returns status information about the specified device by passing a pointer to a status list. The information can be either general or device-specific information. D_STATUS returns information about the internal status of the device or its driver; D_INFO returns information about the external status of the driver and its interface with SOS.

D_CONTROL

[**dev_num**, **control_code**: value; **control_list**: pointer]

This call sends control information to the specified device by passing a pointer to a control list. The information can be either general or device-specific information. D_CONTROL operates on character devices only.

GET_DEV_NUM

[**dev_name**: pointer; **dev_num**: result]

This call returns the device number of the driver whose name is specified by **dev_name**. The file associated with the device need not be open. The device number returned is used in the D_READ, D_WRITE, D_STATUS, D_CONTROL, and D_INFO calls.

D_INFO

[**dev_num**: value; **dev_name**, **option_list**: pointer; **length**: value]

[**slot_num**, **unit_num**, **dev_type**, **sub_type**, **total_blocks**,**manuf_id**, **version_num**: optional result]

This call returns the device name (and optionally, other information) about the device specified by **dev_num**. The file associated with the device need not be open. D_INFO returns information about the device's external status and interface to SOS; D_STATUS returns information about the internal status of the device and its driver.

Files

4.1 Character and Block Files

A *file* is a named, ordered collection of bytes, used to store, transmit, or retrieve information. A file is identified by its name; a byte within the file is identified by its position in the ordered sequence.

SOS recognizes two types of files: character files and block files. A *character file* is treated by SOS as an endless stream of characters, or bytes. SOS can read or write the current byte but cannot go back to a previous byte or forward to a later byte. A character file is an abstraction used to represent a character device. A character file can be read-only, write-only, or read/write, as determined by the device it resides on. A character file is identified by its device name, which is defined in the previous chapter.

A *block file* is treated by SOS as a finite sequence of bytes, each one numbered. Any byte, or group of bytes, in a block file can be accessed by a call to SOS. A block file is so called because it resides in a volume on a block device: the volume is formatted into 512-byte blocks, also numbered. The blocks themselves are of concern only to SOS: the interpreter only reads or writes bytes.

The interpreter need only ask for the particular bytes it wants, using the file READ and WRITE calls. SOS translates these byte-oriented calls into block-oriented *device requests* executed by the device driver. SOS moves the requested bytes between its I/O buffer and the interpreter's data buffer; the driver moves whole blocks containing these bytes to and from the I/O buffer. Device requests are described in the *Apple III SOS Device Driver Writer's Guide.*

4.1.1 Structure of Character and Block Files

Character and block files are quite different in implementation, but are treated similarly. In fact, sequential read and write operations are the same: an interpreter reads a sequence of bytes from its current position in a block file in the same way as it reads a sequence of bytes from a character file.

The bytes in a character file are not numbered and must be accessed sequentially. Each read or write operation can handle a single byte or a sequence of up to 64K bytes. The next operation starts where the last left off. Figure 4-1 shows the structure of a character file.

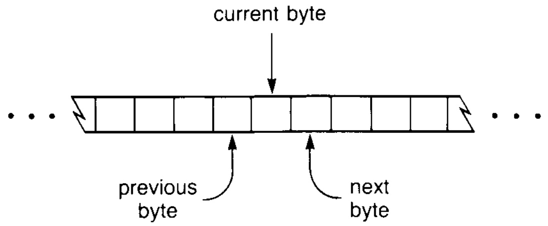


Figure 4-1. Character File Model

The bytes in a block file are numbered from $000000 up to $FFFFFE. A block file can contain up to 16,772,215 bytes (one less than 16 Megabytes). Each read or write operation can handle a single byte or a sequence of up to 64K bytes. The next operation can start anywhere in the file, with no reference to the last. For this reason, a block file is a random-access file. Figure 4-2 shows the structure of a block file.

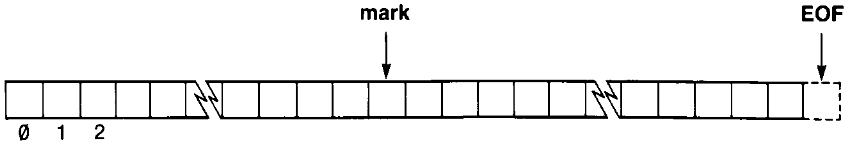


Figure 4-2. Block File Model

A block file's size is defined by its end-of-file marker, or **EOF**, which is the number of bytes that can be read from the file. The interpreter's place in the file is defined by the current position marker, or **mark**, which is the number of the next byte that will be read or written.

Both of these may be moved automatically by SOS or manually by the interpreter.

4.1.2 Open and Closed Files

A file can be *open* or *closed*: an open file can be read from or written to; a closed file cannot.

Initially, a file is closed: access to a closed file is through its *pathname*, defined in section 4.2.3.

When SOS opens a file in response to an OPEN call from an interpreter, SOS creates an *access path* to the file by placing an entry into the *File Control Block* (FCB), which is a table in memory containing information about all open files, and returns a reference number (**ref_num**) to the program that opened the file. This access path determines the way the file may be accessed (read from, written to, renamed, or destroyed). Every time that program accesses that file, it must use that access path and **ref_num**. Some files may have more than one access path, as shown in the Figure 4-3.

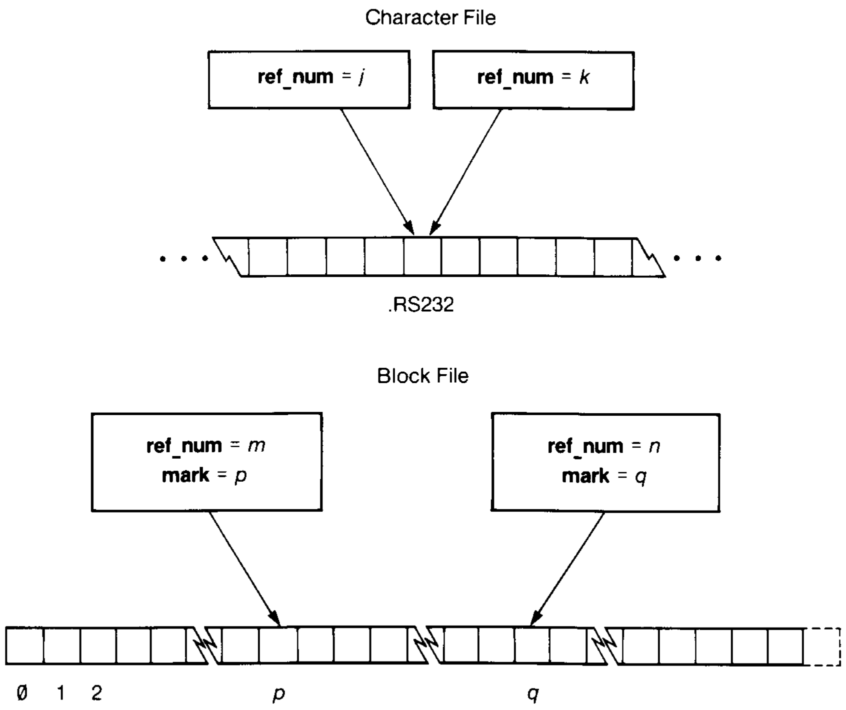


Figure 4-3. Open Files

The character file above has two access paths, along each of which a program can read or write at the current byte, or character. The block file has two access paths, each of which can have a different current position, or **mark**, in the file. Each access path can move its own **mark**, and can read at the position it indicates. Both access paths share a common end-of-file marker, or **EOF**.

In general, a block file can have either (a) one access path open for reading and writing or (b) one or more read-only access paths: it cannot have more than one access path if any access path can write to the file. A character file may have several access paths with write-access.

v1.2 SOS allows a maximum of 16 block-file access paths and 16 character-file access paths to be open at one time.

Each OPEN call to a file creates a new access path (with its own **ref_num**) to that file, which is separate from all the file's other access paths.

When an access path to a file is closed, its FCB entry is deleted and its **ref_num** is released for use by other files.

Certain operations, such as reading and writing, can only be performed on open files; others, such as renaming, can only be performed on closed files.

4.1.3 Volumes

A volume is a piece of random-access storage medium formatted to hold files. A volume is mounted on a block device, and is accessed through that device. Both flexible disks and hard disks are volumes.

Each logical block device corresponds to one volume at any time. If the device uses removable media (like flexible disks), it can access different volumes at different times.

However, a single physical device can correspond to multiple logical devices, each with its own driver and device name. Each of these logical devices would have a volume with a different name. For example, if a disk drive contains a fixed disk and a removable disk, it would normally be treated as two logical devices, each with its own volume. It would have a driver module containing two drivers. The two logical devices would have different names and unit numbers; and the two volumes would have different names.

It is even possible for a single medium to be divided into multiple volumes: a disk holding more than 64K blocks might be so divided, as SOS cannot support volumes larger than 64K blocks. In this case, the physical device is treated as multiple logical devices: the physical device has a single driver module, and each logical device has a uniquely named driver and volume.

On the other hand, a driver for a disk drive containing several fixed disks might treat the disks as one large volume with one name.

Having noted these special cases, we need not discuss them further. They are discussed in the *Apple III SOS Device Driver Writer's Guide*, as the relationships between logical devices and physical devices are established by device drivers. Since SOS and the interpreter deal only with volumes and logical devices, we can ignore physical devices without losing generality. From now on, the word *device* will mean *logical device*.

Every volume must have two special items, each in a fixed place on the medium: a *volume directory* file and a *bit map*. The volume directory file contains information about the volume (such as its name and size), and information about files on the volume. The bit map represents every block on the volume with a bit indicating whether the block is currently allocated to a file, or is free for use.

4.1.3.1 Volume Switching

Some devices (such as flexible-disk drives) have removable media. These devices can access several volumes, though only one at a time. This leads to problems, however, when a file has been opened on one volume in a device, and subsequently that volume has been removed and another substituted for it. If SOS needs to access the open file on the original volume, it will not be able to find the volume it needs.

When this happens, SOS will request that you restore the volume to its original drive. It halts all operations of the computer and displays a message on the screen (see Figure 4-4) naming the volume it needs and the device into which it should be placed. The system will wait until you replace the volume and press the CAPS LOCK (on some keyboards called ALPHA LOCK) key on the keyboard twice.

```
Insert volume: UTILITIES
    in device: .D1
then press the ALPHA LOCK key twice
```

Figure 4-4. The SOS Disk Request

The volume-switching capability is very useful when you need to use many files on various volumes: it allows you to exchange volumes at will (when the device is idle), and still have all files accessible when they are needed.

4.1.3.2 Volume Names

A block device is accessible by two names. The first is the device name, defined in Chapter 3. The second, more useful, name is the *volume name*. The volume name of a block device is the name of the volume currently in the device: the volume name of a flexible-disk drive will change as you insert and remove flexible disks. A block device containing no volume (such as an empty flexible disk drive) has no volume name and, to SOS, does not exist.

A volume name is up to 15 characters long: the first is a letter; the rest can be letters, digits, or periods, in any combination. A volume name is always preceded by a slash (/), but the slash is not part of the name. SOS automatically converts all lowercase letters in a volume name to uppercase. The syntax of a volume name is identical to that of a file name: a diagram is shown in section 4.2.2.

Here are a few legal volume names, with slashes:

/PROGRAMS
/BLOCK.FILES
/CHAP.2B

Here are some volume names that will *not* work, and the reasons why:

/BAD NAME	(contains a space)
/1.TO.10	(first character is a number)
/STEVE'S.PROGRAM	(contains an apostrophe)
/ANTHROPOMORPHOUS	(more than 15 characters)

We strongly recommend using the volume name, rather than the device name, whenever you refer to a block file. This has two advantages:

- The user is protected against volume-swapping.
- The program is more general: it can be used with new mass-storage devices without modification.

4.2 The SOS File System

SOS organizes all files it can access into a hierarchical tree structure, called the SOS file system. The top level of this system is shown in Figure 4-5.

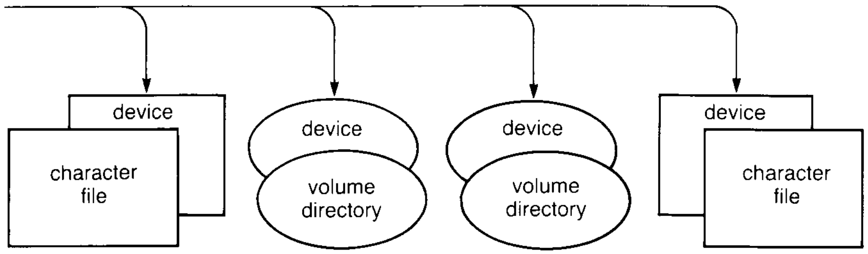


Figure 4-5. Top-Level Files

The top level contains character files and volume directories. Each character file represents one character device; each volume directory represents a volume on a block device, and can directly or indirectly access all files on the volume. Each character file is referred to by its device name; each volume directory is referred to by its volume (preferably) or device name.

By comparing this diagram with that of the SOS device system, you can see that the file system is built on top of the device system: each file overlays a device.

4.2.1 Directory Files and Standard Files

Since a volume on a block device can contain many files, SOS provides a special type of file, the *directory file*, to keep track of them. A directory is a file listing the names and locations of, as well as other information about, other files on the volume. The main directory on the volume is the *volume directory*, whose name is the same as its volume. The volume directory lists both *standard files*, which are block files containing data, and *subdirectory files*, which list other files. (A subdirectory file might not list any files: for example, if you have created a subdirectory file to list a series of future text files but have not yet created them.) If a directory lists a file, we may also say that it "owns" that file, or that is the "parent" of that file.

Now we can fill in our model of the file system, by adding subdirectories and the files they list (see Figure 4-6):

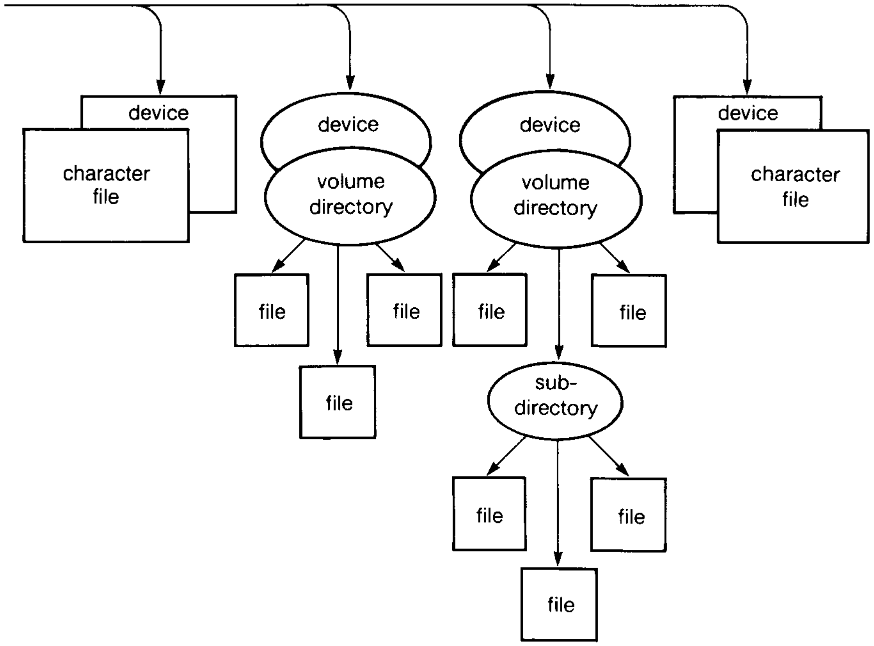


Figure 4-6. The SOS File System

We now have the whole tree: each node is a directory, and each leaf is a character or block file. We will give them names in a minute.

4.2.2 File Names

Each entry in a directory is listed by its *file name*, which distinguishes it from the other entries in that directory. For this reason, each file name in a directory must be unique. A file name is up to 15 characters long: the first is a letter; the rest are letters, digits, or periods, in any combination (see Figure 4-7). SOS automatically converts all lowercase letters in a file name to uppercase.

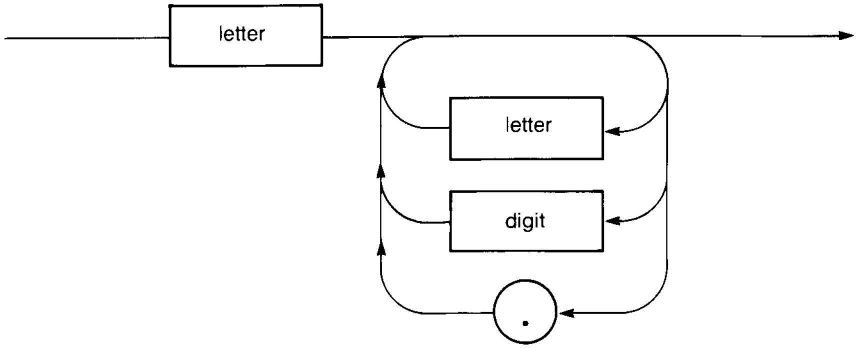


Figure 4-7. File Name Syntax

Here are a few legal file names:

MIKE.2.JULY.80
SORTPROGRAM
LETTER.TO.SUE

Here are some file names that will *not* work, and the reasons why:

BAD NAME	(contains a space)
1.TO.10	(begins with a number)
STEVE'S.PROGRAM	(contains an apostrophe)
ANTHROPOMORPHOUS	(more than 15 characters)

In earlier editions of the *Apple III Owner's Guide*, file names are called local names.

4.2.3 Pathnames

A *pathname* is a sequence of names that defines a path from the root of the file system, through a volume directory and possibly subdirectories, to a specific file.

A pathname uniquely identifies a file. Even if two files with the same file name appear in the system, they can be distinguished by their pathnames.

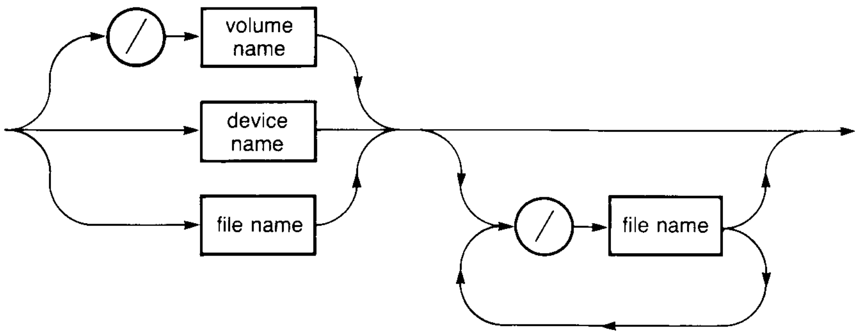


Figure 4-8. Pathname Syntax

A pathname is composed of names and slashes (see Figure 4-8). A pathname begins with a slash and a volume name; a device name; or a file name; more file names may follow. One slash must separate any two successive names, and the last component of a pathname must be a name. As always, a volume name is preceded by a slash, and a device name begins with a period.

Paths always begin at the root of the file system. The first component of the pathname determines the nature of the path.

/vol_name	If the first component is a slash followed by a volume name, the path proceeds from the volume directory.
dev_name	If the first component is the name of a block device (which begins with a period), SOS automatically replaces the device name with the name of the volume directory of the volume in that device, and the path proceeds from that directory.
dev_name	If the sole component is the name of a character device, the pathname specifies its character file. No further file specifications are allowed after a character device name.
file_name	If the first component is a file name, SOS appends the prefix (see below) to the pathname, and the new pathname is evaluated again.

Here is our file system tree again (see Figure 4-9), this time with the file names filled in:

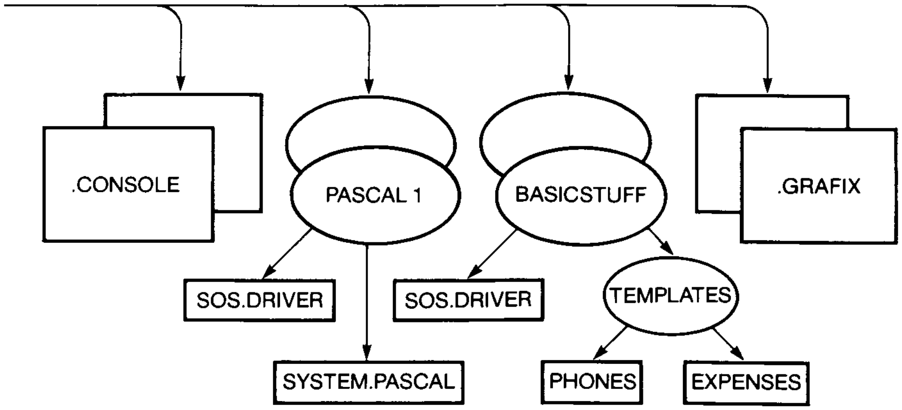


Figure 4-9. Pathnames

The valid pathnames in this file system are

.CONSOLE
.GRAFIX
/PASCAL1
/PASCAL1/SOS.DRIVER
/PASCAL1/SYSTEM.PASCAL
/BASICSTUFF
/BASICSTUFF/SOS.DRIVER
/BASICSTUFF/TEMPLATES
/BASICSTUFF/TEMPLATES/PHONES
/BASICSTUFF/TEMPLATES/EXPENSES

If the volume /PASCAL1 were installed in the device .D1 , then every pathname that included the volume /PASCAL1 would have a synonymous pathname using .D1 : for example, /PASCAL1/SOS.DRIVER would specify the same file as .D1/SOS.DRIVER .

4.2.4 The Prefix and Partial Pathnames

The *prefix* is a pathname that specifies a volume directory or subdirectory file. When SOS boots, the prefix is set to the volume directory of the boot volume.

A *partial pathname* is a pathname that begins with a file name, whereas a full pathname begins with a volume or device name. In other words, a partial pathname begins with a letter, whereas a full pathname begins with a slash or period. When SOS receives a partial pathname, it concatenates the prefix to that pathname with a slash, forming a full pathname. The effect is to allow you to specify a "current directory", or prefix, and refer to files owned by that directory without having to specify the directory's pathname each time. For example, the prefix /PASCAL1 and the partial pathname SOS.DRIVER form the full pathname /PASCAL1/SOS.DRIVER.

The prefix always specifies a volume directory or subdirectory file. The prefix never specifies a standard or character file.

The SOS prefix is not the Pascal prefix. The two may or may not have the same value.

4.3 File and Access Path Information

An interpreter often needs information about a file or an access path. Information about a block file is stored in the file's directory entry. Information about a block file access path is stored in its FCB entry. This section describes file information and access path information for block files only. Information about a character file is stored as the device information of its respective character device (see section 3.3). No corresponding information about an access path to a character file is available through SOS.

The various items of information about a file will be named in boldface, and the same names will be used when these items appear as fields in directories (in Chapter 5) and as parameters for SOS calls (in Chapter 8 and in Volume 2).

4.3.1 File Information

Certain information about a block file, such as a file's name, belongs to the file itself rather than to any of its access paths. This information is stored in that file's directory entry (see section 5.2.4).

An interpreter can read the file information in the directory entry with a GET_FILE_INFO call or change it with a SET_FILE_INFO call, both described in Chapter 10 of Volume 2. No change, however, can be made to any of the file information if the file is open: a SET_FILE_INFO call to do so will have no effect until the file is closed.

This information about a file is kept in the directory entry:

file_name

A closed block file is accessed by its **file_name**. The file name of a block file can be changed, but only when the file is closed. Only the last file name in a pathname can be changed, because the preceding names are the names of open directory files, which are shared with other files.

All access to information about a closed block file is through its **file_name**.

access

Every block file has an **access** attribute field, which determines the ways in which you may use that file. The **access** attributes can be set to prevent you from reading from, writing to, renaming, or destroying a file. It can also tell you whether a file's contents have been changed since the last time a backup copy of the file was made.

EOF and **blocks_used**

The number of bytes in a block file is specified by the end-of-file pointer, or **EOF**. The number of blocks physically used by the file is specified by the **blocks_used** item. In *sparse files*, which we will see later, the **EOF** and **blocks_used** numbers may not correspond as you might expect.

GET_FILE_INFO returns the current value of **EOF** and **blocks_used** only if the file is closed. If it is open, GET_EOF returns the correct value of **EOF**. GET_FILE_INFO returns the values **EOF** and **blocks_used** had when the file was opened.

storage_type, file_type, and **aux_type**

Three items describe the external and internal arrangement of each block file. The **storage_type** indicates whether the file is a directory file or a standard file, and how the file is stored on its block device: this item is used only by SOS. The **file_type** classifies the contents of the file; and the **aux_type** can be used by an interpreter as an additional description of the contents of the file: these two items are used only by the interpreter.

A description of the identification codes and their meanings is given later in this chapter.

creation and **last_mod**

These items record the dates and times at which a block file was initially created and last updated. These values are drawn from the system clock or the last known time.

4.3.2 Access Path Information

Other information about a block file, such as an interpreter's position in a file, belongs to the access path rather than the file itself. This information is stored in the access path's entry in the File Control Block.

Access path information can be changed only while that access path is open. When the access path is closed, certain items, such as the **mark**, disappear, and others, such as the **EOF**, update the file information in the directory entry.

This information about the access path is kept in the FCB entry:

ref_num

When an access path to a file is opened, SOS assigns that access path a unique reference number, or **ref_num**. All subsequent references to that access path must be made with that **ref_num**.

EOF and **mark**

Each access path to an open block file has one attribute defining the end of file, the **EOF**, and another defining the current position in the file, the **mark**. Both of these may be moved automatically by SOS or manually by the interpreter.

The **EOF** pointer is the number of bytes in the file. This is equivalent to pointing one position beyond the last byte in the file, since the first byte is byte number 0: in an empty file (containing zero bytes), **EOF** points at byte number 0. The value of the **mark** cannot exceed the value of **EOF**.

The **EOF** is peculiar in that it appears both in the file's directory entry and in the access path's FCB entry. When a file is open for writing, the two values of the **EOF** may differ. The current **EOF** is stored in the access path's FCB entry: this **EOF** is returned by a GET_EOF call to the **ref_num**. The value of **EOF** in the file's directory entry is updated only when the access path is closed: this **EOF** is returned by a GET_FILE_INFO call to the **file_name**.

It is impossible for two access paths to have different **EOF** values, for in order to change the **EOF**, an access path must have write-access. If it does have write-access, it must be the only access path to that file.

The **mark** automatically moves forward one byte for every byte read from or written to the file. Thus, the **mark** always indicates where the next byte will be read or written.

If, during a WRITE operation, the **mark** meets the **EOF**, both the **mark** and the **EOF** are moved forward one position for every additional byte written to the file. Thus, adding bytes to the end of the file automatically moves the **EOF** up to accommodate the new information. Figure 4-10 shows the automatic movement of **EOF** and **mark**.

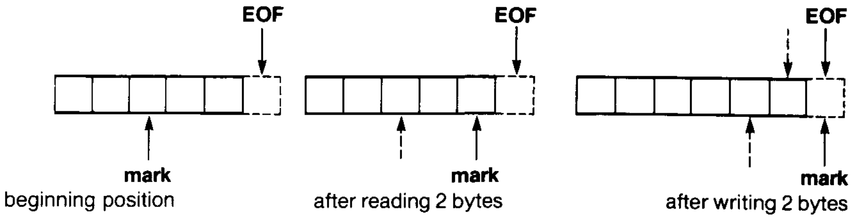


Figure 4-10. Automatic Movement of EOF and Mark

An interpreter can manually move the **EOF** to place it anywhere from the current **mark** position to the maximum byte position possible (see Figure 4-11). The **mark** can also be placed anywhere from the first byte in the file to the current position of the **EOF**.

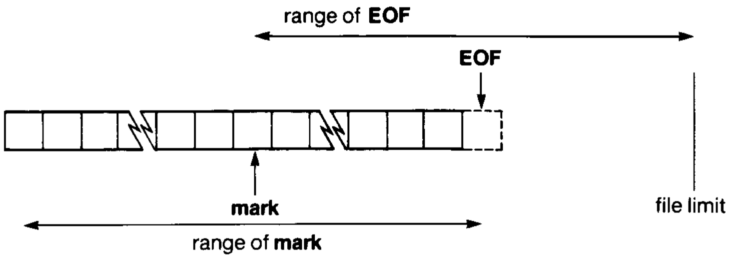


Figure 4-11. Manual Movement of EOF and Mark

The **EOF** is read by the GET_EOF call and manually set by the SET_EOF call; the **mark** is read by the GET_MARK call and manually set by the SET_MARK call.

level

Each access path is given a **level** when it is opened. The **level** of the access path is the value of the system file level at the time the access path was opened. An interpreter can group files by file levels (for example, have user files open at one level, while system files are open at another), and perform group operations on files of like levels.

The system file level has the value 1, 2, or 3. When the system is booted, the level is set to 1. It can be changed by the SET_LEVEL call, and read by the GET_LEVEL call. One use of the file level is to close all files opened by a user program when the interpreter exits that program.

This is done as follows: When the interpreter enters the program, it raises the system file level. Thus all files opened by the program will have a higher **level** than, say, .CONSOLE or the interpreter file. When the interpreter exits the program, it issues a FLUSH call or CLOSE call with a **ref_num** of $00, which closes all files at a **level** equal to or higher than the system file level. Then the interpreter lowers the system file level.

4.3.3 Newline Mode Information

Certain information about a file, called *newline-mode information*, is associated either with the file itself or with an access path to the file, depending on the kind of file. A character file's newline-mode information is associated with the file and its device; a block file's newline-mode information is associated with an access path to the file, and can differ from one access path to another.

When SOS reads from an open file, it can read input as a continuous stream of characters or as a series of lines. In the first case, you ask SOS to read a specific number of bytes: when this number have been read or when the current position has reached the end of file, the READ operation terminates. In the second case, called *newline mode*, the READ will also terminate if a specified character, the *newline character*, is read. The newline character is usually the ASCII CR ($0D), but can be any hex value from $00 to $FF. The newline character is called the *termination character* or *line-termination character* in the *Apple III Standard Device Drivers Manual*.

Newline mode is supported on both character and block files, so that file input/output can be device independent. For example, a program that reads a line of text from a file can treat the keyboard and a disk file exactly the same way.

is_newline and **newline_char**

Newline mode is controlled by two values: **is_newline** turns newline mode on or off; **newline_char** sets the newline character. These two values are set by the NEWLINE call to the access path's **ref_num**.

For a block file, each access path can have separate **is_newline** and **newline_char** values. A character file also has **is_newline** and **newline_char** values, which are also changed by a NEWLINE call to an access path's **ref_num**, but they are the same for all access paths. If either value is changed for one access path, it is changed for all.

4.4 Operations on Files

These operations can be performed on all files:

- OPEN and CLOSE to control access, and READ and WRITE (if its **access** attributes allow) to transfer information from or to the file.
- Change **is_newline** and **newline_char** for an access path, using the NEWLINE call.

These operations can be performed only on block files:

- Examine or change file information, including the name, access, file type, and modification date, using the GET__FILE__INFO and SET__FILE__INFO calls.

These operations can be performed only on closed block files:

- CREATE a new file;
- DESTROY an existing file;

These operations can be performed only on standard files open for writing:

- Set and read the **EOF** pointer, using the SET__EOF and GET__EOF calls.
- Set and read the current position **mark**, using the SET__MARK and GET__MARK calls.

These operations can be performed on directory files:

- OPEN and CLOSE the file.
- READ the file, if it is open.
- DESTROY the file, if it is empty and closed.

4.5 File Calls

These calls deal with files: the calls CREATE through OPEN operate on closed files; the calls NEWLINE through GET__LEVEL operate on open files. The name of each call below is followed by its parameters (in boldface). The input parameters are directly-passed values and pointers to tables. The output parameters are all directly-passed results. The first list is of required parameters; the second list, present for some calls, is of optional parameters. The SOS call mechanism is explained in Chapter 8; the individual calls are described fully in Volume 2, Chapter 9.

CREATE

[**pathname**, **option_list**: pointer; **length**: value]

[**file_type**, **aux_type**, **storage_type**, **EOF**: optional value]

This call creates a standard file or subdirectory file on a block device. A file entry is placed in a directory, and at least one block is allocated.

DESTROY

[**pathname**: pointer]

This call deletes the file specified by the **pathname** parameter by marking the file's directory entry inactive. DESTROY releases all blocks used by that file back to free space on that volume.

The file can be either a standard or a subdirectory file. A volume directory cannot be destroyed except by physically reformatting the medium. A character file can be removed from the system by the System Configuration Program.

RENAME

[**pathname**, **new_pathname**: pointer]

This call changes the name of the file specified by the **pathname** parameter to that specified by **new_pathname**. Only block files may be renamed; character files are "renamed" by the System Configuration Program.

SET_FILE_INFO

[**pathname**, **option_list**: pointer; **length**: value]

[**access**, **file_type**, **aux_type**, **last_mod**: optional value]

This call modifies information in the directory entry of the file specified by the **pathname** parameter. Only block files' information can be modified; character files have no such information associated with them.

You may perform a SET_FILE_INFO on a currently-open file, but the new information will not take effect until the next time the file is OPENed.

GET_FILE_INFO

[**pathname**, **option_list**: pointer; **length**: value]

[**access**, **file_type**, **aux_type**, **storage_type**, **EOF**, **blocks**, **last_mod**: optional result]

This call returns information about the block file specified by the **pathname** parameter.

VOLUME

[**dev_name**, **vol_name**: pointer; **blocks**, **free_blocks**: result]

When given the name of a device, this call returns the volume name of the volume contained in that device, the number of blocks on that volume, and the number of currently unallocated blocks on that volume.

SET_PREFIX

[**pathname**: pointer]

This call sets the operating-system pathname prefix to that specified in **pathname**.

GET_PREFIX

[**pathname**: pointer; **length**: value]

This call returns the current system pathname prefix.

OPEN

[**pathname**: pointer; **ref_num**: result; **option_list**: pointer; **length**: value]

[**req_access**, **pages**: optional value; **io_buffer**: optional pointer]

This call opens an access path to the file specified by **pathname** for reading or writing or both. SOS creates an entry in the file control block and an I/O buffer.

NEWLINE

[**ref_num**, **is_newline**, **newline_char**: value]

This call allows the caller to selectively enable or disable "newline" read mode. Once newline mode has been enabled, any subsequent read request will immediately terminate if the newline character is encountered in the input byte stream.

READ

[**ref_num**: value; **data_buffer**: pointer; **request_count**, **transfer_count**: value]

This call attempts to transfer **request_count** bytes, starting from the current position (**mark**), from the file specified by **ref_num** into the buffer pointed to by **data_buffer**. If newline read mode is enabled and the newline character is encountered before **request_count** bytes have been read, then the **transfer_count** parameter will be less than **request_count** and exactly equal to the number of bytes transferred, including the newline byte.

WRITE

[**ref_num**: value; **data_buffer**: pointer; **request_count**: value]

This call transfers **request_count** bytes, starting from the current file position (**mark**), from the buffer pointed to by **data_buffer** to the open file specified by **ref_num**.

CLOSE

[**ref_num**: value]

This call closes the file access path specified by **ref_num**. Its file-control block is released, and if the file is a block file that has been written to, its write buffer is emptied. The directory entry for the file, if any, is updated. Further file operations using that **ref_num** will fail. If **ref_num** is $00, all files at or above the system file level are closed.

FLUSH

[**ref_num**: value]

This call flushes the file access path specified by **ref_num**. If the file is a block file that has been written to, its I/O buffer is emptied. The access path remains open. If **ref_num** is $00, all files at or above the system file level are flushed.

SET_MARK

[**ref_num**, **base**, **displacement**: value]

This call changes the current file position (**mark**) of the file access path specified by **ref_num**. The **mark** can be changed to a position relative to the beginning of the file, the end of the file, or the current **mark**.

GET_MARK

[**ref_num**: value; **mark**: result]

This call returns the current file position (**mark**) of the file access path specified by **ref_num**.

SET_EOF

[**ref_num**, **base**, **displacement**: value]

This call moves the end-of-file marker (**EOF**) of the specified block file to the indicated position. The **EOF** can be changed to a position relative to the beginning of the file, the end of the file, or the current **mark**.

If the new **EOF** is less than the current **EOF**, then empty blocks at the end of the file are released to the system and their data are lost. The converse is not true: if the new **EOF** is greater than the current **EOF**, then blocks are not allocated, creating a sparse file; reading from these newly created positions before they are written to results in $00 bytes.

GET_EOF

[**ref_num**: value; **EOF**: result]

This call returns the current end-of-file (**EOF**) position of the file specified by **ref_num**.

SET_LEVEL

[**level**: value]

This call changes the current value of the system file level. All subsequent OPENs will assign this level to the files opened. All subsequent CLOSE and FLUSH operations on multiple files (using a **ref_num** of $00) will operate on only those files that were opened with a level greater than or equal to the new level.

GET_LEVEL

[**level**: result]

This call returns the current value of the system file level. See SET_LEVEL, OPEN, CLOSE, and FLUSH.

5

File Organization on Block Devices

When a program accesses a block device, it actually accesses the volume that corresponds to that device. You have already learned of the hierarchical tree structure used by SOS in its file organization, of the naming conventions used to access any file within the tree structure, and of the logical structure of a file as a sequence of bytes; this chapter explains the physical implementation of these structures on any volume.

The first part of the chapter (section 5.1) discusses what is on a volume, the second (section 5.2) describes directory files, the third part of the chapter (section 5.3) discusses standard files, and the final part of the chapter (section 5.4) provides a graphic summary of the organization of information on volumes.

The focus of this chapter is on how SOS works, not on how to use it. For this reason, we have chosen to explain details of implemention that are not strictly necessary for an interpreter writer to know, in order to make the working of SOS more concrete. The only section that is of immediate practical use to an interpreter writer is section 5.2 on the formats of directory files. The rest of the chapter explains the implementation of the file system: these sections should be regarded as examples, not as specifications.

In this manual, we will distinguish the SOS interface, which is supported, and the SOS implementation, which is not. We will support the hierarchical tree structure of the file system and the logical structures of character and block files. We will also support the storage formats of directory headers and entries, although they may be expanded by appending new fields. However, we may change volume formats and the storage formats of standard files.

Programmers should not rely on the details of implementation, as we may change the storage formats of files in order to improve performance. An interpreter that uses the READ and WRITE calls to access files, and interprets directories as we explain here, will work with future versions of SOS. An interpreter that relies on the current disk-allocation scheme or index-block structure may not work with future versions.

5.1 Format of Information on a Volume (SOS 1.2)

This section explains how SOS 1.2 organizes information on a 280-block flexible disk: it should be regarded as an example, not a general specification for volume formats.

In accessing a volume, SOS requests a logical block from the device corresponding to that volume. Logical blocks may be supported physically by tracks and sectors, or cylinders and heads, or other divisions. This translation is done by the device driver: the physical location of information on a volume is unimportant to SOS. This chapter discusses the organization of information on a volume in terms of blocks, numbered starting with 0.

When the volume is formatted, information needed by SOS is placed in specific logical blocks. A *bootstrap loader* program is placed in blocks 0 and 1 of the volume. This program loads SOS from the volume when CONTROL-RESET is pressed. Block 2 of the volume is the first block, or *key block*, of the *volume directory file*: it contains descriptions and locations of all the files in the volume directory, as well as the location of the volume bit map. The volume directory occupies a number of consecutive blocks (4 for SOS 1.2), and normally is immediately followed by the *volume bit map*, which records whether each block on the volume is used or unused. The volume bit map occupies consecutive blocks, one for every 4,096 blocks (or fraction thereof) on the volume. The rest of the blocks on the disk contain either subdirectory file information, standard file information, or garbage (such as parts of deleted files). The first blocks of a volume look something like this (Figure 5-1):

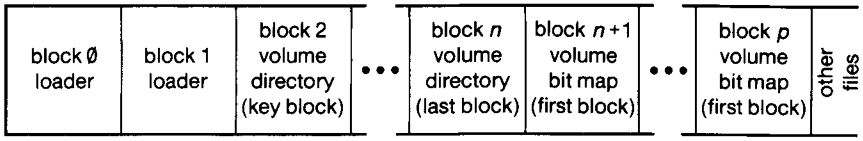


Figure 5-1. Blocks on a Volume

The precise format of the volume directory, volume bit map, subdirectory files and standard files are explained in the following sections.

5.2 Format of Directory Files

The format of the information contained in volume directory and subdirectory files is quite similar. Each directory file is a linked list of one or more blocks: each block contains pointers to the preceding and following blocks, a series of entries, and unused bytes at the end. The first block, called the *key block*, has no preceding block, so its preceding-block pointer is zero; the last block has no following block, so its following-block pointer is zero.

Most entries in a directory describe other files, which can be either standard files or directories: these entries are called *file entries*. The first entry in the key block of a directory contains information about the directory itself, not about another file: this entry is called the *directory header*.

The format of a directory file is represented in Figure 5-2.

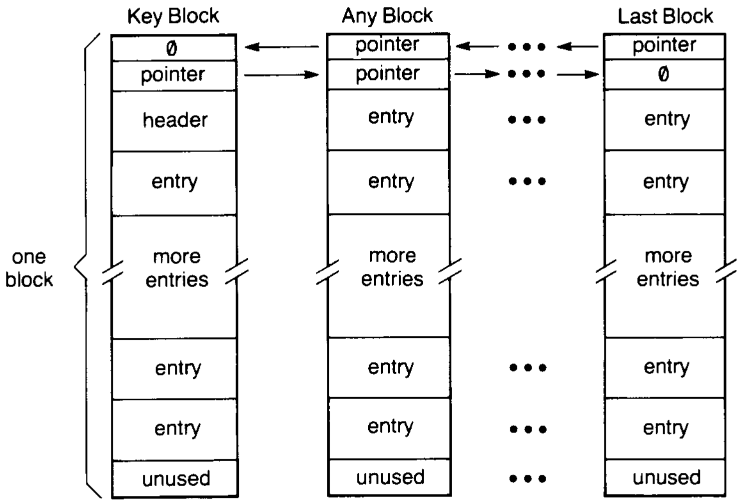


Figure 5-2. Directory File Format

The header entry is the same length as all other entries. As will be described below, the only organizational difference between a volume directory file and a subdirectory file is in the header.

5.2.1 Pointer Fields

The first four bytes of each block used by a directory file contain pointers to the preceding and succeeding blocks, respectively, of the directory file. Each pointer is a two-byte logical block number, low byte first, high byte second. The key block of a directory file has no preceding block: its first pointer is zero. Likewise, the last block in a directory file has no successor: its second pointer is zero. If a directory occupies only one block, both pointers are zero.

A pointer of value zero causes no ambiguity: no directory block could occupy block 0, as blocks 0 and 1 are reserved for the bootstrap loader.

All block pointers used by SOS have the same format: low byte first, high byte second.

5.2.2 Volume Directory Headers

Block 2 of a volume is the key block of that volume's directory file. One finds the volume directory header at byte position 0004 of the key block, immediately following the block's two pointers.

Figure 5-3 illustrates the structure of a volume directory header: following the figure is a description of each field. If you compare Figure 5-3 with Figure 5-4, you will notice that the two header types have the same structure for the first 12 fields, from **storage_type** to **file_count**; after that, the two diverge. However, similarly named fields have different meanings for the two types, so we have described each type separately.

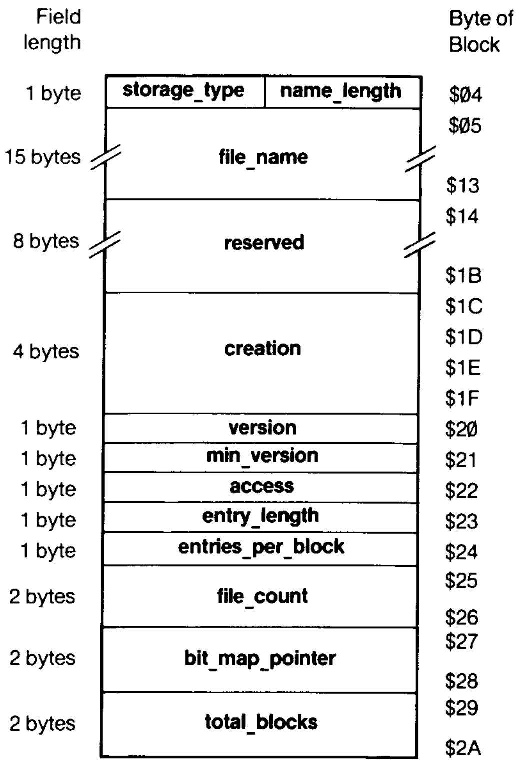


Figure 5-3. The Volume Directory Header

storage_type and **name_length** (1 byte):

Two four-bit fields are packed into this byte. A value of $F in the high four bits (the **storage_type**) identifies the current block as the key block of a volume directory file. The low four bits contain the length of the volume's name (see the **file_name** field, below). The **name_length** can be changed by a RENAME call.

file_name (15 bytes):

The first **name_length** bytes of this field contain the volume's name. This name must conform to the file name (or volume name) syntax explained in Chapter 4. The name does not begin with the slash that usually precedes volume names. This field can be changed by the RENAME call.

reserved (8 bytes):

This field is reserved for future expansion of the file system.

creation (4 bytes):

This field holds the date and time at which this volume was initialized. The format of these bytes is described in section 5.4.2.2.

version (1 byte):

This is the version number of SOS under which this volume was initialized. This byte allows newer versions of SOS to determine the format of the volume, and adjust their directory interpretation to conform to older volume formats.

v1.2 For SOS 1.2, **version** = 0.

min_version (1 byte):

This is the minimum version number of SOS that can access the information on this volume. This byte allows older versions of SOS to determine whether they can access newer volumes.

v1.2 For SOS 1.2, **min_version** = 0.

access (1 byte):

This field determines whether this volume directory may be read, written, destroyed, and renamed. The format of this field is described in section 5.4.2.3.

entry_length (1 byte):

This is the length in bytes of each entry in this directory. The volume directory header itself is of this length.

v1.2 For SOS 1.2, **entry_length** = $27.

entries_per_block (1 byte):

This is the number of entries that are stored in each block of the directory file.

v1.2 For SOS 1.2, **entries_per_block** = $0D.

file_count (2 bytes):

This is the number of active file entries in this directory file. An active file is one whose **storage_type** and **name_length** are not 0. See section 5.2.4 for a description of file entries.

bit_map_pointer (2 bytes):

This is the block address of the first block of the volume's bit map. The bit map occupies consecutive blocks, one for every 4,096 blocks (or fraction thereof) on the volume. You can calculate the number of blocks in the bit map from the **total_blocks** value, described below.

The bit map has one bit for each block on the volume: a value of 1 means the block is free; 0 means it is in use.

total_blocks (2 bytes):

This is the total number of blocks on the volume.

5.2.3 Subdirectory Headers

The key block of every subdirectory file is pointed to by an entry in another directory (explained below). A subdirectory header begins at byte position $0004 of the key block of that subdirectory file, immediately following the two pointers. Its internal structure is quite similar to that of a volume directory header. Figure 5-4 illustrates the structure of a subdirectory header. A description of all the fields in a subdirectory header follows the figure.

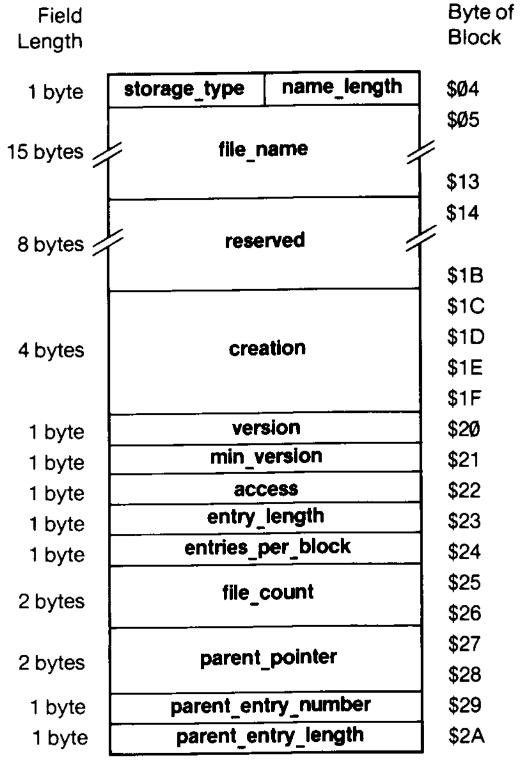


Figure 5-4. The Subdirectory Header

storage_type and **name_length** (1 byte):

Two four-bit fields are packed into this byte. A value of $E in the high four bits (the **storage_type**) identifies the current block as the key block of a subdirectory file. The low four bits contain the length of the subdirectory's name (see the **file_name** field, below). The **name_length** can be changed by a RENAME call.

file_name (15 bytes):

The first **name-length** bytes of this field contain the subdirectory's name. This name must conform to the file name syntax explained in Chapter 4. This field can be changed by the RENAME call.

reserved (8 bytes):

This field is reserved for future expansion of the file system.

creation (4 bytes):

This is the date and time at which this subdirectory was created. The format of these bytes is described in section 5.4.2.2.

version (1 byte):

This is the version number of SOS under which this subdirectory was created. This byte allows newer versions of SOS to determine the format of the subdirectory, and to adjust their directory interpretations accordingly.

v1.2 For SOS 1.2, **version** = 0.

min version (1 byte):

This is the minimum version number of SOS that can access the information in this subdirectory. This byte allows older versions of SOS to determine whether they can access newer subdirectories.

v1.2 For SOS 1.2, **min_version** = 0.

access (1 byte):

This field determines whether this subdirectory may be read, written, destroyed, and renamed. The format of this field is described in section 5.4.2.3. A subdirectory's **access** byte can be changed by the SET__FILE__INFO call.

entry_length (1 byte):

This is the length in bytes of each entry in this subdirectory. The subdirectory header itself is of this length.

v1.2 For SOS 1.2, **entry_length** = $27.

entries_per_block (1 byte):

This is the number of entries that are stored in each block of the directory file.

v1.2 For SOS 1.2, **entries_per_block** = $0D.

file_count (2 bytes):

This is the number of active file entries in this subdirectory file. An active file is one whose **storage_type** and **name_length** are not 0. See the next section for more information about file entries.

parent_pointer (2 bytes):

This is the block address of the directory file block that contains the entry for this subdirectory. This two byte pointer is stored low byte first, high byte second.

parent_entry_number (1 byte):

This is the entry number for this subdirectory within the block indicated by **parent_pointer**.

parent_entry_length (1 byte):

This is the **entry_length** for the directory that owns this subdirectory file. Note that with these last three fields one can calculate the precise position on a volume of this subdirectory's file entry.

v1.2 For SOS 1.2, **parent_entry_length** =$27.

5.2.4 *File Entries*

Immediately following the pointers in any block of a directory file are a number of entries. The first entry in the key block of a directory file is a header; all other entries are file entries. Each entry has the length specified by that directory's **entry_length** field, and each file entry contains information that describes, and points to, a single subdirectory file or standard file.

An entry in a directory file may be active or inactive; that is, it may or may not describe a file currently in the directory. If it is inactive, the **storage_type** and **name_length** fields are zero.

The maximum number of entries, including the header, in a block of a directory is recorded in the **entries_per_block** field of that directory's header. The total number of active file entries, not including the header, is recorded in the **file_count** field of that directory's header.

Figure 5-5 describes the format of a file entry.

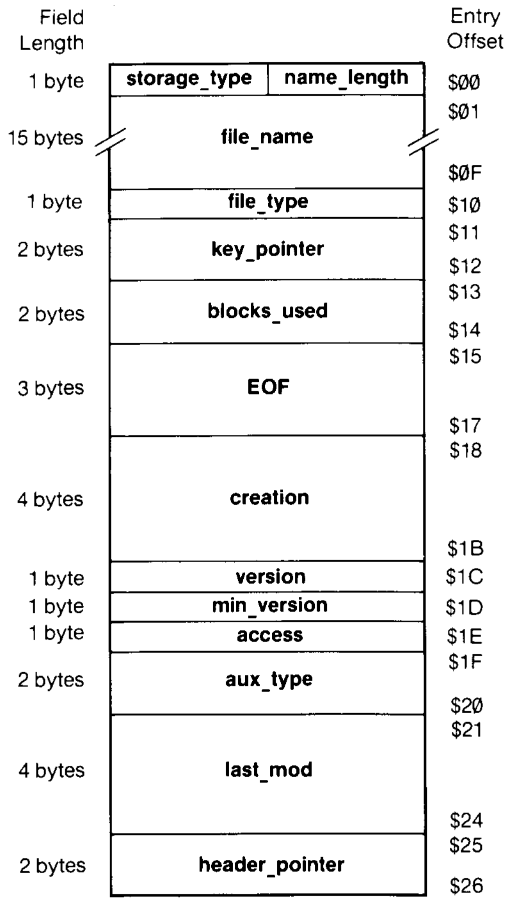


Figure 5-5. The File Entry

storage_type and **name_length** (1 byte):

Two four-bit fields are packed into this byte. The value in the high-order four bits (the **storage_type**) specifies the type of file this entry points to. The values $1, $2, $3, and $D denote seedling, sapling, tree, and subdirectory files, respectively. Seedling, sapling, and tree files, the three forms of a standard file, are described later in this chapter. The low-order four bits contain the length of the file's name (see the **file_name** field, below). If a file entry is inactive, the **storage_type** and **name_length** are zero. The **name_length** can be changed by a RENAME call.

file_name (15 bytes):

The first **name_length** bytes of this field contain the file's name. This name must conform to the file name syntax explained in Chapter 4. This field can be changed by the RENAME call.

file_type (1 byte):

This specifies the internal structure of the file. Section 5.4.2.3 contains a list of the currently defined values of this byte.

key_pointer (2 bytes):

This is the block address of the key block of the subdirectory or standard file described by this file entry.

blocks_used (2 bytes):

This is the total number of blocks actually used by the file. For a subdirectory file, this includes the blocks containing subdirectory information, but not the blocks in the files pointed to. For a standard file, this includes both informational blocks (index blocks) and data blocks. Refer to section 5.3 for more information on standard files.

EOF (3 bytes):

This is a three-byte integer, lowest bytes first, that represents the total number of bytes readable from the file. Note that in the case of sparse files, described later in the chapter, **EOF** may be greater than the number of bytes actually allocated on the disk.

creation (4 bytes):

This is the date and time at which the file pointed to by this entry was created. The format of these bytes is described in section 5.4.2.2.

version (1 byte):

This is the version number of SOS under which the file pointed to by this entry was created. This byte allows newer versions of SOS to determine the format of the file, and adjust their interpretation processes accordingly.

v1.2 For SOS 1.2, **version** = 0.

min_version (1 byte):

This is the minimum version number of SOS that can access the information in this file. This byte allows older versions of SOS to determine whether they can access newer files.

v1.2 For SOS 1.2, **min_version** = 0.

access (1 byte):

This field determines whether this file can be read, written, destroyed, and renamed. The format of this field is described in section 5.4.2.3. The value of this field can be changed by the SET__FILE__INFO call.

aux_type (2 bytes):

This is a general-purpose field in which an interpreter can store additional information about the internal format of a file. For example, BASIC uses this field to store the record length of its data files. This field can be changed by the SET__FILE__INFO call.

last_mod (4 bytes):

This is the date and time that the last CLOSE operation after a WRITE was performed on this file. The format of these bytes is described in section 5.4.2.2. This field can be changed by the SET__FILE__INFO call.

header_pointer (2 bytes):

This field is the block address of the key block of the directory that owns this file entry. This two byte pointer is stored low byte first, high byte second.

5.2.5 Field Formats in Detail

Several of the fields above occur in more than one kind of directory entry. Therefore, we have pulled them out for more detailed explanation here.

5.2.5.1 The **storage_type** Field

The **storage_type**, the high-order four bits of the first byte of an entry, defines the type of header (if the entry is a header) or the type of file described by the entry.

$0	indicates an inactive file entry
$1	indicates a seedling file entry (0 <= **EOF** <= 512 bytes)
$2	indicates a sapling file entry (512 < **EOF** <= 128K bytes)
$3	indicates a tree file entry (128K < **EOF** < 16M bytes)
$D	indicates a subdirectory file entry
$E	indicates a subdirectory header
$F	indicates a volume directory header

SOS automatically changes a seedling file to a sapling file and a sapling file to a tree file when the file's **EOF** grows into the range for a larger type. If a file's **EOF** shrinks into the range for a smaller type, SOS changes a tree file to a sapling file and a sapling file to a seedling file.

5.2.5.2 The **creation** and **last_mod** Fields

The date and time of the creation, and of the last modification, of each file and directory are stored as two four-byte values (see Figure 5-6):

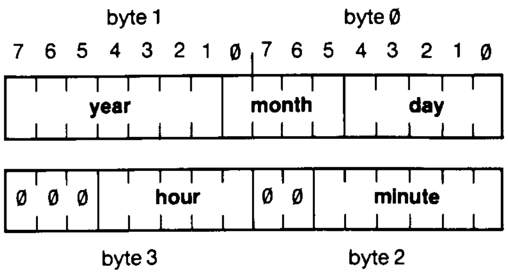


Figure 5-6. Date and Time Format

The values for the year, month, day, hour, and minute are stored as unsigned binary integers, and may be unpacked for analysis. Note that the SOS calls GET_TIME and SET_TIME represent dates and times differently.

5.2.5.3 *The **access** Attributes*

The **access** attribute field determines whether the file can be read from, written to, deleted, or renamed. It also tells whether a backup copy of the file has been made since the file's last modification (see Figure 5-7).

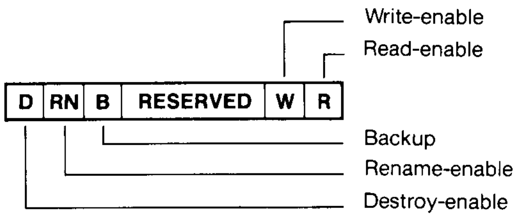


Figure 5-7. The **access** Attribute Field

A bit set to 1 indicates that the operation is enabled; a bit cleared to 0 indicates that the operation is disabled. The reserved bits are always 0.

SOS sets bit 5 (the *backup bit*) of the **access** field to 1 whenever the file is changed (that is, after a CREATE, RENAME, CLOSE after WRITE, or SET_FILE_INFO operation). This bit is cleared to 0 whenever the file is copied by Backup III. This lets Backup III selectively back up files that have been changed since the last backup was made.

Only SOS may change bits 2-4. Only SOS and Backup III may change bit 5.

5.2.5.4 *The **file_type** Field*

The **file_type** field within an entry identifies the type of file described by that entry. This field should be used by interpreters to guarantee file compatibility from one interpreter to the next. The values of this byte are defined below:

$00 = Typeless file (BASIC "unknown" file)
$01 = File containing all bad blocks on the volume
$02 = Pascal or assembly-language code file
$03 = Pascal text file
$04 = BASIC text file; Pascal ASCII file
$05 = Pascal data file
$06 = General binary file
$07 = Font file
$08 = Screen image file
$09 = Business BASIC program file
$0A = Business BASIC data file
$0B = Word Processor file
$0C = SOS system file (DRIVER, INTERP, KERNEL)
$0D,$0E = SOS reserved
$0F = Directory file (see **storage_type**)
$10-$BF = SOS reserved
$C0-$FF = ProDOS reserved

5.2.6 Reading a Directory File

Reading a directory file is straightforward, but your program must be written to allow for possible changes in the entry length and the number of entries per block: future versions of SOS may change these by adding more information at the end of an entry. Since these values are in the directory header, this flexibility is not difficult to achieve.

The first step in reading a directory file is to open an access path to the file, and obtain a **ref_num**. Using the **ref_num** to identify the file, read the first 512 bytes of the file into a buffer. The buffer contains two two-byte pointers, followed by the entries: the first entry is the directory header. Bytes $1F through $20 in the header (bytes $23 through $24 in the buffer) contain the values of **entry_length** and **entries_per_block**.

Once these values are known, an interpreter can read through the entries in the buffer, using a pointer to the beginning of the current entry and a counter indicating the number of entries examined in the current block. Any entry whose first byte is zero is ignored. When the counter equals **entries_per_block**, read the next 512 bytes of the file into the buffer. When a READ returns a **bytes_read** parameter of zero, you have processed the entire directory file.

5.3 Storage Formats of Standard Files

Each active entry in a directory file points (using its **key_pointer** field) to the key block of another directory file or to the key block of a standard file. An entry that points to a standard file contains information about the file: its name, its size, its type, and so on.

Depending on its size, a standard file can be stored in any of the three formats explained below: seedling, sapling, and tree. An interpreter can distinguish between these three (using the file entry's **storage_type** field), but it need not, for an interpreter reads every standard file in exactly the same way, as a numbered sequence of bytes. Only SOS needs to know how a file is stored. Nevertheless, we think it is useful for programmers to understand how SOS stores data on a volume.

The storage formats in this section apply to SOS 1.2. They may change in future versions of SOS.

5.3.1 Growing a Tree File

As a tree file grows, it goes through three storage formats, as explained in the following scenario. In the scenario, we start with an empty, formatted volume, create one file, then increase its size in stages.

v1.2

This scenario is based on the block-allocation scheme used by SOS 1.2 on a 280-block flexible disk, which contains four blocks of volume directory, and one block of volume bit map. This scheme is subject to change in future versions of SOS.

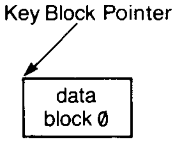
Larger capacity volumes might have more blocks in the volume bit map, but the process would be the same.

A formatted, but otherwise empty, 280-block SOS disk is used like this:

Blocks 0-1	:	Bootstrap Loader
Blocks 2-5	:	Volume Directory
Block 6	:	Volume Bit Map
Blocks 7-279	:	Unused

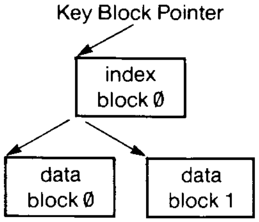
If you open a new standard file, one data block is immediately allocated to that file. An entry is placed in the volume directory, and it points to block 7, the new data block, as the key block for the file. The volume now looks like this:

	Blocks 0-1	:	Bootstrap Loader
	Blocks 2-5	:	Volume Directory
	Block 6	:	Volume Bit Map
→	Block 7	:	Data Block 0
	Blocks 8-279	:	Unused



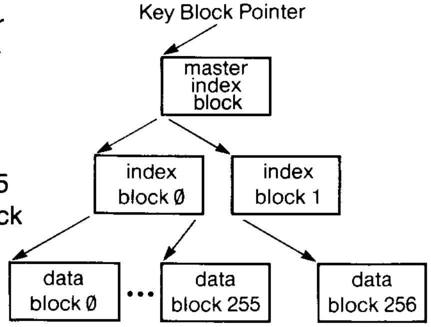
This is a *seedling file*: its key block contains up to 512 bytes of data. If you write more than 512 bytes of data to the file, the file grows into a *sapling file*. As soon as a second block of data becomes necessary, an *index block* is allocated, and it becomes the file's key block: this index block can point to up to 256 data blocks (two-byte pointers). A second data block (for the data that won't fit in the first data block) is also allocated. The volume now looks like this:

Blocks 0-1 : Bootstrap Loader
Blocks 2-5 : Volume Directory
Block 6 : Volume Bit Map
Block 7 : Data Block 0
→ Block 8 : Index Block 0
Block 9 : Data Block 1
Blocks 10-279 : Unused



This sapling file can hold up to 256 data blocks: 128K of data. If the file becomes any bigger than this, the file grows again, this time into a *tree file*. A *master index block* is allocated, and it becomes the file's key block: the master index block can point to up to 128 index blocks, and each of these can point to up to 256 data blocks. Index block 0 becomes the first *subindex block*, which is an index block pointed to by the master index block. In addition, a new subindex block is allocated, and a new data block to which it points. Here's a new picture of the volume:

Blocks 0-1 : Bootstrap Loader
Blocks 2-5 : Volume Directory
Block 6 : Volume Bit Map
Block 7 : Data Block 0
Block 8 : Index Block 0
Blocks 9-263 : Data Blocks 1-255
→ Block 264 : Master Index Block
Block 265 : Index Block 1
Block 266 : Data Block 256
Blocks 267-279 : Unused



As data are written to this file, additional data blocks and index blocks are allocated as needed, up to a maximum of 129 index blocks (one master index block and 128 subindex blocks), and 32,768 data blocks, for a maximum capacity of 16,777,215 bytes of data in a file. If you did the multiplication, you probably noticed that we lost a byte somewhere. The last byte of the last block of the largest possible file cannot be used because **EOF** cannot exceed 16,777,215. If you are wondering how such a large file might fit on a small volume such as a floppy disk, refer to the section on sparse files, later in this chapter.

This scenario shows the growth of a single file on an otherwise empty volume. The process is a bit more confusing when several files are growing (or being deleted) simultaneously. However, the block allocation scheme is always the same: when a new block is needed, SOS always allocates the first unused block in the volume bit map.

5.3.2 *Seedling Files*

A *seedling file* is a standard file that contains no more than 512 data bytes ($0 <= **EOF** <= $200). This file is stored as one block on the volume, and this data block is the file's key block.

v1.2 One block is always allocated for a seedling file, even if no data have been written to the file.

The structure of such a seedling file looks like this (Figure 5-8):

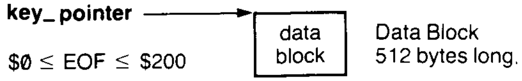


Figure 5-8. Structure of a Seedling File

The file is called a seedling file because, if more than 512 data bytes are written to it, it grows into a sapling file, and thence into a tree file.

The **storage_type** field of an entry that points to a seedling file has the value $1.

5.3.3 *Sapling Files*

A *sapling file* (see Figure 5-9) is a standard file that contains more than 512 and no more than 128K bytes ($200 < **EOF** <= $20000). A sapling file comprises an index block and 1 to 256 data blocks. The index block contains the block addresses of the data blocks.

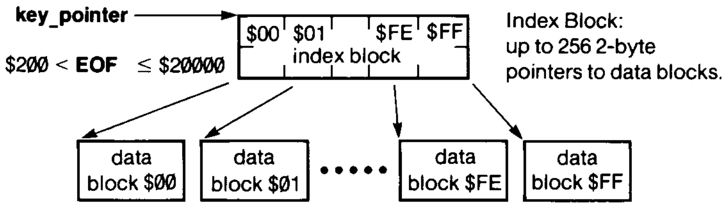


Figure 5-9. Structure of a Sapling File

The key block of a sapling file is its index block. SOS retrieves data blocks in the file by first retrieving their addresses in the index block.

The **storage_type** field of an entry that points to a sapling file has the value $2.

5.3.4 *Tree Files*

A *tree file* (see Figure 5-10) contains more than 128K bytes, and less than 16M bytes ($20000 < **EOF** < $1000000). A tree file consists of a master index block, 1 to 128 subindex blocks, and 1 to 32,768 data blocks. The master index block contains the addresses of the subindex blocks, and each subindex block contains the addresses of up to 256 data blocks.

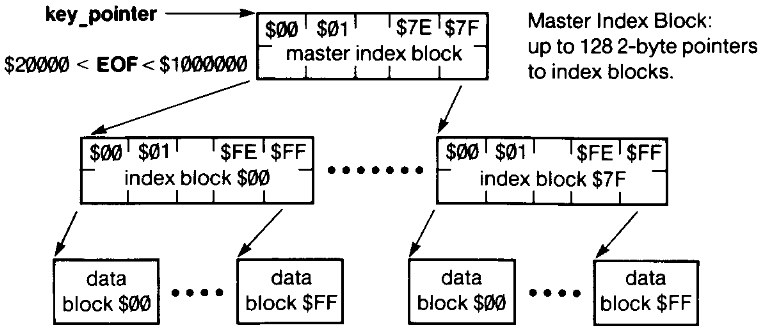


Figure 5-10. The Structure of a Tree File

The key block of a tree file is the master index block. By looking at the master index block, SOS can find the addresses of all the subindex blocks; by looking at those blocks, it can find the addresses of all the data blocks.

The **storage_type** field of an entry that points to a tree file has the value $3.

5.3.5 *Sparse Files*

A *sparse file* is a sapling or tree file in which the number of data bytes that can be read from the file exceeds the number of bytes physically stored in the data blocks allocated to the file. SOS implements sparse files by allocating only those data blocks that have had data written to them, as well as the index blocks needed to point to them.

For example, we can define a file whose **EOF** is 16K, that uses only three blocks on the volume, and that has only four bytes of data written to it. Create a file with an **EOF** of $0. SOS allocates only the key block (a data block) for a seedling file, and fills it with null characters (ASCII $00).

Set the **EOF** and **mark** to position $0565, and write four bytes. SOS calculates that position $0565 is byte $0165 ($0564 - $0200 * 2) of the third block (block $2) of the file. It then allocates an index block, stores the address of the current data block in position 0 of the index block, allocates another data block, stores the address of that data block in position 2 of the index block, and stores the data in bytes $0165 through $0168 of that data block. The **EOF** is $0569.

Set the **EOF** to $4000 and close the file. You have a 16K file that takes up three blocks of space on the volume: two data blocks and an index block. You can read 16384 bytes of data from the file, but all the bytes before $0565 and after $0568 are nulls. Figure 5-11 shows how the file is organized:

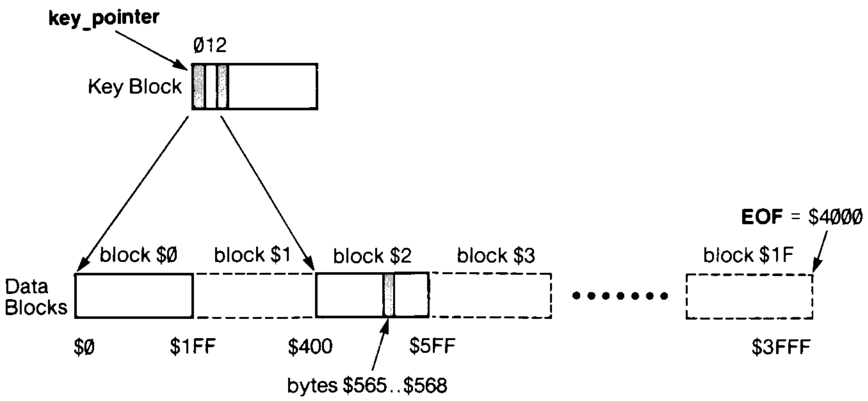


Figure 5-11. A Sparse File

Thus SOS allocates volume space only for those blocks in a file that actually contain data. For tree files, the situation is similar: if none of the 256 data blocks assigned to an index block in a tree file have been allocated, the index block itself is not allocated.

On the other hand, if you CREATE a file with an **EOF** of $4000 (making it 16K bytes, or 32 blocks, long), SOS allocates an index block and 32 data blocks for a sapling file, and fills the data blocks with nulls.

The first data block of a standard file, be it a seedling, sapling, or tree file, is always allocated.

If you read a sparse file, then write it, the copy will not be sparse: all the phantom blocks will be written out as blocks full of nulls. The Apple III System Utilities program, on the other hand, can distinguish between sparse files and non-sparse files and make a sparse copy of a sparse file. Backup III also handles sparse files correctly, but it should not be used to make copies, because when it backs up a file, it clears the file's backup bit, so that a backup of all modified files will overlook the sparse file.

5.3.6 *Locating a Byte in a Standard File*

The **mark** is a three-byte pointer that is normally used to specify a logical byte position within a standard file, using the standard model of a block file. It can also be used to pinpoint the block number and byte number within that block where that byte can be found on a volume. To do so, the **mark** is divided into three fields, shown in Figure 5-12:

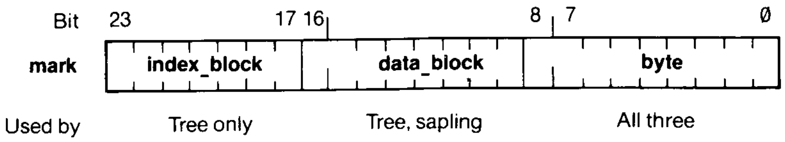


Figure 5-12. Format of **mark**

index_block (7 bits):

If the file is a tree file, this field tells which subindex block points to the data block. If *i* = **index_block**, the low byte of the subindex block address is at byte *i* of the master index block; the high byte is at byte (*i*+$100).

data_block (8 bits):

If the file is a tree file or a sapling file, this field tells which data block is pointed to by the selected index block. If *j* = **data_block**, the low byte of the data block address is at byte *j* of the index block; the high byte is at byte (*j* + $100).

byte (9 bits):

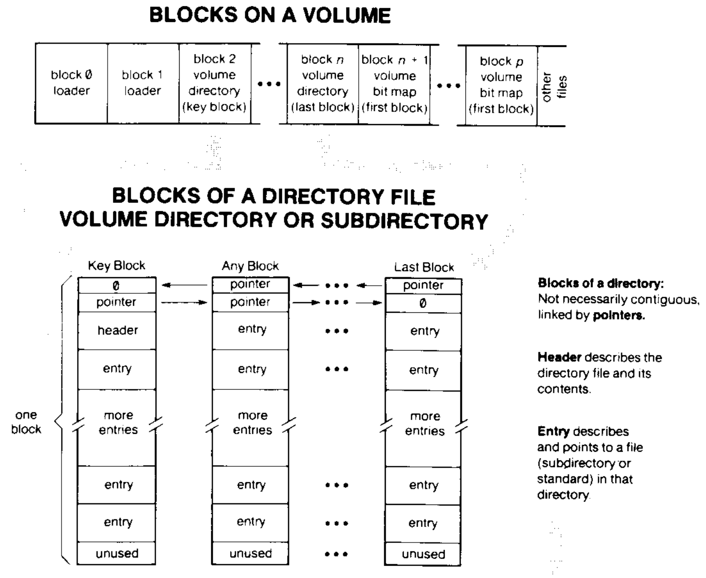
For tree, sapling, and seedling files, this field tells the absolute position of the byte within the selected data block.

This format for **mark** applies to SOS 1.2. Future versions of SOS may use indexing schemes that divide the 24 bits differently. If an interpreter uses **mark** as a three-byte pointer to a logical byte position in a file, it will be unaffected by such changes; if it meddles with index blocks, it may fail catastrophically, trashing your disk in the process, under some future version of SOS.

5.4 *Chapter Overview*

The following figures summarize the information in this chapter.

- Figure 5-13, Disk Organization, shows disk layout and directory structure.
- Figure 5-14, Header and Entry Fields, explains the individual fields in the preceding figure.



Blocks of a directory: Not necessarily contiguous, linked by **pointers.**

Header describes the directory file and its contents.

Entry describes and points to a file (subdirectory or standard) in that directory.

HEADER VOLUME DIRECTORY Found in key block of volume directory.

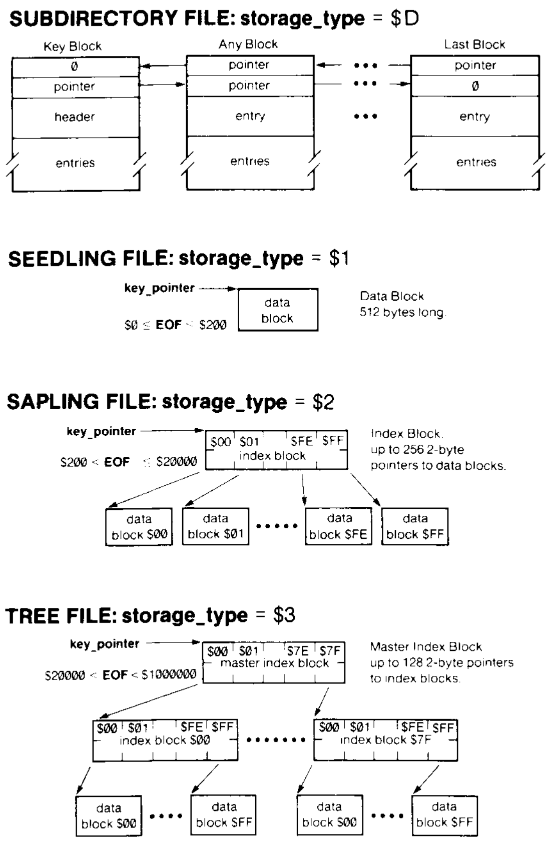
Field length	Field	Byte of Block
1 byte	storage_type | name_length	$04
15 bytes	file_name	$05 $13
8 bytes	reserved	$14 $1B
4 bytes	creation	$1C $1D $1E $1F
1 byte	version	$20
1 byte	min_version	$21
1 byte	access	$22
1 byte	entry_length	$23
1 byte	entries_per_block	$24
2 bytes	file_count	$25 $26
2 bytes	bit_map_pointer	$27 $28
2 bytes	total_blocks	$29 $2A

HEADER SUBDIRECTORY Found in key block of subdirectory.

Field Length	Field	Byte of Block
1 byte	storage_type | name_length	$04
15 bytes	file_name	$05 $13
8 bytes	reserved	$14 $1B
4 bytes	creation	$1C $1D $1E $1F
1 byte	version	$20
1 byte	min_version	$21
1 byte	access	$22
1 byte	entry_length	$23
1 byte	entries_per_block	$24
2 bytes	file_count	$25 $26
2 bytes	parent_pointer	$27 $28
1 byte	parent_entry_number	$29
1 byte	parent_entry_length	$2A

FILE ENTRY SUBDIRECTORY OR STANDARD FILE Found in any directory file block.

Field Length	Field	Entry Offset
1 byte	storage_type | name_length	$00
15 bytes	file_name	$01 $0F
1 byte	file_type	$10
2 bytes	key_pointer	$11 $12
2 bytes	blocks_used	$13 $14
3 bytes	EOF	$15 $17
4 bytes	creation	$18 $1B
1 byte	version	$1C
1 byte	min_version	$1D
1 byte	access	$1E
2 bytes	aux_type	$1F $20
4 bytes	last_mod	$21 $24
2 bytes	header_pointer	$25 $26

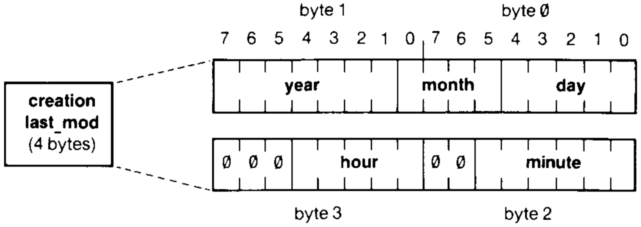


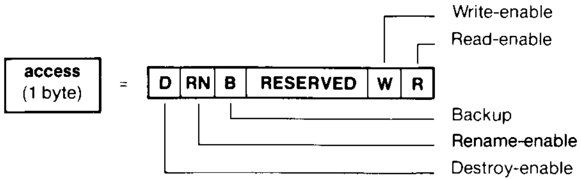
Data Block
512 bytes long.

Index Block.
up to 256 2-byte
pointers to data blocks.

Master Index Block
up to 128 2-byte pointers
to index blocks.

Figure 5-13. Disk Organization





storage_type
(4 bits)

$0 = inactive file entry
$1 = seedling file entry
$2 = sapling file entry
$3 = tree file entry
$D = subdirectory file entry
$E = subdirectory header
$F = volume directory header

file_type
(1 byte)

$00 = typeless file
$01 = bad block file
$02 = Pascal or assembly code file
$03 = Pascal text file
$04 = Basic text; Pascal ASCII file
$05 = Pascal data file
$06 = General binary file
$07 = Font file
$08 = Screen image file
$09 = BASIC program file
$0A = BASIC data file
$0B = Word Processor file
$0C = SOS system file (DRIVER, INTERP, KERNEL)
$0D = Reserved
$0E = Reserved
$0F = Directory file
$10-$BF = SOS reserved
$C0-$FF = ProDOS reserved

name_length = length of file_name ($1-$F)
file_name = $1-$F ASCII characters: first = letters rest are letters, digits, periods.
key_pointer = Block address of file's key block
blocks_used = total blocks for file
EOF = byte number for end of file ($0-$FFFFFF)
version, min_version = 0 for SOS 1.2
entry_length = $27 for SOS 1.2
entries_per_block = $0D for SOS 1.2
aux_type = defined by interpreter
file_count = total files in directory
bit_map_pointer = block address of bit map
total_blocks = total blocks on volume
parent_pointer = block address containing entry
parent_entry_number = number in that block
parent_entry_length = $27 for SOS 1.2
header_pointer = block address of key block of entry's directory.

Figure 5-14. Header and Entry Fields

6

Events and Resources

6.1 Interrupts and Events

An *interrupt* is a signal from a peripheral device to the CPU. When the CPU receives an interrupt, it transfers control to SOS, which saves the current state of the executing program and calls an interrupt handler, located in the driver of the interrupting device. After the interrupt is handled, control is returned to the program that was interrupted.

Interrupts allow device drivers to operate their devices asynchronously. By using interrupts, a device can operate more efficiently and allow the interpreter to continue running while a long I/O operation is in progress. For example, when you send a long buffer of text to the .PRINTER driver, the driver does not process the text all at once; instead, it immediately returns control to the interpreter, and the interpreter can do something else while the interrupt-driven .PRINTER driver processes the buffer for output.

The Apple III/SOS system fully supports interrupts from any internal or external peripheral device capable of generating them. To use the system efficiently, an interpreter must be designed to work properly even if interrupted. Thus, the interpreter cannot contain any time-dependent code (such as timing loops), except to provide a guaranteed minimum time.

Interrupts are discussed in detail in the *Apple III SOS Device Driver Writer's Guide.*

Interrupts are ranked in priority by the priorities of the devices on which they occur. Each device has a unique priority, assigned at system configuration time. In addition, when an interrupt occurs on a device, all further interrupts from that device are locked out until that interrupt has been fully processed. For these reasons, SOS never has to deal simultaneously with two interrupts of equal priority. Conflicts between interrupts of different priorities are resolved in favor of the higher priority: a higher-priority interrupt can suspend processing of a lower-priority interrupt, but not vice versa.

SOS also supports the detection and handling of *events*. An event is a signal from a device driver to an interpreter that something of interest to the interpreter has happened. When an event of sufficient priority occurs, SOS suspends the interpreter and saves its state, then calls an event handler to process the event, then returns control to the portion of the interpreter that was suspended. By using events, an interpreter can respond to outside occurrences without spending all its time watching out for them.

The most common kind of event is triggered by a software response to a hardware interrupt: a device driver (such as the .CONSOLE driver) defines a certain occurrence (such as a press of the space bar) as an event, and allows interpreters or assembly-language modules to respond to that event. In principle, however, events need not be triggered by interrupts: an event can signal, for example, an overflow on a communication card, a "message received" condition on a network interface, or a "new volume mounted" condition on a mass-storage device. Any occurrence or condition a driver can detect can be signaled as an event.

v1.2

SOS currently supports two events, both detected by the .CONSOLE driver: the Any-Key Event and the Attention Event. Both of these are produced by interrupts from the keyboard. These events are described in the *Apple III Standard Device Drivers Manual.* Additional events may be defined by a device driver: for details, see the *Apple III SOS Device Driver Writer's Guide.*

The most common event sequence is illustrated below. An event is *armed* when the interpreter prepares a device driver to signal a certain occurrence (in this case, a keypress) as an event. The interpreter supplies the address of a subroutine to be called when the expected event occurs.

When the device driver detects the event (in this case, by means of an interrupt), the driver places the event into a queue and returns to the interrupted process, whether interpreter or SOS. This is illustrated by Figure 6-1.

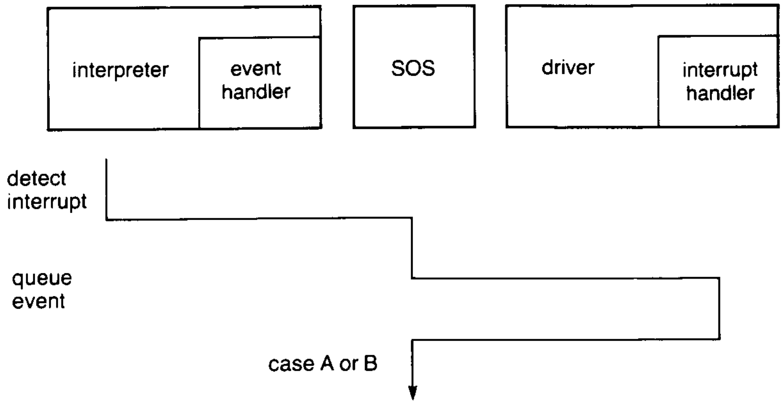


Figure 6-1. Queuing An Event

Any time SOS is ready to return control to the interpreter, such as after executing a call or processing an interrupt, it checks the event queue. If it finds an event of a priority above the preset *event fence* (see Figure 6-2), SOS calls an event-handler subroutine within the interpreter. When the event has been processed, SOS returns control to the main body of the interpreter.

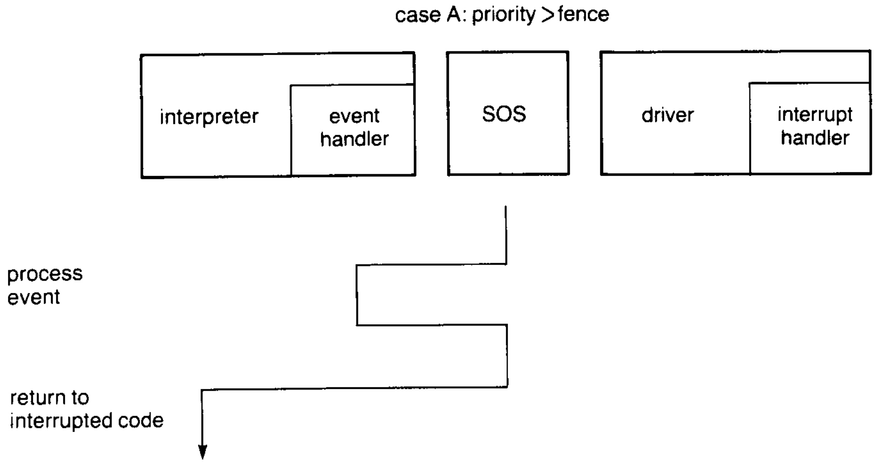


Figure 6-2. Handling An Event: Case A

If SOS finds no event above the fence (see Figure 6-3), the event remains queued until the fence is set (by a SET__FENCE call) below the event's priority. Then, the event will be processed as soon as the call is completed.

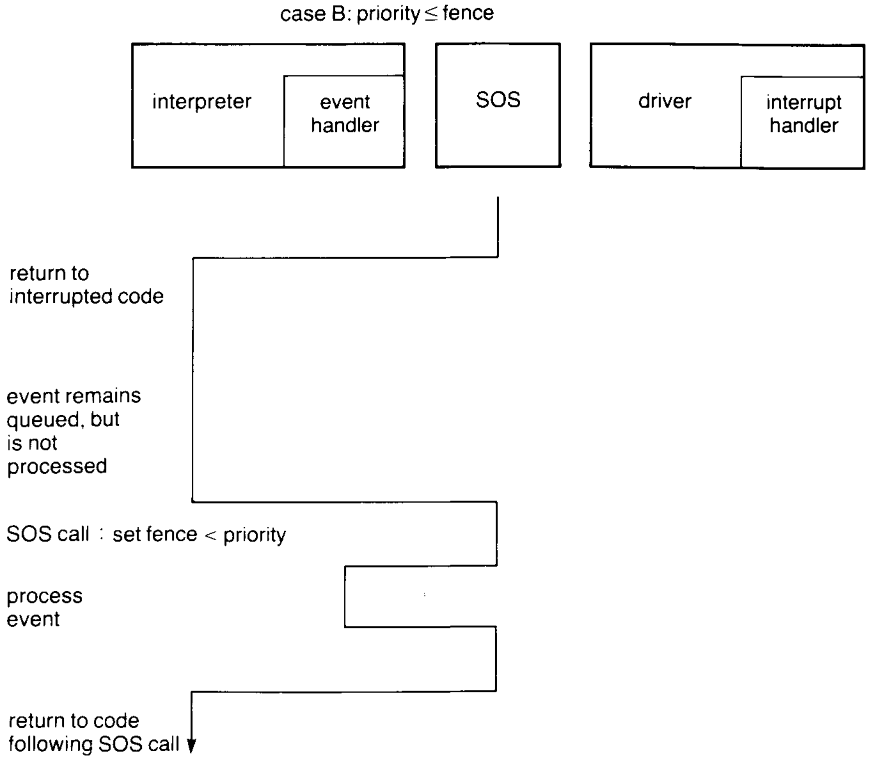


Figure 6-3. Handling An Event: Case B

An event need not be triggered by an interrupt: it can occur as a result of any operation within a device driver. But events are detected only by device drivers, and are handled only by an event-handler subroutine within an interpreter. An event handler will be called only after a SOS call or an interrupt is processed.

6.1.1 *Arming and Disarming Events*

SOS has not defined a uniform mechanism for arming and disarming events: this is left up to the device driver that supports the event. The two existing events are armed and disarmed by D_CONTROL calls to the .CONSOLE driver.

An interpreter arms an event by passing three items to the device driver: the address of the event handler, a one-byte event identifier (ID), and a one-byte event priority. The event ID indicates the nature of the event, and allows the event handler to distinguish different events. For example, the event ID for the Any-Key Event is 1; the event ID for the Attention Event is 2. The event priority indicates the importance of the event, and determines when, or whether, the event will be processed.

An interpreter disarms an event by arming it with a priority of zero: this ensures that it will be ignored.

6.1.2 *The Event Queue*

More than one event can be armed at once, and more than one event can occur during a driver's operation. SOS has a priority-queue scheme for keeping simultaneous events in order.

When a driver detects an event, it assigns an ID, a priority, and an event-handler address to the event. (These are the values the interpreter passed to the driver when the event was armed.) The ID, priority, and address are placed in an *event queue* (see Figure 6-4) maintained by SOS.

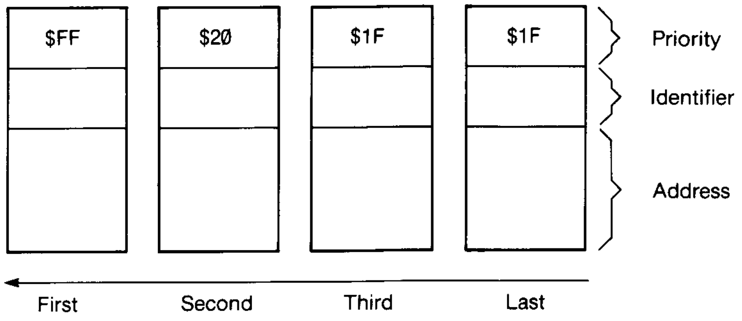


Figure 6-4. The Event Queue

The queue is arranged in order by priority: an event of higher priority will be handled first. The highest priority is $FF: this priority guarantees that an event will be handled before any other event. Events of equal priority are queued first-in, first-out (FIFO): an event with the same priority as another event already in the queue is placed after the other event. Events of priority $00 can never be handled, so they are not queued.

6.1.3 *The Event Fence*

The priority ordering of the event queue determines not only when an event will be handled, but also whether it will be handled at all. SOS maintains an event fence (see Figure 6-5) that determines which events will be processed and which will not.

The fence is a value from $00 to $FF that is compared to the priority value of each event in the queue. Only those events whose priority is greater than the fence will be handled: setting the fence to $FF ensures that no events will be handled.

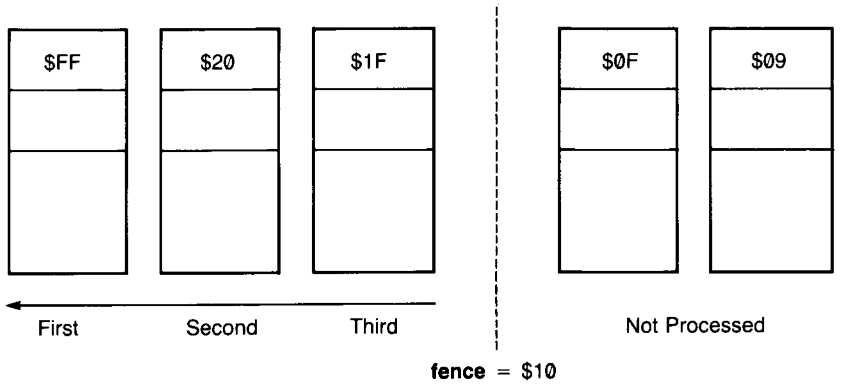


Figure 6-5. The Event Fence

All events above the fence are handled, in order, and removed from the queue before SOS returns control to the suspended portion of the interpreter. Events below the fence remain in the queue, and may be handled when the fence is lowered.

Two SOS calls, SET_FENCE and GET_FENCE, allow an interpreter to set and read the value of the fence. If the interpreter lowers the fence while events are in the queue, previously queued events whose priority values are greater than or equal to the new value of the fence will be handled immediately after the call is completed.

6.1.4 Event Handlers

An event handler is a subroutine in the interpreter that is called by SOS in response to an event, under certain conditions. An event can only be processed when the interpreter is executing. If a SOS call is being executed when an event occurs, the event is queued; after the call is executed, SOS will call the interpreter's event handler if the event's priority is higher than the event fence. When the event handler is called, the previous state of the machine is stored on the interpreter's stack, and the event ID byte is stored in the accumulator; then the event is deleted from the queue.

Among the items saved on the stack is the current value of the event fence. The fence is then raised to the level of the current event until the event has been processed: this ensures that no event of lower priority will preempt the current event, now that the current event is no longer in the queue. Figure 6-6 illustrates the system status during event handling.

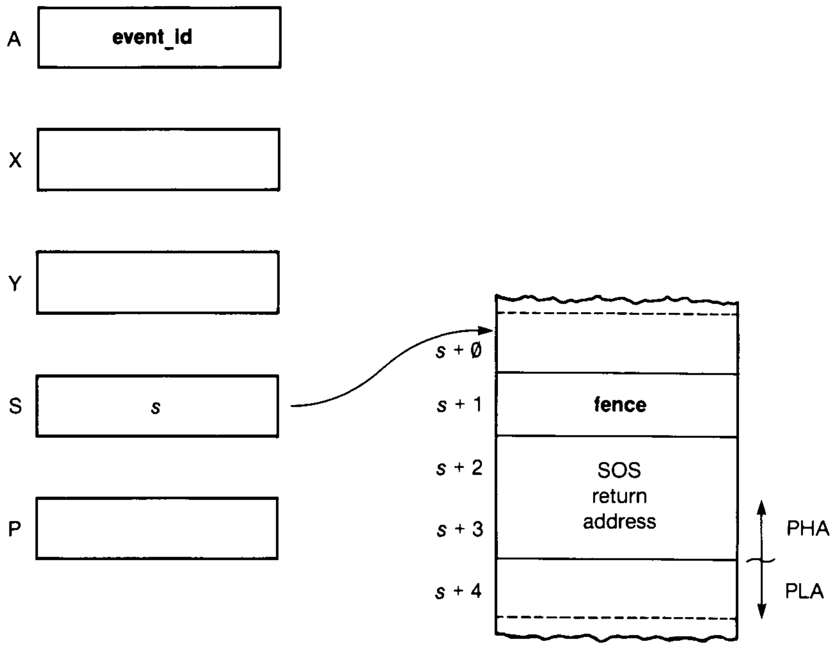


Figure 6-6. System Status during Event Handling

The event handler uses the event ID to determine the reason it was called and to take appropriate action.

When the event handler is finished, it returns control to SOS via an RTS; SOS then restores the system to its previous state, and returns control to the suspended portion of the interpreter. Since the previous state included the event fence, any fence set by the event handler will be lost, unless that fence value is passed to the body of the interpreter and reestablished by it.

6.1.5 *Summary of Interrupts and Events*

- Interrupts are generated by hardware; events are generated by software.
- Interrupts are ranked by the priorities assigned to the devices they occur on; events are ranked by the priorities assigned to them by the drivers that detect them.
- Interrupts are stacked; events are queued.
- Interrupts are handled by an interrupt handler in a device driver; events are detected and queued by a device driver, and processed by an event handler in the interpreter.
- Interrupts can preempt the interpreter or SOS; events can only preempt the interpreter.
- Interrupts cannot be disabled by the interpreter; events can be disabled by setting the event fence to $FF.

6.2 Resources

The Apple III has two resources accessible by special SOS calls: the system clock and the analog ports.

6.2.1 *The Clock*

The Apple III system clock runs continuously: when the computer is turned off, the clock runs on batteries. It keeps time down to the millisecond, and can be read and set by SOS.

The clock is set and read by two calls: SET_TIME and GET_TIME. To set the time, the calling program writes it as an ASCII string into an 18-byte buffer in memory, then passes SOS the address of the buffer: SOS then sets the clock to the specified time. To read the time, the calling program passes SOS the address of an 18-byte buffer: SOS then writes the current time into this buffer.

If the computer has no functioning clock, SOS responds to a SET_TIME call by saving the time it receives. SOS returns this time unchanged upon a subsequent GET_TIME call.

Both calls express the time as an 18-byte ASCII string of the following format:

Y Y Y Y M M D D W H H N N S S U U U

The meaning of each field is as below:

Field	**Meaning**	**Minimum**	**Maximum**
YYYY:	Year	1900	1999
MM:	Month	00	12 December
DD:	Date	00 or 01	28, 30, or 31
W:	Day	01 Sunday	07 Saturday
HH:	Hour	00 Midnight	23 11:00 p.m.
NN:	Minute	00	59
SS:	Second	00	59
UUU:	Millisecond	000	999

For example, Monday, December 29, 1980, at 9:30 a.m. would be specified by the string "198012290093000000".

On input, SOS replaces the first two digits of the year with "19" and ignores the day of the week and the millisecond. SOS calculates the day from the year, month, and date.

SOS does not check the validity of the input data. The clock rejects any invalid combination of month and date. February 29 is always rejected.

The clock does not roll over the year.

6.2.2 *The Analog Inputs*

The GET_ANALOG call reads the analog and digital inputs from an Apple III Joystick connected to port A or B on the back of the Apple III. It can also read compatible signals from other devices.

6.2.3 TERMINATE

The TERMINATE call provides a clean exit from an interpreter.
It clears memory, clears the screen, and displays the message INSERT SYSTEM DISKETTE AND REBOOT on the screen. The TERMINATE call is useful as part of a protection scheme that locks out the NMI. Such a scheme allows only one way of leaving the program, and erases it completely afterward.

Before using this call, an interpreter must close all open files. This will ensure that no half-written buffers are left in limbo.

6.3 Utility Calls

These calls deal with the system clock/calendar, the event fence, the analog input ports, and other general system resources. The name of each call below is followed by its parameters (in boldface). The input parameters are directly-passed values and pointers to tables. The output parameters are all directly-passed results. The SOS call mechanism is explained in Chapter 8; the individual calls are described fully in Chapters 9 through 12 of Volume 2.

SET_FENCE

fence: value

This call changes the current value of the user event fence to the value specified in the **fence** parameter. Events with priority less than or equal to the fence will not be serviced until the fence is lowered.

GET_FENCE

fence: result

This call returns the current value of the user event fence.

SET_TIME

time: pointer

This call sets the current date and time. SET_TIME attempts to set the hardware clock whether it is operational or not. It also stores the new time in system RAM as the last known valid time: this time will be returned by all subsequent GET_TIME calls if the hardware clock is absent or malfunctioning.

GET_TIME

time: pointer

This call returns the current date and time from the system clock. If the clock is not operating, it returns the last known valid date and time from system RAM. If the system knows no last valid time, GET_TIME returns a string of 18 ASCII zeros.

GET_ANALOG

joy_mode: value; **joy_status**: result

This call reads the analog and digital inputs from an Apple III Joystick connected to port A or B on the back of the Apple III.

TERMINATE

This call zeros out memory, clears the screen, displays INSERT SYSTEM DISKETTE & REBOOT in 40-column black-and-white text mode on the screen, and hangs, until the user presses CONTROL-RESET to reboot the system. This call uses no parameters.

Interpreters and Modules

This chapter describes the two kinds of assembly-language programs that you can use: interpreters and modules. It discusses their structures, operating environments, and special characteristics; it explains how to create them and how to get them successfully loaded into the system.

7.1 Interpreters

The *interpreter* is the assembly-language program that SOS loads into memory from the file SOS.INTERP and executes at boot time. The interpreter can be a *stand-alone interpreter*, like Apple Writer III, or it can be a *language interpreter*, like the BASIC and Pascal interpreters. A stand-alone interpreter, normally an application program, provides the interface between you and SOS. A language interpreter can either provide this interface directly, as does BASIC, or support a program that does, as does Pascal, or both. A language interpreter can load and run your program in response to your command, or it can load and run a greeting program at boot time.

The interpreter is stored in its entirety in the file SOS.INTERP in the volume directory of the boot diskette. Additional functions can be added to the interpreter by use of assembly-language modules (see section 7.4).

An interpreter can

- Make SOS calls;
- Store and retrieve information in memory; and
- Handle events.

The SOS calls made by an interpreter can interact with you through devices, store or retrieve data, or request memory segments in which to store data. The memory accesses made by an interpreter can manipulate any information in the memory segments owned by the interpreter. The events handled by the interpreter can let it respond to special circumstances detected by device drivers.

7.1.1 Structure of an Interpreter

An interpreter is stored in a file named SOS.INTERP in the volume directory of a boot diskette. The data in this file consists of two parts: a header and a part containing code—as shown in Figure 7-1.

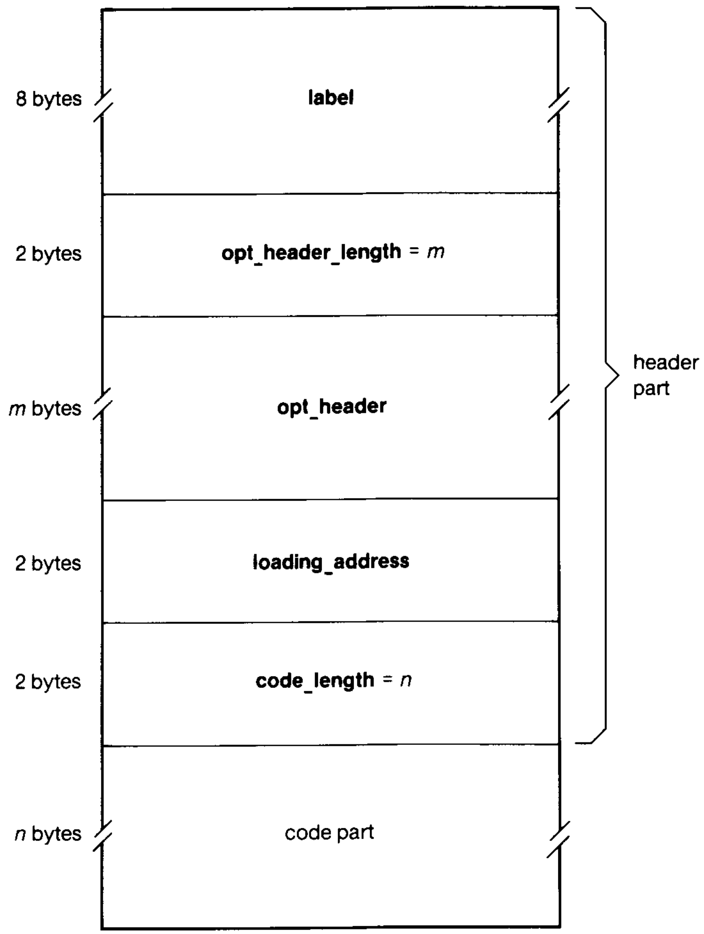


Figure 7-1. Structure of an Interpreter

The header consists of five fields, described below:

label (8 bytes):

This field contains eight characters

SOS NTRP

including the space. This is a label that identifies this file as an interpreter. The letters are all uppercase ASCII with their high bits cleared.

opt_header_length (2 bytes):

The next field contains the length of an optional header information block: if no optional header block is supplied, these bytes should be set to $0000. The length does not include the two bytes of the **opt_header_length** field itself.

opt_header (**opt_header_length** bytes):

If the previous field is nonzero, the optional header block comes here.

loading_address (2 bytes):

This field is the loading address (in current-bank notation) of the code part that must go into the highest bank of the system.

code_length (2 bytes):

This field is the length in bytes of the code part, excluding the header.

For example, an interpreter that begins at location $9250 in the highest bank of the system, is $25AF bytes long, and has no optional header would have a header part like this:

```
.ASCII   "SOS NTRP"         ; label for SOS.INTERP
.WORD    0000               ; opt_header_length = 0
.WORD    9250               ; loading_address
.WORD    25AF               ; code_length
```

Interpreters are always absolute code, and must start at a fixed location. A program in relocatable format cannot be used as an interpreter.

The header is immediately followed by the code part of the interpreter. During a system bootstrap operation, the code part is placed at the address given in the header, so that the first byte of code resides in the location specified by **loading_address** (location $2:9250 for the above example, in a 128K system). When loading is completed, execution of the interpreter begins at this location: the header part is discarded.

SOS requires only that the first byte of the code part be executable interpreter code; the rest of the code part of the interpreter may be in any format.

7.1.2 Obtaining Free Memory

An interpreter can use any and all memory that is not already allocated to SOS or device drivers, but first it must request this memory from SOS. The REQUEST_SEG and FIND_SEG calls to SOS can be used by an interpreter to request an area of memory in which to store data.

By allocating a segment of memory for its exclusive use, the interpreter ensures that no other code—the SOS file system, a device driver, an invocable module—will use that segment for another purpose. SOS allocates by an honor system: it protects allocated memory from conflict, but cannot prevent the use of unallocated memory. You can avoid memory conflict entirely by always allocating memory before use and deallocating it after use.

Using unallocated memory can have dramatic results. When an interpreter overwrites a file's I/O buffer, the system crashes. It does so to avoid trashing a disk: since the buffer contains block-allocation information as well as the interpreter's data, SOS would compromise the entire disk if it wrote out a buffer altered by the interpreter. To avoid this, SOS comes down with a SYSTEM FAILURE 16 message. When this happens, the data in the I/O buffer, as well as the data in memory, are lost.

The piece of interpreter code given below uses the FIND_SEG call (described in Chapter 12 of Volume 2) and the segment-to-extended address conversion described in section 2.2.3.1. It requests a 1K segment of memory (consisting of four adjacent memory pages) and fills that segment with zeros.

The first part of this procedure is the call to SOS to find a segment of the appropriate size. This is done with a FIND_SEG call.

```
FINDSEG    .EQU   041

FINDIT     BRK                  ;  Perform the SOS call
           .BYTE  FINDSEG       ;    FIND_SEG
           .WORD  FSPARAMS      ;    with the required parameters here.
           BEQ    CONVERT       ;  IF successful, THEN process addresses.
           LDA    PAGES         ;    ELSE see how big it can be.
           BNE    FINDIT        ;  IF any free memory exists, THEN ask again.
           JMP    ERRORHALT     ;    ELSE stop execution.

FSPARAMS
           .BYTE  06            ; Six parameters for FIND_SEG:
SRCHMOD    .BYTE  00            ; Seg must be in one bank
SEGID      .BYTE  11            ; I'll call it seg. 11.
PAGES      .WORD  04            ; Ask for 1K of memory
BASE       .WORD  0000          ; "base" result parameter
LIMIT      .WORD  0000          ; "limit" result parameter
SEGNUM     .BYTE  00            ; "seg_num" result parameter
EXTLIMIT                        ; Place to store (extended form of)
           .WORD  00            ; limit bank and page.
```

Once the FIND_SEG call succeeds, the values at BASE and LIMIT contain addresses in segment-address form of the first and last pages in the segment. Now the base and limit addresses must be converted into extended form to be used in clearing the memory in that segment. The first part of this process is determining where the segment is located: in the S-bank, in bank 0, or in another bank in bank-switched memory.

```
CONVERT    LDA    BASE          ;  Get bank number of segment
           BEQ    SZBANK        ;   Is it in bank 0?
           CMP    #0F           ;   Is it in low S-bank?
           BEQ    SZBANK
           CMP    #10           ;   Is it in high S-bank?
           BEQ    SZBANK
```

For the general case (any bank but S or 0), the conversion involves calculating the proper X-byte and creating the two-byte address for the pointer.

```
ANYBANK    CLC                         ; Turn bank number into X-byte
           ADC    #7F                  ; XX = $80 + bb - 1
           STA    1651                 ; Store it in X-page for pointer.
           LDA    BASE + 1             ; Get page number in bank
           CLC                         ; Turn into high part of address
           ADC    #60                  ; NNNN := pp00 + $6000
           STA    51                   ; Store into zero-page pointer
           LDA    #00                  ; Create low part of $00
           STA    50                   ; Store into zero-page pointer
           LDA    LIMIT                ; Get bank number of segment.
           CLC                         ; Turn into X-byte.
           ADC    #7F                  ; XX = $80 + bb - 1
           STA    EXTLIMIT             ; Store it in X-page for pointer.
           LDA    LIMIT + 1            ; Get page number of limit.
           CLC                         ; Turn into extended form for
           ADC    #60                  ;   later comparison with page
           STA    EXTLIMIT + 1         ;   being zeroed,
           JMP    CLEARIT              ; and proceed to clear the segment.
```

For the case where the segment resides in bank 0 or the S-bank, the conversion is much easier: just use an X-byte of $8F and create the proper two-byte address.

```
SZBANK     LDA    #8F                  ; Use an X-byte of $8F
           STA    1651
           LDA    BASE + 1             ; Get page number in bank
           STA    51
           LDA    #00                  ; Create low part of $00
           STA    50
           LDA    #8F                  ; Use limit X-byte of $8F
           STA    EXTLIMIT
           LDA    LIMIT + 1            ; Convert page number of limit
           STA    EXTLIMIT + 1         ;   to extended form.
```

Now an extended pointer has been created and is stored in locations $0050, $0051, and $1651. This pointer indicates the beginning of the memory range allocated by SOS in the FIND_SEG call.

A process similar to the above can be used to convert the limit segment address into another extended pointer to define the end of the segment.

Remember that the limit address specifies the last page in the segment. Converting the limit address into a pointer using the method shown above will give you a pointer to the beginning of this page, not the end. Keep this in mind when comparing two pointers derived from base and limit segment addresses.

Once the pointers are set up, a simpler form of the increment loop described in section 2.4.2.1 can be used to scan through every location in the segment and, in this example, set each byte to $00. Because the FIND_SEG call requested that the entire segment reside in one bank, the increment loop does not need to increment the X-byte of the pointer, or compare the base X-byte to the limit X-byte.

```
        LDY    #00             ; Use Y as an index in each page.
STORE   LDA    #00             ; Value to put in each location.
        STA    (50),Y          ; Extended-address operation.
        INY                    ; Do next byte in page.
        BNE    STORE
        INC    51              ; Move to next page.
        LDA    51              ; Get high part of address.
        CMP    EXTLIMIT + 1    ; Compare with high part of limit.
        BCC    STORE           ; If pointer.high < = limit.high,
        BEQ    STORE           ;   clear another page.
```

A program that wishes to use more than 32K bytes of memory must handle the incrementing and comparing of X-bytes in a loop like this:

```
        LDY    #0              ; Use Y as an index in each page
STORE   LDA    #0              ; Value to put in each location.
        STA    (50),Y          ; Extended-address operation.
        INY                    ; Do next byte in page
        BNE    STORE

        INC    51              ; Move to next page
        BNE    CHECK           ; If same bank, check limit
        LDA    #80             ;   else
        STA    51              ;     set page to $80
        INC    1651            ;     and increment X-byte

CHECK   LDA    1651            ; Compare X-byte to
        CMP    EXTLIMIT        ;   limit X-byte
        BCC    STORE           ; If less than, clear page

        LDA    51              ;   else compare page
        CMP    EXTLIMIT + 1    ;     to limit page
        BCC    STORE           ; If less than
        BEQ    STORE           ; or equal, clear page
```

7.1.3 Event Arming and Response

To arm an event, an interpreter may pass the starting address of its event handler to a device driver that can detect the event. When the event occurs, the interpreter's event handler will be called. One way to arm an event is by a D_CONTROL call to a device driver.

For example, assume that the .CONSOLE device driver defines a certain keypress as an event. An interpreter that wishes to use this feature would include a subroutine that is to be called each time that key is pressed. The interpreter would make a D_CONTROL call to the .CONSOLE driver, passing it the ASCII code of the keypress to detect and the address of the event handler. When the key is pressed, the console queues the event handler's address, and SOS calls the event handler to handle the keypress.

The D_CONTROL calls that arm an event for a given device driver are described in the documentation accompanying that driver. For the .CONSOLE events, see the *Apple III Standard Device Drivers Manual.*

7.2 A Sample Interpreter

This section illustrates the design and construction of a very simple interpreter. The example is simple, but has all the parts an interpreter must have. It shows how SOS calls are made (see Chapter 8 for a full explanation), and how events are handled. The complete listing of the interpreter is shown in the next section; in this section we explain portions in detail.

This model is intended for demonstration only. It does not fully show all features of SOS (such as memory allocation) available to an interpreter, nor does it contain comprehensive error-checking and debugging aids. Use this model only to gain insight into the construction of an interpreter; please do not base your own designs upon it.

This program, SCREENWRITER, reads a byte from the keyboard, then writes it out to the screen, without filtering out control characters. It writes explicitly, without using screen echo.

The interpreter contains an event mechanism. When CONTROL-Q is read, the console driver detects it as an event. The event is processed when control next returns to the interpreter. If the character typed before the CONTROL-Q is ESC, the event handler beeps thrice and issues a TERMINATE call; if not, the event handler just beeps thrice.

This interpreter is deliberately inconsistent in style, in order to show different ways of coding SOS calls. Some calls are coded in line; some, as subroutines. Some are coded with a macro, SOS; some are not. The macro itself can use the SOS call number, or the number can be given the name of the call, via an .EQUate statement.

The syntax for a SOS call using the SOS macro is

SOS **call_num**, **parameter_list** pointer

For example, the call

SOS READ, READLIST

uses the label READ, which has been defined as $CA by an .EQUate. This call could also have been coded as

SOS 0CA, READLIST

READLIST is a pointer to the required parameter list. In this sample interpreter, the required list precedes the call, as the Apple III Pascal Assembler accepts backward references more readily than forward references.

Here is the macro definition for a SOS call block:

```
.MACRO   SOS                        ; Macro def for SOS call block
BRK                                 ; Begin SOS call block
.BYTE    %1                         ; call_num
.WORD    %2                         ; parameter_list pointer
.ENDM                               ; end of macro definition
```

After the header and parameter lists for various calls (shown in the complete listing, but not in this section), comes the main interpreter program, which is in two sections. The first section, the initialization block, opens the console and gets its dev_num; turns off screen echo; passes its ref_num and dev_num to subroutines; arms the attention event; and sets the fence.

```
BEGIN     .EQU    *
          JSR     OPENCONS          ; Open .CONSOLE
          JSR     GETDNUM           ; Get dev_num
          JSR     SETCONS           ; Disable echo
          JSR     ARMCTRLQ          ; Arm attention event
          SOS     60, FENLIST       ; Set event fence to 0:
                                    ;   here we coded "60" directly

          LDA     REF               ; Set up ref_num
          STA     RREF              ;   for reads
          STA     WREF              ;   and writes
```

The main program loop uses a two byte I/O buffer, the second byte of which is always a line feed (LF). The main program reads a byte from the keyboard into the first byte of the I/O buffer, then checks whether that byte is a carriage return (CR): if so, both bytes in the buffer will be written; if not, only the first byte will be written. This is done by setting the value of the write count (WCNT in the listing, or **bytes** in the call definition) to 2 or 1, respectively. The loop repeats indefinitely; the only exit from the program is through the event-handler subroutine, HANDLER.

The numbers preceded by a dollar sign, like $010, are local labels. The numbers are decimal, not hex.

```
$010      SOS     READ, RCLIST      ; Read in one byte:
                                    ;   here we used READ for 0CA
          LDA     RCNT              ; IF no bytes were read
          BEQ     $010              ;   THEN go read again

          STA     WCNT              ; Set up write count
          LDA     BUFFER
          CMP     #0D               ; IF first byte in buffer is CR
          BNE     $020              ;   THEN write out LF also
          INC     WCNT

$020      SOS     WRITE, WPLIST     ; Write out 1 or 2 bytes

          JMP     $010              ; Repeat ad infinitum
```

The first subroutine is OPENCONS, which opens the .CONSOLE file for reading and writing. It consists of a single SOS OPEN call, and is coded with the parameter lists preceding the call block, which here is coded without a macro.

```
COLIST      .BYTE   04              ; 4 required parameters for OPEN
            .WORD   CNAME           ;   pathname pointer
CREF        .BYTE   00              ;   ref_num returned here
            .WORD   COPLIST         ;   option_list pointer
            .BYTE   01              ;   length of opt parm list

COPLIST     .BYTE   03              ; Open for reading and writing

OPENCONS                            ; Here we didn't use a macro.
            BRK                     ; Begin SOS call block
            .BYTE   0C8             ; Open the console.
            .WORD   COLIST          ; Pointer to parameter list
            LDA     CREF            ; Save the result ref_num
            STA     REF             ;   for READs and WRITEs.
            RTS
```

The next subroutine, GETDNUM, which returns the dev_num of .CONSOLE, is coded similarly, except that it has no optional parameter list.

The SETCONS subroutine suppresses screen echo on the .CONSOLE file. This is a very simple example of a D__CONTROL call, as the control list is only one byte long; the next is more complex.

```
SETLIST     .BYTE   03              ; 3 required parms for D__CONTROL
CNUM        .BYTE   00              ;   dev_num of .CONSOLE
            .BYTE   0B              ;   control_code = 0B: screen echo
            .WORD   CONLIST         ;   control_list pointer

CONLIST     .BYTE   FALSE           ;   Disable screen echo

SETCONS
            LDA     CONSNUM         ; Set up device number
            STA     CNUM            ;   of .CONSOLE
            SOS     D__CNTL, SETLIST
            RTS
```

The ARMCTRLQ subroutine arms the Attention Event for CONTROL-Q. The D__CONTROL call in this subroutine sends the event priority, event ID, event-handler address, and the attention character code to the .CONSOLE driver.

```
DCLIST      .BYTE   03              ; 3 required parms for D__CONTROL
DNUM        .byte   00              ;   dev_num of .CONSOLE goes here
            .BYTE   6               ;   control_code = 06:
                                    ;       Arm Attention Event
            .WORD   CLIST           ;   control_list pointer

CLIST                               ; Control list
            .BYTE   0FF             ;   Event priority
            .BYTE   02              ;   Event ID
            .WORD   HANDLER         ;   Event handler address
BANK        .BYTE   00              ;   Event handler bank
            .BYTE   11              ;   Attention character = CTRL-Q

ARMCTRLQ
            LDA     BREG            ; Set up bank number
            STA     BANK            ;   of event handler
            LDA     CONSNUM         ; Set up device number
            STA     DNUM            ; for control request
            SOS     D__CNTL, DCLIST ; D__CONTROL call macro
            RTS
```

The next subroutine, HANDLER, is the attention event handler. It reads the attention character (CONTROL-Q) from .CONSOLE, then beeps thrice. If the previous character was ESCAPE, the program terminates. A buffer separate from the main I/O buffer is used for reading the attention character, as otherwise the attention character would sometimes clobber the character in the buffer before it could be written to the screen.

The buffer BELLS contains three BEL characters, separated by a number of SYNC characters. When written to the console, these cause a total delay of about 150 ms. HBLK1 and HBLK2 are required parameter lists for the READ and WRITE calls. HBUF1 is a one-byte buffer for the attention character.

```
BELLS     .EQU    *                          ; Buffer with BELs and delay:
          .BYTE   07                         ; BEL
          .BYTE   16,16,16,16,16,16,16,16,16 ; SYNCs
          .BYTE   07                         ; BEL
          .BYTE   16,16,16,16,16,16,16,16,16 ; SYNCs
          .BYTE   07                         ; BEL
BELLEN    .EQU    *-BELLS                    ; Calculate buffer length

HBLK1     .BYTE   04                         ; 4 required parameters for READ
HREF1     .BYTE   00                         ;   ref_num
          .WORD   HBUF1                      ;    data_buffer pointer
          .WORD   0001                       ;    request_count
          .WORD   0000                       ;    transfer_count

HBUF1     .BYTE   0                          ; Buffer for reading attention char

HBLK2     .BYTE   03                         ; 3 required parameters for WRITE
HREF2     .BYTE   00                         ;    ref_num
          .WORD   BELLS                      ;    data_buffer pointer
          .WORD   BELLEN                     ;    request_count

HBLK3     .BYTE   01                         ; 1 required parameter for CLOSE
          .BYTE   00                         ;    ref_num = 0: CLOSE all files

HBLK4     .BYTE   00                         ; 0 required parms for TERMINATE
```

These data structures are followed by the actual code of the event handler. Here the SOS calls are coded using macros.

```
HANDLER
          LDA     REF                        ; Set up reference numbers
          STA     HREF1                      ;    for console READ
          STA     HREF2                      ;    and console WRITE

          SOS     READ, HBLK1                ; Read attention character

          SOS     WRITE, HBLK2               ; Write three BELs to .CONSOLE

          LDA     BUFFER
          CMP     #1B                        ; IF last keystroke was ESCAPE
          BNE     $010

          SOS     0CC, HBLK3                 ;    THEN CLOSE all files
          SOS     065, HBLK4                 ;          and TERMINATE

$010      JSR     ARMCTRLQ                   ;    ELSE re-arm attention event
          RTS                                ;          and resume execution
```

The TERMINATE call could have been coded in the following perverse way:

```
TERM      BRK                          ; Begin SOS call
          .BYTE    065                 ; call_num for TERMINATE
          .WORD    TERM                ; parameter_list pointer
```

Since the TERMINATE call has no parameters, the required parameter list need be only an ASCII null ($00). Thus TERM, the **parameter_list** pointer, points to the BRK that begins the call.

A simpler coding, using a macro, is this:

```
TERM      SOS      065, TERM           ; Pointer to BRK
```

The following pages contain a complete listing of the program, including all subroutines and parameter lists, as well as the code necessary to generate a valid header.

7.2.1 Complete Sample Listing

```
PAGE -   0
Current memory available:   17406
0000|                                 .ABSOLUTE
0000|                                 .NOPATCHLIST
0000|                                 .NOMACROLIST
2 blocks for procedure code  16136 words left
```

```
PAGE -   1 SCREENWR  FILE:


0000|                         .PROC SCREENWRITER
Current memory available:   16881
0000|
0000|                    ;******************************************************************
0000|                    ;
0000|                    ;       Screenwriter Program
0000|                    ;
0000|                    ;       Sample Interpreter for SOS Reference Manual
0000|                    ;
0000|                    ;       Don Reed and Thomas Root, 11 August 1982
0000|                    ;
0000|                    ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
0000|                    ;
0000|                    ;       This program reads bytes from the keyboard, then writes
0000|                    ;       them out to the screen, without filtering out control
0000|                    ;       characters.  It writes explicitly, without using screen
0000|                    ;       echo.
0000|                    ;
0000|                    ;       The interpreter contains an event mechanism.  When
0000|                    ;       CONTROL-Q is read, the console driver detects it as an
0000|                    ;       event.  The event is processed when control next returns
0000|                    ;       to the interpreter.  If the character typed before the
0000|                    ;       CONTROL-Q is ESC, the event handler beeps thrice and
0000|                    ;       issues a TERMINATE call; if not, the event handler just
0000|                    ;       beeps thrice.
0000|                    ;
0000|                    ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
0000|                    ;
0000|                    ;       Note on programming style: the style of this program is
0000|                    ;       deliberately inconsistent, to show several ways to code
0000|                    ;       SOS calls.  They can be coded in line; they can be coded
0000|                    ;       as subroutines.  They can be coded with or without a
0000|                    ;       macro, SOS.  The macro itself can use the SOS call number,
0000|                    ;       or it can use the name, via an .EQUate.  In general,
0000|                    ;       data structures appear before the code using them: this
0000|                    ;       is recommended practice with the Apple III Pascal
0000|                    ;       Assembler.
0000|                    ;
0000|                    ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
0000|                    ;
0000|                    ;       The source file for the Screenwriter program is replicated
0000|                    ;       as SCREENWRIT.TEXT on the ExerSOS disk.
0000|                    ;
0000|                    ;******************************************************************
```

```
PAGE -   2 SCREENWR  FILE:


0000|                            .PAGE
0000|                    ;******************************************************************
0000|                    ;
0000|                    ;       Header Part of File
0000|                    ;
0000|                    ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
0000|
0000| 9000               START   .EQU    9000            ; Code begins at $9000
0000|                            .ORG    START-0E        ;   Leave 12 bytes for header
8FF2|
8FF2| 53 4F 53 20 4E 54 52       .ASCII  "SOS NTRP"      ; label for SOS.INTERP
8FF9| 50
8FFA| 0000                       .WORD   0000            ; opt_header_length = 0
8FFC| 0090                       .WORD   START           ; loading_address
8FFE| ****                       .WORD   CODELEN         ; code_length
9000|
9000| 4C ****                    JMP     BEGIN           ; Jump to beginning of code
9003|
9003|                    ;******************************************************************
```

```
PAGE -   3  SCREENWR  FILE:

9003|                             .PAGE
9003|                      ;**************************************************************
9003|                      ;
9003|                      ;      Macros, Equates, and Global Data Area
9003|                      ;
9003|                      ;      The syntax for a SOS call using the macro below is
9003|                      ;
9003|                      ;      SOS     call_num, parameter_list pointer
9003|                      ;
9003|                      ;      The macro definition for a SOS call block using the above
9003|                      ;      format is below:
9003|                      ;
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|
9003|                             .MACRO  SOS             ; Macro def for SOS call block
9003|                             BRK                     ; Begin SOS call block
9003|                             .BYTE   %1              ; call_num
9003|                             .WORD   %2              ; parameter_list pointer
9003|                             .ENDM                   ; end of macro definition
9003|
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|                      ;
9003|                      ;      Here are .EQUates for call_nums:
9003|                      ;
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|
9003| 00CA                 READ    .EQU    0CA             ; call_num for READ
9003| 00CB                 WRITE   .EQU    0CB             ; call_num for WRITE
9003| 0083                 D_CNTL  .EQU    083             ; call_num for D_CONTROL
9003|
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|                      ;
9003|                      ;      Here are more .EQUates:
9003|                      ;
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|
9003| 0000                 FALSE   .EQU    00
9003| 0080                 TRUE    .EQU    80
9003|
9003| FFEF                 BREG    .EQU    0FFEF           ; Bank register
```

```
PAGE -   4  SCREENWR  FILE:

9003|                             .PAGE
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|                      ;
9003|                      ;      These variables are used for communication between the
9003|                      ;      main program and the OPENCONS subroutine.
9003|                      ;
9003|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9003|
9003| 08                   CNAME   .BYTE   08              ; name_length
9004| 2E 43 4F 4E 53 4F 4C         .ASCII  ".CONSOLE"      ; pathname of console
900B| 45
900C| 00                   REF     .BYTE   00              ; Console ref_num
900D| 00                   CONSNUM .BYTE   00              ; Console dev_num
900E|
900E|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
900E|                      ;
900E|                      ;      Here is the data buffer for the READ and WRITE calls in
900E|                      ;      the main program.  Only the first byte is written into;
900E|                      ;      one or both are written out.
900E|                      ;
900E|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
900E|
900E| 00 0A                BUFFER  .BYTE   00, 0A          ; data buffer with trailing LF
9010|
9010|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9010|                      ;
9010|                      ;      Here are required parameter lists for SOS calls in the
9010|                      ;      main program.
9010|                      ;
9010|                      ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9010|
9010| 01                   FENLIST .BYTE   01              ; 1 parameter for SET_FENCE
9011| 00                           .BYTE   00              ;   fence = 0
9012|
9012| 04                   RCLIST  .BYTE   04              ; 4 parameters for READ
9013| 00                   RREF    .BYTE   00              ;   ref_num
9014| 0E90                         .WORD   BUFFER          ;   data_buffer pointer
9016| 0100                         .WORD   0001            ;   request_count
9018| 0000                 RCNT    .WORD   0000            ;   transfer_count
901A|
901A| 03                   WPLIST  .BYTE   03              ; 3 parameters for WRITE
901B| 00                   WREF    .BYTE   00              ;   ref_num (from OPEN call)
901C| 0E90                         .WORD   BUFFER          ;   data_buffer pointer
901E| 0000                 WCNT    .WORD   0000            ;   request_count
9020|
9020|                      ;**************************************************************
```

```
PAGE -   5  SCREENWR  FILE:

9020|                              .PAGE
9020|                      ;*****************************************************************
9020|                      ;
9020|                      ;       Main Program Code
9020|                      ;
9020|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9020|                      ;
9020|                      ;       This is the setup portion, executed at the start.
9020|                      ;
9020|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9020|
9020| 9020                 BEGIN   .EQU    *
9020| 20 ****                      JSR     OPENCONS        ; Open .CONSOLE
9023| 20 ****                      JSR     GETDNUM         ; Get dev_num
9026| 20 ****                      JSR     SETCONS         ; Disable_echo
9029| 20 ****                      JSR     ARMCTRLQ        ; Arm attention event
902C|                              SOS     60, FENLIST     ; Set event fence to 0:
9030|                                                      ;   here we coded "60" directly
9030|
9030| AD 0C90                      LDA     REF             ; Set up ref_num
9033| 8D 1390                      STA     RREF            ;   for reads
9036| 8D 1B90                      STA     WREF            ;   and writes
9039|
9039|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9039|                      ;
9039|                      ;       This is the main loop, executed until termination.
9039|                      ;
9039|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9039|
9039|                      $010    SOS     READ, RCLIST    ; Read in one byte:
903D|                                                      ;   here we used READ for 0CA
903D| AD 1890                      LDA     RCNT            ; IF no bytes were read
9040| F0F7                         BEQ     $010            ;   THEN go read again
9042|
9042| 8D 1E90                      STA     WCNT            ; Set up write count
9045| AD 0E90                      LDA     BUFFER
9048| C9 0D                        CMP     #0D             ; IF first byte in buffer is CR
904A| D0**                         BNE     $020            ;   THEN write out LF also
904C| EE 1E90                      INC     WCNT
904F|
904F|                      $020    SOS     WRITE, WPLIST   ; Write out 1 or 2 bytes
9053|
9053| 4C 3990                      JMP     $010            ; Repeat ad infinitum
9056|
9056|                      ;*****************************************************************
```

```
PAGE -   6  SCREENWR  FILE:

9056|                              .PAGE
9056|                      ;*****************************************************************
9056|                      ;
9056|                      ;       SUBROUTINES
9056|                      ;
9056|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9056|                      ;
9056|                      ;       OPENCONS:  open the .CONSOLE file for reading
9056|                      ;
9056|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9056|
9056| 04                   COLIST  .BYTE   04              ; 4 required parameters for OPEN
9057| 0390                         .WORD   CNAME           ;   pathname pointer
9059| 00                   CREF    .BYTE   00              ;   ref_num returned here
905A| ****                         .WORD   COPLIST         ;   option_list pointer
905C| 01                           .BYTE   01              ;   length_of opt parm list
905D|
905D| 03                   COPLIST .BYTE   03              ; Open for reading and writing
905E|
905E|
905E|                      OPENCONS                        ; Here we didn't use a macro.
905E| 00                           BRK                     ; Begin SOS call block.
905F| C8                           .BYTE   0C8             ; Open the console.
9060| 5690                         .WORD   COLIST          ; Pointer to parameter list
9062| AD 5990                      LDA     CREF            ; Save the result ref_num
9065| 8D 0C90                      STA     REF             ;   for READs and WRITEs.
9068| 60                           RTS
9069|
9069|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9069|                      ;
9069|                      ;       GETDNUM:  Get the device number of .CONSOLE
9069|                      ;
9069|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9069|
9069| 02                   GDLIST  .BYTE   02              ; 2 parameters for GET_DEV_NUM
906A| 0390                         .WORD   CNAME           ;   dev_name pointer
906C| 00                   GDNUM   .BYTE   00              ;   dev_num goes here
906D|
906D|
906D|                      GETDNUM
906D| 00                           BRK
906E| 84                           .BYTE   84              ; Call GET_DEV_NUM
906F| 6990                         .WORD   GDLIST
9071| AD 6C90                      LDA     GDNUM           ; Save the result dev_num
9074| 8D 0D90                      STA     CONSNUM         ;   for console control
9077| 60                           RTS
```

```
PAGE -    7  SCREENWR  FILE:

9078|                              .PAGE
9078|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9078|                      ;
9078|                      ;        SETCONS:  set the .CONSOLE file to suppress screen echo
9078|                      ;
9078|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9078|
9078| 03                   SETLIST .BYTE   03              ; 3 required parms for D_CONTROL
9079| 00                   CNUM    .BYTE   00              ;   dev_num of .CONSOLE
907A| 0B                           .BYTE   0B              ;   control_code = 0B: screen echo
907B| ****                         .WORD   CONLIST         ;   control_list pointer
907D|
907D| 00                   CONLIST .BYTE   FALSE           ;   Disable screen echo
907E|
907E|
907E|                      SETCONS
907E| AD 0D90                      LDA     CONSNUM         ; Set up device number
9081| 8D 7990                      STA     CNUM            ;   of .CONSOLE
9084|                              SOS     D_CNTL, SETLIST
9088| 60                           RTS
9089|
9089|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9089|                      ;
9089|                      ;        ARMCTRLQ:  Arm the Attention Event for CONTROL-Q
9089|                      ;
9089|                      ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9089|
9089| 03                   DCLIST  .BYTE   03              ; 3 required parms for D_CONTROL
908A| 00                   DNUM    .BYTE   00              ;   dev_num of .CONSOLE goes here
908B| 06                           .BYTE   06              ;   control_code= 06:
908C|                                                      ;       Arm Attention Event
908C| ****                         .WORD   CLIST           ;   control_list pointer
908E|
908E|                      CLIST                           ; Control list
908E| FF                           .BYTE   OFF             ;   Event priority
908F| 02                           .BYTE   02              ;   Event ID
9090| ****                         .WORD   HANDLER         ;   Event handler address
9092| 00                   BANK    .BYTE   00              ;   Event handler bank
9093| 11                           .BYTE   11              ;   Attention char = CTRL-Q
9094|
9094|
9094|                      ARMCTRLQ
9094| AD EFFF                      LDA     BREG            ; Set up bank number
9097| 8D 9290                      STA     BANK            ;   of event handler
909A| AD 0D90                      LDA     CONSNUM         ; Set up device number
909D| 8D 8A90                      STA     DNUM            ; for control request
90A0|                              SOS     D_CNTL, DCLIST  ; D_CONTROL call macro
90A4| 60                           RTS
```

```
PAGE -    8  SCREENWR  FILE:

90A5|                               .PAGE
90A5|                       ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
90A5|                       ;
90A5|                       ;       HANDLER:  Attention event handler subroutine
90A5|                       ;
90A5|                       ;       This subroutine reads the attention character (CONTROL-Q)
90A5|                       ;       from .CONSOLE, then beeps thrice.  If the previous
90A5|                       ;       character was ESCAPE, the program terminates.
90A5|                       ;
90A5|                       ;       A buffer separate from the main data buffer is used for
90A5|                       ;       reading the attention character, as otherwise the
90A5|                       ;       attention character would sometimes clobber the character
90A5|                       ;       in the data buffer before it could be written.
90A5|                       ;
90A5|                       ;       The buffer BELLS contains three BEL characters, separated
90A5|                       ;       by a number of SYNC characters.  When written to the
90A5|                       ;       console, these cause a total delay of about 150 ms.
90A5|                       ;
90A5|                       ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
90A5|
90A5| 90A5                  BELLS   .EQU    *               ; Buffer with BELs and delay:
90A5| 07                            .BYTE   07                                      ; BEL
90A6| 16 16 16 16 16 16 16          .BYTE   16, 16, 16, 16, 16, 16, 16, 16, 16      ; SYNCs
90AD| 16 16
90AF| 07                            .BYTE   07                                      ; BEL
90B0| 16 16 16 16 16 16 16          .BYTE   16, 16, 16, 16, 16, 16, 16, 16, 16      ; SYNCs
90B7| 16 16
90B9| 07                            .BYTE   07                                      ; BEL
90BA| 0015                  BELLEN  .EQU    *-BELLS         ; Calculate buffer length
90BA|
90BA| 04                    HBLK1   .BYTE   04              ; 4 required parameters for READ
90BB| 00                    HREF1   .BYTE   00              ;   ref_num
90BC| ****                          .WORD   HBUF1           ;   data_buffer pointer
90BE| 0100                          .WORD   0001            ;   request_count
90C0| 0000                          .WORD   0000            ;   transfer_count
90C2|
90C2| 00                    HBUF1   .BYTE   0               ; Buffer for attention character
90C3|
90C3| 03                    HBLK2   .BYTE   03              ; 3 required parameters for WRITE
90C4| 00                    HREF2   .BYTE   00              ;   ref_num
90C5| A590                          .WORD   BELLS           ;   data_buffer pointer
90C7| 1500                          .WORD   BELLEN          ;   request_count
90C9|
90C9| 01                    HBLK3   .BYTE   01              ; 1 required parameter for CLOSE
90CA| 00                            .BYTE   00              ;   ref_num = 0: CLOSE all files
90CB|
90CB| 00                    HBLK4   .BYTE   00              ; 0 required parms for TERMINATE
```

```
PAGE -   9  SCREENWR  FILE:


90CC|                                 .PAGE
90CC|                       HANDLER
90CC| AD 0C90                         LDA     REF             ; Set up reference numbers
90CF| 8D BB90                         STA     HREF1           ;   for console READ
90D2| 8D C490                         STA     HREF2           ;   and console WRITE
90D5|
90D5|                                 SOS     READ, HBLK1     ; Read attention character
90D9|
90D9|                                 SOS     WRITE, HBLK2    ; Write three BELs to .CONSOLE
90DD|
90DD| AD 0E90                         LDA     BUFFER
90E0| C9 1B                           CMP     #1B             ; IF last keystroke was ESCAPE
90E2| D0**                            BNE     $010
90E4|
90E4|                                 SOS     0CC, HBLK3      ;   THEN CLOSE all files
90E8|                                 SOS     065, HBLK4      ;        and TERMINATE
90EC|
90EC| 20 9490               $010      JSR     ARMCTRLQ        ;   ELSE re-arm attention event
90EF| 60                              RTS                     ;        and resume execution
90F0|
90F0|                       ;*****************************************************************
90F0|
90F0|                       ;*****************************************************************
90F0|                       ;
90F0|                       ;         End of program -- calculate length
90F0|                       ;
90F0|                       ;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
90F0|
90F0| 00F0                  CODELEN .EQU    *-START         ; Calculate number of bytes in
90F0|                                 .END                    ;    program
```

```
PAGE -  10  SCREENWR  FILE:   SYMBOLTABLE DUMP


AB - Absolute     LB - Label      UD - Undefined      MC - Macro
RF - Ref          DF - Def        PR - Proc           FC - Func
PB - Public       PV - Private    CS - Consts


ARMCTRLQ LB 9094|  BANK     LB 9092|  BEGIN    LB 9020|  BELLEN   AB 0015|
CLIST    LB 908E|  CNAME    LB 9003|  CNUM     LB 9079|  CODELEN  LB 00F0|
COPLIST  LB 905D|  CREF     LB 9059|  DCLIST   LB 9089|  DCNTL    AB 0083|
GDLIST   LB 9069|  GDNUM    LB 906C|  GETDNUM  LB 906D|  HANDLER  LB 90CC|
HBLK4    LB 90CB|  HBUF1    LB 90C2|  HREF1    LB 90BB|  HREF2    LB 90C4|
READ     AB 00CA|  REF      LB 900C|  RREF     LB 9013|  SCREENWR PR ----|
START    AB 9000|  TRUE     AB 0080|  WCNT     LB 901E|  WPLIST   LB 901A|


BELLS    LB 90A5|  BREG     AB FFEF|  BUFFER   LB 900E
COLIST   LB 9056|  CONLIST  LB 907D|  CONSNUM  LB 900D
DNUM     LB 908A|  FALSE    AB 0000|  FENLIST  LB 9010
HBLK1    LB 90BA|  HBLK2    LB 90C3|  HBLK3    LB 90C9
OPENCONS LB 905E|  RCLIST   LB 9012|  RCNT     LB 9018
SETCONS  LB 907E|  SETLIST  LB 9078|  SOS      MC ----
WREF     LB 901B|  WRITE    AB 00CB|
```

```
PAGE -  11  SCREENWR  FILE:


Current minimum space is 15687 words.

Assembly complete:      394 lines
     0   Errors flagged on this Assembly
```

7.3 Creating Interpreter Files

The Apple III Pascal Assembler reads a source text file of assembly-language statements and creates a code file consisting of a header block, a code section, and a relocation section, if the code file is relocatable. A SOS interpreter file must be in a format different from the standard code file format that is used for a module:

- It must be in absolute format, beginning at the proper memory location.
- It must have a special header that identifies the file as an interpreter, and the standard header and trailer must be removed.
- It must be named SOS.INTERP before it can be booted.

A utility program, MakeInterp, transforms code files into interpreter files. Its use is described in Appendix C.

7.4 Assembly-Language Modules

An interpreter that is too large to fit into the the memory space allocated for it can be split up into a main interpreter and one or more assembly-language modules. An interpreter can also use modules if it is made to be extensible, or if it wishes to swap sections of machine code in and out of memory. A language interpreter may use modules to allow the user programs it interprets to call assembly-language subroutines.

SOS does not directly support creating, loading, or maintaining modules: modules are defined, loaded, and called by the interpreter only.

Whereas an interpreter must be written and assembled in absolute code, a module can be in either absolute or relocatable format. A stand-alone interpreter performing an application will probably only have to support absolute modules, if any. A language interpreter, however, may support relocatable modules, as do the BASIC and Pascal interpreters.

7.4.1 Using Your Own Modules

An interpreter can use the REQUEST_SEG call to request a fixed memory segment in free memory, then load a 6502 code file into this space and execute its code. An interpreter can execute modules located in bank-switched memory by using the technique described in section 2.4.1.

In this way, an interpreter can have several sections of overlay code—subroutines that are swapped into a certain memory space only when they are needed, and are replaced by other code when their usefulness is expended. This is illustrated in Figure 7-2.

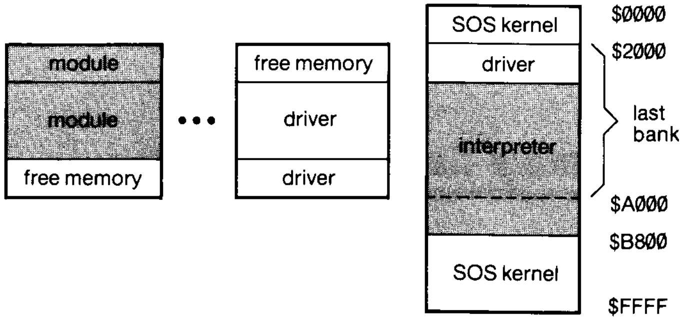


Figure 7-2. Interpreter and Modules

Rather than allocating free memory, an interpreter can also overlay code into itself and execute it without bank-switching. This technique is dangerous unless you carefully control which parts of the interpreter are being overwritten.

7.4.2 BASIC and Pascal Modules

The Apple Pascal and Business BASIC languages both have facilities for loading assembly-language modules or linking them with a Pascal or BASIC program. The modules are in the relocatable format produced by the Apple Pascal Assembler: the Pascal and BASIC interpreters are both designed to load, relocate, and execute files in this format.

The BASIC and Pascal interpreters each place a module in a convenient place in memory, then use the relocation information in the code file to alter the program code to run in its new location. A BASIC program communicates with modules via PERFORM and EXFN statements; a Pascal program uses EXTERNAL PROCEDURE and FUNCTION calls. Whereas invokable modules used by BASIC are loaded dynamically at run time, modules used by Pascal are linked in with the Pascal host program during a post-compilation linking phase, and are stored as part of the final code file.

Both the BASIC and Pascal interpreters pass parameters to their modules via the interpreter's stack. The modules remove and store the return information, then pull the parameter bytes off the stack and process them. When they are finished, they push the return information back on the stack and perform an RTS.

A module used by the BASIC or Pascal interpreter does not need to know any entry points in the interpreter.

A module can access your programs or data by means of pointer parameters. The interpreter passes the two bytes of the pointer on the stack, and sets up the X-bytes of the pointer in a fixed location in the interpreter's X-page. The module pulls the pointer off the stack and stores its pointers in the proper places in the zero page: it can then use extended addressing to access the host program's data structures.

You can find more information on the use of assembly-language modules with Pascal in the *Apple III Pascal Program Preparation Tools* manual, in the chapter The Assembler.

7.4.3 Creating Modules

Modules can be in either of two formats: absolute and relocatable. The absolute form is easier to load, but less versatile. If you can be sure a particular region of memory will be available for a module, you can assemble that module to fit into that region, and write a routine into your interpreter to load that module into that region. In doing so, you must take into consideration whether assembling a module to run in a particular region will affect the interpreter's memory requirements. You can also do this with a number of modules: you can even assemble several modules for the same region, if they are to be used one at a time and swapped in as needed.

Relocatable modules can go anywhere in free memory, so they can more easily be used by machines of different memory sizes, driver sets, and so forth. A language interpreter that supports modules will probably support relocatable modules. However, such an interpreter must take care of the relocation itself. This task goes beyond the scope of this manual. The data formats of relocatable assembly-language code files are described in Appendix E; more detail is in the *Apple III Pascal Technical Reference Manual.* If you are designing an interpreter that supports relocatable modules and need further assistance, contact the Apple PCS Division Technical Support Department.

8

Making SOS Calls

8.1 Types of SOS Calls

An interpreter communicates with SOS primarily through SOS calls. A SOS call is a request that SOS perform an action or return some information about a file, device, or memory segment.

SOS calls fall into four categories:

- File calls, which manipulate files according to the file model presented in Chapter 4;
- Device calls, which manipulate devices according to the device model presented in Chapter 3;
- Memory calls, which allocate and release memory for interpreters and keep track of areas of free memory; and
- Utility calls, which access the system clock, the event fence, and other resources.

The individual SOS calls are presented in Volume 2. The way a SOS call is made, however, is the same regardless of the function of the particular call; the remainder of this section discusses how an interpreter makes SOS calls.

8.2 Form of a SOS Call

A SOS call has three parts: the call block, the required parameter list, and the optional parameter list. Not every call has every part. The parts need not be in any particular order, and need not be contiguous, as they are linked by pointers.

8.2.1 The Call Block

A SOS call begins with the *call block*, a four-byte sequence executed as part of an interpreter's code. Figure 8-1 is a diagram of a call block, along with the code implementing it:

```
54F9|00       CALBLK  BRK
54FA|C8               .BYTE 0C8 ; OPEN
54FB|2052             .WORD PLIST ; PTR
```

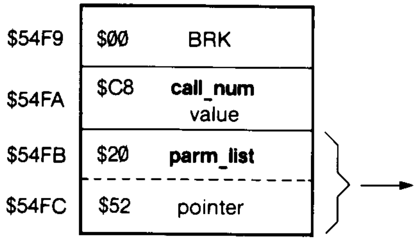


Figure 8-1. SOS Call Block

The SOS call block has three fields:

BRK (1 byte):

This field always contains the BRK opcode, $00;

call_num (1 byte):

This field contains the SOS call number, which must correspond to a valid SOS call.

parm_list (2 bytes):

This field contains a pointer to the *required parameter list* for this SOS call. The **parm_list** is an address in S-bank notation, $*nnnn*, which specifies a location in the current bank or in the S-bank, *never* in the zero page. The location specified contains the first byte of the required parameter list for the call being made: the required parameter list is described below.

If the **call_num** or the **parm_list** is invalid, SOS returns an error code to the caller.

If the format of the SOS call is correct, SOS performs the requested action. After the call is completed, SOS restores the state of the machine (the values in the X- and Y-registers and all status flags except Z and N) and returns control to the caller. If an error was encountered, the error code is returned in the accumulator. If the call was error-free, the accumulator returns $00. You can think of a SOS call as a 4-byte LDA #ERRORCODE instruction; you can check for the presence of an error code with the BEQ and BNE instructions.

8.2.2 The Required Parameter List

The *required parameter list* is a table in memory that the interpreter uses to communicate with SOS. It is from here that a SOS call gets the information it needs, and it is also hère that the call returns information to the caller.

Each SOS call expects a certain number of parameters: the number and type of parameters is different for each call. But the first byte of the required parameter list for any SOS call always contains the number of parameters for the call (not the number of bytes in the list). SOS checks this number against the number of parameters the call is expecting, to verify that you've supplied the correct list for that call. If the numbers don't match, SOS returns an error message.

Figure 8-2 is a required parameter list:

```
54F9|00    CALBLK  BRK
54FA|C8            .BYTE   0C8 ; OPEN
54FB|2052          .WORD   PLIST ; PTR

5220|04    PLIST   .BYTE   04
5221|0652  PATHN   .WORD   FILE1 ; PTR
5223|00    REFNUM  .BYTE   00 ; VALUE
5224|0052  OPLIST  .WORD   REQACC ; PTR
5226|04    OPLEN   .BYTE   04 ; VALUE
```

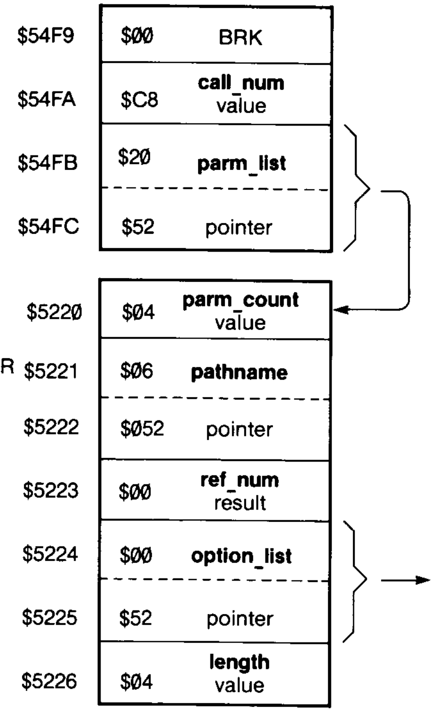


Figure 8-2. The Required Parameter List

This list contains all the required parameters for the call. A value must be supplied for each parameter: no default values are assumed. The number of parameters and the length of the required parameter list are constant for any one SOS call, and usually different for every call.

Parameters are of the four types listed below.

- A *value* parameter is 1, 2, or 4 data bytes passed from the caller to SOS. The caller places a value in the proper field of the parameter list, destroying its previous contents; SOS reads it without changing it.
- A *result* parameter is 1, 2, or 4 data bytes returned by SOS to the caller. SOS places a result in the proper field of the parameter list, destroying its previous contents; the caller reads the result without changing it.

- A *value/result* parameter is 1, 2, or 4 data bytes that are read and modified by SOS: the value and the result share the same space. The caller places a value in the proper field of the parameter list, destroying its previous contents; SOS reads the value and replaces it with a result, destroying the value. Few parameters are of this type.
- A *pointer* parameter is a 2-byte address (in any format—see section 8.3.1 below) that specifies the beginning of a buffer established by the caller. SOS uses the pointer to read information from the buffer or to return data to the same buffer. Pointers allow you to exchange variable-length data with SOS. Pointers are discussed in more detail in section 8.3.

The calling program supplies a pointer to SOS: SOS never returns or alters a pointer. It either reads from or writes to the buffer the pointer points to.

Some required parameter lists can be used for more than one call, usually for a pair of complementary calls. In the case of GET__FILE__INFO and SET__FILE__INFO (which read and change miscellaneous information about a file), you can call the former, examine its results in the required parameter list, perhaps change them, and call the latter with the same required parameter list to make your changes take effect.

8.2.3 The Optional Parameter List

Some SOS calls have parameters that need not be supplied for their simplest operation. These parameters are stored in an *optional parameter list*. A pointer (**option_list**) in the required parameter list specifies the first byte in the optional parameter list, and a **length** parameter in the required parameter list indicates how many bytes of optional parameters are supplied. Figure 8-3 is an optional parameter list:

```
54F9|00    CALBLK  BRK
54FA| C8           .BYTE   0C8 ; OPEN
54FB|2052          .WORD   PLIST ; PTR

5220|04    PLIST   .BYTE   04
5221|0652  PATHN   .WORD   FILE1 ; PTR
5223|00    REFNUM  .BYTE   00 ; VALUE
5224|0052  OPLIST  .WORD   REQACC ; PTR
5226|04    OPLEN   .BYTE   04 ; VALUE

5200|03    REQACC  .BYTE   03 ; VALUE
5201|04    PAGES   .WORD   04 ; VALUE
5202|0055  IOBUF   .WORD   00 ; PTR
```

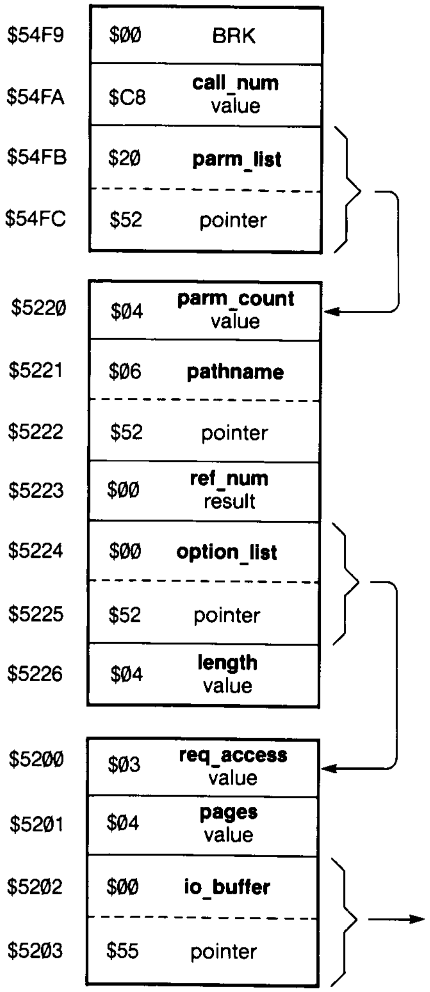


Figure 8-3. Optional Parameter List

You can supply any number of optional parameters, depending upon what you want the call to do. If the length of the optional parameter list is $00, the call will expect no optional parameters. If the length is nonzero, the call will expect as many optional parameters as can fit in that number of bytes.

Some calls supply default values for optional parameters that are not supplied; see the individual call description.

8.3 Pointer Address Extension

Some parameters in the parameter lists are pointers, which are simply addresses of other data structures (usually buffers) in memory. You can supply these addresses in S-bank, current-bank, indirect, or extended format, all of which are described in section 2.1.

When you make a SOS call involving a buffer, you must give a pointer to the buffer, and the number of bytes to be acted on. For example, the READ call requires a **data_buffer** pointer and a **request_count** parameter specifying how many bytes are to be read. SOS takes care of incrementing the pointer to read successive bytes: you need only tell it how to find the first byte.

There are two kinds of pointers:

- A *direct pointer* is a two-byte address in current-bank or S-bank format. This address is that of the beginning of the buffer in the current or S-bank.
- A *indirect pointer* is a two-byte address whose high byte is $00. This address specifies a zero-page location: the location contains the indirect or extended address of the beginning of the buffer in memory.

SOS converts both kinds of pointers into extended addresses. It does not change the pointers in your parameter list: instead it moves them to its own zero page so it can use them as extended addresses. The following paragraphs describe how SOS handles different kinds of pointers.

For all pointer conversions, SOS checks only that the pointer indicates a valid location: it does not ensure that the structure pointed to is in a valid place. It does not verify that the location pointed to actually exists in system RAM. There are limits on how big and where the buffer can be: such restrictions are discussed with each conversion.

8.3.1 Direct Pointers

A direct pointer can specify a location in either the S-bank or the current bank. If the latter, the current bank can be either bank 0 or some other bank. These cases are considered here.

Figure 8-4 shows a direct pointer:

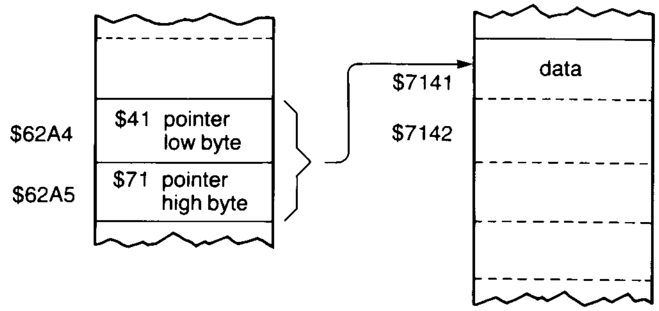


Figure 8-4. A Direct Pointer

8.3.1.1 Direct Pointers to S-Bank Locations

SOS moves the pointer directly to its zero page without conversion, and sets the X-byte of the pointer to $00 to form a normal indirect address.

Original Pointer		**Extended Form**	
$*nnnn*	$A000 to $B7FF	$*00:nnnn*	$00:A000 to $00:B7FF

A buffer that begins in the S-bank must reside in a contiguous region of S-bank memory. For example, if you start reading from a buffer beginning at location $A000 and read $200 bytes, you will cover the address range $A000 to $A1FF. If you read beyond $B7FF, you will run into SOS's region.

8.3.1.2 Direct Pointers to Current Bank Locations

SOS converts such pointers to extended form. If the current bank is not bank 0, SOS creates an X-byte based on the caller's current bank number, *b*. The result is converted to ensure that the resulting pointer specifies neither the zero page nor the last page of a bank pair.

Original Pointer (bank <> 0)	**Extended Form**
$nnnn $2000 to $21FF	*$xx:nnnn* $8b-1:8000 to $8b-1:81FF
$nnnn $2200 to $9FFF	*$xx:nnnn* $8b:0200 to $8b:7FFF

If the current bank is bank 0, then the address is converted to an extended address whose X-byte is $8F.

Original Pointer (bank = 0)	**Extended Form**
$0:nnnn $0:2000 to $0:9FFF	*$8F:nnnn* $8F:2000 to $8F:9FFF

A buffer that begins in switched memory must lie entirely within switched memory. If a buffer begins between $b:2000 and $b:9FFF, it can extend up to 64K bytes, and can wrap across bank boundaries, if b is not zero. For example, if you start reading from a buffer at $b:9F00 and read $200 bytes, you will cover the ranges $b:9F00 to $b:9FFF and $b+1:2000 to $b+1:20FF. However, the buffer may not go into the address range $A000 to $FFFF.

8.3.2 Indirect Pointers

Indirect pointers are always stored on the caller's zero page. The two-byte value in the parameter list is the address of the pointer on zero page. When SOS processes an indirect pointer, it moves the two bytes of the pointer from the caller's zero page to its own zero page, and also moves the X-byte of that pointer to its own X-page.

An indirect pointer can have an X-byte equal or unequal to zero: if it is equal to zero, the bank number can likewise be equal or unequal to zero. These cases are considered here.

Figure 8-5 shows an indirect pointer:

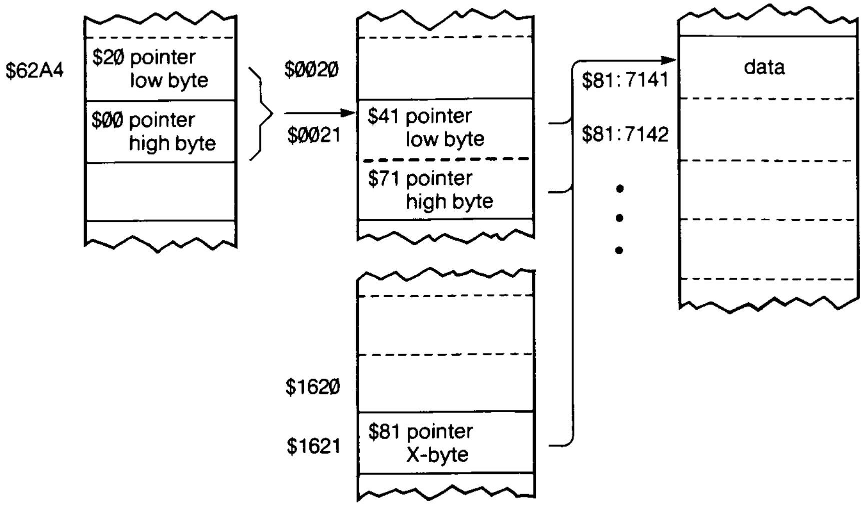


Figure 8-5. An Indirect Pointer

8.3.2.1 Indirect Pointers with an X-Byte of $00

These pointers are converted by SOS to full extended addresses, as in the direct-pointer examples above. An indirect pointer with an X-byte of 00 is identical to a direct pointer and follows the cases shown above. SOS creates an X-byte based on the caller's current bank number, *b*. The address may be converted to prevent it from pointing to the zero page, as shown in the first line below.

Original Pointer	**Extended Form**
$00:nnnn $00:$2000 to $00:$21FF	*$xx:nnnn* $8b-1:8000 to $8b-1:81FF
$00:nnnn $00:$2200 to $00:$9FFF	*$xx:nnnn* 8b:0200 to $8b:7FFF

If the current bank is bank 0, the address is converted to an extended address whose X-byte is $8F.

Original Pointer (bank = 0)	**Extended Form**
$00:nnnn $00:2000 to $00:9FFF	*$8F:nnnn* $8F:2000 to $8F:9FFF

A buffer that begins in switched memory must lie entirely within switched memory. If a buffer begins between $b:2000 and $b:9FFF, it can extend up to 64K bytes, and can wrap across bank boundaries, if b is not zero. For example, if you start reading from a buffer at $b:9F00 and read $200 bytes, you will cover the ranges $b:9F00 to $b:9FFF and $b+1:2000 to $b+1:20FF. However, the buffer may not go into the address range $A000 to $FFFF.

8.3.2.2 *Indirect Pointers with an X-Byte Between $80 and $8F*

These pointers are invalid if they point to the zero page or stack:

Original Pointer	**Extended Form**
$80:nnnn $80:0000 to $80:01FF	Invalid
$8x:nnnn $8b:0000 to $8b:00FF	Invalid

The range of addresses in the second line could be replaced by alternate form, $8b-1:8000 to $8b-1:80FF. This trick doesn't work in the first case, as bank 0 is the lowest bank.

Indirect pointers that have an X-byte between $80 and $8E are converted only to ensure that addresses produced by indexing on them do not point to the zero page. The pointers below are converted:

Original Pointer	**Extended Form**
$8x:nnnn $8b:0100 to $80:01FF	*$8x:nnnn* $8b-1:8100 to $8b-1:81FF
$8x:nnnn $8b:FF00 to $80:FFFF	*$8x:nnnn* $8b+1:7F00 to $8b+1:7FFF

The pointers below are unchanged:

Original Pointer	**Extended Form**
$8x:nnnn $8b:0200 to $8b:FEFF	*$8x:nnnn* $8b:0200 to $8b:FEFF
$8F:nnnn $8F:2000 to $8F:B7FF	*$8F:nnnn* $8b:2000 to $8b:B7FF

The X-byte $8F is a special case that looks like a direct pointer if *b* is zero.

The buffer that the above address points to can contain up to $FFFF bytes, and can wrap from one switched bank to another. SOS will handle all the pointer manipulations automatically. A buffer cannot, however, cross over into S-bank space; and it must reside in no more than three adjacent banks.

8.4 Name Parameters

Many SOS calls use device names, volume names, or pathnames as parameters. Since a name is a variable-length string of characters, it cannot be included in a parameter list: you must supply a pointer to a name. The pointer can be specified in any of the formats described above. Figure 8-6 illustrates the format of a name parameter.

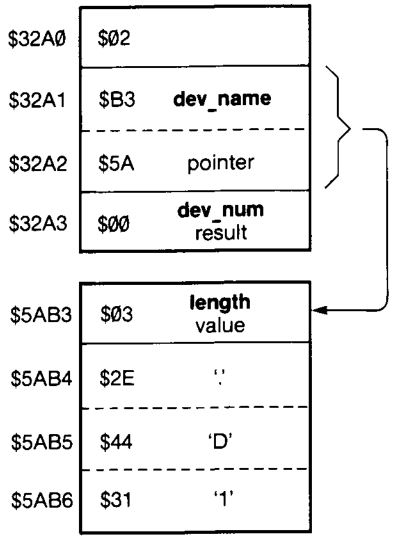


Figure 8-6. Format of a Name Parameter

The first byte pointed to by the parameter contains the number of characters in the rest of the name; the bytes immediately following contain the individual characters in sequence.

Device and volume names can contain up to 15 characters: such names use 2 to 16 bytes of storage. Pathnames can be up to 255 characters in length: such names require 2 to 256 bytes of storage.

8.5 SOS Call Error Reporting

After execution of a SOS call, the accumulator contains the error code reported by the call, and the N and Z status flags are updated accordingly. All other registers are returned to their state before the call. If the call was completed successfully, the accumulator contains $00: a BEQ instruction can detect a successful SOS call.

Error numbers range from $01 to $FF. Errors can be classified into groups by their error numbers:

- Error codes $01 through $05 indicate a problem with the form of the SOS call, or its parameters or pointers.
- Error codes $10 through $2F indicate device call errors. Either a requested operation is not supported by SOS, or the operation cannot be performed due to interface problems with a device. Some of these errors can also be produced by file calls.
- Error codes $30 through $3F are generated by individual device drivers, and they indicate a problem in a particular device.
- Error codes $40 through $5A indicate file call errors.
- Error codes $70 through $7F indicate utility call errors.
- Error codes $E0 through $EF indicate memory call errors.

These errors can be generated by SOS for any SOS call:

$01: Invalid SOS call number (BADSCNUM)

The byte immediately following the BRK instruction ($00) in the SOS call block is not the number of a currently defined SOS call.

$02: Invalid caller zero page (BADCZPAGE)

SOS requires that the interpreter use page $1A as its zero page when calling SOS.

$03: Invalid indirect pointer X-byte (BADXBYTE)

The extension (X-) byte of an indirect pointer is invalid. Legal values for this byte are

$00	Indirect, current bank
$80 through $8E	Indirect, extend bank
$8F	Indirect, S/0 bank

$04: Invalid SOS call parameter count (BADSCPCNT)

The first byte of the required parameter list contains a parameter count not expected by the specified SOS call. Either the call number is incorrect or the call is using the wrong required parameter list.

$05: SOS call pointer out of bounds (BADSCBNDS)

A SOS call pointer parameter is within a proscribed range of memory. Either the required parameter list resides on zero page or a pointer is attempting to point into SOS. The proscribed memory ranges are:

$0100 through	$01FF	Restricted for SOS
$B800 through	$FFFF	Restricted for SOS
$xx:0000 through	$xx:00FF	Zero Page
$8F:0100 through	$8F:01FF	Restricted for SOS
$8F:B800 through	$8F:FFFF	Restricted for SOS

Index

Page references in Volume 2 are shown in square brackets [ ].