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
20325 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
Acknowledgements


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

Figures and Tables xi

Preface xvii

xvii Scope Of This Manual
xviii Using this Manual
xviii About the Examples
xviii Notation and Symbols
xviii Numeric Notation
xix Special Symbols

1 The Abstract Machine 1

2 1.1 About Operating Systems
2 1.1.1 An Abstract Machine
2 1.1.2 A Resource Manager
3 1.1.3 A Common Foundation for Software
3 1.2 Overview of the Apple III
5 1.2.1 The Interpreter
5 1.2.2 SOS
6 1.2.3 Memory
7 1.2.4 Files
8 1.2.5 Devices
8 1.2.6 The 6502 Instruction Set
2 Programs and Memory

10 2.1 Addressing Modes
10 2.1.1 Bank-Switched Memory Addressing
13 2.1.2 Enhanced Indirect Addressing
16 2.2 Execution Environments
17 2.2.1 Zero Page and Stack
18 2.2.2 The Interpreter Environment
19 2.2.3 SOS Kernel Environment
20 2.2.4 SOS Device Driver Environment
22 2.2.5 Environment Summary
23 2.3 Segment Address Notation
25 2.3.1 Memory Calls
27 2.4 Memory Access Techniques
27 2.4.1 Subroutine and Module Addressing
29 2.4.2 Data Access
30 2.4.2.1 Bank-Switched Addressing
31 2.4.2.2 Enhanced Indirect Addressing
32 2.4.3 Address Conversion
33 2.4.3.1 Segment to Bank-Switched
33 2.4.3.2 Segment to Extended
34 2.4.3.3 Extended to Bank-Switched
36 2.4.4 Pointer Manipulation
36 2.4.4.1 Incrementing a Pointer
37 2.4.4.2 Comparing Two Pointers
38 2.4.5 Summary of Address Storage

3 Devices

40 3.1 Devices and Drivers
40 3.1.1 Block and Character Devices
40 3.1.2 Physical Devices and Logical Devices
41 3.1.3 Device Drivers and Driver Modules
41 3.1.4 Device Names
43 3.2 The SOS Device System
43 3.3 Device Information
45 3.4 Operations on Devices
46 3.5 Device Calls

4 Files

50 4.1 Character and Block Files
50 4.1.1 Structure of Character and Block Files
52 4.1.2 Open and Closed Files
53 4.1.3 Volumes
54 4.1.3.1 Volume Switching
55 4.1.3.2 Volume Names
56 4.2 The SOS File System
57 4.2.1 Directory Files and Standard Files
58 4.2.2 File Names
59 4.2.3 Pathnames
61 4.2.4 The Prefix and Partial Pathnames
62 4.3 File and Access Path Information
62 4.3.1 File Information
64 4.3.2 Access Path Information
67 4.3.3 Newline Mode Information
68 4.4 Operations on Files
69 4.5 File Calls
Volume 2: The SOS Calls

Figures and Tables vii

Preface ix

9 File Calls and Errors 1

2  9.1 File Calls
53  9.2 File Call Errors

10 Device Calls and Errors 57

58  10.1 Device Calls
71  10.2 Device Call Errors

11 Memory Calls and Errors 73

74  11.1 Memory Calls
88  11.2 Memory Call Errors

12 Utility Calls and Errors 89

90  12.1 Utility Calls
104  12.2 Utility Call Errors

A SOS Specifications 105

106 Version
106 Classification
106 CPU Architecture
106 System Calls
106 File Management System
107 Device Management System
108 Memory/Buffer Management System
108 Additional System Functions
109 Interrupt Management System
109 Event Management System
109 System Configuration
109 Standard Device Drivers

B ExerSOS 113

114 B.1 Using ExerSOS
117 B.2 The Data Buffer
118 B.3 The String Buffer
119 B.4 Leaving ExerSOS

C Make Interp 121

D Error Messages 123

124 D.1 Non-Fatal SOS Errors
126 D.2 Fatal SOS Errors
128 D.3 Bootstrap Errors
3 Devices

39

42 Figure 3-1 Device Name Syntax
43 Figure 3-2 The SOS Device System

4 Files

49

51 Figure 4-1 Character File Model
51 Figure 4-2 Block File Model
52 Figure 4-3 Open Files
55 Figure 4-4 The SOS Disk Request
57 Figure 4-5 Top-Level Files
58 Figure 4-6 The SOS File System
59 Figure 4-7 File Name Syntax
60 Figure 4-8 Pathname Syntax
61 Figure 4-9 Pathnames
65 Figure 4-10 Automatic Movement of EOF and Mark
66 Figure 4-11 Manual Movement of EOF and Mark

5 File Organization on Block Devices

75

77 Figure 5-1 Blocks on a Volume
78 Figure 5-2 Directory File Format
80 Figure 5-3 The Volume Directory Header
83 Figure 5-4 The Subdirectory Header
86 Figure 5-5 The File Entry
90 Figure 5-6 Date and Time Format
90 Figure 5-7 The access Attribute Field
95 Figure 5-8 Structure of a Seedling File
96 Figure 5-9 Structure of a Sapling File
96 Figure 5-10 The Structure of a Tree File
98 Figure 5-11 A Sparse File
99 Figure 5-12 Format of mark
100 Figure 5-13 Disk Organization
102 Figure 5-14 Header and Entry Fields

6 Events and Resources

103

106 Figure 6-1 Queuing An Event
106 Figure 6-2 Handling An Event: Case A
107 Figure 6-3 Handling An Event: Case B
109 Figure 6-4 The Event Queue
110 Figure 6-5 The Event Fence
111 Figure 6-6 System Status during Event Handling
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation in Text</th>
<th>Notation in Listings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>$3A5</td>
<td>3A5</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>$BAD1</td>
<td>0BAD1</td>
</tr>
<tr>
<td>Label</td>
<td>BAD1</td>
<td>BAD1</td>
</tr>
</tbody>
</table>

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.

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.
The Abstract Machine

2  1.1 About Operating Systems
2   1.1.1 An Abstract Machine
2   1.1.2 A Resource Manager
3   1.1.3 A Common Foundation for Software
3   1.2 Overview of the Apple III
5   1.2.1 The Interpreter
5   1.2.2 SOS
6   1.2.3 Memory
7   1.2.4 Files
8   1.2.5 Devices
8   1.2.6 The 6502 Instruction Set
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.
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.

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.

Programs and Memory

10 2.1 Addressing Modes
10 2.1.1 Bank-Switched Memory Addressing
13 2.1.2 Enhanced Indirect Addressing
16 2.2 Execution Environments
17 2.2.1 Zero Page and Stack
18 2.2.2 The Interpreter Environment
19 2.2.3 SOS Kernel Environment
20 2.2.4 SOS Device Driver Environment
22 2.2.5 Environment Summary
23 2.3 Segment Address Notation
25 2.3.1 Memory Calls
27 2.4 Memory Access Techniques
27 2.4.1 Subroutine and Module Addressing
29 2.4.2 Data Access
30 2.4.2.1 Bank-Switched Addressing
31 2.4.2.2 Enhanced Indirect Addressing
32 2.4.3 Address Conversion
33 2.4.3.1 Segment to Bank-Switched
33 2.4.3.2 Segment to Extended
34 2.4.3.3 Extended to Bank-Switched
36 2.4.4 Pointer Manipulation
36 2.4.4.1 Incrementing a Pointer
37 2.4.4.2 Comparing Two Pointers
38 2.4.5 Summary of Address Storage
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](image)

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:

\[
\begin{align*}
\$nnnn & \quad \text{S-Bank Address} \\
\$2000 \text{ to } \$9FFF & \quad \text{Current-Bank Address} \\
\$A000 \text{ to } \$FFFF & \quad \text{S-Bank Address}
\end{align*}
\]

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:

\[
\begin{align*}
\$b:nnnn & \quad \text{Bank-Switched Addresses} \\
\$0:2000 \text{ to } \$0:9FFF & \\
\$1:2000 \text{ to } \$1:9FFF & \\
\ldots & \\
\$E:2000 \text{ to } \$E:9FFF
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0:2000</td>
<td>First location in bank 0</td>
</tr>
<tr>
<td>$2:9FFF</td>
<td>Last location in bank 2</td>
</tr>
<tr>
<td>$F:32A4</td>
<td>Invalid: there is no bank $F.</td>
</tr>
<tr>
<td>$1:B700</td>
<td>Non-standard: use S-bank specification $B700</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

**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.

$8x:nnnn$ $80:0100$ 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 $8n:0000$ to $8n:00FF$: 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 $8(n+1):7FFF$.

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$80:8000</td>
<td>First location in bank $1</td>
</tr>
<tr>
<td>$81:7FFF</td>
<td>Last location in bank $1</td>
</tr>
<tr>
<td>$03:2215</td>
<td>Not an extended address: X-byte ignored</td>
</tr>
<tr>
<td>$81:002E</td>
<td>Invalid: use $80:802E</td>
</tr>
<tr>
<td>$81:FF2E</td>
<td>Risky: use $82:7F2E</td>
</tr>
</tbody>
</table>

**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 ($1B00 through $1AFF).

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](image)

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).

<table>
<thead>
<tr>
<th>Environment Attribute</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ Interrupts</td>
<td>Enabled</td>
</tr>
<tr>
<td>NMI Interrupts</td>
<td>Enabled or Disabled</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>Full speed</td>
</tr>
<tr>
<td>Zero Page</td>
<td>Page $1A</td>
</tr>
<tr>
<td>Stack Page</td>
<td>Page $1B</td>
</tr>
<tr>
<td>Extend Page</td>
<td>Page $16</td>
</tr>
<tr>
<td>Bank</td>
<td>Highest</td>
</tr>
</tbody>
</table>

Table 2-3. Interpreter Environment

Of the above environment attributes, only the bank register (location $FFFF) 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).
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.

<table>
<thead>
<tr>
<th>Environment Attribute</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ Interrupts</td>
<td>Enabled</td>
</tr>
<tr>
<td>NMI Interrupts</td>
<td>Enabled</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>Full speed</td>
</tr>
<tr>
<td>Zero Page</td>
<td>Page $18$</td>
</tr>
<tr>
<td>Stack Page</td>
<td>Page $01$</td>
</tr>
<tr>
<td>Extend Page</td>
<td>Page $14$</td>
</tr>
<tr>
<td>Bank</td>
<td>S-bank</td>
</tr>
</tbody>
</table>

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.

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.

<table>
<thead>
<tr>
<th>Environment Attribute</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ Interrupts</td>
<td>Enabled or Disabled</td>
</tr>
<tr>
<td>NMI Interrupts</td>
<td>Enabled or Disabled</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>Full Speed or Fixed 1 MHZ</td>
</tr>
<tr>
<td>Zero Page</td>
<td>Page $18$</td>
</tr>
<tr>
<td>Stack Page</td>
<td>Page $01$</td>
</tr>
<tr>
<td>Extend Page</td>
<td>Page $14$</td>
</tr>
<tr>
<td>Bank</td>
<td>Interpreter's or Lower</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Interpreter*</th>
<th>Kernel</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can perform a SOS call</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Can call SOS subroutines</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can be interrupted</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes**</td>
</tr>
<tr>
<td>Can respond to IRQ</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can respond to NMI</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can disable interrupts</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can detect and queue an event</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can respond to an event***</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Can access interpreter memory</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can access free memory</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* 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](image)

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

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 $9F$.

<table>
<thead>
<tr>
<th>Segment Address</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$01:30</td>
<td>Page beginning at $01:3000</td>
</tr>
<tr>
<td>$04:62</td>
<td>Page beginning at $04:6200</td>
</tr>
<tr>
<td>$00:9F</td>
<td>Page beginning at $00:9F00</td>
</tr>
</tbody>
</table>

**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.

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,
      ; as it is not located
8B42| 8D EFFF  STA  BREG
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
A703| 48      PHA  ; Save it on the stack
A704| A9 01    LDA  #01  ; Switch in
A706| 8D EFFF  STA  BREG
A709| 20 0033  JSR  3300  ; bank $1
A70C| 68      PLA  ; Call the module
A70D| 8D EFFF  STA  BREG
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 FFEF ; Define bank register
0001| ORG 0A3AA ; Code starts here
A3AA| AD EEFF LDA BREG ; Save current bank register
A3AD| 48 PHA
A3AE| A0 00 LDY #00 ; Perform a loop to
A3B0| 8C EEFF STY BREG ; zero all locations
A3B3| 98 TYA ; from $0:9800 to
A3B4| 90 0098 LOOP STA 9800,Y ; $0:98FF;
A3B7| C8 INY
A3B8| D0 FB BNE LOOP
A3BA| 68 PLA ; Store bank register
A3BB| 8D EEFF 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 ; $iohi 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:FFFF, 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,

\[
\text{if } (bb = 0F) \text{ or } (bb = 10) \\
\text{then } B := 0 \\
\text{else } B := bb; \\
\text{NNNN := pp}00
\]

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

<table>
<thead>
<tr>
<th>Segment</th>
<th>Bank-Switched</th>
</tr>
</thead>
<tbody>
<tr>
<td>$04:63</td>
<td>$(4):(6300) = $4:6300</td>
</tr>
<tr>
<td>$07:89</td>
<td>$(7):(8900) = $7:8900</td>
</tr>
<tr>
<td>$10:1F</td>
<td>$(0):(1F00) = $0:1F00</td>
</tr>
</tbody>
</table>

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 $XX:NNNN$,

\[
\begin{align*}
\text{if} \quad (bb = $00$) &\quad \{\text{zero bank}\} \\
\quad \text{or} (bb = $0F$) &\quad \{\text{low S-bank}\} \\
\quad \text{or} (bb = $10$) &\quad \{\text{high S-bank}\} \\
\text{then} \\
\quad \text{begin} \\
\quad \quad XX &:= $8F$ ; \\
\quad \quad NNNN &:= pp00 \\
\quad \text{end} \\
\text{else} &\quad \{\text{general case}\} \\
\quad \text{begin} \\
\quad \quad XX &:= $80+bb-1$ ; \\
\quad \quad NNNN &:= pp00+$6000$ \\
\quad \text{end;}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Segment</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>$09:2A$</td>
<td>$(80+9-1):(2A00+6000)$</td>
</tr>
<tr>
<td>$02:94$</td>
<td>$(80+2-1):(9400+6000)$</td>
</tr>
<tr>
<td>$0F:1E$</td>
<td>$(8F):(1E00)$</td>
</tr>
</tbody>
</table>

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 $B:NNNN$,

\[
\begin{align*}
\text{if} \quad (xx = $0F$) &\quad \text{then} \\
\quad \text{begin} \\
\quad \quad B &:= $0$ ; \\
\quad \quad NNNN &:= nnnn \\
\quad \text{end} \\
\text{else} &\quad \text{if} \quad (nnnn < $8000$) \quad \text{then} \\
\quad \quad \text{begin} \\
\quad \quad \quad B &:= xx-$80$ ; \\
\quad \quad \quad NNNN &:= nnnn+$2000$ \\
\quad \quad \text{end} \\
\text{else} &\quad \text{begin} \\
\quad \quad \quad B &:= xx-$80+1$ ; \\
\quad \quad \quad NNNN &:= nnnn-$6000$ \\
\quad \text{end;}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Extended</th>
<th>Bank-switched</th>
</tr>
</thead>
<tbody>
<tr>
<td>$86:4365$</td>
<td>$(86-80):(4365+2000)$</td>
</tr>
<tr>
<td>$82:EFB4$</td>
<td>$(82-7F):(EFB4-6000)$</td>
</tr>
</tbody>
</table>

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 $80:000F$.

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 ;
  if (xx = $8F$) and (nnnn > $20FF$)
    then begin
      xx := $80$ ;
      nnnn := nnnn-$2000$
    end;
  if (nnnn > $FFE$)
    then begin
      nnnn := nnnn-$8000$ ;
      xx := xx + 1
    end;
  until xx > $8D$
```

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 $80:FF00$ and $80:FFFF$, 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 $xx <= XX$. 

![Figure 2-10. Increment Path](image)
if \((\ (xx = XX) \quad \text{and} \quad (nnnn = NNNN) \ )\) \quad [1]

or \((\ (xx = XX-1) \quad \text{and} \quad (XX \leftrightarrow 8F) \quad \text{and} \quad (nnnn = NNNN + 8000) \ )\) \quad [2]

or \((\ (xx = 00) \quad \text{and} \quad (XX = 8F) \quad \text{and} \quad (nnnn = NNNN - 2000) \ )\) \quad [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).
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

50  4.1 Character and Block Files
    4.1.1 Structure of Character and Block Files
52  4.1.2 Open and Closed Files
53  4.1.3 Volumes
    4.1.3.1 Volume Switching
    4.1.3.2 Volume Names
56  4.2 The SOS File System
57  4.2.1 Directory Files and Standard Files
58  4.2.2 File Names
59  4.2.3 Pathnames
61  4.2.4 The Prefix and Partial Pathnames
62  4.3 File and Access Path Information
    4.3.1 File Information
64  4.3.2 Access Path Information
67  4.3.3 Newline Mode Information
68  4.4 Operations on Files
69  4.5 File Calls
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 $FFFFFFE$. 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](image)

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.

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)

![Image](https://example.com/image.png)

**Figure 4-4.** The SOS Disk Request

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.

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
/CHAP2B

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](image)

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):

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.

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 /BASICSTUFF
/.GRAFIX /BASICSTUFF/SOS.DRIVER
/PASCAL1 /BASICSTUFF/TEMPLATES
/PASCAL1/SOS.DRIVER /BASICSTUFF/TEMPLATES/PHONES
/PASCAL1/SYSTEM.PASCAL /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 S0S 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.

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.
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 has 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 ($\text{\$D}$), but can be any hex value from $\text{\$00}$ to $\text{\$FF}$. 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 newpline_char**

Newline mode is controlled by two values: *is_newline* turns newline mode on or off; *newpline_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 *newpline_char* values. A character file also has *is_newline* and *newpline_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

77  5.1  Format of Information on a Volume (SOS 1.2)
78  5.2  Format of Directory Files
    5.2.1  Pointer Fields
79  5.2.2  Volume Directory Headers
82  5.2.3  Subdirectory Headers
85  5.2.4  File Entries
89  5.2.5  Field Formats in Detail
    5.2.5.1  The storage_type Field
79  5.2.5.2  The creation and last_mod Fields
90  5.2.5.3  The access Attributes
91  5.2.5.4  The file_type Field
91  5.2.6  Reading a Directory File
92  5.3  Storage Formats of Standard Files
    5.3.1  Growing a Tree File
95  5.3.2  Seedling Files
95  5.3.3  Sapling Files
96  5.3.4  Tree Files
97  5.3.5  Sparse Files
98  5.3.6  Locating a Byte in a Standard File
99  5.4  Chapter Overview
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):

```
block 0
loader

block 1
loader

block 2
volume
directory
(key block)

block n
volume
directory
(last block)

block n + 1
volume
bit map
(first block)

block p
volume
bit map
(first block)

other
files
```

**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.

![Diagram of 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 $\text{SF}$ 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.

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.

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.

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.

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 $00004 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.

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 = $BD.

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Entry</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>file_name</td>
<td>$00</td>
<td></td>
</tr>
<tr>
<td>1 byte</td>
<td>file_type</td>
<td>$01</td>
<td></td>
</tr>
<tr>
<td>2 bytes</td>
<td>key_pointer</td>
<td>$0F</td>
<td></td>
</tr>
<tr>
<td>2 bytes</td>
<td>blocks_used</td>
<td>$10</td>
<td></td>
</tr>
<tr>
<td>3 bytes</td>
<td>EOF</td>
<td>$11</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>creation</td>
<td>$12</td>
<td></td>
</tr>
<tr>
<td>1 byte</td>
<td>version</td>
<td>$13</td>
<td></td>
</tr>
<tr>
<td>1 byte</td>
<td>min_version</td>
<td>$14</td>
<td></td>
</tr>
<tr>
<td>1 byte</td>
<td>access</td>
<td>$15</td>
<td></td>
</tr>
<tr>
<td>2 bytes</td>
<td>aux_type</td>
<td>$16</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>last_mod</td>
<td>$17</td>
<td></td>
</tr>
<tr>
<td>2 bytes</td>
<td>header_pointer</td>
<td>$18</td>
<td></td>
</tr>
</tbody>
</table>

**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.

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.

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):
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).

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.

---

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:

This file now has two data blocks and one data pointer block.
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 ($0 \leq EOF \leq 200$). 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](image)

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.
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 $\$0$).

Set the **EOF** and **mark** to position $\$0565$, and write four bytes. SOS calculates that position $\$0565$ is byte $\$0165$ ($\$0564 - \$0200 \times 2$) 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:
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 sapling, 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](image)

index_block (7 bits):
If the file is a tree file, this field tells which subindex block points to the data block. If $i = \text{index block}$, 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 $j = \text{data block}$, 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.
Figure 5-13. Disk Organization
**Events and Resources**

6.1.1 Arming and Disarming Events  
6.1.2 The Event Queue  
6.1.3 The Event Fence  
6.1.4 Event Handlers  
6.1.5 Summary of Interrupts and Events  
6.2 Resources  
6.2.1 The Clock  
6.2.2 The Analog Inputs  
6.2.3 TERMINATE  
6.3 Utility Calls

---

**Figure 5-14. Header and Entry Fields**

- **Storage Type** (4 bytes)
  - $0 = inactive file entry
  - $1 = seeding file entry
  - $2 = sapling file entry
  - $3 = tree file entry
  - $4 = subdirectory file entry
  - $C = 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
  - $OA = BASIC data file
  - $OB = Word Processor file
  - $OC = SOS system file (DRIVER, INTERP, KERNEL)
  - $OD = Reserved
  - $OE = Reserved
  - $OF = Directory file
  - $10-$BF = SOS reserved
  - $C0-$FF = ProDOS reserved

- **Access** (1 byte)
  - D = data file
  - R = read-only file
  - N = name only file
  - B = backup file
  - S = system file
  - W = write-only file
  - R = read-enabled
  - E = file is extendable

- **File Name**
  - length = length of file name ($1-$5F)
  - file_name = $1-$5F 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 ($0-$fffffff)

- **Version Min Version**
  - 0 for SOS 1.2

- **Entry Length**
  - $27 for SOS 1.2

- **Entries Per Block**
  - $2D 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.
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.

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.

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 6-2. Handling An Event: Case A

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](image)

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.
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.

\[
\text{event_id}
\]

\[
\text{SOS return address}
\]

\[
\text{PHA}
\]

\[
\text{PLA}
\]

**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.

### 6.1.4 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.
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:

```
YYYY M M D D W H H N N S S U U U
```

The meaning of each field is as below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>Year</td>
<td>1900</td>
<td>1999</td>
</tr>
<tr>
<td>MM</td>
<td>Month</td>
<td>00</td>
<td>12</td>
</tr>
<tr>
<td>DD</td>
<td>Date</td>
<td>00 or 01</td>
<td>28, 30, or 31</td>
</tr>
<tr>
<td>W</td>
<td>Day</td>
<td>01 Sunday</td>
<td>07 Saturday</td>
</tr>
<tr>
<td>HH</td>
<td>Hour</td>
<td>00 Midnight</td>
<td>23 11:00 p.m.</td>
</tr>
<tr>
<td>NN</td>
<td>Minute</td>
<td>00</td>
<td>59</td>
</tr>
<tr>
<td>SS</td>
<td>Second</td>
<td>00</td>
<td>59</td>
</tr>
<tr>
<td>UUU</td>
<td>Millisecond</td>
<td>000</td>
<td>999</td>
</tr>
</tbody>
</table>

For example, Monday, December 29, 1980, at 9:30 a.m. would be specified by the string "198012290930000000".

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_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

118  7.1  Interpreters
119  7.1.1  Structure of an Interpreter
121  7.1.2  Obtaining Free Memory
125  7.1.3  Event Arming and Response
125  7.2  A Sample Interpreter
131  7.2.1  Complete Sample Listing
143  7.3  Creating Interpreter Files
143  7.4  Assembly-Language Modules
144  7.4.1  Using Your Own Modules
145  7.4.2  BASIC and Pascal Modules
146  7.4.3  Creating 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](image-url)
The header consists of five fields, described below:

**label** (8 bytes):
This field contains eight characters included in 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
       .WORD   FIND__SEG
       .WORD   FSPARAMS  ; with the required parameters here.
       CONV    IF successful, THEN process addresses.
       PAGES   ELSE see how big it can be.
       BNE     IF any free memory exists, THEN ask again.
       JMP     ELSE stop execution.
       ERRORHALT

FSPARAMS .BYTE   06  ; Six parameters for FIND__SEG:
SRCHMOD  .BYTE   00  ; Seg must be in one bank
SEGOID   .BYTE   11  ; I'll call it seg. 11.
PAGES    .WORD   04  ; Ask for 1K of memory
BASE     .WORD   0
LIMIT    .WORD   0  ; “base” result parameter
SEGNUM   .BYTE   0  ; “seg_num” result parameter
EXTLIMIT .BYTE   0  ; Place to store (extended form of)
       .WORD   0  ; 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.

```
CONV    LDA    BASE  ; Get bank number of segment
       BEQ    SZBANK  ; Is it in bank 0?
       CMP    #0F    ; Is it in low S-bank?
       BEQ    SZBANK  ; Is it in high S-bank?
       CMP    #10    ;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 := ppbb + $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 ; Get page number in bank
       LDA    BASE + 1 ; Create low part of $00
       STA    50 ; Use limit X-byte of $8F
       LDA    #8F  ; Convert page number of limit
       STA    EXTLIMIT ; to extended form.
       LDA    LIMIT + 1 ;
       STA    EXTLIMIT + 1 ;
```

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
LDA #00
STA (50),Y
INY
BNE STORE
INC 51
LDA 51
CMP EXTLIMIT + 1
BCC STORE
BEQ STORE

STORE

; Use Y as an index in each page.
; Value to put in each location.
; Extended-address operation.
; Do next byte in page.
; Move to next page.
; Get high part of address.
; Compare with high part of limit.
; If pointer.high <= limit.high,
; 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
LDA #0
STA (50),Y
INY
BNE STORE
INC 51
BNE CHECK
LDA #0
STA 51
INC 1651

CHECK

; Use Y as an index in each page
; Value to put in each location.
; Extended-address operation.
; Do next byte in page.
; Move to next page
; If same bank, check limit
; else
; set page to $80
; and increment X-byte
; Compare X-byte to
; limit X-byte
; if less than, clear page
; else compare page
; to limit page
; if less than
; 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
  BRK
  .BYTE %1
  .WORD %2
  ENDM
```

Macro def for SOS call block
Begin SOS call block
call_num
parameter_list pointer
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:
  LDA RCNT : here we used READ for 0CA
  BEQ $010 : IF no bytes were read
  STA WCNT : THEN go read again
  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 ; path name pointer
.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
.BYTE  0C8 ; Here we didn't use a macro.
.BYTE  028 ; Begin SOS call block
.WORD  COLIST ; Open the console.
LDA  CREF ; Pointer to parameter list
STA  REF ; Save the result ref_num
STA  COPLIST ; for READs and WRITEs.

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

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

```
HANDLER
LDA REF
STA HREF1
STA HREF2
SOS READ, HBLK1
SOS WRITE, HBLK2
LDA BUFFER
CMP #1B
BNE $010
SOS 0CC, HBLK3
SOS 065, HBLK4
$010 JSR ARMCTRLQ
RTS
```

Set up reference numbers
for console READ
and console WRITE
Read attention character
Write three BELs to .CONSOLE
IF last keystroke was ESCAPE
THEN CLOSE all files
and TERMINATE
ELSE re-arm attention event
and resume execution
PROC SCREENWRITER

; Screenwriter Program
; Sample Interpreter for SOS Reference Manual
; Don Reed and Thomas Root, 11 August 1982

; This program reads bytes from the keyboard, then writes
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.

Note on programming style: the style of this program is
deliberately inconsistent, to show several ways to code
SOS calls. They can be coded in line; they can be coded
as subroutines. They can be coded with or without a
macro, SOS. The macro itself can use the SOS call number,
or it can use the name, via an .EQUATE. In general,
data structures appear before the code using them: this
is recommended practice with the Apple III Pascal
Assembler.

; The source file for the Screenwriter program is replicated
as SCREENWRITText on the EmuSOS disk.

;******************************************************************
**PAGE**

*Macros, Equates, and Global Data Area*

The syntax for a SOS call using the macro below is:

```
SOS call_num, parameter_list pointer
```

The macro definition for a SOS call block using the above format is below:

```
.MACRO SOS
    ; Macro def for SOS call block
    BNE
    ; Begin SOS call block
    .BYTE x1
    ; call_num
    .WORD x2
    ; parameter_list pointer
    .ENDM
    ; end of macro definition

```

Here are .EQUates for call_nums:

```
READ   .EQU 0CA
WRITE  .EQU 0CB
D_CNTL .EQU 083
```

Here are more .EQUates:

```
FALSE  .EQU 00
TRUE   .EQU 80
```

BPREF  .EQU OFPEF ; Bank register

---

**PAGE**

*These variables are used for communication between the main program and the OPERCONS subroutine.*

```
CHANG  .BYTE   08   ; name length
       .ASCII "CONSOS"
       ; pathName of console
REF    .BYTE   00   ; Console ref_num
COMNUM .BYTE   00   ; Console dev_num
```

Here is the data buffer for the READ and WRITE calls in the main program. Only the first byte is written into; one or both are written out.

```
BUFFER .BYTE   00, 0A   ; data buffer with trailing LF
```

Here are required parameter lists for SOS calls in the main program:

```
ENTRY .BYTE 01   ; parameter for REF_FENCE
        .BYTE 00   ; fence = 0
```

```
RCLIST .BYTE 04   ; 4 parameters for READ
REF   .BYTE 00   ; ref num
        .WORD BUFFER
        ; data buffer pointer
        .WORD 0001
        ; request_count
        .WORD 0000
        ; transfer_count
```

```
WPLIST .BYTE 03   ; 3 parameters for WRITE
```
BEGIN .EQU *
JSR OPENCONS ; Open .CONSOLE
JSR GETDEVNUM ; Get dev num
JSR SETCONS ; Set cons
JSR ARMCTRLQ ; Arm attention event
SOS 60, PENTLIST ; Set event fence to 0;
here we coded "60" directly
LDA REF ; Set up ref num
STA RREF ; for reads
STA WREF ; and writes
SOS READ, COLIST ; Read in one byte:
here we used READ for OCA
BEQ S010 ; IF no bytes were read
S010 ; THEN go read again
LDA RCHAN ; Set up write count
STA WCHAN ; IF first byte in buffer is CR
BNE S010 ; THEN write out LF also
INC WCHAN ; Write out 1 or 2 bytes
SOS WRITE, WPLIST ; Repeat ad infinitum
PAGE - 6 SCREEN FILE:
OPENCONS ; Open the .CONSOLE file for reading
COPLIST .BYTE 03 ; Open for reading and writing
LDA C8 ; Here we didn't use a macro.
RTS
BREAK ; Begin SOS call block.
OPENCONS ; Open the console.
C0NUM .BYTE 00 ; Device number goes here
BREAK ; Call GET_DEV_NUM
PAGE - 5 SCREEN FILE:
PAGE - 7 SCREENFILE:

PAGE

SECTIONS: set the .CONSOLE file to suppress screen echo

SETLIST .BYTE 03 ; 3 required params for D_CONTROL

CONUM .BYTE 00 ; dev_num of .CONSOLE

CONLIST .BYTE 06 ; control_code = 06: screen echo

CONLIST .WORD CONLIST ; control_list pointer

CONLIST .BYTE FALSE ; Disable screen echo

SECTIONS
LDA CONSUM ; Set up device number
STA CONUM ; of .console

SOS D_CNTL, SETLIST

RTS

SECTIONS
LDA CONSUM ; Set up device number
STA CONUM ; of .console

SOS D_CNTL, SETLIST

RTS

SECTIONS
LDA CONSUM ; Set up device number
STA CONUM ; of .console

SOS D_CNTL, SETLIST

RTS

PAGE - 8 SCREENFILE:

PAGE

HANDLE: Attention event handler subroutine

This subroutine reads the attention character (CONTROL=0) from .CONSOLE, then beeps twice. If the previous character was ESCAPE, the program terminates.

A buffer separate from the main data buffer is used for reading the attention character, as otherwise the attention character would sometimes clobber the character in the data buffer before it could be written.

The buffer BELLS contains three BEL characters, separated by a byte of SYNC characters. When written to the console, these cause a total delay of about 150 ms.

SECTIONS

BELLS .EQU * ; Buffer with BELs and delays

BELLS .BYTE 07 ; BEL

BELLS .BYTE 16, 16, 16, 16, 16, 16, 16, 16 ; SYNCs

BELLS .BYTE 07 ; BEL

BELLS .BYTE 16, 16, 16, 16, 16, 16, 16, 16 ; SYNCs

BELLS .BYTE 07 ; BEL

BELLEN .EQU **BELLS ; Calculate buffer length

BELLEN .BYTE 04 ; 4 required parameters for READ

HELLEN .BYTE 00 ; ref_num

HELEN .WORD 0001 ; data buffer pointer

HELEN .WORD 0002 ; request_count

HELEN .WORD 0003 ; transfer_count

HELLEN .BYTE 00 ; Buffer for attention character

HELLEN .BYTE 03 ; 3 required parameters for WRITE

HELLEN .BYTE 00 ; ref_num

HELLEN .WORD 0000 ; data buffer pointer

HELLEN .WORD BELL ; request_count

HELLEN .BYTE 01 ; 1 required parameter for CLOSE

HELLEN .BYTE 00 ; ref_num = 0: CLOSE all files

HELLEN .BYTE 00 ; 0 required params for TERMINATE

SECTIONS
LDA DEFF ; Set up bank number
STA BREG ; of event handler

LDA CONSUM ; Set up device number
STA D_CNTL ; for control request

SOS D_CNTL, D_CONTROL call macro

RTS
PAGE - 9 SCRNWR FILE:

PAGE

900C AD OC00
900D B800
900E C400
900F 2000
9010 9490
9011 0100
9012 0100
9013 0100
9014 0100
9015 0100
9016 0100
9017 0100
9018 0100
9019 0100
901A 0100
901B 0100
901C 0100
901D 0100
901E 0100
901F 0100
9020 0100
9021 0100
9022 0100
9023 0100
9024 0100
9025 0100
9026 0100
9027 0100
9028 0100
9029 0100
902A 0100
902B 0100
902C 0100
902D 0100
902E 0100
902F 0100
9030 0100
9031 0100
9032 0100
9033 0100
9034 0100
9035 0100
9036 0100
9037 0100
9038 0100
9039 0100
903A 0100
903B 0100
903C 0100
903D 0100
903E 0100
903F 0100
9040 0100
9041 0100
9042 0100
9043 0100
9044 0100
9045 0100
9046 0100
9047 0100
9048 0100
9049 0100
904A 0100
904B 0100
904C 0100
904D 0100
904E 0100
904F 0100
9050 0100
9051 0100
9052 0100
9053 0100
9054 0100
9055 0100
9056 0100
9057 0100
9058 0100
9059 0100
905A 0100
905B 0100
905C 0100
905D 0100
905E 0100
905F 0100
9060 0100
9061 0100
9062 0100
9063 0100
9064 0100
9065 0100
9066 0100
9067 0100
9068 0100
9069 0100
906A 0100
906B 0100
906C 0100
906D 0100
906E 0100
906F 0100
9070 0100
9071 0100
9072 0100
9073 0100
9074 0100
9075 0100
9076 0100
9077 0100
9078 0100
9079 0100
907A 0100
907B 0100
907C 0100
907D 0100
907E 0100
907F 0100
9080 0100
9081 0100
9082 0100
9083 0100
9084 0100
9085 0100
9086 0100
9087 0100
9088 0100
9089 0100
908A 0100
908B 0100
908C 0100
908D 0100
908E 0100
908F 0100
9090 0100
9091 0100
9092 0100
9093 0100
9094 0100
9095 0100
9096 0100
9097 0100
9098 0100
9099 0100
909A 0100
909B 0100
909C 0100
909D 0100
909E 0100
909F 0100
90A0 0100
90A1 0100
90A2 0100
90A3 0100
90A4 0100
90A5 0100
90A6 0100
90A7 0100
90A8 0100
90A9 0100
90AA 0100
90AB 0100
90AC 0100
90AD 0100
90AE 0100
90AF 0100
90B0 0100
90B1 0100
90B2 0100
90B3 0100
90B4 0100
90B5 0100
90B6 0100
90B7 0100
90B8 0100
90B9 0100
90BA 0100
90BB 0100
90BC 0100
90BD 0100
90BE 0100
90BF 0100
90C0 0100
90C1 0100
90C2 0100
90C3 0100
90C4 0100
90C5 0100
90C6 0100
90C7 0100
90C8 0100
90C9 0100
90CA 0100
90CB 0100
90CC 0100
90CD 0100
90CE 0100
90CF 0100
90D0 0100
90D1 0100
90D2 0100
90D3 0100
90D4 0100
90D5 0100
90D6 0100
90D7 0100
90D8 0100
90D9 0100
90DA 0100
90DB 0100
90DC 0100
90DD 0100
90DE 0100
90DF 0100
90E0 0100
90E1 0100
90E2 0100
90E3 0100
90E4 0100
90E5 0100
90E6 0100
90E7 0100
90E8 0100
90E9 0100
90EA 0100
90EB 0100
90EC 0100
90ED 0100
90EE 0100
90EF 0100
90F0 0100
90F1 0100
90F2 0100
90F3 0100
90F4 0100
90F5 0100
90F6 0100
90F7 0100
90F8 0100
90F9 0100
90FA 0100
90FB 0100
90FC 0100
90FD 0100
90FE 0100
90FF 0100

PAGE - 10 SCRNWR FILE: SYM TABLE DUMP

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

AB LB UD MC
ARMCTRLOQ LB 9094| BANK LB 9092| BEGIN LB 9020| BELLEN AB 0015|
CLIST LB 908E| CLNAME LB 9003| CNJMB LB 9079| CODELEN LB 00F0|
COPLIST LB 9030| CREFB LB 9059| DCLIST LB 9089| SCNTL AB 0081|
GOLIST LB 9069| GDNUM LB 906C| GETDNMB LB 9046| MANAGER LB 900C|
HLK4 LB 90CA| HBF01 LB 90C2| HREFLI LB 9038| HREFF2 LB 90C4|
READ AB 00CA| REF LB 9030| RREF LB 9013| SCRNWR PB ----|
START AB 9000| TRUE AB 008D| WCTU LB 901E| WLIST LB 901A|

BELL2 LB 90A1| BREG AB 00E| BUFFER LB 900E|
COLIST LB 903A| CMOPLIST LB 907B| CONSMB LB 900D|
DNUM LB 908A| FALSE AB 0000| FENLIST LB 9010|
HLK1 LB 908A| HLK2 LB 90C3| HLK3 LB 9049|
OPENCONS LB 9054| RCLIST LB 9012| RCNT LB 9018|
SETCONS LB 9076| SETLIST LB 9078| SOS MC ----|
WREFL LB 9018| WRITE AB 00C8|

CODELEN .EQU *=START ; Calculate number of bytes in
.EVRD ; program
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](image)

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 runtime, 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.

Making SOS Calls

148 8.1 Types of SOS Calls
148 8.2 Form of a SOS Call
148 8.2.1 The Call Block
150 8.2.2 The Required Parameter List
152 8.2.3 The Optional Parameter List
154 8.3 Pointer Address Extension
155 8.3.1 Direct Pointers
155 8.3.1.1 Direct Pointers to S-Bank Locations
156 8.3.1.2 Direct Pointers to Current Bank Locations
156 8.3.2 Indirect Pointers
157 8.3.2.1 Indirect Pointers with an X-Byte of $00
158 8.3.2.2 Indirect Pointers with an X-Byte Between $80 and $8F
159 8.4 Name Parameters
160 8.5 SOS Call Error Reporting
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:

![Figure 8-1. SOS Call Block](image)

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 here 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   OC8; 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   REFACC; PTR
$5226 04    OPLN   .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:
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 non-zero, 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](image)

#### 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.

<table>
<thead>
<tr>
<th>Original Pointer</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n00n$</td>
<td>$A000$ to $B7FF$</td>
</tr>
<tr>
<td>$00:n00n$</td>
<td>$00:A000$ to $00:B7FF$</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Original Pointer (bank &lt;&gt; 0)</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nnnn $2000 to $21FF</td>
<td>$xx:nnnn $8b-1:8000 to $8b-1:81FF</td>
</tr>
<tr>
<td>$nnnn $2200 to $2FFF</td>
<td>$xx:nnnn $8b:0200 to $8b:7FFFF</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Original Pointer (bank = 0)</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0:nnnn $0:2000 to $0:9FFF</td>
<td>$8F:nnnn $8F:2000 to $8F:9FFFF</td>
</tr>
</tbody>
</table>

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:

![Diagram of 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.

<table>
<thead>
<tr>
<th>Original Pointer</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00:nnnn $00:$2000 to $00:$21FF</td>
<td>$xx:nnnn $8b-1:8000 to $8b-1:81FF</td>
</tr>
<tr>
<td>$00:nnnn $00:$2200 to $00:$9FFF</td>
<td>$xx:nnnn $8b:0200 to $8b:7FFFF</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Original Pointer</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$80:nnnn $80:0000 to $80:01FF</td>
<td>Invalid</td>
</tr>
<tr>
<td>$8x:nnnn $8x:0000 to $8x:00FF</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Original Pointer</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8x:nnnn $8b:0100 to $8b:01FF</td>
<td>$8x:nnnn $8b-1:8100 to $8b-1:81FF</td>
</tr>
<tr>
<td>$8x:nnnn $8b:FF00 to $8b:FFFF</td>
<td>$8x:nnnn $8b+1:7F00 to $8b+1:7FFF</td>
</tr>
</tbody>
</table>

The pointers below are unchanged:

<table>
<thead>
<tr>
<th>Original Pointer</th>
<th>Extended Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8x:nnnn $8b:0200 to $8b:FEFF</td>
<td>$8x:nnnn $8b:0200 to $8b:FEFF</td>
</tr>
<tr>
<td>$8F:nnnn $8F:2000 to $8F:B7FF</td>
<td>$8F:nnnn $8b:2000 to $8b:B7FF</td>
</tr>
</tbody>
</table>

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](image-url)
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 [ ].

A
absolute
  code 120
  mode 29
  modules 143
  or relocatable format 143
access  63, 68, 81, 84, 88, 90, [11],
  [18]
data  10, 27, 29–32
path(s)  52
  information  64–66
  maximum number of  53
  multiple  52
  techniques  27–38
accessing
  a logical device  41
  zero page and stack, warning  17
ACCSERR  [55]
accumulator  110
ADC  31
address(es)  15
  bank-switched  10, 12, 30, 32
  bus  10
  conversion  25, 32–35
  example  122
  current-bank  12, 38
  extended  13, 38
    notation  15
  extension, pointer  154–159
  invalid  13
  limit  122
    notation
      bank-switched  15
      extended  15
      segment  23–27
  of blocks  96, 97
  of event handler  108
  relocatable  [138]
  risky  15
  risky regions  32
  S-bank  12, 38
  segment  24, 38
    notation, S-bank  25
  three-byte  13
  two-byte  12
addressing
  bank-switched memory  10–13,
  30–31
  enhanced indirect  10, 13–16,
  31–32
  indirect-X  13
  indirect-Y  13
modes 10-16
enhanced 8
module 27-29
normal indirect 14
restrictions 15
subroutine 27-29
ALCERR [128]
algorithms 32
reading a directory file 91-92
incrementing a pointer 36-37
sample 27
allocate memory 25
allocation 7, 23
of a segment of memory 121
scheme, block 95
analog inputs 113
AND 31
Apple III, overview of 3-8
Apple III Pascal Assembler 145,
[132], [134]
Apple III Processor xvii
arming events 108, 125
ASCII [139]
ASCII equivalents [117]
Assembler, Apple Pascal 145,
[132], [134]
assembly language 5
code file(s) [131-139]
data formats for relocatable 146
module 19, 118, 143-146
linking 145
loading 145
procedure [136]
attribute tables [136], [137]
programming xvii
asynchronous operations 5
do device drivers 104
attribute table [136], [138]
assembly-language procedure [136]
format of [137]
procedure [136]
.AUDIO [111]
.audio [111]
aux_type 64, 88, [5], [14], [19]
B
B-field 14
backup bit 90, [12], [18]
Backup III 90, [13]
BADGBKPG [88]
BADBRK [127]
BADBUFNUM [128]
BADBUFSIZ [128]
BADCHMODE [88]
BADCTL [71]
BADCTLPARAM [71]
BADCZWPAGE 161
BADDDNUM [71]
BADINT [127]
BADJMODE [104]
BADLSCTCN [56]
BADOP [72]
BADPATH [53]
BADPGCNT [88]
BADREFNUM [54]
BADRECODE [71]
BADSCBND [161]
BADSCNUM [160]
BADSCPCTN [161]
BADSEGNUM [88]
BADSRCHMODE [88]
BADSYSBUF [56]
BADSYSCALL [127]
BADXBYTE 161
BCBERR [128]
bank $8 16
.current 12
.highest 11
.switchable 15
number 15
.pair 13, 14
.highest 15
part of segment address 25
.register 11, 19, 28
restoring contents of 31
.switchable 11
bank-pair field 14
bank-switched address 10, 12,
30, 32
as immediate form 32
notation 15
bank-switched memory
.addressing 10-13, 30-31
bank-switched notation 23
bank-switching 27, 28, 30
for data access 30
for module execution 30
restrictions 28
base 23, 122, [43], [48], [75], [78],
[83]
BASE 122
.base-relative relocation table
[138]
BASIC 118, 143
and Pascal modules 145
interpreter 145
program 145
BCS [139]
bibliography [141]
.bit
backup 90, [12], [18]
destroy-enable [12], [18]
enhanced-addressing 14
.map 54
.readable [12], [18]
.rename-enable [12], [18]
.write-enable [12], [18]
.bit_map_pointer 82
BITMAPADR [56]
.BLOCK [139]
block(s) 77
.addresses of 96, 97
allocation
for sparse files 98
.scheme 95
altering configuration 46
call 148-149, [x]
configuration 43
altering 46
data 93, 96
device 8, 40, 76
.logical 53
.status request $00 [60]
device information (DIB) 43
DIB configuration 43
file 50-56, 62
.control 64
.structure of 50-51
.index 93, 94
.key 77, 82, 93, 97
.logical 77
.master index 94, 96, 97
.maximum index 94
.on a volume 77
.SOS call [103]
.subindex 94, 96
.total 45, 82
blocks_used 63, 87, [19]
BNE [139]
bootstrap
.errors [128]
.loader 77, 93
BRK 149
.instruction 8
BTSERR [55]
.buffer
data 50, [117]
.editing [117]
I/O 50
.space, for drivers 21
.string [117], [118]
BUFTBLFULL [56]

**Call(s)**

- **block**: 148-149, [x]
- **SOS**: [103]
- **choosing**: [114]
- **coding TERMINATE**: 131
- **D_CONTROL**: 128
- **device**: 46-47, [58-71]
- **errors**: [71-72]
- **management**: 5
- **device**: 160, [71-72], [125]
- **file**: 160, [53-56], [125-126]
- **memory**: 160, [88]
- **utility**: 160, [104], [126]
- **file**: 69-73, [2-53]
- **errors**: [53-56]
- **management**: 5
- **FIND_SEG**: 30
- **form of the SOS**: 160
- **memory**: 25-27, [74-87]
- **errors**: [88]
- **management**: 5
- **OPEN**: 128
- **REQUEST_SEG**: 30
- **SOS**: 8
  - **error reporting**: 160
  - **form of a**: 148-154
  - **types of**: 148
  - **utility**: 90-103
  - **errors**: [104]
  - **management**: 5

**Call num**: 149, [xi]

- **capacity of a file, maximum**: 94
- **carry**: 15
- **CFCBFULL**: 53
- **changing device**: name 46
  - **subtype**: 46
  - **type**: 46
- **changing slot number**: 46

**Change_mode**: 81

- **CHANGE_SEG**: 26, [81-82]
- **character**: device 8, 40
  - **control code**: $01 [64]
  - **control code**: $02 [64]
  - **status request**: $01 [60]
  - **status request**: $02 [61]
  - **file(s)**: 50-56, 57
    - **structure of**: 50-51
    - **line-termination**: 67
    - **newline**: 67
    - **null (ASCII $00)**: 97
    - **streams**: 40
    - **termination**: 67
  - **circumvention of programming restrictions**: 3
  - **clock**: 112-113, [95], [97], [98]
  - **rate**: 19
  - **system**: 112
  - **CLOSE**: 66, 68, 72, 90, [39-40]
  - **closed files**: 52-53
  - **closing files before TERMINATE**: 103
  - **CMP**: 31
  - **code**: file(s) 145
    - **data formats of relocatable assembly-language**: 146
    - **organization**: [132]
    - **assembly-language**: [131-139]
    - **code part of**: [135]
    - **fragments, examples**: xiv

**Interpreter, executing**

- **part of a code file**: 119, 121, [132], [135]
- **segments, executing**: 27
- **sharing**: 44
- **procedure**: [136]

**Code length**: 120

- **CODEADDR**: [134]
- **CODELEN**: [134]
- **colon**: 15
- **command interpreter**: [103]
- **common code**: 44
- **common file structure**: 3
- **common foundation for software**: 3
- **defined**: 2
- **communicating with the device**: 42
- **comparing two pointers**: 37-38
- **compatibility with future versions**: 18
- **conditions for enhanced indirect addressing**: 31
- **configuration block**: 43
  - **alter**: 46
  - **DIB**: 43
- **conflicts between interrupts**: 104
  - **with zero page**: 16
- **.CONSOLE**: 66, 105, 108, 125, [109]
- **console**: 40
- **constant, relocation**: [138]
- **control**: block, file 64
  - **flow of**: 27
  - **transfer**: 28
  - **CONTROL-C**: [117]
  - **CONTROL-RESET**: [117]

**Control code**: 63

- **$01**, character device [64]
- **$02**, character device [64]

**Control list**: 63

- **conversions**: 32
- **copy-protection**: [103]
- **copying sparse files**: 98
- **CPTERR**: 55
- **CPU**: 104
- **CREATE**: 68, 69, 90, 98, [3-6]
- **creating interpreter files**: 143
- **creation date and time**: 64, 81, 84, 88, 89-90
- **field**: 89-90
- **current**: bank 12
  - **direct pointers to**: 156
  - **directory**: 62
  - **position marker**: 51
- **current-bank**: address 12, 38
  - **form**: 13
  - **cylinders**: 77

**D**

- **D1**: [109]
- **D2**: [109]
- **D3**: [109]
- **D4**: [109]
- **.D_CONTROL**: 45, 47, 108, 125,
  - **128, [63-64], [118]
- **.D_INFO**: 43, 45, 47, [67-71]
- **.D_STATUS**: 45, 46, [59-61], [118]

**Data**

- **access**: 10, 27, 29-32
  - **bank-switching for**: 30
  - **and buffer storage**: 19
  - **block**: 93, 95, 96
  - **buffer**: 50, [117]
  - **editing**: [117]
  - **formats of relocatable assembly-language code files**: 146
  - **in free memory**: 30
data_block 99

data_buffer [35], [37]
date and time
creation 64, 81, 84, 88, 89-90
format 90
last mod 64, 88, 89-90, [14], [19]
decimal numbers xix
decimal point xix
DESTROY 68, 69, [7-8]
destroy-enable bit [12], [18]
detecting an event 105
device name 43, 60, [23], [65], [67]
device_num 43, [59], [63], [65], [67]
device_type 44, 45, [68]
device(s) 8, 40-42
adding a 46
block 8, 40
call(s) 46-47
errors 160, [125]
changing name of 46
character 8, 40
communicating with the 42
control information 45
correspondence
logical/physical 54
special cases of 54
defined as logical device 54
driver(s) 5, 41, 77, 104, 107, 108, 125
asynchronous operation of 104
environment 20-21
errors, individual 160
graphics 16
standard [109-111]
memory placement 21
independence 7, 67
information 43-44
block (DIB) 43
input 40
logical 40
block 53
management calls 5
multiple logical 54
name(s) 41-42, 44, 50, 55, 60
illegical 42
legal 42
syntax 42
number 44
operations on 45-46
output 40
peripheral 8, 104
physical 40
random-access 7
removing a 46
requests 50
sequential-access 7
status information 45
subtype 44
changing 46
type 44
changing 46
device-independent I/O 67
DIB
configuration block 43
header 43
dictionary 8
current 62
entry 62
procedure [135], [136]
error (DIRERR) 55
file 57-58
format(s) 78-92
header 78
storage formats 76
segment [132], [134]
volume 54, 57, 78
digit(s) 42, 56
hexadecimal 12
direct pointer 154, 155
to S-bank locations 155
directory file, reading a 91-92

DIRERR [55]
DIRFULL [55]
disarming events 108
Disk III driver 41
disk drives 40
disk, flexible 42, 77, 93
DISKSW [72]
dispatching routine 28
displacement [43], [48]
Display/Edit function [117]
DNFERR [71]
dollar signs xviii, xix
driver
device See device driver
module 41
placement of 44
DRIVER FILE NOT FOUND [129]
DRIVER FILE TOO LARGE [129]
DUPERR [54]
DUPVOL [56]

E
E-bit 14
editing data buffer [117]
EMPTY DRIVER FILE [129]
empty file 65
end-of-file marker See EOF
enhanced
addressing bit 14
addressing modes 8
indirect addressing 10, 13-16,
27, 30, 31-32
conditions for 31
ENTER IC [138]
entries_per_block 82, 85, 92
text entry (entries) 86
active 86
directory 62
FCB 53, 62
format compatibility 91
inactive 86
points 145
storage formats of 76

entry_length 81, 84, 92
environment
attributes 19
execution 16-22
interpreter 18-19
SOS device driver 20-21
SOS Kernel 19-20
summary 22
EOF 51, 53, 63, 64-65, 68, 87, 89,
94, 95, 96, 97, 98, [5], [19], [49]
limit 94
movement of automatic 65
manual 65-66
updating 65
EOFERR [55]
EOR 31
eor(s) [124]
bootstrap [128]
device call [125]
file call [125]
messages [123-130]
numbers range 160
reporting, SOS call 160
SOS
fatal [124], [126]
general [124]
non-fatal [124]
utility call [126]
event(s) 5, 104-115
any-key 105
arming, example 129
arming and response 105, 108,
125
attention 105
detecting an 105
disarming 108
existing 108
fence 106, 109-110
F

FCB 52
  entry 53, 62
  FCBERR [128]
  FCBFULL [54]
  fence [91], [93]
  fence, event 106, [91], [93]

field(s)
  formats 89-92
  bank-pair 14
  pointer 79
  FIFO (first-in, first-out) 109
  FILBUSY [55]
  file(s) 7-8, 52
    assembly-language code [133]
    block 50-56, 62
    allocation for sparse 98
    call(s) 69-73, [2]
    errors 160, [125]
    character 50-56, 57
    close 52-53
    closing before TERMINATE [103]
    code 145
    part of a code [135]
    control block 64
    copying sparse 98
    creating interpreter 143
    data formats of relocatable
      assembly-language code 146
    defined 50
    directory 57-58
      format 78-92
      relocatable 120
      or absolute 143
    reading 91-92
    empty 65
    entry (entries) 78, 85-89
      inactive 86, 89
      sapling 89
      seedling 89
      subdirectory 89
      tree 89
    information 62-64
    input/output 67
    interpreter, creating an 143
    level, system 66
    management calls 5
    maximum capacity of a 94
    name(s) 58-59, 60
    illegal 59
    legal 59
    syntax 59
    open 52-53, 63
    operations on 68
    organization 76-99
      code [132]
    sapling 93, 95
    seedling 93, 95
    SOS 56-62
    sparse 63, 94, 97-98
    standard 57-58
    locating a byte in 98-99
    storage formats of 92-99
    structure
      common 16
      hierarchical 8
        of a block 50-51
        of a character 50-51
        of a sapling 96
        of a seedling 95
        of a tree 96
    subdirectory 57, 78
    system
      relationship to device
        system 57
      root of 59
    SOS 55-62
    tree 61
    top-level 57
    tree 94, 96-97
    growing a 92-95
    type 68
    volume directory 77
  file_count 82, 85
  file_name 60, 63, 80, 83, 87
  file_type 64, 87, 91, [4], [13], [18]
  FIND_SEG 26, 30, 121, 122,
    [77-79]
  flexible disk 42, 77, 93, [109]
GET_LEVEL 66, 69, 73, [53]
GET_MARK 66, 68, 72, [45]
GET_PREFIX 70, [27]
GET_SEG_INFO 26, [83–84]
GET_SEG_NUM 26, [85]
GET_TIME 90, 112, 115, [97–98]
.GRAFIX 110
graphics 16, [110]
area 16
device drivers 16
growing a tree file 92

H

hand symbol xv
handler
  event 5, 125
  interrupt 5
handling an event 106, 107
hardware 8, 10
  independence 2
  interrupt 105
header(s) 43, 119
directory 78, 79–82
  subdirectory 82–85, 89
  volume directory 79, 80, 89
header_pointer 89
heads 77
hexadecimal (hex) xviii
digit 12
  numbers xviii
hierarchical file structure 8
hierarchical tree structure 56, 76
high-order nibble [117]
highest bank 11
  pair 15
highest switchable bank 15, 18
highest-numbered bank 23
housekeeping functions 3

I

I/O
  block 51
  buffer 50, 127
  character 51
  device-independent 67
  ERROR [129]
  implementation versus interface 76
  warning 99
INCOMPATIBLE INTERPRETER [129]
  increment loop 124
  one-bank example of 124
  incrementing a pointer 36–37
index block(s) 93, 94, 95
  master 94
  maximum 94
  sub- 94, 96
index_block 99
indexed mode, zero-page 29
indexing 15
  addresses 15
  indirect
    addressing 10
      enhanced 10, 13–16, 27, 30,
      31–32
      normal 14
    operation, normal 31
    pointer(s) 154, 156, 157
      with an X-byte between $80
      and $8F 158
      with an X-byte of $00 157
  indirect-X addressing 13
  indirect-Y addressing 13
input(s)
  analog 113
  device 40
  parameters [116]
  input/output, file 67
  interface versus implementation 76
  warning 99
interface, SOS 76
intermediate form, bank-switched
  addresses as 32
INTERP [139]
interpreter(s) 5, 16, 118–125, 145,
  [132]
  and modules 144
  BASIC 145
code 10
  executing 10
  command [103]
environment 18–19
files, creating 143
language 118
maximun size of 18
memory
  placement 18
  requirements of 146
Pascal 145
return to 29
sample(s) 125–142
  listing, complete 131–142
  stand-alone 118
structure of 119–121
table within 29, 30
INTERPRETER FILE NOT
  FOUND [129]
interpreter-relative relocation
table [139]
interpreter's
  stack 19, 110
  zero page 19
interrupt(s) 5, 104–115
  conflicts between 104
  handler 5, 22, 104
  IRQ 22
  and NMI 20
  ranked in priority 104
  summary of 112
invalid
  address 13
  jumps 29
  regions 15, 16
INVALID DRIVER FILE [129]
io_buffer [31]
I/OERR [72]
IRQ interrupts 20, 22
IS_NEWLINE 67, 68, [33]

J

JMP 27–28, [139]
joy_mode [99]
joy_status [100]
joystick [99]
JSn-B [100]
JSn-Sw [100]
JSn-X [100]
JSn-Y [100]
JSR 27–28
jumps 29
  inside module 29
  invalid 29
  valid 29

K

KERNEL FILE NOT FOUND
  [130]
key_pointer 87, 92
keyboard 40

L

labels xix, 120
  local 127
language interpreter 118
largest possible file 94
last_mod date and time 64, 88,
  89–90, [14], [19]
  field 89–90
LDA 31, [139]
leaving ExerSOS [119]
legal device names 42
legal file names 59
length 152, 3, 11, 17, 25, [30, 67, 116]
letters 42, 56
level 66, 51, 53
level, system file 66
limit 23, 122, [75, 78, 83]
LIMIT 122
link—termination character 67
linked list 78
linker information 133
linking
  assembly-language modules 145
dynamic loading during 145
lists
  required parameter 129,
  150-152
  optional parameter 152-154
loading
dynamic, during linking 145
assembly-language modules 145
routine 134
loading_address 120, 121
locating a byte in a standard
  file 98
logical
  block 77
device 53
byte position 98
device(s) 40
accessing a 41
multiple 54
structures 76
logical/physical device
  correspondence 54
loop, increment 124
low—order nibble 117
LVLERR 56
M
  machine
    abstract 2
    storing the state of the 110
  macro, S0S 126
  MakeInterp [121-122]
  management calls
    device 5
    file 5
    memory 5
    utility 5
  manager, resource 2-3
  manual movement of EOF and
    mark 66
  manuf_id 45, 70
  manufacturer 45
  mark 51, 53, 64-65, 68, 97, 98,
    [45]
    movement of, automatic 65
    movement of, manual 65-66
  marker, current position 51
  master index block 94, 96, 97
  maximum
    number of access paths 53
    capacity of a file 94
    number of index blocks 94
    size of an interpreter 18
  MCTOVFL 127
  media, removable 53, 54
  medium 42, 53
  MEM2SML 127
  memory 6-7, 23
    access techniques 27-38
    addressing, bank-switched
      10-13
    allocation 25, 121
    bookkeeper 7
    call(s) 25-27
    errors 160
    conflict 121
    avoiding 121
  management 7
    calls 5
    obtaining free 121-124
    placement
      interpreter 18
      module 144
    S0S device driver 21
    S0S Kernel 20
    S-bank 19
    segment 7
    size, maximum 6, 10
    unswitched 28
    messages, error [123-130]
    min_version 81, 84, 88
  mode(s)
    absolute addressing 29
    addressing 10-16
    enhanced addressing 8
    newline information 67
    zero-page addressing 29
    indexed 29
    modification date and time 68
  module(s) 5, [132]
    absolute 143
    addressing 27-29
    assembly-language 19, 118,
      143-146
    linking 145
    BASIC invokable 145
    creating 146
    driver 41
    execution, bank-switching
      for 30
    formats 146
    loader [134]
    Pascal 145
    program or data access by 145
    relocatable 143, 146, [132]
  multiple
    access paths 52
    logical devices 54
    volumes 54
N
  name(s) 60, 68
  device 60
  file 58-59, 60
  local 59
  parameter 159-160
  volume 55-56, 60
  name_length 80, 83, 87
  naming conventions 76
  newline
    character 67
    mode 67
  newline_char 67, 68, [33-34]
  newline—mode information 67
  nibble
    high—order 117
    low—order [117]
  NMI 114
  interrupts 20
  NMIHAND [127]
  NORESC [72]
  notation xviii
    and symbols xviii
    bank—switched address 15,
      23
    extended address 15
    numeric xviii
    segment address 23-27
  NOTBLKDEV [56]
  NOTOPEN [72]
  NOTSOS [55]
  NOWRITE [72]
  null characters (ASCII $00) 97
  number(s)
    decimal xix
    device 44
    hexadecimal xiv
    reference 52
    slot 44
    changing 46
unit 44
version 45
numeric notation xviii, xix

O
OPEN 52, 53, 68, 69, 71, [29-32]
call, example 128
operating system 2-3
defined 2
operations
asynchronous 5
normal indirect 31
on devices 45-46
on files 68
sequential read and write 50
Opt_header 120
Opt_header_length 120
Option_list 152, [3], [11], [17], [29], [67]
onoptional parameter list 152-154, [x]
ORA 31
order of event queue 109
organization, code file 132
OUTOFMEM 56
output device 40
overview of the Apple III 3-8
OVRERR [54]

P
page(s) 23, [31], [78], [81], [83]
part of segement address 25
parameter(s)
format of a name 159
input [116]
list, optional 152-154, [x]
required 129, 150-152, [x]
nname 159-160
passing 145
pointer 145
parent_entry_length 85
parent_entry_number 85
parent_pointer 85
parm_count [xi]
parm_list 149
Pascal 118, 143, [132]
and BASIC modules 145
assembler 145, [134]
interrepreter 145
prefix 62
program 145
versus SOS prefixes 62
path(s)
access 52
information 64-66
multiple 52
maximum number of 56
pathname [3], [7], [9], [11], [17], [25], [29]
pathname 52, 59-61
full 62
partial 61-62
syntax 60
valid 61
PERFORM 145
period 42, 56
peripheral device 8, 104
physical device 40, 54
correspondence with logical devices 54
PNFERR [54]
point, decimal xix
pointer(s) 31, 69, 152
address extension 154-159
byte order of 79
comparing two 37
direct 154, 155-156
to current 156
to X-bank 155
extended 123
fields 79
incrementing a 36-37
indirect 154, 156-159
manipulation 36-38
parameters 145
preceding-block 78
self-relative [136], [138]
three-byte 98
POSNERR [55]
prefix(es) 60, 61-62
Pascal 62
restrictions on 62
SOS 62
versus Pascal 62
.PRINTER [111]
printers 40
priority of zero 108
priority-queue scheme 108
.PRIVATE [138]
.PROC [136], [139]
procedure(s) [135], [136]
attribute table [136]
code [136]
dictionary [135]
entries [136]
PROCEDURE NUMBER [138]
procedure-relative relocation table [139]
processing an event 106
Processor, Apple III xvii
Product Support Department 45
program
execution, restrictions on 14
executing from 66
programming
assembly-language xiii
restrictions, circumvention of SOS 3
psuedo-opcode(s) [136]
.FUNC [136]
.PRIVATE [138]
.PROC [136]
.PUBLIC [138]

Q
queuing an event 106

R
range, X-byte 15
READ 67, 68, 71, [35-36]
read and write operations, sequential 50
read-enable bit 12, [18]
reading a directory file 91
ref_num 52, 64, 67, [2], [29], [33], [35], [37], [39], [49], [41], [43], [45], [47]
references, relocation [138]
regions
invalid 15, 16
risky 15, 16
release memory 25
RELEASE_SEG 27, [87]
relocation 146
constant [138]
information 145
references [138]
table(s) [138]
base-relative [138]
intererpreter-relative [139]
procedure-relative [139]
segment-relative [139]
RELOCSEG NUMBER [138]
RENAME 69, 90, [9-10]
req_access [30]
request_count [35], [37]
REQUEST_SEG 25, 121, [75-76]
call 30
required parameter list 129, 150-152, [x]
example 129
resource manager 2-3
defined 2
resources 112-114
restrictions
addressing 15
bank-switching 28
on program execution 14
result 69, 151
return to interpreter 29
risky regions 15, 16
addresses 32
avoiding 37
warning 32
ROM ERROR: PLEASE NOTIFY YOUR DEALER 130
root of file system 59
.RS232 111

S
S–bank 11, 23, 28
address 12, 38
in segment notation 25
locations, direct pointers to 155
memory 19
sample programs, examples xiv
sapling file 93, 95
entry 89
structure of a 96
SBC 31
scheme, priority–queue 108
SCP 43
screen 40
search mode [77]
sectors 77
seedling file 93, 95
entry 89
structure of a 95
seg_address [85]
seg_id [75], [78], [83]
seg_num [76], [78], [81], [83], [85], [87]
segment 23–24
address 24, 38
bank part of 25
conversion 33–35
notation 23–27
page part of 25
allocated from free memory 29
dictionary 132, [134]
memory 7
of memory, allocating a 121
to bank-switched address
conversion 33
to extended address conversion 33
segment–relative relocation
 table 139
SEGNOTFND 88
SEGRODN 88
SEGTBLFULL 88
sequential
 access 51
devices 7
read and write operations 50
serial printer (.PRINTER) 111
SET_EOF 66, 68, 72–73, [47–48]
SET_FENCE 107, 110, 114, [91]
SET_FILE_INFO 63, 68, 70, 88,
90, 152, [11–16]
SET_LEVEL 66, 73, [51]
SET_MARK 66, 68, 72, [43–44]
SET_PREF 70, [25–26]
SET_TIME 90, 112, 115, [95–96]
slash (/) 56, 60
slot number 44
change 46
of zero 44
slot_num 44, [68]
software, common foundation
for 2, 3
Sophisticated Operating System
See SOS
SOS xvi, 3, 5–6, 16, 104
 1.1 xix, 106
 1.2 18, 77, 81, 82, 84, 85, 88, 92,
 93, 95, 99, 105
 1.3 xix, 106
bank 11
call(s) 8
block [103]
form error 160
reporting 160–161
form of 148–154, 160
types of 148
device
driver
environment 20–21
memory placement 21
system 43
disk request 55
errors
fatal [124], [126]
general [124]
non–fatal [124]
file system 56, 58
future versions of 91, 92, 93
implementation 76
interface 76
Kernel 19
environment 19–20
memory placement 20
macro 126
for SOS call block 126
prefix(es) 62
versus Pascal 62
programming restrictions,
circumvention of 3
specifications 106–111
support for 76
system 104
versions xix, [106]
SOS.DRIVER 6, 41
SOS.INTERP 118
SOS.KERNEL 6, 41
sparse file(s) 63, 94, 97–98
block allocation for 98
copying 98
special symbols xv
STA 31
stack 17, 20
interpreter's 145
overflow 127
pages 19
stand-alone interpreter 118
standard device drivers 109–111
standard file(s) 57–58
locating a byte in 98–99
storage formats of 92–99
state of the machine, storing
the 110
status request
$00, block device 60
$01, character device 60
$02, character device 61
status_code [59]
status_list [60]
STK0VFL [127]
stop symbol xv
storage formats
directory headers 76
entries 76
of standard files 92–99
storage_type 64, 80, 83, 87, 89,
92, 95, 96, 97, [5], [19]
string buffer [117], [118]
structure(s)
hierarchical tree 56, 76
logical 76
of a sapling file 96
of a seedling file 95
of a tree file 96
of an interpreter 119–121
of block files 50–51
of character files 50–51
sub_type 44, 45, [69]
subdirectory (subdirectories) 8
file(s) 57, 78
entry 89
header 82, 83, 89
subindex block 94, 96
subroutine addressing 27–29
summary
of address storage 38
of interrupts and events 112
switchable bank 11
highest 15, 18
symbol(s)
eye xix
hand xix
stop xix
v1.2 xix
syntax
device name 42
file name 59
pathname 60
volume name 56
System Configuration Program (SCP) 41, 46
system
clock 112
configuration time 104
file level 66
operating 2–3
status during event handling 111
T
table
procedure attribute [136]
within interpreter 29, 30
Technical Support Department 146
TERMINATE 114, 115, 126, 131,
[xi], [103]
call, coding 131
closing files before [103]
termination character 67, [61],
[64]
three-byte address 13
pointer 98
time
date and
creation 64, 81, 84, 88, 89–90
format 90
last mod 64, 88, 89–90, [14],
[19]
time pointer [95], [97]
time-dependent code 104
timing loop 19, 104
TWO MANY BLOCK DEVICES
[130]
TWO MANY DEVICES [130]
TOOLONG [128]
top-level files 57
total_blocks 45, 82, [23], [70]
tracks 77
transfer control 28
transfer_count [36]
tree file 94, 96–97
entry 89
growing a 92–95
structure of a 96
tree structure, hierarchical 56
tree, file system 61
TYPERR [55]
U
unit number 44
unit_num 44, [68]
unsupported storage type
(TYPERR) [55]
utilities disk 41
utility
call(s) 114
errors 160, [126]
management 5
V
v1.2 symbol xix
and other versions xix
valid
jumps 29
pathnames 61
value 69, 151
value/result parameter 152
VCBERR [128]
version 81, 84, 88
number 45
version_num 45, [70]
VNFERR [54]
vol name 60, [23]
VOLUME 70, [23–24]
volume(s) 53–54, 76
bit map 77, 93
blocks on a 77
directory 54, 57, 78, 93
file 77
header 79, 80, 89
formats 77
multiple 54
name(s) 42, 55–56, 60
advantages of 56
syntax 56
switching 54–55
volume/device correspondence 54
W
warning
address conversion 123
interface versus implementation 99
on accessing zero page and
stack 17
on pointer conversions 155
on sample interpreter 125
pointer
direct 156
indirect 158, 159
risky regions 32
termination 114
unallocated memory 121
_Word [139]
words [133]
WRITE 68, 71, 90, [37–38]
write-enable bit [12], [18]
X
X register 14
X-bank, direct pointers to 155
X-byte 14, 15, 31, 145
between $80 and $8F, indirect
pointers with an 158
format 14
of $00, indirect pointers with
an 157
of $8F 16
range 15
X-page 145
Y
Y-register 15, 32
Z
zero
Interpreter’s 19
page 15, 17, 20, 29
and stack 17, 20
warning on accessing 17
conflicts with 16
priority of 100
zero-page addressing mode 29
zero-page indexed addressing
mode 29
Special Symbols and Numbers
& v1.2 81, 82, 84
$ xviii, xix
$0 16
$8F 16
6502 xvii
instruction set 8
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
Acknowledgements


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 2: The SOS Calls

Figures and Tables

Preface

9 File Calls and Errors

2 9.1 File Calls
3 9.1.1 CREATE
7 9.1.2 DESTROY
9 9.1.3 RENAME
11 9.1.4 SET_FILE_INFO
17 9.1.5 GET_FILE_INFO
23 9.1.6 VOLUME
25 9.1.7 SET_PREFIX
27 9.1.8 GET_PREFIX
29 9.1.9 OPEN
33 9.1.10 NEWLINE
35 9.1.11 READ
37 9.1.12 WRITE
39 9.1.13 CLOSE
41 9.1.14 FLUSH
43 9.1.15 SET_MARK
45 9.1.16 GET_MARK
47 9.1.17 SET_EOF
49 9.1.18 GET_EOF
51 9.1.19 SET_LEVEL
53 9.1.20 GET_LEVEL
53 9.2 File Calls Errors
10 Device Calls and Errors
58 10.1 Device Calls
59 10.1.1 D_STATUS
63 10.1.2 D_CONTROL
65 10.1.3 GET_DEV_NUM
67 10.1.4 D_INFO
71 10.2 Device Calls Errors

11 Memory Calls and Errors
74 11.1 Memory Calls
75 11.1.1 REQUEST_SEG
77 11.1.2 FIND_SEG
81 11.1.3 CHANGE_SEG
83 11.1.4 GET_SEG_INFO
85 11.1.5 SET_SEG_NUM
87 11.1.6 RELEASE_SEG
88 11.2 Memory Call Errors

12 Utility Calls and Errors
90 12.1 Utility Calls
91 12.1.1 SET_FENCE
93 12.1.2 GET_FENCE
95 12.1.3 SET_TIME
97 12.1.4 GET_TIME
99 12.1.5 GET_ANALOG
103 12.1.6 TERMINATE
104 12.2 Utility Call Errors

A SOS Specifications
105
106 Version
106 Classification
106 CPU Architecture
106 System Calls
106 File Management System
107 Device Management System
108 Memory/Buffer Management Systems
108 Additional System Functions
109 Interrupt Management System
109 Event Management System
109 System Configuration
109 Standard Device Drivers

B ExerSOS
113
114 B.1 Using ExerSOS
114 B.1.1 Choosing Calls and Other Functions
116 B.1.2 Input Parameters
117 B.2 The Data Buffer
117 B.2.1 Editing the Data Buffer
118 B.3 The String Buffer
119 B.4 Leaving ExerSOS

C MakeInterp
121

D Error Messages
123
124 D.1 Non-Fatal SOS Errors
124 D.1.1 General SOS Errors
125 D.1.2 Device Call Errors
125 D.1.3 File Call Errors
126 D.1.4 Utility Call Errors
126 D.1.5 Memory Call Errors
126 D.2 Fatal SOS Errors
128 D.3 Bootstrap Errors
Data Formats of Assembly-Language Code Files

132 E.1 Code File Organization
134 E.2 The Segment Dictionary
135 E.3 The Code Part of a Code File
136     E.3.1 The Procedure Dictionary
136     E.3.2 Procedures
136     E.3.3 Assembly-Language Procedure Attribute Tables
138     E.3.4 Relocation Tables
138         E.3.4.1 Base-Relative Relocation Table
139     E.3.4.2 Segment-Relative Relocation Table
139     E.3.4.3 Procedure-Relative Relocation Table
139     E.3.4.4 Interpreter-Relative Relocation Table

Bibliography

141

Index

143
Preface

Volume 2: The SOS Calls comprises the remaining chapters and the appendixes of this manual. The chapter numbers continue the sequence of those in Volume 1.

Volume 2 defines the individual SOS calls. Chapter 9 contains a description of each file call; Chapter 10, each device call; Chapter 11, each memory call; and Chapter 12, each utility call. Each of these chapters is divided into two sections: calls, and errors.

The calls defined in each chapter are arranged in numerical order by call number (for example, CREATE is $C0$). Each call description contains the following information:

- Definition of the call
- Required parameters
- Optional parameters
- Comments
- Errors

The parameter fields are of four types:

- Pointer (2 bytes): The location of a table or parameter list.
- Value (1, 2, or 4 bytes): A parameter passed by the caller to SOS.
- Result (1, 2, or 4 bytes): A parameter returned by SOS to the caller.
- Value/result (1, 2, or 4 bytes): A parameter passed to SOS and back to the caller, possibly changed.
- Unused (any length): Occurs when the same parameter list is used by two calls, one of which ignores some parameters in the list. An unused field can be of any length.
Each SOS call has three parts, described in Chapter 8 of Volume 1:

- The call block
- The required parameter list
- The optional parameter list

They can be diagrammed as shown in Figure 0-1:

Each call description is accompanied by a diagram like that shown in Figure 0-1. Most of the diagrams omit the call block, as these are identical, except for the `call_num`, and show only the required and optional parameter lists. In addition, the `parm_count` (shown in the diagram) is omitted from the required parameter list.

The one exception to this pattern is TERMINATE, for which the call block only is shown, as in Figure 0-2, because it differs from the standard form. See section 12.1.6 for details.

![Figure 0-1. Parts of the SOS Call](image)

![Figure 0-2. TERMINATE Call Block](image)
9.1 File Calls

- CREATE
- DESTROY
- RENAME
- SET_FILE_INFO
- GET_FILE_INFO
- VOLUME
- SET_PREFIX
- GET_PREFIX
- OPEN
- NEWLINE
- READ
- WRITE
- CLOSE
- FLUSH
- SET_MARK
- GET_MARK
- SET_EOF
- GET_EOF
- SET_LEVEL
- GET_LEVEL

9.2 File Calls Errors
9.1 File Calls

This section contains descriptions of all calls that operate on files. These calls operate on closed files and refer to a file by its pathname.

$C0: CREATE
$C1: DESTROY
$C2: RENAME
$C3: SET_FILE_INFO
$C4: GET_FILE_INFO
$C5: VOLUME
$C6: SET_PREFIX
$C7: GET_PREFIX
$C8: OPEN

These calls operate on access paths to open files and refer to the access path by its rel_num, returned by the OPEN call.

$C9: NEWLINE
$CA: READ
$CB: WRITE
$CC: CLOSE
$CD: FLUSH
$CE: SET_MARK
$CF: GET_MARK
$D0: SET_EOF
$D1: GET_EOF
$D2: SET_LEVEL
$D3: GET_LEVEL

9.1.1 CREATE

File Call $C0

This call creates a standard file or subdirectory file on a volume mounted on a block device. A directory entry is established, and at least one block is allocated on the volume.

This call cannot create a volume directory or a character file. Volume directories are "created" by the formatting utility on the Apple III Utilities disk. Character files are "created" by the System Configuration Program.

Required Parameter List

pathname: pointer
This parameter is a pointer to a string in memory containing the pathname of the file to be created: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. The last name in the pathname should be that of a file that does not currently exist in the specified directory, or a DUPERR will result.

option_list: pointer
This is a pointer to the optional parameter list if length (below) is between 1 and 8; otherwise it is ignored.

length: 1 byte value
Range: $0..$08
This is the length in bytes of the optional parameter list. It specifies which optional parameters are supplied.
The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the length to the next defined value.

\[
\begin{align*}
0 &= \text{no optional parameters} \\
1 &= \text{file_type} \\
3 &= \text{file_type through aux_type} \\
4 &= \text{file_type through storage_type} \\
8 &= \text{file_type through EOF}
\end{align*}
\]

Optional Parameter List

**file_type**: 1 byte value  
Range: \$00..$FF  
Default: $00

This is the type identifier for this file. The file_type does not affect the way in which SOS deals with the file. It is used only by interpreters to determine the internal arrangement and meaning of the bytes in the file. These values of file_type are now defined:

- \$00 = Typeless file (BASIC or Pascal "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..$DF = SOS reserved  
- \$E0..$FF = ProDOS reserved

**aux_type**: 2 byte value  
Range: \$00..$FFFF  
Default: \$0000

This is the auxiliary file identifier. It is used by interpreters to store any additional information about the file. BASIC, for example, uses this field to store the record size of its data files. If the file is a volume directory (storage_type is \$0F), these bytes contain the total number of blocks on the volume.

**storage_type**: 1 byte value  
Range: \$01..$0D  
Default: \$01

This indicates whether the file is to be a standard file ($01$) or a subdirectory file ($0D$). All other values are illegal and will result in a TYPERR.

**EOF**: 4 byte value  
Range: \$00000000..$00FFFFFF  
Default: \$00000000

This specifies the amount of space to preallocate for the file. One data block is automatically allocated regardless of the value of EOF; additional data blocks are allocated until the number of bytes in the allocated data blocks equals or exceeds EOF. In addition to the data blocks, index blocks are allocated as necessary.

The maximum creation size for standard files is $0FFFFFFF$, or $8000$ blocks. The maximum creation size for subdirectories is $00000000$, or $80$ blocks. The total number of blocks occupied by a file is the number of data blocks plus the number of index blocks: see Chapter 5 of Volume 1 for more information.

**Comments**

The file created must be a block file. The access attribute of the file is implicitly set to the following:

standard file = \$E3: (destroy, backup, rename, write, read)  
subdirectory = \$E1: (destroy, backup, rename, NO write, read)
Errors

$27: IOERR I/O error
$2B: NOWRITE Volume is write-protected
$40: BADPATH Invalid pathname syntax
$44: PNFERR Path not found
$45: VNFERR Volume not found
$46: FNFERR Subdirectory file not found
$47: DUPERR Attempt to CREATE an existing file
$48: OVRERR Overrun error. Either EOF too large or not enough disk space
$49: DIRFULL Directory is full
$4B: TYPERR Storage_type parameter neither $01 nor $0D
$52: NOTSOS Not a SOS volume
$53: BADLSTCNT Invalid length parameter
$58: NOTBLKDEV Not a block device

9.1.2 DESTROY

File Call $C1

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

The file can be either a standard or subdirectory file. Volume directories cannot be destroyed except by physical reformatting of the medium. Character files are "destroyed" by the System Configuration Program.

Required Parameters

pathname: pointer

This parameter is a pointer to a string containing the pathname of the file to be destroyed: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself.

Comments

A file cannot be destroyed if it is currently open. If the pathname refers to a subdirectory file, then that subdirectory must be completely empty in order for the subdirectory to be destroyed.
9.1.3 RENAME

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.

**Required Parameters**

- `pathname`: pointer
  
  This parameter is a pointer to a string containing the old pathname of the file to be renamed: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. The `pathname` must refer to either a volume directory, subdirectory, or standard file.

- `new_pathname`: pointer
  
  This parameter is a pointer to a string containing the new pathname of the file to be renamed: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. The `new_pathname` can be either a complete or partial pathname. Only the last file name of the new pathname may differ from that in the old pathname.

**Comments**

The file must reside on a block device. Both `pathname` and `new_pathname` must be identical except for the last file name. For example, the path `/VOL.1/FILE.1` can be renamed `/VOL.1/FILE.2` but not `/VOL.2/FILE.X` or `/VOL.1/SUBDIR.A/FILE.X`.

A file may not be renamed while it is open for writing.

If `new_pathname` matches the pathname of an existing file, you will get a DUPEERR.
Errors

$27: IOERR  I/O error
$2B: NOWRITE Volume is write-protected
$40: BADPATH Invalid pathname syntax
$44: PNFERR Path not found
$45: VNDFERR Volume not found
$46: FNFERR File not found
$47: DUPERR Duplicate file name
$4A: CPTERR Incompatible file format
$4B: TYPERR File storage type not supported
$4E: ACCSERR File's access attribute prevents RENAME
$50: FILBUSY File is open. Request denied.
$52: NOTSOS Not a SOS volume
$57: DUPVOL Duplicate volume
$58: NOTBLKDEV Not a block device

9.1.4 SET_FILE_INFO

File Call $C3

This call modifies file information in the directory entry of the block file specified by the pathname parameter. If the file is closed, a SET FILE_INFO call will modify the file information immediately. This information will be returned by any subsequent GET_FILE_INFO calls. If the file is open, no file information will be modified until the file is closed.

Required Parameters

pathname: pointer
This parameter is a pointer to a string containing the file name of the file whose directory entry will be modified: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself.

option_list: pointer
This is a pointer to the optional parameter list if length is between $01 and $0F; otherwise it is ignored.

length: 1 byte value
Range: $00..$0F

This is the length of the optional parameter list. It specifies which optional parameters are supplied. If length equals $00, no optional parameters are supplied: the call does nothing more than error checking.
The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the length to the next defined value.

0 = no optional parameters
1 = access
2 = access through file_type
4 = access through aux_type
F = access through last_mod

**Optional Parameters**

**access:** 1 byte value
Range: $00..$E3
Default: None

This parameter specifies the access allowed to the file. Bits 4 through 2 are reserved for future implementation and must be set to 0, otherwise an ACCSERR will occur.

- Write-enable
- Read-enable
- Backup
- Rename-enable
- Destroy-enable

For bits 7, 6, 1, and 0,

0 = not allowed
1 = allowed

These bits may be altered as the user wishes by the SET__FILE__INFO call.

For bit 5,

0 = backup not needed
1 = backup needed

This bit is always set when a SET__FILE__INFO call is made. Only the Backup III program can clear it.

**file_type:** 1 byte value
Range: $00..$FF
Default: Current value

This the type identifier for this file. The file_type does not affect the way in which SOS deals with the file: it is used only by interpreters to determine the internal arrangement and meaning of the bytes in the file. These values of file_type are now defined:

- $00 = Typeless file (BASIC or Pascal "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)
- $10..$DF = SOS reserved
- $E0..$FF = ProDOS reserved
aux_type: 2 byte value
   Range: $0000..$FFFF
   Default: Current value

This is the auxiliary file identifier. It is used by interpreters to store any additional information about the file. BASIC, for example, uses this field to store the record size of its data files. If the file is a volume directory (storage_type is $0F), these bytes contain the total number of blocks on the volume.

unused: 7 bytes

These bytes are here to maintain symmetry with GET__FILE__INFO, and are always ignored by SET__FILE__INFO.

last_mod: 4 byte value
   Range: $00000000..$FFFFFFF
   Default: Current value

This is the date and time the file was last closed after being written to. It can be set to a user-defined value, or you can use the GET__TIME call (see the Utility calls) and form this value from the current time. The last_mod parameter is organized as two 2-byte words, each stored low byte first:

```
+--------+--------+
| 7 6 5 4| 3 2 1 0 |
+--------+--------+
    |        |
+--------+--------+
| 7 6 5 4| 3 2 1 0 |
+--------+--------+
    |        |
```

The ranges for these fields are as follows:

- Year: $0..99 ($00..$63)
- Month: $0..11 ($00..$0C)
- Day: $0..31 ($00..$1F)
- Hour: $0..23 ($00..$18)
- Minute: $0..59 ($00..$3C)

A zero value for the month or day means that no value was set.

No checking is performed on this parameter. If you use the GET__TIME call, you must pack the 18-byte time parameter from that call into the proper format for the SET__FILE__INFO call's last_mod parameter.

Comments

The default value for all optional parameters that are omitted is the current value of that attribute of the file: for example, omitting the last_mod parameter results in no change to that file's modification date and time.

The same required and optional parameter lists can be used for GET__FILE__INFO. In fact, you can perform a GET__FILE__INFO, examine and perhaps alter the values in the parameter lists, and then perform a SET__FILE__INFO to update the file's attributes.

You can perform SET__FILE__INFO on any block file, regardless of the current value of its access attribute. In this call, therefore, an access error can result only from passing an invalid access parameter.

SET__FILE__INFO affects a file's directory entry only. It does not affect the FCB entry for any access path to the file. Specifically, if you open a file with read/write access, then use a SET__FILE__INFO call to change the access to read-only, you still write to the file via that access path, but you cannot open another access path. This is because the access field in the file's directory entry will not be updated until the file is closed, and the FCB entries will not be updated at all: so, as far as SOS is concerned, this is still a read/write file, for which only one access path is allowed. As soon as you close the file, however, the new access value will be stored in the directory entry, and multiple read-only access paths can be opened.
9.1.5 GET__FILE__INFO

This call returns file information from the directory entry of the block file specified by the **pathname** parameter.

**Required Parameters**

**pathname**: pointer

This parameter is a pointer to a string containing the pathname of the file whose directory entry information will be returned: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself.

**option_list**: pointer

This is a pointer to the optional parameter list if **length** is between $01$ and $0F$; otherwise it is ignored.

**length**: 1 byte value

- Range: $00..0F$

This is the length of the optional parameter list. If **length** equals $00$, no optional parameters are returned: the call does nothing more than error checking.

The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the **length** to the next defined value.
$00 = no optional parameters
$01 = access
$02 = access through file_type
$04 = access through aux_type
$05 = access through storage_type
$09 = access through EOF
$0B = access through blocks_used
$0F = access through last_mod

Optional Parameters

access: 1 byte result
Range: $00..$C3

This parameter returns the access allowed to the file. Bits 4 through 2 are reserved for future implementation and are now set to 0.

write-enable
read-enable
D R N B RESERVED W R
backup
rename-enable
destroy-enable

For bits 7, 6, 1, and 0,
0 = not allowed
1 = allowed

For bit 5,
0 = backup not needed
1 = backup needed

file_type: 1 byte result
Range: $00..$FF

This the type identifier for this file. The file_type does not affect the way in which SOS deals with the file: it is used only by interpreters to determine the internal arrangement and meaning of the bytes in the file. These values of file_type are now defined:

$00 = Typeless file (BASIC or Pascal "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
$0B = Word Processor file
$0C = SOS system file (DRIVER, INTERP, KERNEL)
$0D, $0E = SOS reserved
$0F = Directory file (see storage_type)
$10..$1F = SOS reserved
$E0..$FF = ProDOS reserved

aux_type: 2 byte result
Range: $0000..$FFFF

This is the auxiliary file identifier. It is used by interpreters to store any additional information about the file. BASIC, for example, uses this field to store the record size of its data files. If the file is a volume directory (storage_type is $0F), these bytes contain the total number of blocks on the volume.

storage_type: 1 byte result
Range: $01..$03, $0D, $0F

This byte describes the external format of the file: how the blocks that compose the file are stored on the volume.

$01 = seedling file ( 0 < = EOF < = 512 bytes)
$02 = sapling file ( 512 < EOF < = 128K bytes)
$03 = tree file (128K < EOF < 16M bytes)
$0D = subdirectory file
$0F = volume directory file
These structures are fully explained in Chapter 5. In brief, seedling files are stored as one data block; sapling files are stored as one index block and up to 256 data blocks; tree files are stored as one root index block, up to 127 subindex blocks, and up to 32,767 data blocks. Directories and subdirectories do not use index blocks, and instead are stored as doubly-linked lists of blocks.

**EOF:** 4 byte result

Range: `$00000000..$00FFFFFF`

This is the position of the end of file marker. It indicates the number of bytes readable from the file. This is the EOF value stored in the file's directory entry when the file was created or last closed. It is accurate only if the file is not open for writing. If the file is open for writing, the current EOF (stored in the file's FCB entry) can be read by the GET_EOF call.

**blocks_used:** 2 byte result

Range: `$0000..$FFFF`

If the file is a standard file or subdirectory (storage_type is $01, $02, $03, or $0D), blocks_used is the total number of blocks (including index blocks) currently used by the file.

- If the file is a sparse file, the blocks_used value can be substantially less than one would expect from the EOF.

- If the file is a volume directory (storage_type is $0F), blocks_used is the total number of blocks used by all files on the volume.

**last_mod:** 4 byte result

Range: `$00000000..$FFFFFFF`

This is the date and time the file was last closed after being written to. If the file has never been written to, these bytes are the same as the creation date of the file. SET_FILE_INFO can also change the modification date.

The ranges for these fields are as follows:

- **Year:** $0..99 ($00..$63)
- **Month:** $0..12 ($00..$0C)
- **Day:** $0..31 ($00..$1F)
- **Hour:** $0..24 ($00..$18)
- **Minute:** $0..59 ($00..$3C)

A zero value for the month or day means that no value was set.

**Comments**

This call can be performed when the file is either open or closed. The same required and optional parameter lists can be used for SET_FILE_INFO. A GET_FILE_INFO call to an open file will return file information from the directory entry, not access path information from the FCB entry. This is not surprising, since the GET_FILE_INFO call refers to a file by its pathname, not its ref_num. For example, if you have changed the EOF since the file was opened, GET_FILE_INFO will not return the current value.

**Errors**

- **$27:** IOERR  I/O error
- **$40:** BADPATH  Invalid pathname syntax
- **$44:** PNFERR  Path not found
- **$45:** VNFERR  Volume not found
- **$46:** FNFERR  File not found
- **$4A:** CPTERR  Incompatible file format
- **$4B:** TYPERR  Unsupported file storage type
- **$52:** NOTSOS  Not a SOS volume
- **$53:** BADLSTCNT  Length parameter invalid
- **$58:** NOTBLKDEV  Not a block device
9.1.6  VOLUME

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.

Required Parameters

**dev_name**: pointer

This parameter is a pointer to a string containing the device name: the first byte of the string contains the number of bytes in the device name; the remaining bytes contain the device name itself.

**vol_name**: pointer

This is a pointer to a buffer at least $10$ bytes long into which the volume name will be returned: the first byte in the buffer contains the number of bytes in the volume name; the rest contain the name itself.

**total_blocks**: 2 byte result

Range: $0000..FFFF$

This is the total number of blocks contained by the volume in the specified block device.

**free_blocks**: 2 byte result

Range: $0000..FFFF$

This is the number of unallocated blocks contained by the volume in the specified block device.

Comments

The **dev_name** must point to the name of a block device.
### Errors

<table>
<thead>
<tr>
<th>Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10:</td>
<td>DNFERR</td>
<td>Device not found</td>
</tr>
<tr>
<td>$27:</td>
<td>IOERR</td>
<td>I/O error</td>
</tr>
<tr>
<td>$45:</td>
<td>VNFERR</td>
<td>Volume not found</td>
</tr>
<tr>
<td>$4A:</td>
<td>CPTERR</td>
<td>Incompatible file format</td>
</tr>
<tr>
<td>$52:</td>
<td>NOTSOS</td>
<td>Not a SOS volume</td>
</tr>
<tr>
<td>$58:</td>
<td>NOTBLKDEV</td>
<td>Not a block device</td>
</tr>
</tbody>
</table>

---

### 9.1.7 SET_PREFIX

**File Call $C6**

This call sets the current SOS prefix pathname to that specified by `pathname`.

**Required Parameters**

- `pathname`: pointer

This parameter is a pointer to a string containing the pathname that will replace the current prefix pathname: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. This pathname specifies a volume directory or subdirectory, not a character file or a standard file.

**Comments**

The system prefix is appended to the beginning of any pathname not beginning in a volume name or device name: a volume name is preceded by a slash, and a device name begins with a period.

If the new prefix begins with a volume name, only syntax checking is performed on it: SOS does not verify that the directory file specified by the prefix is actually on line. If the new prefix begins with a device name, SOS substitutes the corresponding volume name: the SOS prefix always begins with a volume name.

The prefix can be reset to null by passing a pathname with a length of zero characters.

Upon system boot, the prefix is initialized to the volume directory name of the boot disk.

*The `pathname` can optionally terminate with a "/".*
9.1.8  GET__PREFIX

This call returns the current SOS prefix pathname.

Required Parameters

pathname: pointer
This parameter is a pointer to a string into which SOS is to store the current prefix pathname: the first byte of the string contains the number of bytes in the prefix; the remaining bytes contain the prefix itself.

length: 1 byte value
  Range:  $00..$FF
  Default: $80
This is the maximum number of bytes in the pathname buffer. This should be set as long as the longest prefix the interpreter accepts: SOS will accept up to 128 ($80) bytes. A BTSERR is returned if the pathname is longer than length.

Comments

If the SOS prefix pathname has been set to the null string (no prefix), the null string is returned.

If the prefix pathname is not null, it is terminated with a slash.

If the first name in the prefix pathname is a volume name, the pathname begins with a slash.

Errors

$4F:  BTSERR    Buffer too small
9.1.9 OPEN

This call causes SOS to open an access path (allowing read-access, write-access, or both) to the file specified by pathname. For this access path, SOS makes an entry in the file control block and allocates a 1024-byte I/O buffer. This buffer holds the contents of one index block (if the file has any) and one data block.

Required Parameters

pathname: pointer

This is a pointer to a string in memory containing the pathname of the file to be opened; the first byte is the number of characters in the pathname; the remaining bytes are the characters of the pathname itself. It may be any block or character file.

ref_num: 1 byte result

Range: $01..$10, $81..$90

The reference number is assigned when an access path to a file is opened. It uniquely identifies an access path to the file: any open-file call will operate on a single access path, not the file itself.

option_list: pointer

This points to optional parameter list if length is between $01 and $04; otherwise it is ignored.
length: 1 byte value
  Range: $00..$04

This is the length in bytes of the optional parameter list. It specifies which optional parameters are supplied.

The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the length to the next defined value.

$00 = no optional parameters
$01 = req_access
$04 = req_access through io_buffer

Optional Parameters

req_access: 1 byte value
  Range: $00..$03
  Default: $00

This is the requested file access. SOS compares this parameter with the file’s current access-attribute byte to ensure that the intended file operations are permitted. A $00 requests as much access as permitted.

$00 = Open as permitted
$01 = Open for reading only
$02 = Open for writing only
$03 = Open for reading and writing

A standard file that is already open for writing may have only one access path: a req_access of $00 will open the existing access path for reading as well. A standard file on a write-protected volume may never be opened for writing; a req_access of $00 will open such a file for reading only.

A character file may have multiple access paths with read-access, write-access, or both, if the file’s device allows such access.

pages: 1 byte value
  Range: $00 or $04
  Default: $00

This is the length in 256-byte pages of a caller-supplied I/O buffer. If equal to $00, then SOS finds its own buffer, ignoring the io_buffer parameter below. If equal to $04, then SOS will use the 1024-byte buffer pointed to by io_buffer. Any value except $00 or $04 is invalid.

If pages is nonzero, you must specify an io_buffer parameter.

In general, it is preferable to let SOS allocate an I/O buffer.

io_buffer: pointer

This is an indirect pointer to a caller-supplied I/O buffer if and only if the pages parameter is nonzero.

Comments

On block files, multiple access paths for read-access are permitted.

On block files, only one access path for writing is permitted: no other access path, even for reading only, is permitted at the same time.

Multiple access paths on character files for both read- and write-access are permitted.

OPEN sets the file level of the opened file to the current system file level (see SET_LEVEL and GET_LEVEL). Unless the file level is raised, a subsequent CLOSE or FLUSH of multiple files will close or flush this file.

The option_list and length parameters are ignored when OPENing character files; no optional parameters are used.
9.1.10 NEWLINE

This call allows the caller to turn newline read mode on or off. Once newline mode has been turned on, any subsequent READ operation will immediately terminate if the newline character is encountered in the input byte stream.

**Required Parameters**

- **ref_num**: 1 byte value
  - Range: $01..$10, $81..$90
  - This is the reference number of the access path, provided by the OPEN call.

- **is_newline**: 1 byte value
  - Range: $00..$FF
  - The high bit of this byte determines whether newline read mode is on or off. If it is set (is_newline > $7F), newline mode is on; otherwise, newline mode is off.

- **newline_char**: 1 byte value
  - Range: $00..$FF
  - This byte indicates the character used to terminate read requests. If newline read mode is off, this parameter is ignored.
Comments

The **newline_char** byte need not have any ASCII interpretation.

A NEWLINE call to a character file implicitly does a D__CONTROL call number 2 (set newline mode) to the device driver represented by that file. This changes the newline mode of all access paths to that character file.

Errors

$43:  BADREFNUM      Bad reference number

9.1.11 READ

This call attempts to transfer **request_count** bytes, starting from the current file position (**mark**), from the I/O buffer of the file access path specified by **ref_num** into the interpreter's data 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.

Required Parameters

**ref_num**: 1 byte value

Range: $01..$10, $81..$90

This is the reference number of the access path to be read from, obtained through an OPEN call.

**data_buffer**: pointer

This is a pointer to the first byte of a caller-supplied buffer at least **request_count** bytes long.

**request_count**: 2 byte value

Range: $0000..$FFFF

This is the number of bytes SOS is to read from the file into the buffer. If **request_count** equals $0000, the READ call does error checking only; no bytes are read.
transfer_count: 2 byte result
    Range: $0000..request_count

If a READ is successful, the number of bytes transferred to the data buffer is returned in this parameter. If a READ is completely unsuccessful, transfer_count equals $0000.

Comments

READ advances the current file position (mark) by one byte for each byte transferred. It will advance the mark up to the end-of-file (EOF) marker, which points one byte past the last byte in the file. READ fails with an EOFERR if and only if the mark already equals EOF; in this case, no bytes are transferred and transfer_count returns zero.

If a READ operation spans several contiguous blocks on a disk, SOS transfers whole blocks directly to the interpreter's data buffer, bypassing the I/O buffer; partial blocks go through the I/O buffer. This optimization improves performance, but is otherwise invisible to the interpreter writer and user.

Errors

$27:  IOERR  I/O error
$43:  BADREFNUM Invalid reference number
$4C:  EOFERR  End of file has been encountered
$4E:  ACCSERR File not open for READING

9.1.12 WRITE

This call attempts to transfer 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.

Required Parameters

ref_num: 1 byte value
    Range: $01..$10, $81..$90

This is the reference number of the file to be written to, obtained by an OPEN call.

data_buffer: pointer

This is a pointer to a caller-supplies buffer from which SOS is to draw the bytes to be written to the file. This pointer is not modified by SOS.

request_count: 2 byte value
    Range: $0000..$FFFF

This is the number of bytes to be written to the file.

Comments

If WRITE ends with an OVRERR, it has written all the bytes that it can to the file: it will not tell you how many it has written. Otherwise, WRITE always succeeds or fails completely.

Bytes written to a file may be stored in an I/O buffer, and sent a buffer-load at a time. For block files, WRITE physically alters the bytes on the volume only when a block of bytes has been written to the file: this occurs automatically when the mark crosses a block boundary. To ensure that information in the buffer has been updated on the volume, use the FLUSH call.
Errors

$27: IOERR  I/O error
$2B: NOWRITE Volume write-protected
$43: BADREFNUM Invalid reference number
$48: OVRERR Not enough room in file or on volume
$4E: ACCSERR Tried to write to read-only file

9.1.13 CLOSE

The file access path specified by ref_num is closed. Its file control block (FCB) entry is deleted, and if the file is a block file that has been written to, its I/O buffer is written to the file. The directory entry of a block file is then updated from the FCB entry. Further file operations using that ref_num will fail.

Required Parameters

ref_num: 1 byte value
Range: $00..$10, $81..$90

This is the reference number of the file to be closed, obtained by an OPEN call.

Comments

If a block file has been written to, a CLOSE call changes the modification date and time of the file to the current date and time.

If ref_num equals $00, all open files are closed whose file level (see SET_LEVEL, GET_LEVEL) is greater than or equal to the current system level.

If an error occurs while closing multiple files, all files that can be closed will be, and CLOSE will return the error number of the last error that occurred. CLOSE will not tell you which files were closed and which were not.
Errors

$27$: IOERR  I/O error
$2B$: NOWRITE  Volume is write-protected
$43$: BADREFNUM  Invalid reference number
$48$: OVRERR  Not enough room on volume

9.1.14  FLUSH

File Call $CD$

If a previous WRITE call has left any data in a block file's I/O buffer, the FLUSH call writes these data to the volume the file is stored on and clears the buffer. If the I/O buffer is empty, FLUSH simply returns an error code of $00$.

**Required Parameters**

ref_num: 1 byte value
   Range: $00..10$

This is the reference number of the block file access path to be FLUSHed, obtained from an OPEN call. Since the file is open for writing, this access path is the only one.

**Comments**

FLUSH must be used only on block file access paths that are open for writing.

If the ref_num equals $00$, all open files are FLUSHed whose file level (see SET_LEVEL, GET_LEVEL) is greater than or equal to the current system file level.

FLUSH is a time-consuming call: if it is used when not needed, performance will suffer.
Errors

$27: IOERR       I/O error
$2B: NOWRITE    Volume is write-protected
$43: BADREFNUM  Invalid reference number
$48: OVRERR     Not enough room on volume
$58: NOTBLKDEV  Not a block device

9.1.15 SET_MARK

This call changes the current file position (mark) of the file access path specified by ref_num. The mark can be changed to an absolute byte position in the file, or to a position relative to the EOF or the current mark.

Required Parameters

ref_num: 1 byte value
          Range: $01..$10

This is the reference number of the block file access path whose mark is to be moved, obtained through an OPEN call.

base: 1 byte value
       Range: $00..$03

This is the starting byte position in the file from which to calculate the new mark position.

$00 = Absolute, byte $00000000..$00FFFFFF
$01 = Backward from EOF
$02 = Forward from current mark
$03 = Backward from current mark

displacement: 4 byte value
              Range: $00000000..$00FFFFFF

This is the number of bytes the mark is to move from the starting location specified by the base parameter. The final computed position must lie between $0 and the current EOF ($0 <= mark <= EOF <= $FFFFFFF).
9.1.16 GET__MARK

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

**Required Parameters**

ref_num1: 1 byte value
  - Range: $01..$10

This is the reference number of the file whose current position is to be returned.

mark: 4 byte result
  - Range: $00000000 through current EOF value

This is the current mark position in the file.

**Errors**

$43: BADREFNUM Invalid reference number
$58: NOTBLKDEV Not a block device
9.1.17  SET_EOF

This call changes the end-of-file marker (EOF) of the block file whose access path is specified by ref_num. The EOF can be changed to an absolute byte position, or to a position relative to the current EOF or the current mark.

If the new EOF is less than the current EOF, empty blocks at the end of the file are released to the system and their data are lost. If the new EOF is greater than the current EOF, blocks are not physically allocated for unwritten data. (This is one way of creating a sparse file.) If a program attempts to read from these newly created logical positions before they have been physically written to, SOS supplies a null ($D0) for each byte requested.

Required Parameters

ref_num: 1 byte value
  Range: $01..$10

This is the reference number of the file whose EOF is to be changed, returned by an OPEN call. It must refer to a block file open for writing, and is thus the file's sole ref_num.
base: 1 byte value
    Range: $00..$03

This is the position in the file from which to calculate the new value of EOF, (the current number of bytes in the file).

    $00 = Absolute, byte $000000..$FFFFF
    $01 = Backward from current EOF
    $02 = Forward from current mark position
    $03 = Backward from current mark position

displacement: 4 byte value
    Range: $0000000..$00FFFFFF

This is the number of bytes the EOF is to move from the starting position specified in the base parameter. The final computed position must be greater than or equal to $000000, and less than or equal to $FFFFFFF.

Comments

The file must be a block file currently open for writing. Since such a file can have only one access path, the ref_num specifies the file, as well as the access path.

This call updates the EOF field in the file control block entry, but not the EOF field in the file's directory entry: the latter is updated only when the access path is closed. For this reason, a GET__FILE__INFO call to an open file will not always return the current EOF. A GET__EOF call will.

Errors

$27: IOERR I/O error
$2B: NOWRITE Volume write-protected
$43: BADREFNUM Invalid reference number
$4D: POSNERR Position out of range
$4E: ACCSERR Tried to move EOF of read-only file
$58: NOTBLKDEV Not a block device
9.1.19 SET_LEVEL

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.

Required Parameters

level: 1 byte value
Range: $01..$03

This specifies the new file level.

Comments

The system file level is set to $01 at boot time.

Errors

$59: LEVLERR Invalid file level
9.1.20 GET__LEVEL

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

Required Parameters

level: 1 byte result
Range: $01..$03

This parameter returns the current file level.

Comments

The file level is set to $01 at boot time.

9.2 File Call Errors

These error messages can be generated by SOS file calls; in addition, some of these calls may generate device call errors, described in section 10.2. Other errors are listed in Appendix D.

$40: Invalid pathname syntax (BADPATH)
The pathname violates the syntax rules in Chapter 4 of Volume 1.

$41: Character File Control Block full (CFCBFULL)
The Character File Control Block (CFCB) table can contain a maximum of $10 entries. Opening the same character file more than once will return the same ref_num (that is, will not consume an additional entry).
$42: Block File or Volume Control Block full (FCBFULL)

The Block File Control Block (BFCB) table can contain a maximum of $10 entries. The Volume Control Block (VCB) table can contain a maximum of $08 entries. Opening the same block file more than once returns a different ref_num and consumes a new entry in the BFCB table. Every volume with an open file on it, whether it is mounted on a device or not, consumes one entry in the VCB table.

$43: Invalid reference number (BADREFNUM)

The ref_num input parameter does not match the ref_num of any currently open file. This error is also returned if the currently open file is marked with a bad storage_type; only $01 through $04, $0D, and $0F are allowed.

$44: Path not found (PFNERR)

Some file name in the pathname refers to a nonexistent file. The pathname's syntax is legal.

$45: Volume not found (VNFERR)

The volume name in the pathname refers to a nonexistent volume directory. The pathname's syntax is legal.

$46: File not found (FNFERR)

The last file name in the pathname refers to a nonexistent file. The pathname's syntax is legal. Note that a missing volume directory file returns VNFERR instead of FNFERR.

$47: Duplicate file name (DUPERR)

An attempt was made to CREATE a file using a pathname that already belongs to a file, or a RENAME was attempted using a new_pathname that already belongs to a file.

$48: Overrun on volume (OVRERR)

An attempt to allocate blocks on a volume during a CREATE or WRITE operation failed due to lack of space on the volume. This error also is returned on an invalid EOF parameter.

$49: Directory full (DIRFULL)

No more entries are left in the root/subdirectory. Thus no more files can be added (CREATED) in this directory until another file is DESTROYed.

$4A: Incompatible file format (CPTERR)

The file is not backward compatible with this version of SOS.

$4B: Unsupported storage type (TYPERR)

The CREATE call accepts only two values for the storage_type parameter: $01 (standard file) or $0D (subdirectory file).

$4C: End of file would be exceeded (EOFERR)

A READ call was attempted when the mark was equal to the EOF.

$4D: Position out of range (POSNERR)

A base/displacement parameter pair produced an invalid mark or EOF.

$4E: Access not allowed (ACCSERR)

The user attempted to access (RENAME, DESTROY, READ from, or WRITE to) a file in a way not allowed by its access attribute.

$4F: Buffer too small (BTSERR)

The user supplied a buffer too small for its purpose.

$50: File busy (FILBUSY)

An attempt was made to RENAME or DESTROY an open file or to OPEN a block file already open for writing.

$51: Directory error (DIRERR)

The directory entry count disagrees with the actual number of entries in the directory file.

$52: Not a SOS volume (NOTSOS)

The volume in the block device contains a directory that is not in SOS format: it may be an Apple II Pascal or DOS 3.3 volume.
$53: Length parameter invalid (BADLSTCNT)
The length supplied for the optional parameter list is invalid.

$54: Out of memory (OUTOFMEM)
There is not enough free memory for the SOS system buffer. The user must release some memory to SOS to allow the system to use it.

$55: Buffer Table full (BUFTBLFULL)
The Buffer Table can contain a maximum of $10 entries.

$56: Invalid system buffer parameter (BADSYSBUF)
The buffer pointer parameter must be an extended indirect pointer.

$57: Duplicate volume (DUPVOL)
A SOS call asked SOS to bring a volume on-line on a particular block device. The request was denied because a volume with the same name on another block device is currently on line and contains a currently open file.

$58: Not a block device (NOTBLKDEV)
Only OPEN, NEWLINE, READ, WRITE, and CLOSE file calls can reference a character file. For example, CREATE is not permitted on the character file .PRINTER .

$59: Invalid level (LVLERR)
The SET_LEVEL call received a parameter less than $01 or greater than $03.

$5A: Invalid bit map address (BITMAPADR)
An index block contained a block number that, according to the bit map, is not physically available on the volume; usually this indicates that the blocks on the volume have been scrambled.
**Device Calls**

These SOS calls operate directly on devices.

- $82$: D__STATUS
- $83$: D__CONTROL
- $84$: GET__DEV__NUM
- $85$: D__INFO

### 10.1.1 D__STATUS

This call returns status information about a particular device. The information can be either general or device-specific information.

D__STATUS returns information about the internal status of the device or its driver;

GET__DEV__INFO returns information about the external status of the driver and its interface with SOS.

#### Required Parameters

- **dev_num**: 1 byte value
  - Range: $01..$18

  This is the device number of the device from which to read status information, obtained from the GET__DEV__NUM call. Each device in the system has a unique device number assigned to it when the system is booted. Device numbers do not change unless the SOS.DRIVER file is changed and the system is rebooted.

- **status_code**: 1 byte value
  - Range: $00..$FF

  This is the number of the status request being made. All device drivers respond to the following requests:

  Block devices only:

  - $00$  Return driver's status byte

  Character devices only:

  - $00$  No effect
  - $01$  Return driver's control block
  - $02$  Return newline status
Device drivers also may respond to other status codes. The complete list of status requests available for a device driver is included in the documentation accompanying that driver.

**status_list**: pointer

This is a pointer to the buffer in which the device driver returns its status. For the three requests above, the buffer is in one of these three formats:

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSY</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>WPR</td>
</tr>
</tbody>
</table>
```

- BSY: If 1, device is busy
- WPR: If 1, device or medium is write-protected

**Figure 10-1.** Block Device Status Request $00

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSY</td>
<td>If 1, device is busy</td>
</tr>
<tr>
<td>WPR</td>
<td>If 1, device or medium is write-protected</td>
</tr>
</tbody>
</table>

The newline character is called the *termination character* in the *Apple III Standard Device Drivers Manual*.

Each driver that defines its own additional status requests also defines buffer formats for those requests; see the manual describing that driver.

**Comments**

The length of the buffer pointed to by **status_list** must vary depending upon the particular status request being made.

**Errors**

- $11$: BADNUM    Invalid device number
- $21$: CTLCODE   Invalid status code
- $23$: NOTOPEN   Character device not open
- $25$: NORESRC   Resource not available
- $30..$3F: Device-specific error
10.1.2 D__CONTROL

This call sends control information to a particular device. The information can be either general or device-specific information. D__CONTROL operates on character devices only.

Required Parameters

dev_num: 1 byte value
   Range: $01..$18

This is the device number of the device to which to send control information, obtained from the GET__DEV__NUM call. Each device in the system has a unique device number assigned to it when the system is booted. Device numbers do not change unless the SOS.DRIVER file is changed and the system is rebooted.

control_code: 1 byte value
   Range: $00..$FF

This is the number of the control request being made. All character device drivers respond to the following requests:

   $00  Reset device
   $01  Restore driver's control block
   $02  Set newline mode and character

Block devices do not respond to any control requests.

Device drivers also may respond to other control requests. The complete list of control requests available for a device driver is included in the documentation accompanying that driver.

control_list: pointer

This is a pointer to the buffer from which the device driver draws the control information. For the two requests above, the buffer is in one of these two formats:
10.1.3 GET_DEV_NUM

This call returns the device number of the driver whose device name is specified. The device need not be open. The dev_num returned is used in the D_STATUS, D_CONTROL, and D_INFO calls.

**Required Parameters**

- **dev_name**: pointer
  - This is a pointer to a string in memory containing the device name of the device whose number is to be returned; the first byte of the string is the number of bytes in the name; the rest are the bytes of the name itself. Note that this a device name, not a pathname.
  - **dev_num**: 1 byte result
    - Range: $01..$18
  - This is the device number of the device specified by dev_name. The name of a device can be changed by the System Configuration Program.

**Errors**

- **$10**: DNFERR  Device name not found

---

**Figure 10-4. Character Device Control Code $01**

The status list for each driver has a different format. See the manual describing that driver.

**Figure 10-5. Character Device Control Code $02**

The newline character is called the termination character in the Apple III Standard Device Drivers Manual.

Each driver that defines its own additional control requests also defines buffer formats for those requests; see the documentation for that driver.

**Comments**

The length of the buffer pointed to by control_list must vary depending upon the particular control request being made.

**Errors**

- **$11**: BADNUM  Invalid device number
- **$21**: CTLCODE  Invalid control code
- **$23**: NOTOPEN  Character device not open
- **$25**: NORESRC  Resource not available
- **$26**: BADOP  No control of block devices allowed
- **$30..$3F**: Device-specific error
10.1.4 D_INFO

This call returns the device name (and optionally, other information) about the device specified by dev_num. The device's character file need not be open. D_INFO returns identifying 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.

Required Parameters

dev_num: 1 byte value
  Range: $01..$18

This is the device number of the device whose information is to be returned, obtained from the GET_DEV_NUM call.

dev_name: pointer

This is a pointer to a sixteen-byte buffer into which SOS is to store the resulting device name: the first byte of the buffer is the number of bytes in the name; the rest are the bytes of the name itself.

option_list: pointer

This is a pointer to the optional parameter list if length is between $00 and $0A; otherwise it is ignored.

length: 1 byte value
  Range: $00..$0A

This is the length in bytes of the optional parameter list. It specifies which optional parameters are supplied.
The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the length to the next defined value.

- $00 = \text{no optional parameters}
- $01 = \text{slot_num}
- $02 = \text{slot_num through unit_num}
- $03 = \text{slot_num through dev_type}
- $05 = \text{slot_num through sub_type}
- $07 = \text{slot_num through total_blocks}
- $09 = \text{slot_num through manuf_id}
- $0B = \text{slot_num through version_num}

Optional Parameters

**slot_num**: 1 byte result  
Range: $00..$04

This is the slot number of the peripheral slot the device uses. Slot numbers $01$ through $04$ correspond to peripheral slots 1 through 4. Slot number $00$ indicates the device does not use a peripheral slot.

**unit_num**: 1 byte result  
Range: $00..$FF

This is the unit number of the device. Devices that are bundled together into one driver module are assigned unit numbers in ascending sequence, beginning with $00$. See the Apple III SOS Device Driver Writer's Guide for more details.

This parameter has nothing to do with the logical unit numbers that Pascal associates with the devices.

**dev_type**: 1 byte result  
Range: $00..$FF

The dev_type byte, along with the following byte, is used for device classification and identification. This field specifies the generic family that the device belongs to.

The dev_type byte for SOS character devices has the following structure:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
<td>R</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Bit 7 is cleared for all character devices.

Bit 6 (W) is **write allowed** byte. It must be set for all character devices that accept data from the Apple III.

Bit 5 (R) is the **read allowed** bit. It must be set for all character devices that send data to the Apple III.

Bit 4 is reserved for future use and must always be cleared.

The dev_type byte for SOS block devices has the following structure:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>Rem</td>
<td>Fmt</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Bit 7 is set for all block devices.

Bit 6 (W) is **write allowed** byte. It must be set for all block devices that accept data from the Apple III.

Bit 5 (R) is the **removable device** bit. It must be set for all block devices that use removable storage media, such as floppy-disk drives.

Bit 4 is set if the driver can also format its device.

**sub_type**: 1 byte result  
Range: $00..$FF

The device subtype identifies the specific device within the generic family specified in dev_type.

unused: 1 byte
Please contact the PCS Division Product Support Department of Apple Computer, Inc. if you wish to be assigned a dev_type, sub_type, manuf_id, or version_num. This will ensure that such codes are unique and are known to SOS and future application programs.

**Errors**

$11: \text{BADNUM} \quad \text{Invalid device number}

### 10.2 Device Call Errors

The errors below are generated by SOS device calls; some of them are also generated by SOS file calls. Other errors are listed in Appendix D.

$18: \text{Device not found (DNFERR)}

The device name passed as a parameter to GET_DEV_NUM is not that of a device that is configured into the system; a device driver with that name was not in the SOS.DRIVER file at the time the system was booted, or that device driver was inactive.

$11: \text{Invalid device number (BADDNUM)}

The dev_num parameter does not contain the device number of a device configured into the system.

$28: \text{Invalid request code (BADREQCODE)}

This error is generated only for device requests, made by SOS to a device driver, and should never be received as a result of a SOS call.

$21: \text{Invalid status or control code (BADCTL)}

The control (for D__CONTROL) or status (for D__STATUS) code is not supported by the device driver being called.

$22: \text{Invalid control parameter list (BADCTL Parm)}

The parameter list specified by the control parameter to the D__CONTROL call is not in the proper format for the control request being made.

---

**total_blocks**: 2 byte result  
Range: $0000..$FFFF

If the device is a block device, this parameter indicates the total number of blocks it can access. If the device is a character device, this parameter returns $0000. The Apple III's built-in disk drive can access $0118 blocks.

**manuf_id**: 2 byte result  
Range: $0000..$FFFF

The manufacturer identification code uniquely identifies the manufacturer of the driver. The currently assigned values are:

- $0000: \text{Unknown}
- $0001: \text{Apple Computer, Inc.}

**version_num**: 2 byte result  
Range: $0000..$9999

This is the version number of the device driver. The format is BCD (binary-coded decimal); no hexadecimal digits from $A$ to $F$ will appear in this result.

**Comments**

The following values for dev_type and sub_type are assigned:

<table>
<thead>
<tr>
<th>dev_name</th>
<th>dev_type</th>
<th>sub_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS232 printer (.PRINTER)</td>
<td>$41</td>
<td>$01</td>
</tr>
<tr>
<td>Silentype printer (.SILENTYPE)</td>
<td>$41</td>
<td>$02</td>
</tr>
<tr>
<td>Parallel printer (.PARALLEL)</td>
<td>$41</td>
<td>$03</td>
</tr>
<tr>
<td>Sound port (.AUDIO)</td>
<td>$43</td>
<td>$01</td>
</tr>
<tr>
<td>System console (.CONSOLE)</td>
<td>$61</td>
<td>$01</td>
</tr>
<tr>
<td>Graphics screen (.GRAFIX)</td>
<td>$62</td>
<td>$01</td>
</tr>
<tr>
<td>Onboard RS232 (.RS232)</td>
<td>$63</td>
<td>$01</td>
</tr>
<tr>
<td>Parallel card (.PARALLEL)</td>
<td>$64</td>
<td>$01</td>
</tr>
<tr>
<td>Disk III (.D1 through .D4)</td>
<td>$E1</td>
<td>$01</td>
</tr>
<tr>
<td>Profile disk (.PROFILE)</td>
<td>$D1</td>
<td>$02</td>
</tr>
</tbody>
</table>
| Block device formatter;  
  Disk III (.FMTD1 ... .FMTD4) | $11      | $01      |
$23: Device not open (NOTOPEN)

The character device being referenced has not been opened by the file OPEN call.

$25: Resources not available (NORESC)

The device driver is unable to acquire the system resources (such as memory, I/O ports, or interrupts) it needs to operate. This error can also occur during a file OPEN call.

$26: Call not supported on device (BADOP)

The requested SOS call is not supported by the device.

$27: I/O error (IOERR)

The device driver is unable to exchange information with the device, due to a bad storage medium or communication line, or some other cause. If this happens on a flexible disk, remove and replace the disk, and try again.

$28: Device write-protected (NOWRITE)

The medium in this block device is write-protected. Remove the write-protect tab and try again.

$2E: Disk switched (DISKSW)

The medium in the block device has been removed and possibly replaced. This message is merely a warning, and occurs only the first time the call is made: the second time the call is made, it will be executed.

Errors $30 through $3F are returned by individual device drivers, and relate to specific error conditions within those drivers. The error codes generated by a device driver are described in the manual describing that device driver.
11.1 Memory Calls

These calls are used by SOS to allocate memory for interpreters, as explained in section 2.3.

$40$: REQUEST__SEG
$41$: FIND__SEG
$42$: CHANGE__SEG
$43$: GET__SEG__INFO
$44$: GET__SEG__NUM
$45$: RELEASE__SEG

11.1.1 REQUEST__SEG

Memory Call $40$

This call requests the contiguous region of memory bounded by the base and limit segment addresses. A new segment is created if and only if no other segment currently occupies part or all of the requested region of memory.

Required Parameters

**base**: 2 byte value
Range: $0020..$10FF
This is the segment address (bank followed by page) of the beginning of the memory range requested.

**limit**: 2 byte value
Range: $0020..$10FF
This is the segment address of the end of the memory range requested.

**seg_id**: 1 byte value
Range: $00..$7F
This is the segment identification code of the requested segment. The caller can use this parameter to identify the type of information that the segment will contain.

The highest four bits of the *seg_id* identify the owner of the segment:

<table>
<thead>
<tr>
<th>Seg_id_range</th>
<th>Owner</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00$ to $0F$</td>
<td>SOS Kernel</td>
<td>System code</td>
</tr>
<tr>
<td>$10$ to $1F$</td>
<td>Interpreter</td>
<td>Interpreter data</td>
</tr>
<tr>
<td>$20$ to $7F$</td>
<td>User</td>
<td>User program and data</td>
</tr>
</tbody>
</table>

The memory system does not check this parameter to ensure that it is in the proper range.
**seg_num:** 1 byte result  
Range: $01..$1F

If the requested segment is available, this parameter returns the segment number of the segment granted. This number must be used to identify the segment in subsequent calls to CHANGE_SEG, RELEASE_SEG, or GET_SEG_INFO.

**Comments**

Both the base and limit segment addresses must reside in switchable banks $00 through $0E, system bank $0F, or system bank $10. In addition, the base address must be less than or equal to the limit address. If the base and limit segment address parameters fail to meet the above criteria, then the segment will not be allocated and error BADBKPG will be returned.

The ranges for base and limit are not continuous: these are the allowable segment addresses:

- $0020..$009F  
- $0120..$019F  
- $0E20..$0E9F  
- $0F00..$0F1F  
- $10A0..$10FF

SOS can keep track of $1F segments

**Errors**

- $E0: BADBKPG  
  Invalid segment address (bank/page pair)  
- $E1: SEGRQDN  
  Segment request denied  
- $E2: SEGTBLFULL  
  Segment table full

### 11.1.2 FIND_SEG

This call searches memory from high memory down, until it finds the first free space that is **pages** pages long and meets the search restrictions in **search_mode**. If such a space is found, it assigns this free space to the caller as a segment (as in REQUEST_SEG), returning both the segment number and the location in memory of the segment. 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.

**Required Parameters**

- **search_mode:** 1 byte value  
  Range: $00..$02

This parameter selects one of three constraints to place upon the segment search:

- $00: may not cross a 32K bank boundary  
- $01: may cross one 32K bank boundary  
- $02: may cross any 32K bank boundary
**seg_id**: 1 byte value  
Range: $00..$7F

This is the segment identification code of the requested segment. The caller can use this parameter to identify the type of information that the segment will contain.

The highest four bits of the **seg_id** identify the owner of the segment:

<table>
<thead>
<tr>
<th>Seg_id_range</th>
<th>Owner</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00 to $0F</td>
<td>SOS Kernel</td>
<td>System code</td>
</tr>
<tr>
<td>$10 to $1F</td>
<td>Interpreter</td>
<td>Interpreter data</td>
</tr>
<tr>
<td>$20 to $7F</td>
<td>User</td>
<td>User program and data</td>
</tr>
</tbody>
</table>

The memory system does not check this parameter to ensure that it is in the proper range.

**pages**: 2 byte value/result  
Range: $0001..$FFFF

This is the number of contiguous pages to search for. If no free space is found that contains this many pages, then the memory system will return in this parameter the size of the largest free space it can find; the SEGRQDN error is also generated. A page count of $00 always returns error BADPBCNT.

**base**: 2 byte result  
Range: $0020..$0E9F

This is the segment address of the beginning of the new segment.

**limit**: 2 byte result  
Range: $0020..$0E9F

This is the segment address of the end of the new segment.

**seg_num**: 1 byte result  
Range: $01..$1F

This is the segment number of the segment granted. This number must be used to identify the segment in subsequent calls to CHANGE__SEG, RELEASE__SEG, or GET__SEG__INFO.

**Comments**

FIND__SEG does not search the system banks $0F and $10.

The **base** and **limit** parameters both return $0000 if the segment is not granted; even though **pages** returns the length of the largest available segment, **base** and **limit** do not return its location.

**Errors**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E1:</td>
<td>SEGRQDN</td>
</tr>
<tr>
<td></td>
<td>Segment request denied</td>
</tr>
<tr>
<td>$E2:</td>
<td>SEGTBLFULL</td>
</tr>
<tr>
<td></td>
<td>Segment table full</td>
</tr>
<tr>
<td>$E5:</td>
<td>BADSRCHMODE</td>
</tr>
<tr>
<td></td>
<td>Invalid search mode parameter</td>
</tr>
<tr>
<td>$E7:</td>
<td>BADPBCNT</td>
</tr>
<tr>
<td></td>
<td>Invalid pages parameter ($00)</td>
</tr>
</tbody>
</table>
11.1.3 CHANGE_SEG

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.

**Required Parameters**

**seg_num**: 1 byte value
   Range: $01..$1F
   This is the segment number of the segment to be changed.

**change_mode**: 1 byte value
   Range: $00..$03
   The change mode indicates which end (base or limit) of the segment to change, and whether to add or release space at that end.
   
   $00$: Release from the base (decrease size)
   $01$: Add before the base (increase size)
   $02$: Add after the limit (increase size)
   $03$: Release from the limit (decrease size)

**pages**: 2 byte value/result
   Range: $0001..$FFFF
   This is the number of pages to add to or release from the segment. If too many pages are added to or removed from the segment, then the segment is not changed, and the maximum number of pages that can be added or removed in the requested change_mode is returned in this parameter, along with a SEGRQDN error.
Comments

You cannot move both ends of a segment at once.

If the segment was granted by FIND_SEGMENT, a CHANGE_SEGMENT operation will not heed the bank-crossing criterion that was used in finding the segment. If you request a segment that does not cross a bank boundary, then increase it with CHANGE_SEGMENT, the larger segment may cross a bank boundary.

Errors

$E1 SEGRQDN Segment request denied
$E3 BADSEGNUM Invalid segment number
$E6 BADCHGMODE Invalid change mode parameter

11.1.4 GET_SEGMENT_INFO

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

Required Parameters

seg_num: 1 byte value
  Range: $01..$1F

This returns the segment number of an existing segment.

base: 2 byte result
  Range: $0020..$109F

This returns the segment address of the beginning of that segment.

limit: 2 byte result
  Range: $0020..$109F

This returns the segment address of the end of that segment.

pages: 2 byte result
  Range: $0001..$FFFF

This returns the number of pages contained by the segment.

seg_id: 1 byte result
  Range: $00..$7F

This returns the identification code of the segment. The highest four bits of the seg_id identify the owner of the segment:

<table>
<thead>
<tr>
<th>Seg_id_range</th>
<th>Owner</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00 to $0F</td>
<td>SOS Kernel</td>
<td>System code</td>
</tr>
<tr>
<td>$10 to $1F</td>
<td>Interpreter</td>
<td>Interpreter data</td>
</tr>
<tr>
<td>$20 to $7F</td>
<td>User</td>
<td>User program and data</td>
</tr>
</tbody>
</table>

Memory Call $43

GET_SEGMENT_INFO $43

0  $05
1  seg_num
2  value
3  base
4  limit
5  result
6  pages
7  result
8  seg_id
result


11.1.5 GET__SEG__NUM

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

**Required Parameters**

**seg_address**: 2 byte value
Range: $0020..$109F

This is the segment address in question.

**seg_num**: 1 byte result
Range: $01..$1F

This is the segment number of the segment that contains the specified segment address.

**Comments**

You may make a subsequent call to GET__SEG__INFO with the resultant segment number to determine the ownership of that segment.

**Errors**

$E0$: BADDRP  Invalid segment address (bank/page pair)
$E4$: SEGNOTFND  Segment not found
11.1.6 RELEASE__SEG

This call releases the memory occupied by the segment specified by `seg_num`, by removing the segment from the segment table. The space formerly occupied by the released segment is returned to free memory. If `seg_num` equals zero, then all nonsystem segments (those with segment identification codes greater than $0F$) will be released.

### Required Parameters

**seg_num**: 1 byte value

Range: $00..$1F

This is the segment number of the segment to be released. If `seg_num` is $00$, then all segments not owned by SOS are released.

### Errors

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E3</td>
<td>BADSEGNUM    Invalid segment number</td>
</tr>
</tbody>
</table>
11.2 Memory Call Errors

The errors below are generated by SOS memory calls. For other errors, see Appendix D.

$E8: Invalid segment address (BADBKPG)
The segment address has an invalid bank number, page number, or both.

$E1: Segment request denied (SEGRQDN)
No segment can be created that meets the caller's size and boundary criteria.

$E2: Segment table full (SEGTBLFULL)
SOS can keep track of no more segments. Existing segments must be released or consolidated if more segments are needed.

$E3: Invalid segment number (BADSEGNUM)
The seg_num passed is not that of a currently existing segment.

$E4: Segment not found (SEGNOTFND)
For GET_SEGMENT_NUM, no segment in the system contains the segment address specified.

$E5: Invalid search_mode parameter (BADSRCHMODE)
For FIND_SEGMENT, the search_mode parameter is invalid (greater than $02).

$E6: Invalid change_mode parameter (BADCHGMODE)
For CHANGE_SEGMENT, the change_mode parameter is invalid (greater than $03).

$E7: Invalid pages parameter (BADPGCNT)
The pages parameter is invalid (equal to $00).
12.1 Utility Calls

The following system calls deal with the system clock/calendar, the event fence, the analog input ports, and other general system resources.

$60:  SET_FENCE
$61:  GET_FENCE
$62:  SET_TIME
$63:  GET_TIME
$64:  GET_ANALOG
$65:  TERMINATE

12.1.1 SET_FENCE

This call changes the current value of the user event fence to the value specified in the fence parameter.

Required Parameters

fence: 1 byte value
Range: $00..$FF

This parameter contains the new value of the user event fence for the operating system's event mechanism. Events with priority less than or equal to the fence will not be serviced until the fence is lowered.

Errors

No errors are possible.
12.1.2 GET__FENCE

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

*Required Parameters*

- `fence`: 1 byte result
  - Range: $00..$FF

This parameter returns the current setting of the user event fence. Events with priority less than or equal to the fence will not be serviced until the fence is lowered.

*Errors*

No errors are possible.
12.1.3 SET__TIME

This call sets the system clock to the contents of a buffer located at the specified address. If the system has no functioning clock, SET__TIME stores the contents of the buffer as the last valid time, to be returned on the next GET__TIME call.

Required Parameters

time: pointer

This is a pointer to an 18-byte buffer containing the current date and time. The information is specified as an 18-byte ASCII string whose format is

YYYY M M D D X H H N N S S X X X

The meaning of each field is as below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY: Year</td>
<td>1900</td>
<td>1999</td>
<td></td>
</tr>
<tr>
<td>MM: Month</td>
<td>00 or 01</td>
<td>12 (December)</td>
<td></td>
</tr>
<tr>
<td>DD: Date</td>
<td>00 or 01</td>
<td>28, 30, or 31</td>
<td></td>
</tr>
<tr>
<td>X: Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HH: Hour</td>
<td>00(Midnight)</td>
<td>23 (11:00 p.m.)</td>
<td></td>
</tr>
<tr>
<td>NN: Minute</td>
<td>00</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>SS: Second</td>
<td>00</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>XXX: Ignored</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, December 29, 1980, at 9:30 a.m., would be specified by the string "19801229093000000".
**Comments**

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 to make sure each field is in the proper range. The clock makes several restrictions: it rejects any invalid combination of month and date. The clock only accepts dates in the range 1..30 if the month is 4, 6, 9, or 11; it only accepts dates in the range 1..28 if the month is 2: February 29 is always rejected.

**SET__TIME** attempts to set the hardware clock, whether or not it is present and functioning. 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 missing or malfunctioning.

The clock does not roll over the year.

The format of the **SET__TIME** string is the same as that of the **GET__TIME** result, except that **SET__TIME** ignores the day of the week and the millisecond fields.

**Errors**

No errors are possible.

---

**12.1.4 ** **GET__TIME**

**Utility Call $63**

This call reads the time from the system clock and returns it to the buffer located at the specified address. If the system has no functioning clock, **GET__TIME** returns the last known valid time.

**Required Parameters**

**time:** pointer

This is a pointer to an 18-byte buffer containing the current date and time. The information is specified as an 18-byte ASCII string whose format is

```
YYYY MM DD WW HH NN SS UU UU
```

The meaning of each field is as below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>Year</td>
<td>1900</td>
<td>1999</td>
</tr>
<tr>
<td>MM</td>
<td>Month</td>
<td>00 or 01</td>
<td>12 (December)</td>
</tr>
<tr>
<td>DD</td>
<td>Date</td>
<td>00 or 01</td>
<td>28, 30, or 31</td>
</tr>
<tr>
<td>W</td>
<td>Day</td>
<td>01 (Sunday)</td>
<td>07 (Saturday)</td>
</tr>
<tr>
<td>HH</td>
<td>Hour</td>
<td>00 (Midnight)</td>
<td>23 (11:00 p.m.)</td>
</tr>
<tr>
<td>NN</td>
<td>Minute</td>
<td>00</td>
<td>59</td>
</tr>
<tr>
<td>SS</td>
<td>Second</td>
<td>00</td>
<td>59</td>
</tr>
<tr>
<td>UUU</td>
<td>Millisecond</td>
<td>0000</td>
<td>9999</td>
</tr>
</tbody>
</table>

For example, Friday, March 21, 1980, at 1:27:41.001 p.m., would be returned as "19800321132741001".

**Comments**

If the hardware clock is not operational, the utility manager retrieves the last known valid time from system RAM. If no last known valid time is stored, **GET__TIME** returns a string of eighteen ASCII zeros: "000000000000000000".
SOS calculates the day of the week from the year, month, and date.

The clock will only generate dates in the range 1..30 if the month is 4, 6, 9, or 11; it will only generate dates in the range 1..28 if month is 2; February 29 will never be generated by a system with a functioning clock. A system without a functioning clock can return February 29 if that month and date have been set by a SET__TIME call.

The clock does not roll over the year.

You must ensure that the buffer pointed to by time can hold all eighteen ($12) bytes, to avoid overwriting other data.

**Errors**

No errors are possible.

### 12.1.5 GET__ANALOG

**Utility Call $64**

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.

**Required Parameters**

joy_mode: 1 byte value

Range: $00..$07

This parameter specifies the joystick inputs to be read. For each value of joy_mode, the following inputs will be read:

<table>
<thead>
<tr>
<th>Joy_mode</th>
<th>Port</th>
<th>Buttons/Switches</th>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>B</td>
<td>JS0-B, JS0-Sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$01</td>
<td>B</td>
<td>JS0-B, JS0-Sw</td>
<td>JS0-X</td>
<td></td>
</tr>
<tr>
<td>$02</td>
<td>B</td>
<td>JS0-B, JS0-Sw</td>
<td></td>
<td>JS0-Y</td>
</tr>
<tr>
<td>$03</td>
<td>B</td>
<td>JS0-B, JS0-Sw</td>
<td>JS0-X</td>
<td>JS0-Y</td>
</tr>
<tr>
<td>$04</td>
<td>A</td>
<td>JS1-B, JS1-Sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$05</td>
<td>A</td>
<td>JS1-B, JS1-Sw</td>
<td>JS1-X</td>
<td></td>
</tr>
<tr>
<td>$06</td>
<td>A</td>
<td>JS1-B, JS1-Sw</td>
<td></td>
<td>JS1-Y</td>
</tr>
<tr>
<td>$07</td>
<td>A</td>
<td>JS1-B, JS1-Sw</td>
<td>JS1-X</td>
<td>JS1-Y</td>
</tr>
</tbody>
</table>

The names for these variables are those used in the *Apple III Owner's Guide*, Appendix C. These eight variables are returned by the joy_status parameter.
**joy_status**: 4 byte result
Range: $00000000..$FFFFFFF

This 4-byte field is treated as one parameter by SOS. Here we subdivide it into four 1-byte fields for clarity: \( n \) represents the numbers of the joystick (1 or 2) as determined by the **joy_mode** parameter.

**JSn-B**: 1 byte result
Range: $00..$FF
This digital output returns $00 if the button is off and returns $FF if the button is on.

**JSn-Sw**: 1 byte result
Range: $00..$FF
This digital output returns $00 if the switch is off and returns $FF if the switch is on.

**JSn-X**: 1 byte result
Range: $00..$FF
This analog output returns a value from $00 to $FF corresponding to the horizontal position of the joystick. A position that was not read (due to the **joy_mode** parameter) returns a byte of $00.

**JSn-Y**: 1 byte result
Range: $00..$FF
This analog output returns a value from $00 to $FF corresponding to the vertical position of the joystick. A position that was not read (due to the **joy_mode** parameter) returns a byte of $00.

**Comments**
An input device other than a joystick can be read, provided (a) it uses the same pins for analog and digital inputs, and (b) each pin produces the correct signals, as described in the *Apple III Owner's Guide*.

Both buttons of the selected joystick are always read and returned.

Reading the analog inputs slows down the execution speed of this call and should be avoided when unnecessary.

**JSn-B, JSn-Sw, JSn-X, and JSn-Y** all return results of $FF if no joystick is attached to the port.

The XNORESRC error will be generated if an attempt is made to read Port A and a device driver (such as the Siltype driver) has already claimed the use of that port.

The **parm_count** is $02, not $05.

**Errors**

$25  XNORESRC  Resource not available
$70  BADJMODE  Invalid joystick mode
12.1.6 TERMINATE

This call clears memory, clears the screen, and displays INSERT SYSTEM DISKETTE & REBOOT in 40-column black-and-white text mode on the screen. The system then hangs, and waits for the user to press CONTROL-RESET and reboot.

Required Parameters

None

Comments

Only the SOS Call Block is shown for this call. Since this call has no parameters, the parameter_count is $00. Thus the parameter_list pointer must point to a byte containing $00. The most convenient such byte is the BRK opcode beginning the TERMINATE call, so this call customarily bites its own tail.

Before issuing a TERMINATE call, the interpreter should close all open files. This will ensure that all I/O buffers are written out, and all file entries updated, while the necessary information still exists.

This call is the recommended way to leave a program. It provides a clean exit to a program, and leaves no traces of it in memory for the user's examination. It can be used in conjunction with a copy-protection scheme to protect a program from piracy. It also provides a hook that could be used to return control to a future command interpreter.

Errors

No errors are possible. This is an excellent call for beginners.
12.2 Utility Call Errors

One error can be generated by one of the utility calls; other errors are listed in Appendix D.

$78: Invalid Joystick Mode (BADJMODE)
The `joy_mode` parameter is greater than $07.
Version: SOS 1.1, 1.2 and 1.3

Classification:
- Single-task, configurable, interrupt-driven operating system.
- File system—hierarchical, tree file structure.
- Device-Independent I/O.

CPU Architecture:
- Address enhanced 6502 instruction set.
- Supports both bank-switched and enhanced indirect addressing.
- Separate execution environments for user and SOS including private zero and stack pages.

System Calls:
- Based on 6502 BRK instruction, pointer, and value parameter types.
- Error codes returned via A register.
- All other CPU registers preserved upon return.
- Optional parameter lists for future expansion.

File Management System:
- Hierarchical file structure.
- Pathname prefix facility.
- Byte-oriented file access to both directory/user files and device files.
- Dynamic, non-contiguous file allocation on block devices.
- Automatic buffering (current index block and data block).
- Dynamic memory allocation of file buffers.
- Block size (512 bytes).
- File protection: rename/destroy/read/write access attributes.
- File level assignment on Open.

Automatic date/time stamping of files.
Automatic volume logging/swapping, supported by system message center.
Multiple volumes per block device can be "open" simultaneously.
Sparse file capability:
- maximum number of active volumes = 8
- maximum disk size = 32 Mbytes
- maximum user file size = 16 Mbytes
- maximum file entries in volume directory = 51
- maximum file entries in a subdirectory = 1663
- file names — maximum 15 characters
- pathnames — maximum 128 characters

File system calls:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>READ</td>
</tr>
<tr>
<td>DESTROY</td>
<td>WRITE</td>
</tr>
<tr>
<td>RENAME</td>
<td>CLOSE</td>
</tr>
<tr>
<td>SET_FILE_INFO</td>
<td>FLUSH</td>
</tr>
<tr>
<td>GET_FILE_INFO</td>
<td>SET_MARK</td>
</tr>
<tr>
<td>VOLUME</td>
<td>GET_MARK</td>
</tr>
<tr>
<td>SET_PREFIX</td>
<td>SET_EOF</td>
</tr>
<tr>
<td>GET_PREFIX</td>
<td>GET_EOF</td>
</tr>
<tr>
<td>OPEN</td>
<td>SET_LEVEL</td>
</tr>
<tr>
<td>NEWLINE</td>
<td>GET_LEVEL</td>
</tr>
</tbody>
</table>

Device Management System:

- Block and character device classes.
- Standardized interface for block and for character devices.
- All devices are named and configurable.
Support for synchronous, interrupt, and DMA-based I/O.
  maximum number of devices = 24
  maximum number of block devices = 12

Device system calls:
  GET_DEV_NUM   D_STATUS
  D_INFO        D_CONTROL

Memory/Buffer Management System:
  All memory allocated as segments.
  Supports maximum of 512 Kbytes RAM.
  System buffers allocated and released dynamically.
  System buffer checksum routine for data integrity.

Memory system calls:
  REQUEST_SEG   GET_SEG_INFO
  FIND_SEG      GET_SEG_NUM
  CHANGE_SEG    REL_SEG

Additional System Functions:
  System clock/calendar
    (year/month/day/weekday/hour/minute/second/millisecond).
  Joysticks: reads X and Y axes, pushbutton, and switch.
  TERMINATE call provides clean program termination and clears memory.

System calls:
  SET_TIME      TERMINATE
  GET_TIME      GET_ANALOG

Interrupt Management System:
  Receives hardware interrupts (IRQ, NMI) and system calls (BRK).
  Hardware resource allocation and deallocation.
  Dispatches to driver interrupt handlers.

Event Management System:
  Priority-based event signaling.
  Event handlers preempted by higher priority events.
  Events with equal priorities process FIFO.
  Event fence delays events with priority less than fence.
  Event system calls:
    SET_FENCE        GET_FENCE

System Configuration:
  Menu-driven system-configuration editor (System Configuration Program).
  Can add, remove, and modify drivers and can select the keyboard-layout and system-character-set tables.

Standard Device Drivers:
  Floppy disk (.D1, .D2, .D3, .D4)
    143,360 bytes (formatted) per volume.
    Automatically reports mounting of a new volume.
    Built into SOS kernel.
  Console (.CONSOLE)
    Interrupt-driven keyboard (supports type-ahead).
    Configurable keyboard-layout table (via SCP).
    Raw-keystroke and no-wait input modes.
    Event handler supports anykey and attention character.
    Optional screen echoing.
Console control modes:
- video on/off
- flush type-ahead buffer
- suspend screen output
- display control characters
- flush screen output

Cursor positioning commands.

Viewport set, clear, save, and restore commands.

Horizontal and vertical scrolling.

Text modes: 24 × 80 and 24 × 40 B&W and 24 × 40 color (normal and inverse).

Configurable system character set table (via SCP).

Character set can be changed under program control at any time.

Screen read command.

Graphics (.GRAFIX)

Displays graphical and textual information simultaneously.

Graphics modes:
- 560 × 192 and 280 × 192 in B&W video.
- 280 × 192 and 140 × 192 in 16 colors.

Point-plotting and line-drawing commands using graphics viewport and pen.

Raster block picture operations.

Color operator table, controls color overwrite.

Transfer modes allow binary operations on the drawing color and the current screen color.

Allows use of either the system character set or an alternate character set to display ASCII text on the screen.

Single or dual graphics screens.

General purpose communications (.RS232)

RS-232-C interface.

Configurable data rates from 110 to 9600 baud.

Configurable protocols, including XON/XOFF, ETX/ACK, and ENQ/ACK.

Interrupt-driven, buffered, bi-directional data transfer.

Hardware handshaking option.

Serial printer (.PRINTER)

RS-232-C interface.

Configurable data rates from 110 to 9600 baud.

Interrupt-driven and buffered (output only).

Hardware handshaking option.

Audio (.AUDIO)

64 volume levels.

Produces tones from 31 to 5090 Hz (over 7 octaves).

Duration range from 0 to 5 sec (increments of 1/60 sec).
B.1 Using ExerSOS
B.1.1 Choosing Calls and Other Functions
B.1.2 Input Parameters
B.2 The Data Buffer
B.2.1 Editing the Data Buffer
B.3 The String Buffer
B.4 Leaving ExerSOS
ExerSOS is a interactive BASIC program that lets you make SOS calls from the keyboard without writing a special assembly-language program to test each call. It is intended to let you try out calls to see how they work. ExerSOS lets you choose a call from a menu, then prompts you for each of the call's input parameters, and gives you the correct output parameters or error message.

### B.1 Using ExerSOS

To use ExerSOS, insert the ExerSOS disk into the built-in drive and press CONTROL-RESET. After the introductory displays you will see the Main Menu.

#### B.1.1 Choosing Calls and Other Functions

The Main Menu presents you with a choice of functions. Typing 0 will EXIT ExerSOS. The first 35 of these functions are SOS calls (listed below by type). The remainder are special functions available within ExerSOS. The full list of functions is:

- **File Calls:**
  - $C0$: CREATE
  - $C1$: DESTROY
  - $C2$: RENAME
  - $C3$: SET_FILE_INFO
  - $C4$: GET_FILE_INFO
  - $C5$: VOLUME
  - $C6$: SET_PREFIX
  - $C7$: GET_PREFIX
  - $C8$: OPEN
  - $C9$: NEWLINE
  - $CA$: READ
  - $CB$: WRITE
  - $CC$: CLOSE
  - $CD$: FLUSH

- **Device Calls:**
  - $D0$: SET_MARK
  - $D1$: GET_MARK
  - $D2$: SET_EOF
  - $D3$: GET_EOF
  - $D4$: SET_LEVEL
  - $D5$: GET_LEVEL

- **Memory Calls:**
  - $E0$: REQUEST_SEG
  - $E1$: FIND_SEG
  - $E2$: CHANGE_SEG
  - $E3$: GET_SEG_INFO
  - $E4$: GET_SEG_NUM
  - $E5$: RELEASE_SEG

- **Utility Calls:**
  - $F0$: SET_FENCE
  - $F1$: GET_FENCE
  - $F2$: SET_TIME
  - $F3$: GET_TIME
  - $F4$: GET_ANALOG

- **ExerSOS Utilities:**
  - $1$: Display Directory
  - $2$: Display Open Files
  - $3$: Display Active Memory Segments
  - $4$: Display/Edit Contents of Data Buffer
B.1.2 Input Parameters

When you select a SOS call from the Main Menu, the display is replaced by a split-screen menu showing the name of the call at the top. The left half of the screen is used for typing input parameters to the call; the right is used to show the resultant SOS call error and any output parameters. You will then be prompted for each input parameter, following the description of the call in the SOS Manual. If you wish to return to the Main Menu, type a backslash (\) and press RETURN.

All parameters have the same names as in this manual, and appear in the same order as in the description of the SOS call in Volume 2. Pointer parameters, however, are omitted, as all values and results are passed interactively, rather than by building a table in memory and passing its address.

In some cases, a range of legal values is displayed; if your entry falls outside that range, you will be prompted again. For example, the first prompt you encounter in the READ call is

```
ref_num [0..255]
```

If you respond to this with an out-of-range value, the prompt will be repeated.

You may also type data in hexadecimal by preceding a value with a dollar sign ($). Some input fields have a fixed dollar sign: these fields require hex input. SOS calls requiring no input display

```
(None)
```

before reporting the results of the call.

When typing an input parameter, you can use the ESCAPE key to edit the input, as in BASIC.

Several SOS calls employ an optional parameter list along with a \textbf{length} parameter. For those calls, ExerSOS asks you for the \textbf{length} and selectively prompts or displays information as requested.

B.2 The Data Buffer

ExerSOS maintains two buffers you should be aware of: the data buffer and the string buffer. ExerSOS alone locates the 16K data buffer in memory. All I/O operations (READ, WRITE) use the data buffer. Hence, a READ call followed by a WRITE call will transfer bytes from one file to another.

In order to ensure the return of this 16K space to the system, always exit ExerSOS through the Main Menu, never by typing CONTROL-C. If you should accidentally exit ExerSOS, reboot by pressing CONTROL-RESET.

B.2.1 Editing the Data Buffer

The Display/Edit function allows you to select any of the 64 256-byte pages of memory occupied by the data buffer, and displays that page in hex with the ASCII equivalents on the right side of the screen. You are then placed in Edit mode with the cursor (denoted by matching "[..]") positioned in the upper-right corner. You can move the cursor through the use of the four arrow keys.

You can alter the contents of a byte by typing a hex digit, (that is, 0..9, A..F, a..f). Note that as you do so, the value you type is placed in the low-order nibble of the target byte, and the value that was in the low-order nibble moves to the high-order nibble. You may terminate the input to a byte by pressing RETURN, which accepts the new value, or ESCAPE, which restores the original value.

If you press ESCAPE while you are in the cursor-positioning phase, you exit from Edit mode and have the choice of returning to the Main Menu or displaying another page of the buffer.
### B.3 The String Buffer

The string buffer is used by many of the calls as temporary storage any time a pathname or device name is passed into or out of a SOS call. Additionally, the D__STATUS and D__CONTROL calls use the string buffer for the STATUS__LIST and CONTROL__LIST, respectively.

The following SOS calls require some further user input:

**D__STATUS**

In addition to the SOS-required input parameters, ExerSOS prompts you for two more items. The first prompt,

**Initialize Buffer [Y/N]**

lets you initialize the string buffer by typing Y, or leave its current contents intact by typing N. Usually, you will initialize it, to make sure no garbage from a previous call obscures your results. However, in some cases, you may wish to make a status call, then change something with a control call, then check the buffer with a status call again: in such a case do not initialize the buffer.

The second prompt,

**Amount of output**

asks you how many bytes of the string buffer you wish to see. If you specify more bytes than are in the status list, the remaining bytes will be either zeros or garbage, depending on your response to the "Initialize?" prompt.

**D__CONTROL**

After you specify the dev_num and control_code, ExerSOS allows you to specify the control list from either of two places. If you type a "0" to the "Length of input" prompt, the call is made from the current value of the string buffer. If you respond to the prompt with a value larger than 0, you are prompted for each byte of the control list. The resultant string is moved into the string buffer.

### B.4 Leaving ExerSOS

To leave ExerSOS, return to the Main Menu and type 0. You will be asked to confirm your intention: type Y to exit (any other reply will return you to the Main Menu). ExerSOS will drop into BASIC, and you will be able to run another BASIC program, or reboot by pressing CONTROL-RESET. If you leave ExerSOS inadvertently, as by typing CONTROL-C, you should reboot. If you try to RUN the program without rebooting, you will have lost the 16K space allocated to the data buffer.
MakelInterp
MakInterp is a program that takes an assembly-language code file produced by the Apple III Pascal Assembler and converts it to the proper format for a bootable SOS.INTERP file. If you are writing an interpreter, this makes it unnecessary for you to know the details of interpreter file format, and protects you from future changes in this format.

To use MakInterp, boot Pascal and insert the ExerSOS disk into, say, .D2. Now execute (that is, type X)

.D2/MAKEINTERP.CODE

Then type the input pathname, the name of the interpreter code file, for example,

.D2/INTERP.CODE

and the output pathname, say,

.D2/SOS.INTERP

As the disk spins, you see this message displayed:

Converting Files

When the conversion is complete, MakInterp displays the message

Files converted

and returns you to the Pascal command line.

All pathnames must be complete, with suffix. If you type any invalid input, you will have to execute the program again.
SOS detects two types of errors:

- Non-fatal SOS errors, occurring during a SOS call, that are detected and flagged;
- Fatal SOS errors, occurring during a SOS call or interrupt sequence, that signal such a substantial irregularity that the system cannot continue to operate.

In addition, the SOS bootstrap loader detects bootstrap errors, which occur only when the system is starting up.

The reporting mechanism for non-fatal SOS errors is discussed in Volume 1, section 8.4. The error code is returned in the accumulator after a SOS call: an error code of $00 means no error was encountered in the call. The error code is normally used by the interpreter to display a message to the user, to repeat an operation, or to take some other action.

Bootstrap errors and fatal errors occur when an error condition is so critical that no recovery is possible. These errors cause their own messages to be displayed on the screen, as no interpreter is in place to interpret them. These errors are discussed in detail in section D.3.

### D.1 Non-Fatal SOS Errors

Explanations of the general system errors are given in section 8.4 of Volume 1. Explanations of the other non-fatal system errors are given in Volume 2. The list below, numerically ordered, is for easy reference. Three things are listed for each error: the error number, a suggested name for the assembly-language routine handling the error, and a suggested error message for the interpreter to display on the screen.

#### D.1.1 General SOS Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$01</td>
<td>BADSCNUM</td>
</tr>
<tr>
<td>$02</td>
<td>BADCZPAGE</td>
</tr>
<tr>
<td>$03</td>
<td>BADXBYTE</td>
</tr>
<tr>
<td>$04</td>
<td>BADSCPCNT</td>
</tr>
<tr>
<td>$05</td>
<td>BADSCBNDX</td>
</tr>
</tbody>
</table>

#### D.1.2 Device Call Errors

(See section 18.2)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>DNFERR</td>
</tr>
<tr>
<td>$11</td>
<td>BADNUM</td>
</tr>
<tr>
<td>$20</td>
<td>BADREQCODE</td>
</tr>
<tr>
<td>$21</td>
<td>BADCTLCODE</td>
</tr>
<tr>
<td>$22</td>
<td>BADCTLPARM</td>
</tr>
<tr>
<td>$23</td>
<td>NOTOPEN</td>
</tr>
<tr>
<td>$25</td>
<td>NORESRC</td>
</tr>
<tr>
<td>$26</td>
<td>BADOP</td>
</tr>
<tr>
<td>$27</td>
<td>IOERROR</td>
</tr>
<tr>
<td>$2B</td>
<td>NOWRITE</td>
</tr>
<tr>
<td>$2E</td>
<td>DISKSW</td>
</tr>
<tr>
<td>$30..$3F</td>
<td></td>
</tr>
</tbody>
</table>

#### D.1.3 File Call Errors

(See section 9.2)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$40</td>
<td>BADPATH</td>
</tr>
<tr>
<td>$41</td>
<td>CFBCFBB</td>
</tr>
<tr>
<td>$42</td>
<td>FCBFULL</td>
</tr>
<tr>
<td>$43</td>
<td>BADREFNUM</td>
</tr>
<tr>
<td>$44</td>
<td>PATHNOTFND</td>
</tr>
<tr>
<td>$45</td>
<td>VNFERR</td>
</tr>
<tr>
<td>$46</td>
<td>FNFERR</td>
</tr>
<tr>
<td>$47</td>
<td>DUPERR</td>
</tr>
<tr>
<td>$48</td>
<td>OVRERR</td>
</tr>
<tr>
<td>$49</td>
<td>DIRFULL</td>
</tr>
<tr>
<td>$4A</td>
<td>CPTERR</td>
</tr>
<tr>
<td>$4B</td>
<td>TYPERR</td>
</tr>
<tr>
<td>$4C</td>
<td>EOFERR</td>
</tr>
<tr>
<td>$4D</td>
<td>POSNERR</td>
</tr>
<tr>
<td>$4E</td>
<td>ACCSERR</td>
</tr>
<tr>
<td>$4F</td>
<td>BTSEERR</td>
</tr>
<tr>
<td>$50</td>
<td>FILBUSY</td>
</tr>
<tr>
<td>$51</td>
<td>DIRERR</td>
</tr>
<tr>
<td>$52</td>
<td>NOTSOS</td>
</tr>
<tr>
<td>$53</td>
<td>BADSTCNT</td>
</tr>
<tr>
<td>$55</td>
<td>BUFTBLFULL</td>
</tr>
</tbody>
</table>
$56: $56: BADSYSBUF  Invalid system buffer parameter
$57: DUPVOL  Duplicate volume
$58: NOTBLKDEV  Not a block device
$59: LVLERR  Invalid level
$5A: BITMAPADDR  Invalid bit map address

D.1.4 Utility Call Errors
(See section 12.2)

$70: BADJOYMODE  Invalid joy_mode parameter

D.1.5 Memory Call Errors
(See section 18.2)

$E0: BADBKPG  Invalid segment address
$E1: SEGRQDN  Segment request denied
$E2: SEGTBLFULL  Segment table full
$E3: BADSEGNUM  Invalid segment number
$E4: SEGNOTFND  Segment not found
$E5: BADSRCMODE  Invalid search_mode parameter
$E6: BADCCHMODE  Invalid change_mode parameter
$E7: BADPGCOUNT  Invalid pages parameter

D.2 Fatal SOS Errors

If SOS encounters an internal error from which it cannot recover, it displays an error message (including the code number of the error that occurred) on the screen, beeps the speaker, and hangs. The only recovery possible is to reboot.

The fatal error codes and conditions are listed below. The phrase following the number is a convenient name for the error, but no interpreter will be able to display it to the user, as SOS will not be around to help.

$01: Invalid BRK (BADBRK)
A BRK software interrupt was encountered within SOS. As SOS is not reentrant, it is not allowed to make SOS calls to itself; making such a call is an unrecoverable error and means that the memory region containing SOS has been scrambled.

$02: Invalid interrupt (BADINT)
An interrupt occurred that cannot be acknowledged by SOS. The 6502's IRQ or NMI line was pulled down, but either polling did not reveal the device that performed the interrupt, or no device driver had claimed that interrupt.

$04: Invalid NMI (NMIHANG)
A request was made for SOS to lock the RESET/NMI key, but a device is currently attempting to perform a NMI interrupt. If the interrupt is not granted and handled within a short time after the request to lock NMI was made, this error will occur.

$05: Event queue overflow (EVQOVFL)
More events (see Chapter 6) have occurred than have been handled. Possibly the event fence is set too high, and few events are being handled.

$06: SOS stack overflow (STKOVFL)
The SOS stack has been pushed to more than 256 bytes, and the data at the bottom of the stack have been overwritten.

$07: Invalid control or status request (BADSYSCALL)
The device system has detected an invalid control or status request.

$08: Too many drivers (MCTOVFL)
Too many device drivers have been created for SOS to keep track of.

$09: Memory too small (MEM2SML)
The Apple III's memory is too small for SOS to operate in; that is, less than 128K bytes.
$0A: Buffer Control Block damaged (VCBERR)
The file system’s Buffer Control Block has been damaged due to a memory failure.

$0B: File Control Block damaged (FCBERR)
The file system’s File Control Block has been damaged due to a memory failure.

$0C: Invalid allocation blocks (ALCERR)
Allocation blocks are invalid.

$0E: Pathname too long (TOOLENG)
A pathname supplied or internally generated contains more than 256 characters. This can result from concatenating a long prefix to a long filename.

$0F: Invalid buffer number (BADBUFNUM)
An internal buffer allocation request has supplied an invalid buffer number.

$10: Invalid buffer size (BADBUFSIZE)
An internal buffer allocation request has supplied an invalid buffer size.

D.3 Bootstrap Errors

If an error occurs during the bootstrap operation, an error message is displayed (in uppercase inverse characters) in the middle of the video screen, the speaker beeps, and the system hangs. Bootstrap errors are not SOS errors, as they occur before SOS has started running; for this reason, they are not numbered. Any bootstrap error is a fatal error; you must insert a proper boot diskette, then hold down the CONTROL key and press the RESET button to reboot.

The following errors can be produced during a bootstrap operation:

**DRIVER FILE NOT FOUND**
There is no file named SOS.DRIVER listed in the volume directory of the boot disk. SOS cannot operate without device drivers, and the drivers must be stored in a file with this name in the volume directory of the disk.

**DRIVER FILE TOO LARGE**
The SOS.DRIVER file is too large to fit into the system’s memory along with the interpreter. Use the System Configuration Program to remove some drivers from this file.

**EMPTY DRIVER FILE**
The SOS.DRIVER file contains no device drivers. SOS requires at least one device driver, .CONSOLE, to operate.

**INCOMPATIBLE INTERPRETER**
The interpreter is either too large or specifies a loading location that conflicts with SOS. This error usually occurs when trying to load an older interpreter with a newer version of SOS.

**INTERPRETER FILE NOT FOUND**
There is no file named SOS.INTERP listed in the volume directory of the boot disk. SOS cannot operate without an interpreter, and the interpreter must be stored in a file with this name, in the volume directory of the disk.

**INVALID DRIVER FILE**
The SOS.DRIVER file is not in the proper format for a driver file. Make sure that the file was created by the System Configuration Program or obtained from a valid Apple III boot disk.

**I/O ERROR**
The loader encountered an I/O error while trying to read the kernel, interpreter, or driver file from the disk in the Apple III’s internal disk drive. Make sure the correct disk is properly inserted in that drive.
KERNEL FILE NOT FOUND

No file named SOS.KERNEL is listed in the volume directory of the boot disk. The files SOS.KERNEL, SOS.INTERP, and SOS.DRIVER must all be present in the volume directory of a disk to be booted.

ROM ERROR: PLEASE NOTIFY YOUR DEALER

Your Apple III contains an older version of the bootstrap ROM that is not supported by this version of SOS. Your Apple dealer should be able to replace the ROM at no cost. If you receive this message, please contact your dealer or nearest Apple Service Center.

 TOO MANY BLOCK DEVICES

The SOS.DRIVER file contains too many device drivers for block devices. Use the System Configuration Program to remove some of the block device drivers from this file.

 TOO MANY DEVICES

The SOS.DRIVER file, while small enough to fit into memory, contains too many device drivers for SOS to keep track of. Use the System Configuration Program to remove some drivers from this file.

Data Formats of Assembly-Language Code Files

E.1 Code File Organization
E.2 The Segment Dictionary
E.3 The Code Part of a Code File
E.3.1 The Procedure Dictionary
E.3.2 Procedures
E.3.3 Assembly-Language Procedure Attribute Tables
E.3.4 Relocation Tables
E.3.4.1 Base-Relative Relocation Table
E.3.4.2 Segment-Relative Relocation Table
E.3.4.3 Procedure-Relative Relocation Table
E.3.4.4 Interpreter-Relative Relocation Table
Interpreters can load additional code modules. When you write an interpreter, you may want to make these code modules relocatable. This appendix describes the relocation information generated by the Apple III Pascal Assembler.

Appendix E is derived from the *Apple III Pascal Technical Reference Manual*. Read that manual if you want more detailed information.

Most of the information about assembly-language code files described in the *Apple III Pascal Technical Reference Manual* is addressed to Pascal programmers. However, if you want to use Pascal Assembler code files when you write an interpreter, you need to deal with only two general areas: the overall organization of the code file, and the data structures generated for various pseudo-opcodes by the Pascal Assembler.

### E.1 Code File Organization

An assembly-language code file consists of a segment dictionary and a code part, as shown in Figure E-1:

![Figure E-1. An Assembly-Language Code File](image)

The first block of a code file generated by the Pascal Assembler is in the standard format for block 0 of a Pascal code file; this block is called the *segment dictionary*. The remaining blocks of the file constitute the code part of the code file, which is a single code segment in this kind of file. The code part is followed by linker information: in an assembly-language code file, this information is unused.

Be especially careful in reading this section: words (two bytes of data) are used as well as bytes. Be sure you know which type each number refers to.
E.2  The Segment Dictionary

Since the code part is a single segment, most of the information in the segment dictionary is unused. Figure E-2 shows the information that is used.

<table>
<thead>
<tr>
<th>Byte</th>
<th>High Byte</th>
<th>Low Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CODEADDR = 1 &lt;br&gt; (relative block number)</td>
<td>(segment 1)</td>
</tr>
<tr>
<td>7</td>
<td>CODELENG &lt;br&gt; (in bytes)</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

Figure E-2. A Segment Dictionary

Two 2-byte fields in this block are relevant when you write a module loader. The first starts at byte 4 and is the starting block number (relative to the beginning of the file) of the code generated by the Pascal Assembler; call this CODEADDR, because that is the field name in the Pascal declaration. The second starts at byte 6 and is the length, in bytes, of the code; call it CODELENG, for the same reason.

Your loading routine should begin loading at the relative block number (usually 1) indicated by CODEADDR, and should load the number of bytes indicated by CODELENG.

E.3  The Code Part of a Code File

Following the segment dictionary is the code part, which contains the procedure dictionary and the procedures themselves. This is diagrammed in Figure E-3.

Figure E-3. The Code Part of a Code File
E.3.1 The Procedure Dictionary

The low byte of the last word of the procedure dictionary is at the address CODELENG-2; the structure grows down toward lower addresses, as shown in Figure E-3. To decipher the structure, look at the word whose location is calculated by CODEADDR * 512 + CODELENG-2. The low byte should contain 1. The high byte tells you the number of procedures in the code file. Each use of the pseudo-opcodes .PROC or .FUNC increments this number. Below this word is a sequence of words that contain self-relative pointers to the last word of each procedure in the code file.

A self-relative pointer contains the absolute distance, in bytes, between the low byte of the pointer and the low byte of the word to which it points. To find the address referred to by a self-relative pointer, subtract the value of the pointer from the address of its location.

The number of a procedure is an index into the procedure dictionary: the nth word in the dictionary (counting down from higher addresses) contains a pointer to the top (high address) of the code of procedure number n. As ϕ is not a valid procedure number, the ϕth word of the dictionary is used to store a Pascal-specific descriptor (usually 1) and the number of procedures in the code file (as described above).

E.3.2 Procedures

Each procedure consists of two parts: the procedure code, and the procedure attribute table. The procedure code is contained in the lower portion of the procedure and grows upward toward the higher addresses.

E.3.3 Assembly-Language Procedure Attribute Tables

A procedure's attribute table provides information needed to execute the procedure. Procedure attribute tables are pointed to by entries in the procedure dictionary of each code file.

The format of the attribute table of an assembly-language procedure is illustrated in Figure E-4.

The other type of attribute table is described in the Apple III Pascal Technical Reference Manual.
The highest word in the attribute table of an assembly-language procedure always has 0 in its PROCEDURE NUMBER field. This 0 can be used as a flag to indicate to your loading routine that relocation references may need changing to agree with the other information in the attribute table. The RELOCSEG NUMBER field must contain 0.

The second-highest word of the attribute table is the ENTER IC field: a self-relative pointer to the first executable instruction of the procedure. Following this are four relocation tables; from high address to low address, they are base-relative, segment-relative, procedure-relative, and interpreter-relative.

### E.3.4 Relocation Tables

A relocation table is a sequence of records that contain information necessary to relocate any relocatable addresses used by code within the procedure. These addresses must be relocated whenever the code file containing the procedure is loaded into or moved within memory.

The format of all four relocation tables is the same: the highest word of each table specifies the number of entries (possibly 0) that follow at the lower addresses in the table. The remainder of each table contains the one-word self-relative pointers to locations in the procedure code that must be changed by the addition of the appropriate relative relocation constant, which is known to your interpreter when the code is loaded.

#### E.3.4.1 Base-Relative Relocation Table

Every reference to a label associated with the pseudo-opcodes .PUBLIC and .PRIVATE generates an entry into this table. In the Pascal environment, these opcodes flag references to data global to the Pascal program.

#### E.3.4.2 Segment-Relative Relocation Table

References to labels associated with .REF generate segment-relative relocation entries. The offsets in this table are relative to the beginning of the code portion of the code file: the address of the lowest byte of the code module is added to each of the addresses pointed to in the relocation table. Additionally, references to .PROC or .FUNC names generate entries into this table.

#### E.3.4.3 Procedure-Relative Relocation Table

Addresses pointed to by the procedure-relative relocation table must be relocated relative to the lowest address of the procedure. The address of the lowest byte in the procedure must be added to the contents of the words pointed to in the relocation table. The relevant Assembler directives are .BYTE, .WORD, .BLOCK, and .ASCII. Additionally, any non-relative reference (that is, JMP or LDA, but not BNE or BCS) generates an entry into this table.

#### E.3.4.4 Interpreter-Relative Relocation Table

Entries into this table are generated by references to labels defined by the .INTERP pseudo-opcode. The Pascal System uses this to index into a jump table in the interpreter.
These manuals, in addition to the present one, explain the workings of the Apple III and its system software:


These books explain 6502 assembly-language programming:


Index

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

A
absolute
  code 120
  mode 29
  modules 143
  or relocatable format 143
access 63, 68, 81, 84, 88, 90, [11], [18]
  data 10, 27, 29–32
  path(s) 52
    information 64–66
    maximum number of 53
    multiple 52
    techniques 27–38
accessing
  a logical device 41
  zero page and stack, warning 17
ACCSERR [55]
accumulator 110
ADC 31
address(es) 15
  bank-switched 10, 12, 30, 32
  bus 10
  conversion 25, 32–35
  example 122
current-bank 12, 38
extended 13, 38
  notation 15
extension, pointer 154–159
invalid 13
limit 122
notation
  bank-switched 15
  extended 15
  segment 23–27
  of blocks 96, 97
  of event handler 108
  relocatable [138]
  risky 15
  risky regions 32
  S-bank 12, 38
  segment 24, 38
  notation, S-bank 25
three-byte 13
two-byte 12
addressing
  bank-switched memory 10–13, 30–31
  enhanced indirect 10, 13–16, 31–32
  indirect-X 13
  indirect-Y 13
modes 10–16
enhanced 8
module 27–29
normal indirect 14
restrictions 15
subsection 27–29
ALCERR [128]
algorithm 32
reading a directory file 91–92
incrementing a pointer 36–37
sample 27
allocate memory 25
allocation 7, 23
of a segment of memory 121
scheme, block 95
analog inputs 113
AND 31
Apple III, overview of 3–8
Apple III Pascal Assembler 145, [132], [134]
Apple III Processor xvii
arming events 108, 125
ASCII [139]
ASCII equivalents [117]
Assembler, Apple Pascal 145, [132], [134]
assembly language 5
code file(s) [131–139]
data formats for relocatable 146
module 19, 118, 143–146
linking 145
loading 145
procedure [136]
attribute tables [136], [137]
programming xvii
asynchronous operations 5
of device drivers 104
attribute table [136], [138]
assembly-language procedure [136]

format of [137]
procedure [136]
.AUDIO [111]
audio [111]
aux_type 64, 88, [5], [14], [19]

B
B field 14
backup bit 90, [12], [18]
Backup III 90, [13]
BADBKPG [88]
BADBRK [127]
BADBUFFNUM [128]
BADBUFFSZ [128]
BADCHGMODE [88]
BADCTL [71]
BADCTLPARM [71]
BADCZPAGE 161
BADDDNUM [71]
BADINT [127]
BADJMODE [104]
BADLSCTNT [56]
BADOP [72]
BADPATH [53]
BADPGCNT [88]
BADREFNUM [54]
BADREGCODE [71]
BADSCBNDS 161
BADSCNUM 160
BADSCPCNT 161
BADSEGNUM [88]
BADSRCHMODE [88]
BADSYSBUF [56]
BADSYSCALL [127]
BADXBYTE 161
BCBERR [128]
bank
$0 16
current 12
highest 11
switchable 15

number 15
pair 13, 14
highest 15
part of segment address 25
register 11, 19, 28
restoring contents of 31
switchable 11
bank-switched field 14
bank-switched address 10, 12, 30, 32
as intermediate form 32
notation 15
bank-switched memory addressing 10–13, 30–31
bank-switched notation 23
bank-switching 27, 28, 30
for data access 30
for module execution 30
restrictions 28
base 23, 122, [43], [48], [75], [78], [83]
BASE 122
base-relative relocation table
[B138]
BASIC 118, 143
and Pascal modules 145
interpreter 145
program 145
BCS [139]
bibliography [141]
bit
backup 90, [12], [18]
destroy-enable [12], [18]
enhanced-addressing 14
map 54
read-enable [12], [18]
rename-enable [12], [18]
write-enable [12], [18]
bit_map_pointer 82
BITMAPADR [56]
.BLOCK [139]

block(s) 77
addresses of 96, 97
allocation for sparse files 98
scheme 95
altering configuration 46
call 148–149, [x]
configuration 43
altering 46
data 93, 96
device 8, 40, 76
logical 53
status request $00 [60]
device information (DIB) 43
DIB configuration 43
file 50–56, 62
control 64
structure of 50–51
index 93, 94
key 77, 82, 93, 97
logical 77
master index 94, 96, 97
maximum index 94
on a volume 77
SOS call [103]
subindex 94, 96
total 45, 82
blocks_used 63, 87, [19]
BNE [139]
bootstrap errors [128]
loader 77, 93
BRK 149
instruction 8
BTSERR [55]
buffer
data 50, [117]
editing [117]
I/O 50
space, for drivers 21
string [117], [118]
BUFTBLFULL [56]
**.BYTE** [139]

byte 99, [133]
extension 14, 31 (See also X-byte)
locating in a standard file 98-99
numbering 51
order of pointers 79
position, logical 98

**C**
call(s)
block 148-149, [x]

SOS [103]
choosing [114]
coding TERMINATE 131
D_CONTROL 128
device 46-47, [58-71]
errors [71-72]
management 5
errors
device 160, [71-72], [125]
file 160, [53-56], [125-126]
memory 160, [88]
utility 160, [104], [126]
file 69-73, [2-53]
errors [53-56]
management 5
FIND_SEG 30
form of the SOS 160
memory 25-27, [74-87]
errors [88]
management 5
OPEN 128
REQUEST_SEG 30
SOS 8
error reporting 160
form of a 148-154
types of 148
utility [90-103]
errors [104]
management 5

call_num 149, [xi]
capacity of a file, maximum 94
carry 15
CFCBFULL [53]
changing device
name 46
subtype 46
type 46
changing slot number 46
change mode [81]
CHANGE_SEG 26, [81-82]
character
device 8, 40
control code $01 [64]
control code $02 [64]
status request $01 [60]
status request $02 [61]
file(s) 50-56, 57
structure of 50-51
line-termination 67
newline 67
null (ASCII $00) 97
streams 40
termination 67
circumvention of programming restrictions 3
clock 112-113, [95], [97], [98]
rate 19
system 112
CLOSE 66, 68, 72, 90, [39-40]
closed files 52-53
closing files before TERMINATE [103]
CMP 31
code
file(s) 145
data formats of relocatable assembly-language 146
organization [132]
assembly-language [131-139]
code part of [135]
fragments, examples xiv
interpreter, executing 10
part of a code file 119, 121,
[132], [135]
segments, executing 27
sharing 44
procedure [136]
code length 120
CODEADDR [134]
CODELEN [134]
colon 15
command interpreter [103]
common code 44
common file structure 3
common foundation for software 3
defined 2
communicating with the device 42
comparing two pointers 37-38
compatibility with future versions 18
conditions for enhanced indirect addressing 31
configuration block 43
alter 46
DIB 43
conflicts between interrupts 104
with zero page 16
.CONSOLE 66, 105, 108, 125,
[109]
console 40
constant, relocation [138]
control
block, file 64
flow of 27
transfer 28
CONTROL-C [117]
CONTROL-RESET [117]

**control code** [63]

$01, character device [64]
$02, character device [64]

**control_list** [63]

conversions 32
copy-protection [103]
copying sparse files 98
CPTERR [55]
CPU 104
CREATE 68, 69, 90, 98, [3-6]
creating interpreter files 143
creation date and time 64, 81, 84,
88, 89-90
field 89-90

current
bank 12
direct pointers to 156
directory 62
position marker 51
current-bank address 12, 38
form 13

cylinders 77

**D**
.D1 [109]
.D2 [109]
.D3 [109]
.D4 [109]

.D_CONTROL 45, 47, 108, 125,
[109]
.D_INFO 43, 45, 47, [67-71]
.D_STATUS 45, 46, [59-61], [118]
data

access 10, 27, 29-32
bank-switching for 30
and buffer storage 19
block 93, 95, 96
buffer 50, [117]
editing [117]

formats of relocatable assembly-language code
files 146
in free memory 30
data_block 99
data_buffer [35], [37]
date and time
   creation 64, 81, 84, 88, 89-90
   format 90
last mod 64, 88, 89-90, [14], [19]
decimal numbers xix
decimal point xix
DESTROY 68, 69, [7-8]
destroy-enable bit [12], [18]
detecting an event 105
device 43, 60, [23], [65], [67]
   dev_name 43, [59], [63], [65], [67]
device_num 44, 45, [68]
device(s) 8, 40-42
   adding a 46
   block 8, 40
call(s) 46-47
   errors 160, [125]
   changing name of 46
   character 8, 40
   communicating with the 42
   control information 45
   correspondence
   logical/physical 54
   special cases of 54
defined as logical device 54
driver(s) 5, 41, 77, 104, 107, 108, 125
   asynchronous operation of 104
environment 20-21
   errors, individual 160
   graphics 16
   standard [109-111]
memory placement 21
indication 7, 67
information 43-44
   block (DIB) 43
input 40
logical 40
   block 53
management calls 5
multiple logical 54
name(s) 41-42, 44, 45, 50, 55, 60
   illegal 42
   legal 42
   syntax 42
number 44
operations on 45-46
output 40
physical 8, 104
random-access 7
removing a 46
requests 50
sequential-access 7
status information 45
subtype 44
changing 46
type 44
   changing 46
device-independent I/O 67
DIB
   configuration block 43
   header 43
dictionary 8
   current 62
   entry 62
   procedure [135], [136]
error (DIRERR) [55]
   file 57-58
   format(s) 78-92
   header 78
   storage formats 76
   segment [132], [134]
   volume 54, 57, 78
digit(s) 42, 56
   hexadecimal 12
direct pointer 154, 155
   to S-bank locations 155
directory file, reading a 91-92
DIRERR [55]
DIRFULL [55]
disarming events 108
Disk III driver 41
disk drives 40
disk, flexible 42, 77, 93
DISKSW [72]
dispatching routine 28
displacement [43], [48]
Display/Edit function [117]
DNFERR [71]
dollar signs xviii, xix
driver
   device See device driver
   module 41
   placement of 44
DRIVER FILE NOT FOUND [129]
DRIVER FILE TOO LARGE [129]
DUPERR [54]
DUPVOL [56]
E
E-bit 14
   editing data buffer [117]
EMPTY DRIVER FILE [129]
empty file 65
end-of-file marker See EOF
enhanced
   addressing bit 14
   addressing modes 8
   indirect addressing 10, 13-16,
   27, 30, 31-32
   conditions for 31
ENTER IC [138]
entries_per_block 82, 85, 92
entry (entries) 86
   active 86
   directory 62
   FCB 53, 62
   format compatibility 91
   inactive 86
points 145
storage formats of 76
entry_length 81, 84, 92
environment
   attributes 19
   execution 16-22
   interpreter 18-19
   SOS device driver 20-21
   SOS Kernel 19-20
summary 22
EOF 51, 53, 63, 64-65, 68, 87, 89,
   94, 95, 96, 97, 98, [5], [19], [49]
   limit 94
   movement of automatic 65
   manual 65-66
   updating 65
EOFERR [55]
EOR 31
error(s) [124]
   bootstrap [128]
   device call [125]
   file call [125]
   messages [123-130]
   numbers range 160
   reporting, SOS call 160
SOS
   fatal [124], [126]
   general [124]
   non-fatal [124]
   utility call [126]
event(s) 5, 104-115
   any-key 105
   arming, example 129
   arming and response 105, 108,
   125
   attention 105
   detecting an 105
disarming 108
   existing 108
fence 106, 109-110
field(s)  89–92
formats  89–92
bank-pair  14
pointer  79
FIFO (first-in, first-out)  109
FILBUSY  [55]
file(s)  7–8, 52
  assembly-language code  [133]
  block  50–56, 62
  allocation for sparse  98
  call(s)  69–73, [2]
  errors  160, [125]
  character  50–56, 57
  closed  52–53
  closing before TERMINATE  [103]
  code  145
    part of a code  [135]
  control block  64
  copying sparse  98
  creating interpreter  143
  data formats of relocatable
    assembly-language code 146
  defined  50
  directory  57–58
    format  78–92
    relocatable  120
    or absolute  143
  reading  91–92
  empty  65
  entry (entries)  78, 85–89
    inactive  86, 89
    sapling  89
    seedling  89
    subdirectory  89
    tree  89
  information  62–64
  input/output  67
  interpreter, creating an  143
  level, system  66
  management calls  5
  maximum capacity of a  94
  name(s)  58–59, 60
    illegal  59
    legal  59
    syntax  59
  open  52–53, 63
  operations on  68
  organization  76–99
    code  [132]
  sapling  93, 95
  seedling  93, 95
  SOS  56–62
  sparse  63, 94, 97–98
  standard  57–58
  locating a byte in  98–99
  storage formats of  92–99
structure
  common  3
  of a block  50–51
  of a character  50–51
  of a sapling  96
  of a seedling  95
  of a tree  96
  subdirectory  57, 78
  system
    relationship to device
    system  57
    root of  59
    SOS  55–62
  tree  61
  top-level  57
  tree  94, 96–97
  growing a  92–95
  type  68
  volume directory  77

floppy disk  See flexible disk
flow of control  27
FLUSH  66, 72, [37], [41–42]
FNMERR  [54]
form
  bank-switched  13
  current-bank-switched  13
  of a SOS call  148, 160
format(s)
  absolute or relocatable  143
  date and time  90
  directory file  78
  of attribute table  [137]
  of directory files  78
  of information on a volume  77
  of name parameter  159
  of relocatable assembly-
    language code files, data 146
  relocatable  120
  volume  77
free memory  23
  data in  30
  obtaining  121–124
  segment allocated from  29
free_blocks  [23]
.FUNC  [136], [139]
FUNCTION  145
future versions
  compatibility with  18
  of SOS  91, 92, 93
G
general purpose communications
  (RS232)  [111]
GET_ANALOG  113, 115,
  [99–101]
GET_DEV_NUM  43, 44, 45, 47,
  [65]
GET_EOF  65, 66, 68, 73, [49]
GET_FENCE  110, 114, [93]
GET_FILE_INFO  63, 65, 68, 70,
  152, [17–21]
GET_LEVEL 66, 69, 73, [53]
GET_MARK 66, 68, 72, [45]
GET_PREFIX 70, [27]
GET_SEG_INFO 26, [83-84]
GET_SEG_NUM 26, [85]
GET_TIME 90, 112, 115, [97-98]
.GRAFIX 110
graphics 16, [110]
area 16
device drivers 16
growing a tree file 92

I
I/O
block 51
buffer 50, 127
character 51
device-independent 67
ERROR [129]
implementation versus interface
76
warning 99
INCOMPATIBLE INTERPRETER
[129]
increment loop 124
one-bank example of 124
incrementing a pointer 36-37
index block(s) 93, 94, 95
master 94
maximum 94
sub- 94, 96
index block 99
indexed mode, zero-page 29
indexing 15
indirect
addressing 10
enhanced 10, 13-16, 27, 30, 31-32
normal 14
operation, normal 31
pointer(s) 154, 155, 157
with an X-byte between $80
and $8F 158
with an X-byte of $00 157
indirect-X addressing 13
indirect-Y addressing 13
input(s)
analog 113
device 40
parameters [116]
input/output, file 67
interface versus implementation
76
warning 99
interface, SOS 76
intermediate form, bank-switched
addresses as 32
.INTERP [139]
interpreter(s) 5, 16, 118-125, 145,
[132]
and modules 144
BASIC 145
code 10
executing 10
command [103]
environment 18-19
files, creating 143
language 118
maximum size of 18
memory
placement 18
requirements of 146
Pascal 145
return to 29
sample(s) 125-142
listing, complete 131-142
stand-alone 118
structure of 119-121
table within 29, 30
INTERPRETER FILE NOT
FOUND [129]
interpreter-relative relocation
table [139]
interpreter's
stack 19, 110
zero page 19
interrupt(s) 5, 104-115
conflicts between 104
handler 5, 22, 104
IRQ 22
and NMI 20
ranked in priority 104
summary of 112
invalid
address 13
jumps 29
regions 15, 16
INVALID DRIVER FILE [129]
io_buffer [31]
IORD [72]
IRQ interrupts 20, 22
IS_newline 67, 68, [33]

J
JMP 27-28, [139]
joy_mode [99]
joy_status [100]
joystick [99]
JSn-B [100]
JSn-Sw [100]
JSn-X [100]
JSn-Y [100]
JSR 27-28
jumps 29
inside module 29
invalid 29
valid 29

K
KERNEL FILE NOT FOUND
[130]
key_pointer 87, 92
keyboard 40

L
labels xix, 120
local 127
language interpreter 118
largest possible file 94
last_mod date and time 64, 88,
89-90, [14], [19]
field 89-90
LDA 31, [139]
leaving ExerSOS [119]
legal device names 42
legal file names 59
length 152, [3], [11], [17], [25], [30], [67], [116]
letters 42, 56
level 66, [51], [53]
level, system file 66
limit 23, 122, [75], [78], [83]
LIMIT 122
line-termination character 67
linked list 78
linker information [133]
linking
assembly-language modules 145
dynamic loading during 145
lists
required parameter 129, 150-152
optional parameter 152-154
loading
dynamic, during linking 145
assembly-language modules 145
routine [134]
loading_address 120, 121
locating a byte in a standard file 98
logical
block 77
device 53
byte position 98
device(s) 40
accessing a 41
multiple 54
structures 76
logical/physical device correspondence 54
loop, increment 124
low-order nibble [117]
LVLERR [56]

M
machine
abstract 2
storing the state of the macro, SOS 110
MakInterp [121-122]
management calls
device 5
file 5
memory 5
utility 5
manager, resource 2-3
manual movement of EOF and mark 66
manuf_id 45, [70]
manufacturer 45
mark 51, 53, 64-65, 68, 97, 98, [45]
movement of, automatic 65
movement of, manual 65-66
marker, current position 51
master index block 94, 96, 97
maximum
number of access paths 53
capacity of a file 94
number of index blocks 94
size of an interpreter 18
MCTOVL [127]
media, removable 53, 54
medium 42, 53
MEM2SML [127]
memory 6-7, 23
access techniques 27-38
addressing, bank-switched 10-13
allocation 25, 121
bookkeeper 7
call(s) 25-27
ers, 160
error 121
conflict 121
avoiding 121
management 7
calls 5
obtaining free 121-124
placement
interpreter 18
module 144
SOS device driver 21
SOS Kernel 20
S-bank 19
segment 17
size, maximum 6, 10
switched 28
messages, error [123-130]
min_version 81, 84, 88
mode(s)
absolute addressing 29
addressing 10-16
enhanced addressing 8
newline information 67
zero-page addressing 29
indexed 29
modification date and time 68
module(s) 5, [132]
absolute 143
addressing 27-29
assembly-language 19, 118,
143-146
linking 145
BASIC invokable 145
creating 146
driver 41
execution, bank-switching for 30
formats 146
loader [134]
Pascal 145
program or data access by 145
relocatable 143, 146, [132]
multiple
access paths 52
logical devices 54
volumes 54

N
name(s) 60, 68
device 60
file 58-59, 60
local 59
parameter 159-160
volume 55-56, 60
name_length 80, 83, 87
naming conventions 76
new_pathname [9]
NEWLINE 67, 68, 69, 71, [33-34]
newline
character 67
mode 67
newline_char 67, 68, [33]
nnewline-mode information 67
nibble
high-order [117]
low-order [117]
NMI 114
interrupts 20
NMIHANG [127]
NORESC [72]
notation xviii
and symbols xviii
bank-switched address 15, 23
extended address 15
numeric xviii
segment address 23-27
NOTBLKDEV [56]
NOTOPEN [72]
NOTSOS [55]
NOWRITE [72]
null characters (ASCII $00) 97
number(s)
decimal xix
device 44
hexadecimal xiv
reference 52
slot 44
changing 46
unit 44
version 45
numeric notation xviii, xix

O
OPEN 52, 53, 68, 69, 71, [29–32]
call, example 128
operating system 2–3
defined 2
operations
asynchronous 5
normal indirect 31
on devices 45–46
on files 68
sequential read and write 50
opt_header 120
opt_header_length 120
option_list 152, [3], [11], [17],
[29], [67]
optional parameter list 152–154,
[x]
ORA 31
order of event queue 109
organization, code file [132]
OUTOFMEM 56
output device 40
overview of the Apple III 3–8
OVRRERR [54]

P
page(s) 23, [31], [78], [81], [83]
part of segment address 25
parameter(s)
format of a name 159
input [116]
list,
optional 152–154, [x]
required 129, 150–152, [x]
name 159–160
passing 145
pointer 145
parent_entry_length 85
parent_entry_number 85
parent_pointer 85
parm_count [xi]
parm_list 149
Pascal 118, 143, [132]
and BASIC modules 145
assembler 145, [134]
interpreter 145
prefix 62
program 145
versus SOS prefixes 62
path(s)
access 52
information 64–66
multiple 52
maximum number of 56
pathname [3], [7], [9], [11], [17],
[25], [29]
pathname 52, 59–61
full 62
partial 61–62
syntax 60
valid 61
PERFORM 145
period 42, 56
peripheral device 8, 104
physical device 40, 54
 correspondence with logical
devices 54
PNFERR [54]
point, decimal xix
pointer(s) 31, 69, 152
address extension 154–159
byte order of 79
comparing two 37
direct 154, 155–156
to current 156
to X-bank 155
extended 123
fields 79
incrementing a 36–37
indirect 154, 156–159
manipulation 36–38
parameters 145
preceding-block 78
self-relative [136], [138]
three-byte 98
POSNERR [55]
prefix(es) 60, 61–62
Pascal 62
restrictions on 62
SOS 62
versus Pascal 62
.PRINTER [111]
printers 40
priority of zero 108
priority–queue scheme 108
.PRIVATE [138]
.PROC [136], [139]
procedure(s) [135], [136]
attribute table [136]
code [136]
dictionary [135]
entries [136]
PROCEDURE NUMBER [138]
procedure–relative relocation
table [139]
processing an event 106
Processor, Apple III xvii
Product Support Department 45
program
execution, restrictions on 14
exiting from 66
programming
assembly–language xiii
restrictions, circumvention of
SOS 3
pseudo-opcode(s) [136]
.FUNC [136]
.PRIVATE [138]
.PROC [136]
.PUBLIC [138]
.PUBLIC [138]
Q
queuing an event 106
R
range, X-byte 15
READ 67, 68, 71, [35–36]
read and write operations,
sequential 50
read-enable bit [12], [18]
reading a directory file 91
ref_num 52, 64, 67, [2], [29], [33],
[35], [37], [39], [49]
[41], [43], [45], [47]
references, relocation [138]
regions
invalid 15, 16
risk 15, 16
release memory 25
RELEASE_SEG 27, [87]
relocation 146
constant [138]
information 145
references [138]
table(s) [138]
base–relative [138]
interpreter–relative [139]
procedure–relative [139]
segment–relative [139]
RELOCSEG_NUMBER [138]
RENAME 69, 90, [9–10]
req_access [30]
request_count [35], [37]
REQUEST_SEG 25, 121, [75–76]
call 30
required parameter list 129,
150–152, [x]
example 129
resource manager 2–3
defined 2
resources 112–114
restrictions
addressing 15
bank-switching 28
on program execution 14
result 69, 151
return to interpreter 29
risky regions 15, 16
addresses 32
avoiding 37
warning 32
ROM ERROR: PLEASE NOTIFY
YOUR DEALER [130]
root of file system 59
.RS232 [111]

S
S–bank 11, 23, 28
address 12, 38
in segment notation 25
locations, direct pointers to 155
memory 19
sample programs, examples xiv
sapling file 93, 95
entry 89
structure of a 96
SBC 31
scheme, priority–queue 108
SCP 43
screen 40
search_mode [77]
sectors 77
seedling file 93, 95
entry 89
structure of a 95
seg_address [85]
seg_id [75], [78], [83]
seg_num [76], [78], [81], [83],
[85], [87]
segment 23–24
address 24, 38
bank part of 25
conversion 33–35
notation 23–27
page part of 25
allocated from free memory 29
dictionary [132], [134]
memory 7
of memory, allocating a 121
to bank-switched address
conversion 33
to extended address conversion 33
segment–relative relocation

table [139]
SEGNOFNDF [88]
SEGROD [88]
SEGTLFULL [88]
sequential
access 51
devices 7
read and write operations 50
serial printer (PRINTER) [111]
SET_EOF 66, 68, 72–73, [47–48]
SET_FENCE 107, 110, 114, [91]
SET_FILE_INFO 63, 68, 70, 88,
90, 152, [11–16]
SET_LEVEL 66, 73, [51]
SET_MARK 66, 68, 72, [43–44]
SET_PREFIX 70, [25–26]
SET_TIME 90, 112, 115, [95–96]
slash (/) 56, 60
slot number 44
change 46
of zero 44
slot_num 44, [68]
software, common foundation
for 2, 3
Sophisticated Operating System
See SOS
SOS xiv, 3, 5–6, 16, 104
1.1 xiv, [106]
1.2 18, 77, 81, 82, 84, 85, 88, 92,
93, 95, 99, 105
1.3 xiv, [106]
bank 11
call(s) 8
block [103]
form error 160
reporting 160–161
form of 148–154, 160
types of 148
device
driver
environment 20–21
memory placement 21
system 43
disk request 55
ersors
fatal [124], [126]
genral [124]
non–fatal [124]
file system 58, 58
future versions of 91, 92, 93
implementation 76
interface 76
Kernel 19
environment 19–20
memory placement 20
macro 126
for SOS call block 126
prefix(es) 62
versus Pascal 62
programming restrictions,
circumvention of 3
specifications [105–111]
support for 76
system 104
versions xiv, [106]
SOS.DRIVER 6, 41
SOS.INTERP 118
SOS.KERNEL 6, 41
sparse file(s) 63, 94, 97–98
block allocation for 98
copying 98
special symbols xv
STA 31
stack 17, 20
interpreter's 145
overflow [127]
pages 19
stand–alone interpreter 118
standard device drivers [109–111]
standard file(s) 57–58
locating a byte in 98–99
storage formats of 92–99
state of the machine, storing
the 110
status request
$00, block device [60]
$01, character device [60]
$02, character device [61]
status_code [59]
status_list [60]
STKOVFL [127]
stop symbol xv
storage formats
directory headers 76
entries 76
of standard files 92–99
storage_type 64, 80, 81, 83, 87, 89,
92, 95, 96, 97, [5], [19]
string buffer [117], [118]
structure(s)
for hierarchical tree 56, 76
logical 76
of a sapling file 96
of a seedling file 95
of a tree file 96
of an interpreter 119–121
of block files 50–51
of character files 50–51
sub_type 44, 45, [69]
subdirectory (subdirectories) 8
file(s) 57, 78
entry 89
header 82, 83, 89
subindex block 94, 96
subroutine addressing 27–29
summary
  of address storage 38
  of interrupts and events 112
switchable bank 11
  highest 15, 18
symbol(s)
  eye xix
  hand xix
  stop xix
  v1.2 xix
syntax
  device name 42
  file name 59
  pathname 60
  volume name 56
System Configuration Program
  (SCP) 41, 46
system
  clock 112
  configuration time 104
  file level 66
  operating 2-3
  status during event handling 111

T
  table
    procedure attribute [136]
    within interpreter 29, 30
  Technical Support Department 146
TERMINATE 114, 115, 126, 131,
  [xii], [103]
call, coding 131
closing files before [103]
termination character 67, [61],
  [64]
three-byte
  address 13
  pointer 98

V
  v1.2 symbol xix
  and other versions xix
time
  date and
    creation 64, 81, 84, 88, 89-90
    format 90
  last mod 64, 88, 89-90, [14],
    [19]
  time pointer [95], [97]
time-dependent code 104
timing loop 19, 104
TOO MANY BLOCK DEVICES
  [130]
TOO MANY DEVICES [130]
TOOLONG [128]
top-level files 57
total_blocks 45, 82, [23], [70]
tracks 77
transfer control 28
transfer_count [36]
tree file 94, 96-97
  entry 89
  growing a 92-95
  structure of a 96
tree structure, hierarchical 56
tree, file system 61
  TYPERR [55]

U
  unit number 44
  unit_num 44, [68]
  unsupported storage type
    (TYPERR) [55]
  utilities disk 41
utility
  call(s) 114
    errors 160, [126]
    management 5

W
  warning
    address conversion 123
    interface versus implementation
      99
    on accessing zero page and
      stack 17
    on pointer conversions 155
    on sample interpreter 125
    pointer
      direct 156
      indirect 158, 159
    risky regions 32
termination 114
  unallocated memory 121
  jumps 29
  pathnames 61
  value 69, 151
  value/result parameter 152
  VCBERR [128]
  version 81, 84, 88
  number 45
  version_num 45, [70]
  VNFERR [54]
  vol_name 60, [23]
  VOLUME 70, [23-24]
  volume(s) 53-54, 76
    bit map 77, 93
    blocks on a 77
    directory 54, 57, 78, 93
      file 77
      header 79, 80, 89
    formats 77
    multiple 54
    name(s) 42, 55-56, 60
      advantages of 56
      syntax 56
    switching 54-55
  volume/device correspondence
    54

X
  X register 14
  X-bank, direct pointers to 155
  X-byte 14, 15, 31, 145
    between $80 and $8F, indirect
      pointers with an 158
    format 14
    of $00, indirect pointers with
      an 157
    of $8F 16
    range 15
  X-page 145

Y
  Y-register 15, 32

Z
  zero
    interpreter's 19
    page 15, 17, 20, 29
    and stack 17, 20
    warning on accessing 17
    conflicts with 16
    priority of 108
zero-page addressing mode 29
zero-page indexed addressing
  mode 29

Special Symbols and Numbers
  & v1.2 81, 82, 84
  $ xviii, xix
  $0 16
  $8F 16
  6502 xvii
    instruction set 8