ProDOS 8 Technical Reference Manual

The ProDOS 8 Technical Reference Manual is identical in content to the previously published ProDOS Technical Reference Manual.
Information in this manual covers ProDOS 8 through 1.1.1. For details on changes to more recent versions, see ProDOS Technical Note #23.

Contents

- **Figures and Tables** ... xiii

- **Preface** ... xv
  - About ProDOS ... xv
  - About This Manual ... xvi
  - What These Mean ... xvii
  - About the Apple IIc ... xvii

- **Chapter 1 - Introduction** ... 1
  - 1.1 - What Is ProDOS? ... 2
    - 1.1.1 - Use of Disk Drives ... 3
    - 1.1.2 - Volume and File Characteristics ... 5
    - 1.1.3 - Use of Memory ... 5
    - 1.1.4 - Use of Interrupt Driven Devices ... 6
    - 1.1.5 - Use of Other Devices ... 6
  - 1.2 - Summary ... 7

- **Chapter 2 - File Use** ... 9
  - 2.1 - Using Files ... 10
    - 2.1.1 - Pathnames ... 10
    - 2.1.2 - Creating Files ... 13
    - 2.1.3 - Opening Files ... 13
    - 2.1.4 - The EOF and MARK ... 14
    - 2.1.5 - Reading and Writing Files ... 15
    - 2.1.6 - Closing and Flushing Files ... 16
    - 2.1.7 - File Levels ... 17

  - 2.2 - File Organization ... 17

        - 2.2.1 - Directory Files and Standard Files ... 17
        - 2.2.2 - File Structure ... 18
        - 2.2.3 - Sparse Files ... 20

- **Chapter 3 - Memory Use** ... 21
    - 3.1 - Loading Sequence ... 22
    - 3.2 - Volume Search Order ... 23
    - 3.3 - Memory Map ... 23
        - 3.3.1 - Zero Page ... 25
        - 3.3.2 - The System Global Page ... 25
        - 3.3.3 - The System Bit Map ... 25

- **Chapter 4 - Calls to the MLI** ... 27
    - 4.1 - The Machine Language Interface ... 28
    - 4.2 - Issuing a Call to the MLI ... 29
        - 4.2.1 - Parameter Lists ... 31
        - 4.2.2 - The ProDOS Machine Language Exerciser ... 31
    - 4.3 - The MLI Calls ... 32
        - 4.3.1 - Housekeeping Calls ... 32
        - 4.3.2 - Filing Calls ... 33
        - 4.3.3 - System Calls ... 35

    - 4.4 - Housekeeping Calls ... 36
        - 4.4.1 - CREATE ($C0) ... 36
        - 4.4.2 - DESTROY ($C1) ... 40
        - 4.4.3 - RENAME ($C2) ... 42
        - 4.4.4 - SET_FILE_INFO ($C3) ... 43
        - 4.4.5 - GET_FILE_INFO ($C4) ... 47
        - 4.4.6 - ON_LINE ($C5) ... 51
        - 4.4.7 - SET_PREFIX ($C6) ... 54
        - 4.4.8 - GET_PREFIX ($C7) ... 55
    - 4.5 - Filing Calls ... 56
        - 4.5.1 - OPEN ($C8) ... 56
        - 4.5.2 - NEWLINE ($C9) ... 58
        - 4.5.3 - READ ($CA) ... 59
        - 4.5.4 - WRITE ($CB) ... 61
        - 4.5.5 - CLOSE ($CC) ... 63
        - 4.5.6 - FLUSH ($CD) ... 64
        - 4.5.7 - SET_MARK ($CE) ... 65
        - 4.5.8 - GET_MARK ($CF) ... 66
        - 4.5.9 - SET_EOF ($D0) ... 67
        - 4.5.10 - GET_EOF ($D1) ... 68

        - 4.5.11 - SET_BUF ($D2) ... 69
        - 4.5.12 - GET_BUF ($D3) ... 70
    - 4.6 - System Calls ... 71
        - 4.6.1 - GET_TIME ($82) ... 71
        - 4.6.2 - ALLOC_INTERRUPT ($40) ... 72
        - 4.6.3 - DEALLOC_INTERRUPT ($41) ... 73
    - 4.7 - Direct Disk Access Commands ... 73
        - 4.7.1 - READ_BLOCK ($80) ... 74
        - 4.7.2 - WRITE_BLOCK ($81) ... 75
    - 4.8 - MLI Error Codes ... 77

- **Chapter 5 - Writing a ProDOS System Program** ... 81
    - 5.1 - System Program Requirements ... 82
        - 5.1.1 - Placement in Memory ... 82
        - 5.1.2 - Relocating the Code ... 84
        - 5.1.3 - Updating the System Global Page ... 84
        - 5.1.4 - The System Bit Map ... 84
            - 5.1.4.1 - Using the Bit Map ... 85
        - 5.1.5 - Switching System Programs ... 86
            - 5.1.5.1 - Starting System Programs ... 86
            - 5.1.5.2 - Quitting System Programs ... 87
    - 5.2 - Managing System Resources ... 89
        - 5.2.1 - Using the Stack ... 89
        - 5.2.2 - Using the Alternate 64K RAM Bank ... 89
            - 5.2.2.1 - Protecting Auxiliary Bank Hi-Res Graphics Pages ... 89
            - 5.2.2.2 - Disconnecting /RAM ... 90
            - 5.2.2.3 - How to Treat RAM Disks With More Than 64K ... 91
            - 5.2.2.4 - Reinstalling /RAM ... 92
        - 5.2.3 - The System Global Page ... 94
        - 5.2.4 - Rules for Using the System Global Page ... 94
    - 5.3 - General Techniques ... 98
        - 5.3.1 - Determining Machine Configuration ... 98
            - 5.3.1.1 - Machine Type ... 98
            - 5.3.1.2 - Memory Size ... 98
            - 5.3.1.3 - 80-Column Text Card ... 99
        - 5.3.2 - Using the Date ... 99
        - 5.3.3 - System Program Defaults ... 100
        - 5.3.4 - Finding a Volume ... 100
        - 5.3.5 - Using the RESET Vector ... 101
    - 5.4 - ProDOS System Program Conventions ... 101

- **Chapter 6 - Adding Routines to ProDOS** ... 103
  - 6.1 - Clock/Calendar Routines ... 104
    - 6.1.1 - Other Clock/Calendars ... 106
  - 6.2 - Interrupt Handling Routines ... 106
    - 6.2.1 - Interrupts During MLI Calls ... 108
    - 6.2.2 - Sample Interrupt Routine ... 109
  - 6.3 - Disk Driver Routines ... 112
    - 6.3.1 - ROM Code Conventions ... 112
    - 6.3.2 - Call Parameters ... 114

- **Appendix A - The ProDOS BASIC System Program** ... 117
  - A.1 - Memory Map ... 118
  - A.2 - HIMEM ... 120
    - A.2.1 - Buffer Management ... 121
  - A.3 - The BASIC Global Page ... 123
    - A.3.1 - BASIC.SYSTEM Commands From Assembly Language ... 131
    - A.3.2 - Adding Commands to the BASIC System Program ... 134
      - A.3.2.1 - BEEP Example ... 136
      - A.3.3.2 - BEEPSLOT Example ... 138
    - A.3.3 - Command String Parsing ... 140
  - A.4 - Zero Page ... 142
  - A.5 - The Extended 80-Column Text Card ... 143

- **Appendix B - File Organization** ... 145
  - B.1 - Format of Information on a Volume ... 146
  - B.2 - Format of Directory Files ... 147
    - B.2.1 - Pointer Fields ... 148
    - B.2.2 - Volume Directory Headers ... 148
    - B.2.3 - Subdirectory Headers ... 151
    - B.2.4 - File Entries ... 154
    - B.2.5 - Reading a Directory File ... 157
  - B.3 - Format of Standard Files ... 159
    - B.3.1 - Growing a Tree File ... 159
    - B.3.2 - Seedling Files ... 161
    - B.3.3 - Sapling Files ... 162
    - B.3.4 - Tree Files ... 163
    - B.3.5 - Using Standard Files ... 163
    - B.3.6 - Sparse Files ... 164
    - B.3.7 - Locating a Byte in a File ... 166
  - B.4 - Disk Organization ... 167

        - B.4.1 - Standard Files ... 169
        - B.4.2 - Header and Entry Fields ... 170
            - B.4.2.1 - The storage_type Attribute ... 171
            - B.4.2.2 - The creation and last_mod Fields ... 171
            - B.4.2.3 - The access Attribute ... 172
            - B.4.2.3 - The file_type Attribute ... 172
    - B.5 - DOS 3.3 Disk Organization ... 174

- **Appendix C - ProDOS, the Apple III, and SOS** ... 175
    - C.1 - ProDOS, the Apple III, and SOS ... 176
    - C.2 - File Compatibility ... 176
    - C.3 - Operating System Compatibility ... 177
        - C.3.1 - Comparison of Input/Output ... 177
        - C.3.2 - Comparison of Filing Calls ... 177
        - C.3.3 - Memory Handling Techniques ... 178
        - C.3.4 - Comparison of Interrupts ... 178

- **Appendix D - The ProDOS Machine Language Exerciser** ... 179
    - D.1 - How to Use It ... 180
    - D.2 - Modify Buffer ... 181

- **Index** ... 183

- **Tell Apple Card**

- **Quick Reference Card**

Figures and Tables

- **Chapter 1 - Introduction** ... 1
    - Figure 1-1 - A Simplified Diagram of ProDOS ... 2
    - Figure 1-2 - A Typical ProDOS Directory Structure ... 4
    - Figure 1-3 - The Levels of ProDOS ... 8

- **Chapter 2 - File Use** ... 9
    - Figure 2-1 - A Typical ProDOS Directory Structure ... 12
    - Figure 2-2 - Automatic Movement of EOF and MARK ... 15
    - Figure 2-3 - Directory File Structure ... 18
    - Figure 2-4 - Block Organization of a Directory File ... 19

- Figure 2-5 - Block Organization of a Standard File ... 19

- **Chapter 3 - Memory Use** ... 21
  - Figure 3-1 - Memory Map ... 24

- **Chapter 5 - Writing a ProDOS System Program** ... 81
  - Figure 5-1 - Memory Map ... 83
  - Figure 5-2 - Memory Representation in the System Bit Map ... 84
  - Figure 5-3 - Page Number to Bit-Map Bit Conversion ... 85

- **Chapter 6 - Adding Routines to ProDOS** ... 103
  - Figure 6-1 - ProDOS Date and Time Locations ... 104

- **Appendix A - The ProDOS BASIC System Program** ... 117
  - Figure A-1 - Memory Map ... 119
  - Table A-1 - HIMEM and Program Workspace ... 120
  - Figure A-2 - The Movement of HIMEM ... 121
  - Figure A-3 - Zero Page Memory Map ... 142

- **Appendix B - File Organization** ... 145
  - Figure B-1 - Blocks on a Volume ... 147
  - Figure B-2 - Directory File Format ... 148
  - Figure B-3 - The Volume Directory Header ... 149
  - Figure B-4 - The Subdirectory Header ... 152
  - Figure B-5 - The File Entry ... 155
  - Figure B-6 - Structure of a Seedling File ... 162
  - Figure B-7 - Structure of a Sapling File ... 162
  - Figure B-8 - Structure of a Tree File ... 163
  - Figure B-9 - A Sparse File ... 165
  - Figure B-10 - Disk Organization ... 168
  - Figure B-11 - Standard Files ... 169
  - Figure B-12 - Header and Entry Fields ... 170
  - Figure B-13 - Date and Time Format ... 171
  - Figure B-14 - The access Attribute Field ... 172
  - Table B-1 - The ProDOS File_Types ... 173
  - Figure B-15 - Tracks and Sectors to Blocks ... 174

Preface

The *ProDOS Technical Reference Manual* is the last of three manuals that describe ProDOS(TM), the most powerful disk operating system available for the Apple II.

- The *ProDOS User's Manual* tells how to copy, rename, and remove ProDOS files using the ProDOS Filer program, and how to move files from DOS disks to ProDOS disks using the DOS-ProDOS Conversion program.
- *BASIC Programming With ProDOS* describes ProDOS to a user of the BASIC system program. It explains how to store information on ProDOS disks and to retrieve information from ProDOS disks using Applesoft BASIC.
- This manual, the *ProDOS Technical Reference Manual*, explains how to use the machine-language routines upon which the Filer program, the DOS-ProDOS Conversion program, and the BASIC system program are based. Appendix A reveals a more technical side of the BASIC system program.

About ProDOS

The set of machine-language routines described in this manual provides a consistent and interruptible interface to any of the disk devices manufactured by Apple Computer, Inc. for the Apple II. They are designed to be used in programs written in the 6502 machine language.

This manual

- describes the files that these routines create and access
- tells how each of the routines is used
- explains how to combine the routines into an application program
- tells how to write and install routines to be used when an interrupt is detected
- tells how to write a routine that automatically reads the date from a clock/calendar card when a file is created or modified
- explains how to attach other devices to ProDOS.

Some advantages of programs written using these ProDOS machine-language routines are:

- They store information on disks using a hierarchical directory structure.
- They are able to access all disk devices manufactured by Apple Computer, Inc. for the Apple II.
- They can read data from a Disk II drive at a rate of approximately eight kilobytes per second (compared to one kilobyte per second for DOS).
- They are interruptible.
- They have the same disk and directory format as Apple III SOS disks.
- Calls to ProDOS are very similar to calls to SOS; programs can be readily developed for both the Apple II and the Apple III. Appendix C explains the similarities and differences between ProDOS and SOS.

About This Manual

Apple II
In this manual the name *Apple II* implies the Apple II Plus, the Apple IIe, and the Apple IIc, as well as the Apple II, unless it specifically states otherwise.

This manual is written to serve as a learning tool and a reference tool. It assumes that you have had some experience with the 6502 assembly language, and that you are familiar with the Apple II's internal structure.

If you have read *BASIC Programming With ProDOS* and you want to find out more about how the BASIC system program works, refer first to Appendix A. If you still want more details, Chapters 1 through 3 tell what ProDOS is and how it works. If you plan to write machine-language programs that use ProDOS, you will also need to read Chapters 4 and 5. Chapter 6 shows techniques for adding various devices to the ProDOS system.

This manual does not explain 6502 assembly language. If you plan to read beyond Chapter 3, you should be familiar with the 6502 assembly language and with the ProDOS Editor/Assembler.

What These Mean

By the Way: Text set off in this manner presents sidelights or interesting points of information.

Important!
Text set off in this manner -- and with a tag in the margin -- presents important information.

Warning
Warnings like this indicate potential problems or disasters.

About the Apple IIc

Although the Apple IIc has no slots for peripheral cards, it is configured as if it were an Apple IIe with

- 128 Kbytes of RAM
- serial I/O cards in slots 1 and 2
- an 80-column text card in slot 3
- a mouse (or joystick) card in slot 4
- a disk controller (for two disk drives) in slot 6.

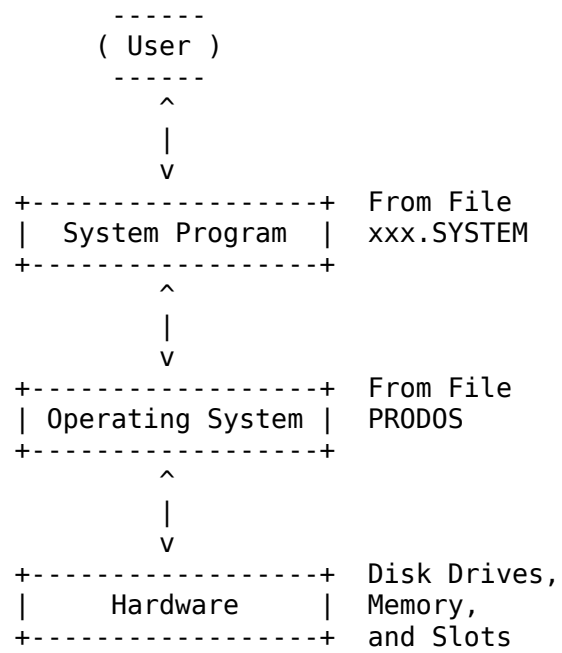
Chapter 1 Introduction

This chapter contains an overview of ProDOS and of the material explained in the rest of this manual. It presents a conceptual picture of the organization and capabilities of ProDOS. It also tells you where in the manual each aspect of ProDOS is explained.

1.1 - What Is ProDOS?

ProDOS is an operating system that allows you to manage many of the resources available to an Apple II. It functions primarily as a disk operating system, but it also handles interrupts and provides a simple means for memory management. ProDOS marks files with the current date and time, taken from a clock/calendar card if you have one.

All ProDOS startup disks have two files in common: PRODOS and XXX.SYSTEM (Chapter 2 explains the possible values for XXX). The file PRODOS contains the ProDOS operating system; it performs most of the communication between a **system program** and the computer's hardware. The file XXX.SYSTEM contains a system program, the program that usually communicates between the user and the operating system. Figure 1-1 shows a simplified block diagram of the ProDOS system.



A ProDOS system program -- such as the BASIC system program (file BASIC.SYSTEM on the *ProDOS BASIC Programming Examples* disk), the ProDOS Filer (file FILER on the *ProDOS User's Disk*), or the DOS-ProDOS Conversion program (file CONVERT on the *ProDOS User's Disk*) -- is an assembly-language program that accepts commands from a user, makes sure they are valid, and then takes the appropriate action. One course of action is to make a call to the Machine Language Interface (MLI), the portion of the operating system that receives, validates, and issues operating system commands.

Calls to the MLI give you control over various aspects of the hardware. MLI calls can be divided into housekeeping calls, filing calls, memory calls, and interrupt handling calls. The way that the MLI communicates with disk drives, memory, and interrupt driven devices is described in the following sections.

Calls to the MLI: see Chapter 4.

> **About System Programs:** If you have dealt with system programs before, you may be a bit confused about the term as used in this manual. True system programs are neither application programs (such as a word processor) nor operating systems: they provide an easy means of making operating system calls from application programs.
>
> As used in this manual, *system program* refers to a program that is written in assembly language, makes calls to the Machine Language Interface, and adheres to a set of conventions, making it relatively easy to switch from one system program to another. System programs can be identified by their file type.
>
> In short, it is the structure of a program, not its function, that makes a program a ProDOS system program.
>
> The rules for organizing system programs are given in Chapter 5.

1.1.1 - Use of Disk Drives

Although ProDOS is able to communicate with several different types of disk drives, the type of disk drive and the slot location of the drive need not be known by the system program: the MLI takes care of such details. Instead disks -- or, more accurately, volumes of information -- are identified by their volume names.

The information on a volume is divided into files. A **file** is an ordered collection of bytes, having a name, a type, and several other properties. One important type of file is the **directory file**: a directory file contains the names and location on the volume of other files. When a disk is formatted using the Format a Volume option of the ProDOS Filer program, a main directory file for the volume is automatically

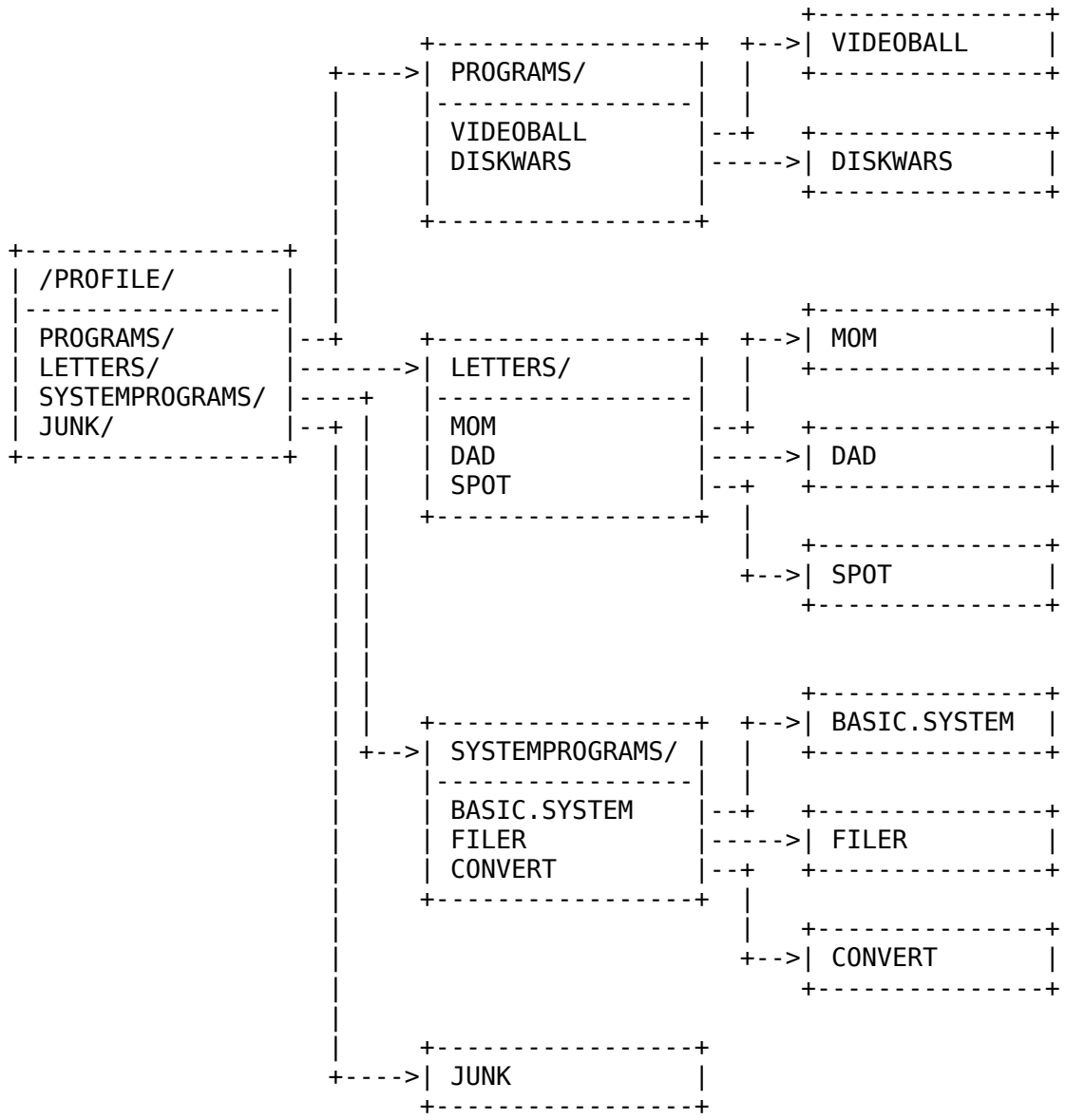
placed on the disk. It is called the disk's **volume directory** file, and it has the same name as the volume itself. Although it is initially empty,

a volume directory file has a maximum capacity of 51 files.

Any file in the volume directory may itself be a directory file (called a **subdirectory**), and any file within a subdirectory can also be a subdirectory. Using directory files, you can arrange your files so that they can be most easily accessed and manipulated. This is especially useful when you are working with large capacity disk drives such as the ProFile. A sample directory structure is shown in Figure 1-2.

Directory structures are described in Chapter 2.

Figure 1-2. A Typical ProDOS Directory Structure



The filing calls, described in Chapter 4, provide all functions necessary for the access and manipulation of files.

1.1.2 - Volume and File Characteristics

Programs that make filing calls to the ProDOS Machine Language Interface can take advantage of the following features:

- Access to all ProDOS formatted disks; maximum capacity 32 megabytes on a volume.
- Files can be stored in up to 64 levels of readable directory and subdirectory files.
- A volume directory holds up to 51 entries.
- Subdirectories can hold as many files as needed; they become larger as files are added to them.
- There are over 60 distinct file identification codes; some are predefined, others can be defined by the system program. For compatibility, existing file types should be used.
- Up to eight files can be open for access simultaneously.
- A file can hold up to 16 megabytes of data.
- Disks can be accessed by block number as well as by file.
- If the data in a file is not sequential, the logical size of the file can be bigger than the amount of disk space used.

The use of files is described in Chapter 2; their format is given in Appendix B.

1.1.3 - Use of Memory

ProDOS treats memory as a sequence of 256-byte **pages**. It represents the status of each page, used or unused, as a single bit in a portion of memory called the **system bit map**.

When ProDOS initializes itself, it marks all the pages in memory it needs to protect. Once running, it sets the corresponding bit in the bit map for each new page it uses; when it releases the page, it clears the bit.

If your program allows the user to read information into specific areas of memory, you can use the bit map to prevent ProDOS from overwriting the program.

The arrangement of ProDOS in memory is described in Chapter 3.

1.1.4 - Use of Interrupt Driven Devices

Certain devices generate interrupts, signals that tell the controlling computer (in this case an Apple II), that the device needs attention.

ProDOS is able to handle up to four interrupting devices at a time. To add an interrupt driven device to your system:

1. Place an interrupt handling routine into memory.
2. Mark the block of memory as used.
3. Use the MLI call that adds interrupt routines to the system.
4. Enable the device.

This causes the routine to be called each time an interrupt occurs. If you install more than one routine, the routines will be called in the order in which they were installed.

To remove an interrupt handling routine:

1. Disable the device.
2. Unmark its block in memory
3. Use the MLI call that removes interrupt routines from the system.

Warning:
Failure to follow these procedures in sequence may cause system error.

The use of interrupt driven devices is described in Chapter 6.

1.1.5 - Use of Other Devices

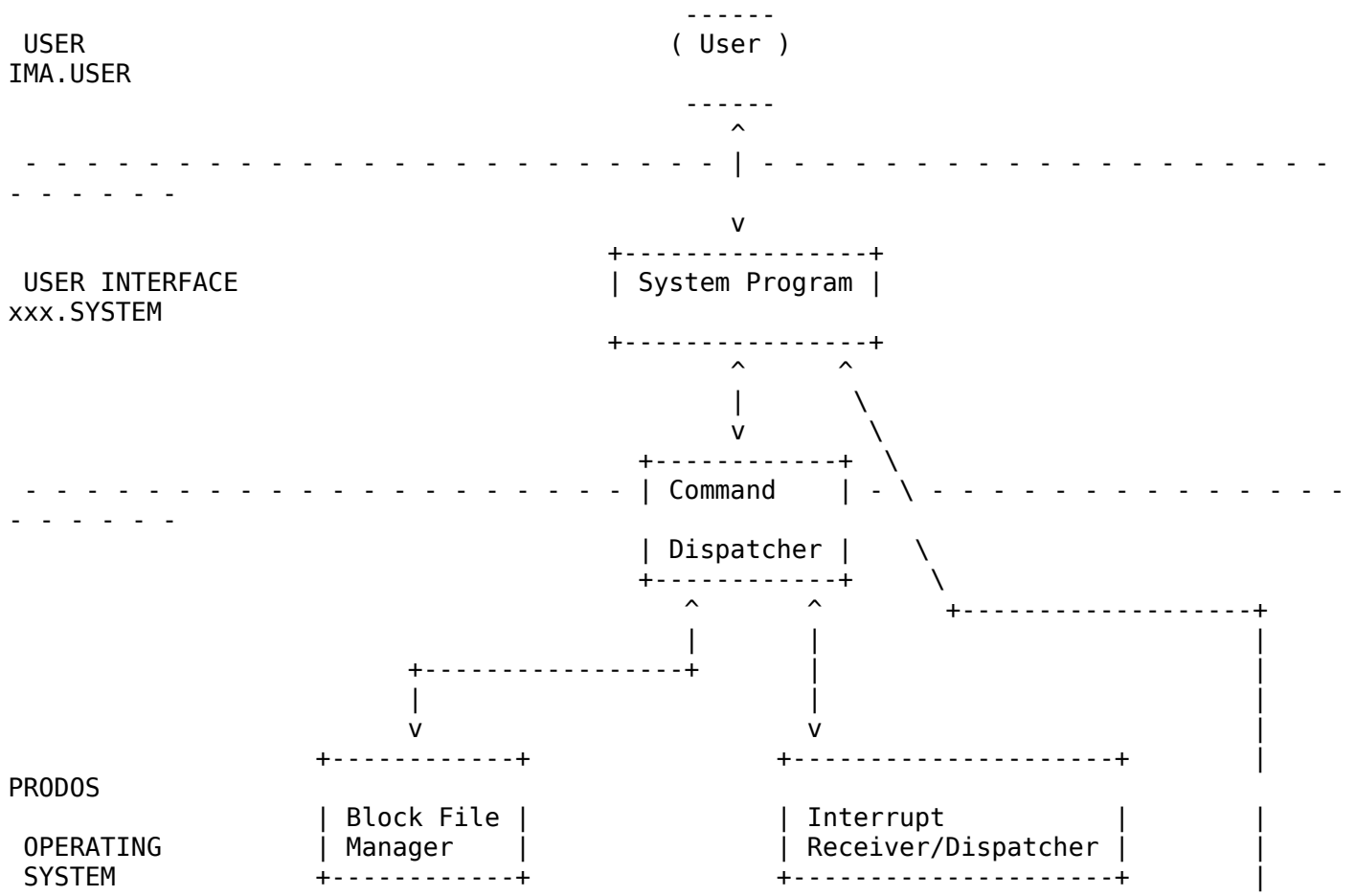
Other than disks, ProDOS communicates only with clock/calendar cards. If your system has a clock/calendar card that follows ProDOS protocols (see Chapter 6), ProDOS automatically sets up a routine so that it can read from the clock before marking files with the time. If you have some other type of clock, you must write your own routine, place it in memory, and tell ProDOS where the routine is located.

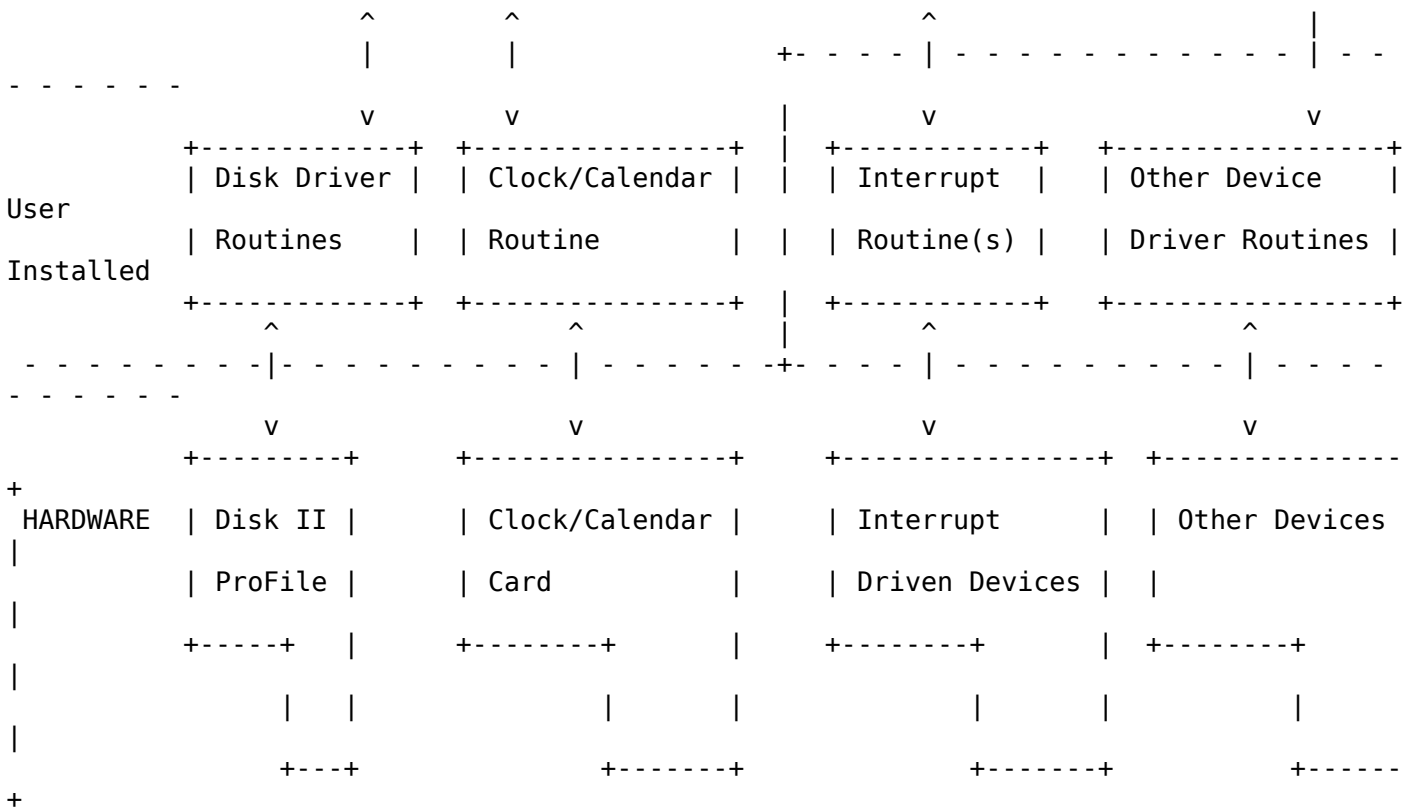
1.2 - Summary

Figure 1-3 illustrates the entire mechanism used by ProDOS and shows the interaction between the levels of ProDOS. A complete ProDOS system consists of the Machine Language Interface, a system program, and some external routines. If you wish your system to operate with interrupt driven devices, a clock/calendar card, or other external devices, you must supply routines that communicate with these devices.

The system program takes commands from the user and issues them to the Command Dispatcher portion of the Machine Language Interface or to independently controlled devices. The Command Dispatcher validates each command before passing it to the Block File Manager (which also manages memory) or to the Interrupt Receiver/Dispatcher. The Block File Manager calls a disk driver routine and the clock/calendar routine if necessary; the Interrupt Receiver/Dispatcher calls the interrupt handling routines.

Figure 1-3. The Levels of ProDOS





The following chapters describe the implementation of this mechanism. After reading through Chapter 5, you will be ready to start writing your own system programs. After reading through Chapter 6, you will be able to write your own external routines.

Chapter 2
File Use

Chapter 1 introduced you to the concepts of volumes and files. This chapter explains how files are named, how they are created and used and a little about how they are organized on disks. When you have finished reading this chapter you will be nearly ready to start using the ProDOS Machine Language Interface filing calls.

The technical details of file organization are given in Appendix B.

2.1 - Using Files

A ProDOS **filename** or **volume name** is up to 15 characters long. It may contain capital letters (A-Z), digits (0-9), and periods (.), and it must begin with a letter. Lowercase letters are automatically converted to uppercase. A filename must be unique within its directory. Some examples are

```
LETTERS
JUNK1
BASIC.SYSTEM
```

By the Way: On the Apple II, an ASCII character is read from the keyboard and printed to the screen with its high bit set. ProDOS clears this high bit.

2.1.1 - Pathnames

A ProDOS **pathname** is a series of filenames, each preceded by a slash (/). The first filename in a pathname is the name of a volume directory. Successive filenames indicate the path, from the volume directory to the file, that ProDOS must follow to find a particular file. The maximum length for a pathname is 64 characters, including slashes. Examples are

```
/PROFILE/GAMES/DISKWARS
/PROFILE/JUNK1
/PROFILE/SYSTEMPROGRAMS/FILER
```

All calls that require you to name a file will accept either a pathname or a **partial pathname**. A partial pathname is a portion of a pathname that doesn't begin with a slash or a volume name. The maximum length for a partial pathname is 64 characters, including slashes. These partial pathnames are all derived from the sample pathnames above.

The partial pathnames are

```
DISKWARS
JUNK1
SYSTEMPROGRAMS/FILER
FILER
```

ProDOS automatically adds the **prefix** to the front of partial pathnames to form full pathnames. The prefix is a pathname that indicates a directory; it is internally stored by ProDOS. To locate a file by its

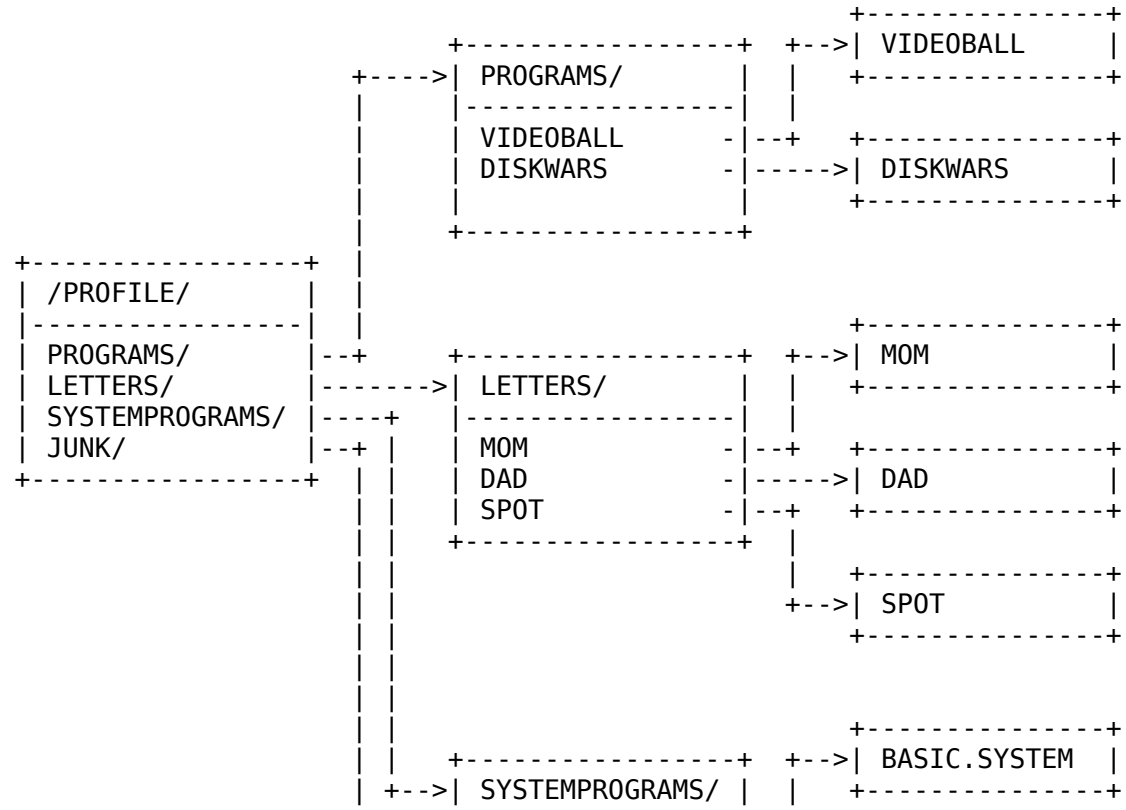
pathname, ProDOS must look through each file in the path. If you specify a partial pathname, however, ProDOS jumps straight to the prefix directory and starts searching from there. Thus disk accesses are faster when you set the prefix and use partial pathnames.

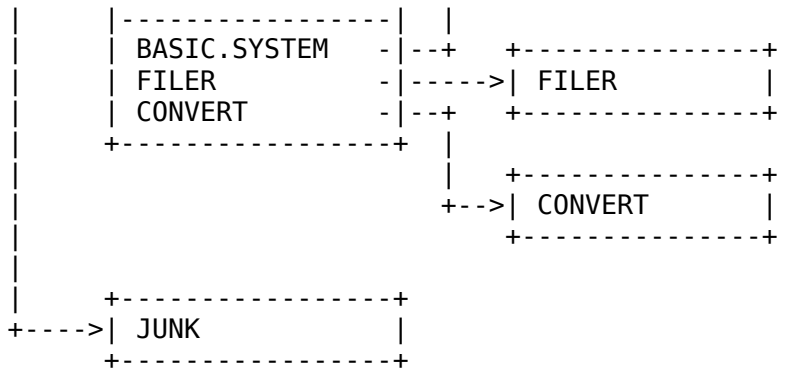
For the partial pathnames listed above to indicate valid files, the prefix should be set to **/PROFILE/GAMES/**, **/PROFILE/**, **/PROFILE/**, and **/PROFILE/SYSTEMPROGRAMS/**, respectively. The slashes at the end of these prefixes are optional; however, they are convenient reminders that prefixes indicate directory files.

The maximum length for a prefix is 64 characters. The minimum length for a prefix is zero characters, known as a **null prefix**. You set and read the prefix using the MLI calls, SET_PREFIX and GET_PREFIX, respectively. The 64 character limits for the prefix and partial pathname combine to create a maximum pathname of 128 characters.

Figure 2-1 illustrates a typical directory structure; it contains all the files mentioned above.

Figure 2-1. A Typical ProDOS Directory Structure





2.1.2 - Creating Files

A file is placed on a disk by the CREATE call. When you create a file, you assign it the following properties:

- A pathname. This pathname is a unique path by which the file can be identified and accessed. This pathname must place the file within an existing directory.
- An access byte. The value of this byte determines whether or not the file can be written to, read from, destroyed, or renamed.
- A file_type. This byte indicates to other system programs the type of information to be stored in the file. It does not affect, in any way, the contents of the file.
- A storage_type. This byte determines the physical format of the file on the disk. There are only two different formats: one is used for directory files, the other for non-directory files.
- A creation_date and a creation_time.

When you create a file, these properties are placed on the disk. The file's name can be changed using the RENAME call; other properties can be altered using the SET_FILE_INFO call. The disk storage format of these properties is given in Appendix B.

Once a file has been created, it remains on the disk until it is destroyed (using the DESTROY call).

2.1.3 - Opening Files

Before you can read information from or write information to a file you must use the OPEN call to open the file for access. When you open a file you specify:

- A pathname. This pathname must indicate a previously created file that is on a disk mounted in a disk drive.
- The starting address in memory of an I/O buffer. Each open file requires its own 1024-byte buffer for the transfer of information to and from the file.

The OPEN call returns a reference number (ref_num). All subsequent references to the open file must use this reference number. The file remains open until you use the CLOSE call.

Each open file's I/O buffer is used by the system the entire time the file is open. Thus it is wise to keep as few files open as possible. A maximum of eight files can be open at a time.

When you open a file, some of the file's characteristics are placed into a region of memory called a **file control block**. Several of these characteristics -- the location in memory of the file's buffer, a pointer to the end of the file (the EOF), and a pointer to the current position in the file (the file's MARK) -- are accessible to system programs via MLI calls, and may be changed while the file is open.

It is important to be aware of the differences between a file on the disk and an open file in memory. Although some of the file's characteristics and some of its data may be in memory at any given time, the file itself still resides on the disk. This allows ProDOS to manipulate files that are much larger than the computer's memory capacity. As a system program writes to the file and changes its characteristics, new data and characteristics are written to the disk.

Warning
In is crucial that you close all files before turning off the computer or pressing [CONTROL]-[RESET]. This is the only way than you can ensure that all written data has been placed on the disk. See also the FLUSH call.

2.1.4 - The EOF and MARK

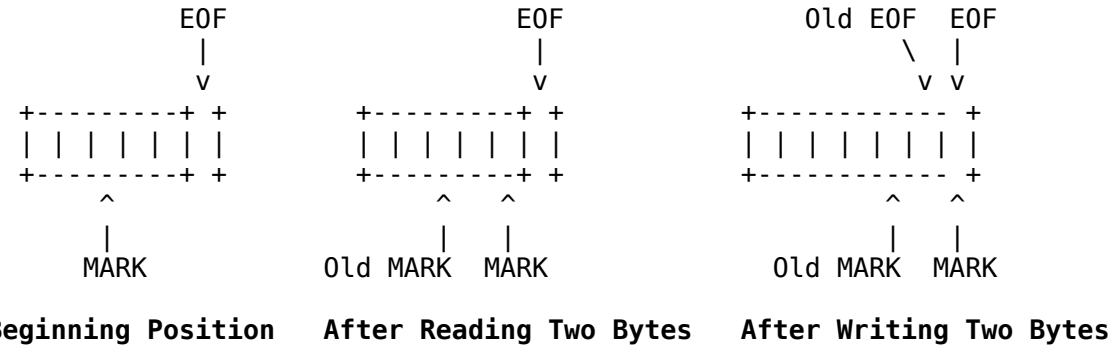
To aid the tasks of reading from and writing to files, each open file has one pointer indicating the end of the file, the EOF, and another defining the current position in the file, the MARK. Both are moved automatically by ProDOS, but can also be independently moved by the system program.

The EOF is the number of readable bytes in the file. Since the first byte in a file has number 0, the EOF, when treated as a pointer, points one position past the last character in the file.

When a file is opened, the MARK is set to indicate the first byte in the file. In is automatically moved forward one byte for each byte written to or read from the file. The MARK, then, always indicates the next byte to be read from the file, or the next byte position in which to write new data. It cannot exceed the EOF.

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 advances the EOF to accommodate the new information. Figure 2-2 illustrates the relationship between the MARK and the EOF.

Figure 2-2. Automatic Movement of EOF and MARK



A system program can place the EOF anywhere, from the current MARK position to the maximum possible byte position. The MARK can be placed anywhere from the first byte in the file to the EOF. These two functions can be accomplished using the SET_EOF and SET_MARK calls. The current values of the EOF and the MARK can be determined using the GET_EOF and GET_MARK calls.

2.1.5 Reading and Writing Files

READ and WRITE calls to the MLI transfer data between memory and a file. For both calls, the system program must specify three things:

- The reference number of the file (assigned when the file was opened).
- The location in memory of a buffer (data_buffer) that contains, or

is to contain, the transferred data. Note that this cannot be the same buffer that was specified when the file was opened.
- The number of bytes to be transferred.

When the request has been carried out, the MLI passes back to the system program the number of bytes that it actually transferred.

A read or write request starts at the current MARK, and continues until the requested number of bytes has been transferred (or, on a read, until the end of file has been reached). Read requests can also terminate when a specified character is read. You turn on this feature and set the character(s) on which reads will terminate using the NEWLINE call. It is typically used for reading lines of text that are terminated by carriage returns.

By the Way: Neither a READ nor a WRITE call necessarily causes a disk access. It is only when a read or write crosses a 512-byte (block) boundary that a disk access occurs.

2.1.6 - Closing and Flushing Files

When you finish reading from or writing to a file, you must use the CLOSE call to close the file. When you use this call, you specify

- the reference number of the file (assigned when the file was opened).

CLOSE writes any unwritten data to the file, and it updates the file's size in the directory, if necessary. Then it frees the 1024-byte io_buffer for other uses and releases the file's reference number.

Information in the file's directory, such as the file's size, is normally updated only when the file is closed. If you were to press [CONTROL]-[RESET] (typically halting the current program) while a file is open, data written to the file since it was opened could be lost and the integrity of the disk could be damaged. This can be prevented by using the FLUSH call. To use FLUSH you specify

- the reference number of the file (assigned when the file was opened).

If you press [CONTROL]-[RESET] while an open but flushed file is in

memory, there is no loss of data and no damage to the disk.
Both the CLOSE and FLUSH calls, when used with a reference number of 0, normally cause all open files to be closed or flushed. Specific groups of files can be closed or flushed using the **system level**.

2.1.7 - File Levels

When a file is opened, it is assigned a level, according to the value of a specific byte in memory (the system level). If the system level is never changed, the CLOSE and FLUSH calls, when used with a reference number of 0, cause all open files to be closed or flushed. But if the level has been changed since the first file was opened, only the files having a file level greater than or equal to the current system level are closed or flushed.

The system level feature is used, for example, by the BASIC system program to implement the EXEC command. An EXEC file is opened with a level of 0, then the level is set to 7. A BASIC CLOSE command (intended to close all files opened within the EXEC program) closes all files at or above level 7, but the EXEC file itself remains open.

2.2 - File Organization

This portion of the chapter describes in general terms the organization of files on a disk. It does not attempt to teach you everything about file organization: its purpose is to familiarize you with the terms and concepts required by the filing calls.

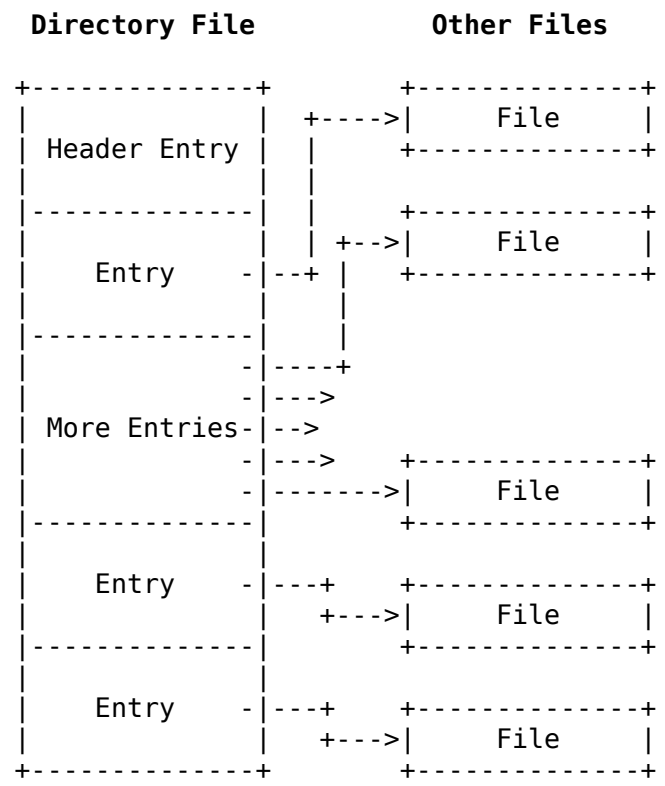
Appendix B elaborates on the subject of
file organization.

2.2.1 - Directory Files and Standard Files

Every ProDOS file is a named, ordered sequence of bytes that can be read from, and to which the rules of MARK and EOF apply. However, there are two types of files: **directory files** and **standard files**. Directory files are special files that describe and point to other files on the disk. They may be read from, but not written to (except by ProDOS). All nondirectory files are standard files. They may be read from and written to.

A directory file contains a number of similar elements, called **entries**. The first entry in a directory file is the header entry: it holds the name and other properties (such as the number of files stored in that directory) of the directory file. Each subsequent entry in the file describes and points to some other file on the disk. Figure 2-3 represents the structure of a directory file.

Figure 2-3. Directory File Structure



The files described and pointed to by the entries in a directory file can be standard files or other directory files.

A system program does not need to know the details of directory structure to access files with known names. Only operations on unknown files (such as listing the files in a directory) require the system program to examine a directory's entries. For such tasks, refer to Appendix B.

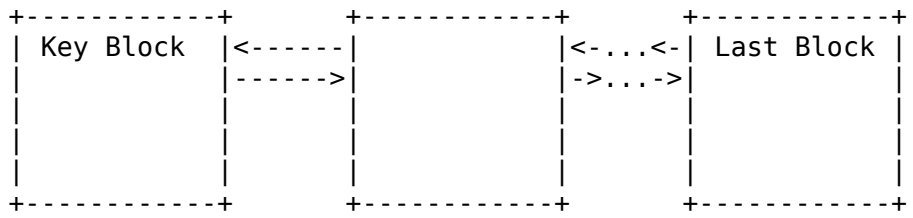
Standard files have no such predefined internal structure: the format of the data depends on the specific file type.

2.2.2 - File Structure

Because directory files are generally smaller than standard files, and because they are sequentially accessed, ProDOS uses a simpler form of storage for directory files. Both types of files are stored as a set of 512-byte blocks, but the way in which the blocks are arranged on the disk differs.

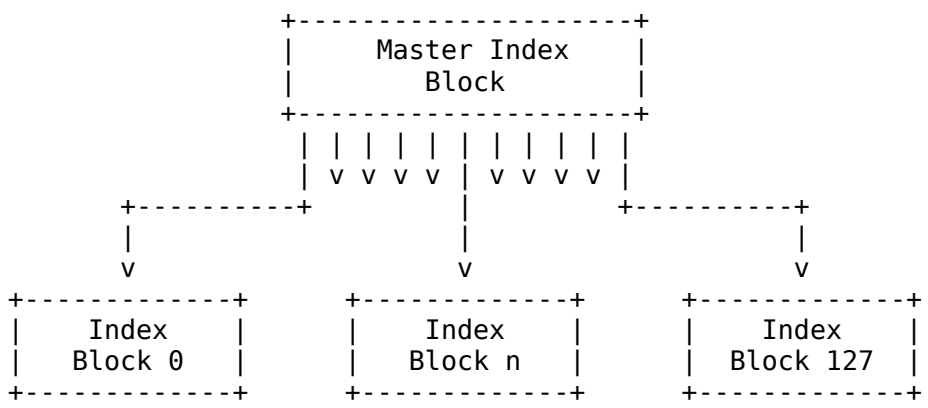
A directory file is a linked list of blocks: each block in a directory file contains a pointer to the next block in the directory file as well as a pointer to the previous block in the directory. Figure 2-4 illustrates this structure.

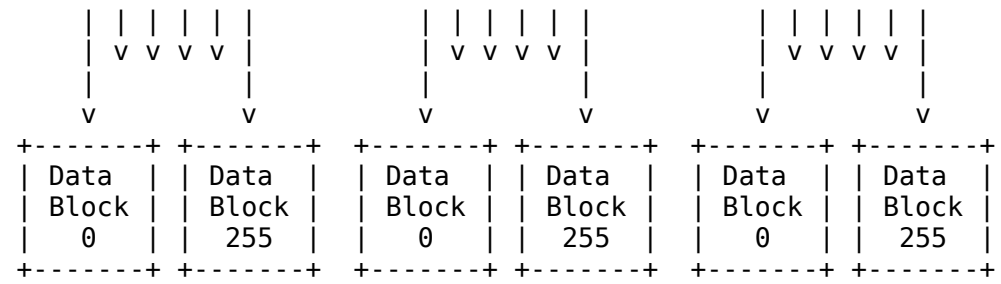
Figure 2-4. Block Organization of a Directory File



Data files, on the other hand, are often quite large, and their contents may be randomly accessed. It would be very slow to access such large files if they were organized sequentially. Instead ProDOS stores standard files using a **tree structure**. The largest possible standard file has a **master index block** that points to 128 **index blocks**. Each index block points to 256 **data blocks** and each data block can hold 512 bytes of data. The block organization of the largest possible standard file is shown in Figure 2-5.

Figure 2-5. Block Organization of a Standard File





Most standard files do not have this exact organization. ProDOS only writes a subset of this structure to the file, depending on the amount of data written to the file. This technique produces three distinct forms of standard file: seedling, sapling, and tree files.

Appendix B describes the three forms of standard file.

2.2.3 - Sparse Files

In most instances a program writes data sequentially into a file. By writing data, moving the EOF and MARK, and then writing more data, a program can also write nonsequential data to a file. For example, a program can open a file, write ten characters of data, and then move the EOF and MARK (thereby making the file bigger) to $3FE0 before writing ten more bytes of data. The file produced takes up only three blocks on the disk (a total of 1536 bytes), yet over 16,000 bytes can be read from the file. Such files are known as **sparse files**.

> **Important!**
> The fact that more data can be read from the file than actually resides on the disk can cause a problem. Suppose that you were trying to copy a sparse file from one disk to another. If you were to read data from one file and write it to another, the new file would be much larger than the original because data that is not actually on the disk can be read from the file. Thus if your system program is going to transfer sparse files, you must use the information in Appendix B to determine which data segments should be copied, and which should not.
>
> The ProDOS Filer automatically preserves the structure of sparse files on a copy.

Chapter 3
Memory Use

This chapter explains the way the Machine Language Interface uses memory. It tells how much memory system programs have available to them, how system programs should manage this free memory, and it discusses the contents of important areas of memory while ProDOS is inn use.

3.1 - Loading Sequence

When you start up your Apple II from a ProDOS startup disk -- one that contains both the MLI (ProDOS) and a system program (XXX.SYSTEM) -- a complex loading sequence is initiated.

A preliminary loading program is stored in the read-only memory (**boot ROM**) on a disk drive's controller card; the main part of the **loader program**, as it is called, resides in blocks 0 and 1 of every ProDOS-formatted disk.

When you turn on your computer, or use a **PR#** or **IN#** command to reference a disk drive from Applesoft, or otherwise transfer control to the ROM on the disk-drive controller card when a ProDOS startup disk is in the drive, this is what happens:

1. The program in the ROM reads the loader program from blocks 0 and 1 of the disk, places it into memory starting at location $800, and then executes it.
2. This loader program looks for the file with the name PRODOS and type $FF (containing the MLI) in the volume directory of the startup disk, loads it into memory starting at location $2000, and executes it.
3. The MLI ascertains the computer's memory size and moves itself to its final location, as shown in Figure 3-1. Next it determines what devices are in what slots and it sets up the **system global page**, described in the section "The System Global Page," for this system configuration.
4. The MLI then searches the volume directory of the boot disk for the first file with the name XXX.SYSTEM and type $FF, loads it into memory starting at $2000, and executes it.

If PRODOS cannot be found, the loader reports to the user that it is unable to load ProDOS. If no XXX.SYSTEM program is found, ProDOS displays the message **UNABLE TO FIND A SYSTEM FILE**.

The rules for system programs are described in Chapter 5.

The MLI is entirely memory resident. Once it is in memory, it neither moves, nor does it require any additional disk accesses (although the system program might). The memory configuration that results from this loading process is described in the section "Memory Map."

3.2 - Volume Search Order

When a program or user requests access to a volume that ProDOS has not yet accessed, it must search through the volumes that are currently online for the requested volume. The order in which it searches the devices is determined during step 3 above.

The first volume checked is /RAM, if present, then the startup volume (generally slot 6, drive 1). The search then checks slots in descending slot order, starting with slot 7. In any slot, drive 1 is searched before drive 2.

For example, if there are two Disk II drives in slot 6, two Disk II drives in slot 5, and a ProFile in slot 7, the search order is:

/RAM
Slot 6, drive 1
Slot 6, drive 2
Slot 7
Slot 5, drive 1
Slot 5, drive 2

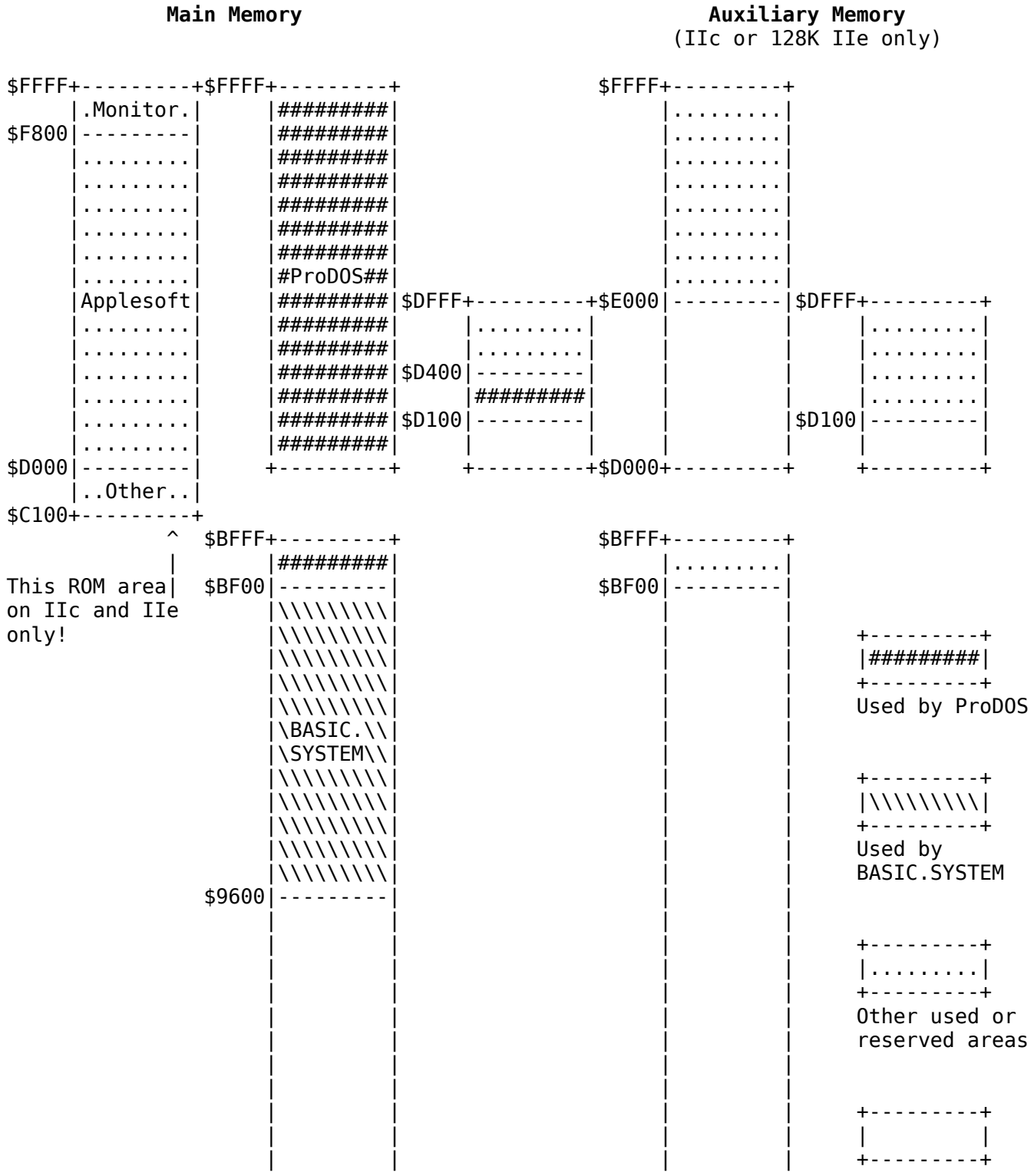
The startup volume is the volume in the highest numbered slot that can be identified by the system as a startup volume. This sequence is kept in the device list in the ProDOS global page and can be altered.

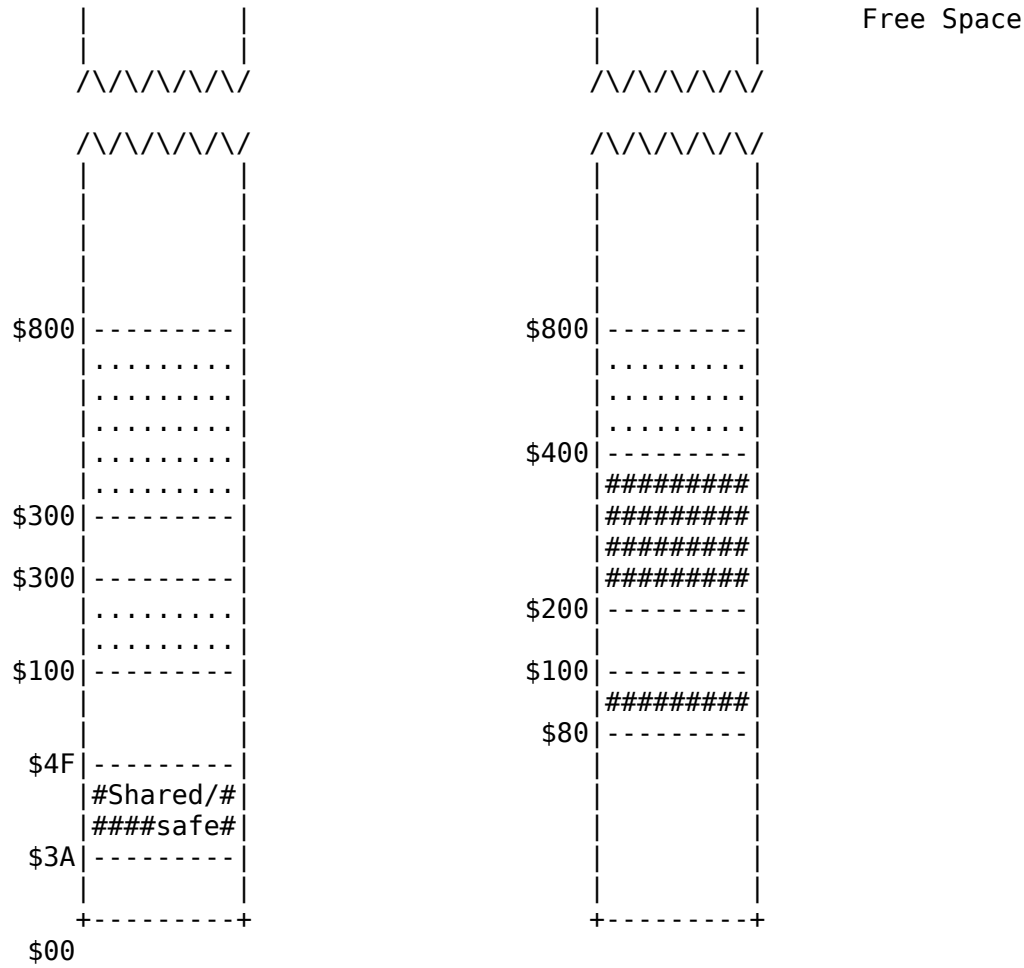
Note: If the startup volume is a hard disk, the search order is from slot 7 to slot 1.

3.3 - Memory Map

ProDOS requires at least 64 kilobytes of memory. Figure 3-1 is the ProDOS memory map.

Figure 3-1. Memory Map





A system program as large as $8F00 (36608) bytes can be loaded into a 64K system. The total amount of space available to a system program running on a 64K system is $B700 (46848) bytes.

3.3.1 - Zero Page

The ProDOS Machine Language Interface uses zero-page locations $40-$4E, but it restores them before it completes a call. The disk-driver routines, called by the MLI, use locations $3A through $3F. These locations are not restored. See Chapter 4 for details.

3.3.2 - The System Global Page

The $BF-page of memory, addresses $BF00 through $BFFF, contains the system's global variables. This section of memory is special because no matter what system ProDOS is booted on, the global page is always in the same location. Because of this it serves as the communication link between system programs and the operating system. The MLI

places all information that might be useful to a system program in these locations. These locations are defined and described in Chapter 5.

3.3.3 - The System Bit Map

ProDOS uses a simple form of memory management that allows it to protect itself and the user's data from being overwritten by ProDOS buffer allocation. It represents the lower 48K of the Apple II's random-access memory using twenty-four bytes of the system global page: one bit for each 256-byte page of RAM in the lower 48K of the Apple II. These twenty-four bytes are called the **system bit map**.

When ProDOS is started up, it protects the zero page, the stack, and the global page, by setting the bits that correspond to the used pages. If at all possible, a system program should not use pages of memory that are already used. If this is not possible, the system program must close all files and clear the bit map, leaving pages 0, 1, 4 through 7, and BF (zero page, stack, text, and ProDOS global page) protected. If an error occurs on the close, the program should ask the user to restart the system. See Chapter 5 for details.

While a system program is using the MLI, there are only three calls that affect the setting of the bit map: OPEN, CLOSE, and SET_BUF. When the system program opens a file, it must specify the starting address of a 1024-byte file buffer. As long as the file is open, this buffer is a part of the system, and is marked off in the bit map. When the file is closed, the buffer is released, and its bits are cleared.

In general, a system program requires the used pages of memory to be contiguous, or touching. This leaves the maximum possible unbroken memory space for the reading and manipulation of data. Suppose a system program opens several files and then closes the one that was opened first. In most cases, this causes a vacant 1K area to appear. The GET_BUF and SET_BUF calls can be used to find this vacant area, and to move another file's buffer into this space.

Refer to Chapter 5 for a specific example
of using the system bit map.

Chapter 4
Calls to the MLI

This chapter is about the ProDOS Machine Language Interface (MLI), which provides a simple way to use disk files from machine-language programs. This chapter describes

- the organization of the MLI on a functional basis
- how to make calls to the MLI from machine-language programs
- the MLI calls themselves
- the MLI error codes.

4.1 - The Machine Language Interface

The ProDOS MLI is a complete, consistent, and interruptible interface between the machine-language programmer and files on disks. It is entirely independent of the ProDOS BASIC system program; thus, it serves as a base upon which other system programs can be written. Its filename is PRODOS. It consists of:

- the **Command Dispatcher**, which accepts and dispatches calls from a machine-language program. It validates each call's parameters, updates the system global page, and then jumps to the appropriate routine of the Block File Manager.
- the **Block File Manager**, which carries out all valid calls to the MLI. The Block File Manager keeps track of all mounted disks, manages the condition of all opened files, and does some simple memory mangagement. It performs all disk access (reads and writes) via calls to disk-driver routines.
- **Disk Driver Routines**, which perform the reading and writing of data.
- the **Interrupt Handler**, which allows up to four interrupt handling routines to be attached to ProDOS. The Interrupt Handler keeps four vectors to interrupt routines. When an interrupt occurs, these routines are called, in sequence, until one of them claims the interrupt.

4.2 - Issuing a Call to the MLI

A program sends a call to the Machine Language Interface by executing a JSR (jump to subroutine) to address $BF00 (referred to below as MLI). The call number and a two-byte pointer (low byte first) to the call's parameter list must immediately follow the call. Here is an example of a call to the MLI:

```
SYSCALL  JSR MLI         ;Call Command Dispatcher
         DB  CMDNUM      ;This determines which call is being made
         DW  CMDLIST     ;A two-byte pointer to the parameter list
         BNE ERROR       ;Error if nonzero
```

Upon completion of the call, the MLI returns to the address of the JSR plus 3 (in the above example, the BNE statement); the call number and parameter list pointer are skipped. If the call is successful, the C-flag is cleared and the Accumulator is set to zero. If the call is unsuccessful, the C-flag is set and the Accumulator is set to the error code. The register status upon call completion is summarized below. Note that the value of the N-flag is determined by the Accumulator and that the value of the V-flag is undefined.

```
                    N  Z  C  D  V   Acc    PC    X  Y  SP
Successful call:    0  1  0  0  x    0    JSR+3  unchanged
Unsuccessful call:  x  0  1  0  x  error  JSR+3  unchanged
                                      code
```

Here is an example of a small program that issues calls to the MLI. It tries to create a text file named NEWFILE on a volume named TESTMLI. If an error occurs, the Apple II beeps and prints the error code on the screen. Both the source and the object are given so you can enter it from the Monitor if you wish (remember to use a formatted disk named /TESTMLI).

```
-------------------------------------------------------------------------
SOURCE   FILE #01 =>TESTCMD
----- NEXT OBJECT FILE NAME IS TESTCMD.0
2000:        2000    1         ORG  $2000
2000:        2000    1         ORG  $2000
2000:                2 *
2000:        FF3A    3 BELL    EQU  $FF3A     ;Monitor BELL routine
2000:        FD8E    4 CROUT   EQU  $FD8E     ;Monitor CROUT routine
2000:        FDDA    5 PRBYTE  EQU  $FDDA     ;Monitor PRBYTE routine
2000:        BF00    6 MLI     EQU  $BF00     ;ProDOS system call
2000:        00C0    7 CRECMD  EQU  $C0       ;CREATE command number
2000:                8 *
2000:20 06 20        9 MAIN    JSR  CREATE    ;CREATE "/TESTMLI/NEWFILE"
2003:D0 08   200D   10         BNE  ERROR     ;If error, display it
2005:60             11         RTS            ;Otherwise done
```

```
2006:                12 *
2006:20 00 BF        13 CREATE  JSR  MLI       ;Perform call
2009:C0              14         DFB  CRECMD    ;CREATE command number
200A:17 20           15         DW   CRELIST   ;Pointer to parameter list
200C:60              16         RTS
200D:                17 *
200D:20 DA FD        18 ERROR   JSR  PRBYTE    ;Print error code
2010:20 3A FF        19         JSR  BELL      ;Ring the bell
2013:20 8E FD        20         JSR  CROUT     ;Print a carriage return
2016:60              21         RTS
2017:                22 *
2017:07              23 CRELIST DFB  7         ;Seven parameters
2018:23 20           24         DW   FILENAME  ;Pointer to filename
201A:C3              25         DFB  $C3       ;Normal file access permitted
201B:04              26         DFB  $04       ;Make it a text file
201C:00 00           27         DFB  $00,$00   ;AUX_TYPE, not used
201E:01              28         DFB  $01       ;Standard file
201F:00 00           29         DFB  $00,$00   ;Creation date (unused)
2021:00 00           30         DFB  $00,$00   ;Creation time (unused)
2023:                31 *
2023:10              32 FILENAME DFB ENDNAME-NAME ;Length of name
2024:2F 54 45 53     33 NAME    ASC  "/TESTMLI/NEWFILE" ;followed by the name
2034:        2034    34 ENDNAME EQU  *
--------------------------------------------------------------------------
```

The parameters used in **TESTCMD** are explained in the following sections. The MLI error codes are summarized in Section 4.7.

4.2.1 - Parameter Lists

As defined above, each MLI call has a two-byte pointer to a parameter list. A parameter list contains information to be used by the call and space for information to be returned by the call. There are three types of elements used in parameter lists: values, results, and pointers.

A **value** is a one or more byte quantity that is passed to the Block File Manager (BFM). Values help determine the action taken by the BFM.

A **result** is a one or more byte space in the parameter list into which the Block File Manager will place a value. From results, programs can get information about the status of a volume, file, or interrupt, or about the success of the call just completed.

A **pointer** is a two-byte memory address that indicates the location of data, code, or a space in which the Block File Manager can place or receive data. All pointers are arranged low byte first, high byte second.

The first element in every parameter list is the **parameter count**, a

one-byte value that indicates the number of parameters used by the call (not including the parameter count). This byte is used to verify that the call was not accidental.

4.2.2 - The ProDOS Machine Language Exerciser

To help you learn to use the ProDOS Machine Language Interface, there is a useful little program called the ProDOS Machine Language Exerciser. It allows you to execute MLI calls from a menu; it has a hexadecimal memory editor for reviewing and altering the contents of buffers; and it has a catalog command.

Instructions for using the Machine Language Exerciser program are in Appendix D.

When you use it to make an MLI call, you request the call by its call number, then you specify its parameter list, just as if you were coding the call in a program. When you press [RETURN], the call is executed. Using the Exerciser, you can try out sequences of MLI calls before actually coding them.

4.3 - The MLI Calls

The MLI calls can be divided into three groups: housekeeping calls, filing calls, and system calls.

4.3.1 - Housekeeping Calls

The housekeeping calls perform operations such as creating, deleting, and renaming, which cannot be used on open files. They are used to change a file's status, but not the information that is in the file. They refer to files by their pathnames, and each requires a temporary buffer, which is used during execution of the call. The housekeeping calls are:

CREATE
: Creates either a standard file or a directory file. An entry for the file is placed in the proper directory on the disk, and one block of disk space is

allocated to the file.

DESTROY
: Removes a standard file or directory file. The entry for the file is removed from the directory and all the file's disk space is released. If a directory is to be destroyed, it must be empty. A volume directory cannot be destroyed except by reformatting the volume.

RENAME
: Changes the name of a file. The new name must be in the same directory as the old name. This call changes the name in the entry that describes that file, and if it is a directory file, also the name in its header entry.

SET_FILE_INFO
: Sets the file's type, the way it may be accessed, and/or its modification date and time.

GET_FILE_INFO
: Returns the file's type, the way it may be accessed, the way it is stored on the disk, its size in blocks, and the date and time at which it was created and last modified.

ON_LINE
: Returns the slot number, drive number, and volume name of one or all mounted volumes. This information is placed in a user-supplied buffer.

SET_PREFIX
: Sets the pathname that is used by the operating system as a prefix. The prefix must indicate an existing directory on a mounted volume.

GET_PREFIX
: Returns the value of the current system prefix.

4.3.2 - Filing Calls

The filing calls cause the transfer of data to or from files. The first filing call, OPEN, must be used before any of the others can be used. The OPEN call specifies a file by its pathname; the other filing calls refer to files by the reference number returned by the OPEN call. In addition, an input/output buffer (io_buffer), is allocated to the open file; subsequent data transfers go through this buffer. The reference number remains assigned and the buffer remains allocated until the file is closed. The filing calls are:

OPEN
: Prepares a file to be accessed. This call causes a file control block (FCB) to be allocated to the file, and a reference number to be returned (A reference number is really a file control block number). In addition, an input/output buffer is allocated for data transfers to and from the file.

NEWLINE
: Sets conditions for reading from the file. This call turns on and turns off the capability of read requests to terminate when a particular character (such as a carriage return) is read.

READ
: Causes the transfer of a requested number of characters from a file to a specified memory buffer, and updates the current position (MARK) in the file. Characters are read according to the rules set by the NEWLINE call.

WRITE
: Causes the transfer of a requested

number of characters from a specified buffer to a file, and updates the current position (MARK) in the file and the end of file (EOF), if necessary.

CLOSE
: Transfers any unwritten data from a file's input/output buffer to the file, releases the file's io_buffer and file control block, and updates the file's directory entry, if necessary. The file's reference number is released for use by subsequently opened files.

FLUSH
: Transfers any unwritten data from a file's input/output buffer to the file, and updates the file's directory entry, if necessary.

SET_MARK
: Changes the current position in the file. The current position is the absolute position in the file of the next character to be read or written.

GET_MARK
: Returns the current position in the file. The current position is the absolute position in the file of the next character to be read or written.

SET_EOF
: Changes the logical size of the file (the end of file).

GET_EOF
: Returns the logical size of the file.

SET_BUF
: Assigns a new location for the input/output buffer of an open file.

GET_BUF
: Returns the current location of the

input/output buffer of an open file.

4.3.3 - System Calls

System calls are those calls that are neither housekeeping nor filing calls. They are used for getting the current date and time, for installing and removing interrupt routines, and for reading and writing specific blocks of a disk. The system calls are:

GET_TIME
: If your system has a clock/calendar card, and if a routine that can read from the clock is installed, then it places the current date and time in the system date and time locations.

ALLOC_INTERRUPT
: Places a pointer to an interrupt-handling routine into the system interrupt vector table.

DEALLOC_INTERRUPT
: Removes a pointer to an interrupt handling routine from the system interrupt vector table.

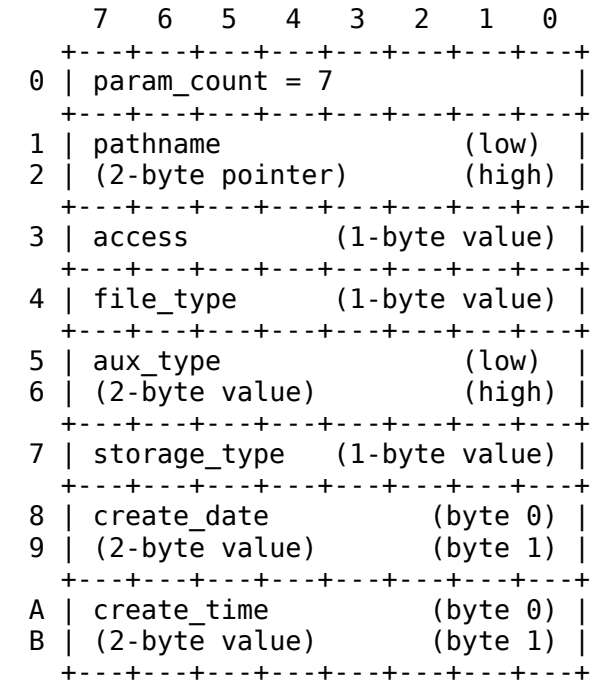
READ_BLOCK
: Reads one specific block (512 bytes) of information from a disk into a user specified data buffer. This call is file independent.

WRITE_BLOCK
: Writes a block of information from a user specified data buffer to a specific block of a disk. This call is file independent.

4.4 - Housekeeping Calls

Each of the following sections contains a description of a housekeeping call, including its parameters and the possible errors that may be returned.

4.4.1 - CREATE ($C0)



Every disk file except the volume directory file must be created using this call. There are two organizationally distinct types of file storage: tree structure (storage_type = $1), used for standard files; and linked list (storage_type = $D), used for directory files.

Pathname specifies the name of the file to be created and the directory in which to insert an entry for the new file. One block (512 bytes) of disk space is allocated, and the entry's key_pointer field is set to indicate that block. Access, in most cases, should be set to $E3 (full access permitted). File_type and aux_type may be anything, but it is strongly recommended that conventions be followed (see below).

Parameters

param_count
(1-byte value)
Parameter count: 7 for this call.

pathname
(2-byte pointer)

Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the prefix is attached to the front to make a full pathname. The pathname string is not changed.

access
(1-byte value)

Access permitted: This byte defines how the file will be accessible. Its format is:

```
  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+
|D |RN|B |Reserved|W |R |
+--+--+--+--+--+--+--+--+

D:   Destroy enable bit
RN:  Rename enable bit
B:   Backup needed bit
W:   Write enable bit
R:   Read enable bit
```

For all bits, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved for future definition and must always be disabled. Usually access should be set to $C3.

If the file is destroy, rename, and write enabled, it is **unlocked**. If all three are disabled, it is **locked**. Any other combination of access bits is called **restricted access**.

The backup bit (B) is always set by this call.

file_type
(1-byte value)

File type: This byte describes the contents of the file. The currently defined file types are listed below.

File Type	Preferred Use

```
$00             Typeless file (SOS and ProDOS)
$01             Bad block file
$02 *           Pascal code file
$03 *           Pascal text file
$04             ASCII text file (SOS and ProDOS)
$05 *           Pascal data file
$06             General binary file (SOS and ProDOS)
$07 *           Font file
$08             Graphics screen file
$09 *           Business BASIC program file
$0A *           Business BASIC data file
$0B *           Word Processor file
$0C *           SOS system file
$0D,$0E *       SOS reserved
$0F             Directory file (SOS and ProDOS)
$10 *           RPS data file
$11 *           RPS index file
$12 *           AppleFile discard file
$13 *           AppleFile model file
$14 *           AppleFile report format file
$15 *           Screen library file
$16-$18 *       SOS reserved
$19             AppleWorks Data Base file
$1A             AppleWorks Word Processor file
$1B             AppleWorks Spreadsheet file
$1C-$EE         Reserved
$EF             Pascal area
$F0             ProDOS added command file
$F1-$F8         ProDOS user defined files 1-8
$F9             ProDOS reserved
$FA             Integer BASIC program file
$FB             Integer BASIC variable file
$FC             Applesoft program file
$FD             Applesoft variables file
$FE             Relocatable code file (EDASM)
$FF             ProDOS system file
```

Note: The file types marked with a * in the above list apply to Apple III SOS only; they are not used by ProDOS. For the file_types used by Apple III SOS only, refer to the *SOS Reference Manual*.

---

aux_type
(2-byte value)

Auxiliary type: This two-byte field is used by the system program. The BASIC system program uses it (low byte first) to store text-file record size or binary-file load address, depending on the file_type.

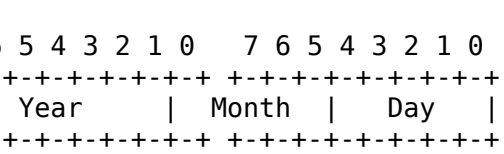
storage_type
(1-byte value)

File kind: This byte describes the physical

organization of the file. storage_type = $0D is a linked directory file; storage_type = $01 is a standard file.

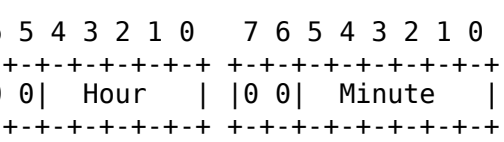
create_date
(2-byte value)

This 2-byte field may contain the date on which the file was created. Its format is:



create_time
(2-byte value)

This 2-byte field may contain the time at which the file was created. Its format is:



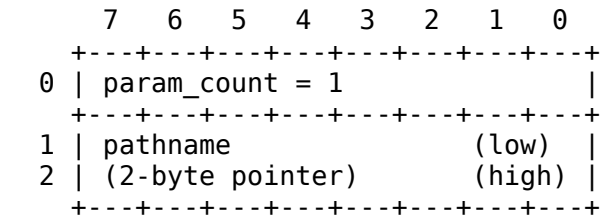
See Chapter 6 for information about the use of ProDOS with a clock/calendar card.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$40 - Invalid pathname syntax
$44 - Path not found
$45 - Volume directory not found
$46 - File not found
$47 - Duplicate filename
$48 - Overrun error: not enough disk space
$49 - Directory full
-- ProDOS can have no more than 51 files in a volume directory.
$4B - Unsupported storage_type
$53 - Invalid parameter

$5A - Bit map disk address is impossible

4.4.2 - DESTROY ($C1)



Deletes the file specified by pathname by removing its entry from the directory that owns it, and by returning its blocks to the volume bit map. Volume directory files and open files cannot be destroyed. Subdirectory files must be empty before they can be destroyed.

Parameters

param_count
(1-byte value)
: Parameter count: 1 for this call.

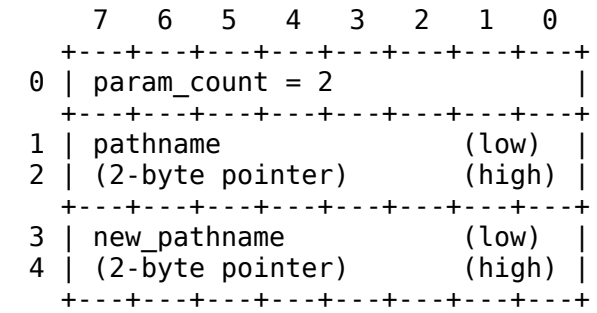
pathname
(2-byte pointer)
: Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the prefix is attached to the front to make a full pathname.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$40 - Invalid pathname syntax
$44 - Path not found
$45 - Volume directory not found
$46 - File not found
$4A - Incompatible file format
$4B - Unsupported storage_type

$4E - Access error: destroy not enabled
$50 - File is open: request denied
$5A - Bit map disk address is impossible

4.4.3 - RENAME ($C2)



Changes the name of the file specified by pathname to that specified by new_pathname. Both pathname and new_pathname must be identical except for the rightmost filename (they must indicate files in the same directory). For example, the path /EGG/ROLL can be renamed /EGG/PLANT, but not /JELLY/ROLL or /EGG/DRUM/ROLL.

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

pathname
(2-byte pointer)
: Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the prefix is attached to the front to make a full pathname.

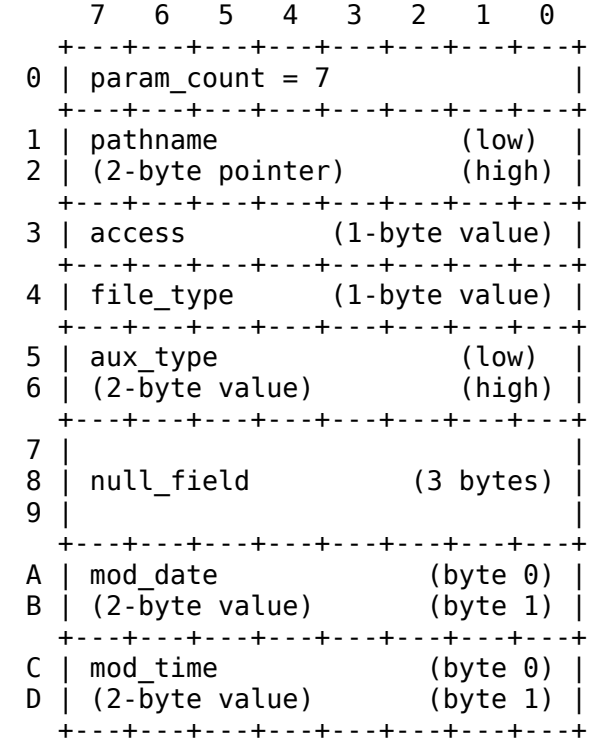
new_pathname
(2-byte pointer)
: New pathname pointer: This two-byte pointer (low byte first) indicates the location of the new pathname. It has the same syntax as pathname.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$40 - Invalid pathname syntax
$44 - Path not found
$45 - Volume directory not found
$46 - File not found

$47 - Duplicate filename
$4A - Incompatible file format
$4B - Unsupported storage_type
$4E - Access error: rename not enabled
$50 - File is open: request denied
$57 - Duplicate volume

4.4.4 - SET_FILE_INFO ($C3)



Modifies information in the specified file's entry field. This call can be performed when the file is either open or closed. However, new access attributes are not used by an open file until the next time the file is opened (that is, this call doesn't modify existing file control blocks).

You should use the GET_FILE_INFO call to read a file's attributes

into a parameter list, modify them as needed, and then use the same parameter list for the SET_FILE_INFO call.

Parameters

param_count
(1-byte value)
Parameter count: 7 for this call.

pathname
(2-byte pointer)
Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the prefix is attached to the front to make a full pathname.

access
(1-byte value)
Access permitted: This byte determines how the file may be accessed. Its format is:

```
  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+
|D |RN|B |Reserved|W |R |
+--+--+--+--+--+--+--+--+

D:   Destroy enable bit
RN:  Rename enable bit
B:   Backup needed bit
W:   Write enable bit
R:   Read enable bit
```

For all bits, 1 = enabled, 0 = disabled. Bits 2 through 4 are used internally and should be set to 0. Usually access should be set to $C3.
If the file is destroy, rename, and write enabled it is **unlocked**. If all three are disabled, it is **locked**. Any other combination of access bits is called **restricted access**.
The backup bit (B) is set by this call.

file_type
(1-byte value)

File type: This byte describes the contents of a file. The currently defined file types are listed below.

File Type	Preferred Use
$00	Typeless file (SOS and ProDOS)
$01	Bad block file
$02 *	Pascal code file
$03 *	Pascal text file
$04	ASCII text file (SOS and ProDOS)
$05 *	Pascal data file
$06	General binary file (SOS and ProDOS)
$07 *	Font file
$08	Graphics screen file
$09 *	Business BASIC program file
$0A *	Business BASIC data file
$0B *	Word Processor file
$0C *	SOS system file
$0D,$0E *	SOS reserved
$0F	Directory file (SOS and ProDOS)
$10 *	RPS data file
$11 *	RPS index file
$12 *	AppleFile discard file
$13 *	AppleFile model file
$14 *	AppleFile report format file
$15 *	Screen library file
$16-$18 *	SOS reserved
$19	AppleWorks Data Base file
$1A	AppleWorks Word Processor file
$1B	AppleWorks Spreadsheet file
$1C-$EE	Reserved
$EF	Pascal area
$F0	ProDOS added command file
$F1-$F8	ProDOS user defined files 1-8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS system file

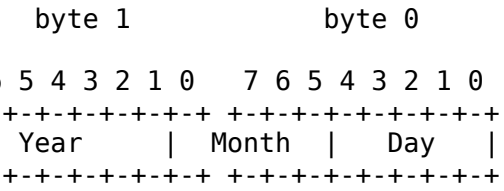
Note: The file types marked with a * in the above list apply to Apple III SOS only; they are not used by ProDOS. For the file_types used by Apple III SOS only, refer to the *SOS Reference Manual*.

aux_type
(2-byte value)

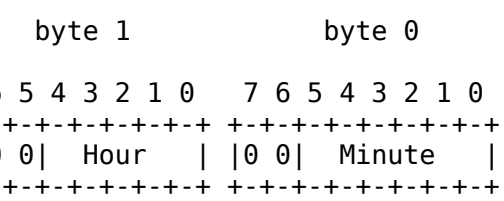
Auxiliary type: This two-byte field is used by the system program. The BASIC system program uses it (low byte first) to store record size or load address, depending on the file_type.

null_field
(3 bytes)
Null field: These three bytes preserve symmetry between this and the GET_FILE_INFO call.

mod_date
(2-byte value)
This 2-byte field should contain the current date. It has this format:



mod_time
(2-byte value)
This 2-byte field should contain the current time. It has this format:



See Chapter 6 for information about the use or ProDOS with a clock/calendar card.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$40 - Invalid pathname syntax
$44 - Path not found
$45 - Volume directory not found
$46 - File not found
$4A - Incompatible file format
$4B - Unsupported storage_type
$4E - Access error: file not write enabled

$53 - Invalid value in parameter list
$5A - Bit map disk address is impossible

4.4.5 - GET_FILE_INFO ($C4)

```
      7   6   5   4   3   2   1   0
    +---+---+---+---+---+---+---+---+
  0 | param_count = $A              |
    +---+---+---+---+---+---+---+---+
  1 | pathname               (low)  |
  2 | (2-byte pointer)       (high) |
    +---+---+---+---+---+---+---+---+
  3 | access        (1-byte result) |
    +---+---+---+---+---+---+---+---+
  4 | file_type     (1-byte result) |
    +---+---+---+---+---+---+---+---+
  5 | aux_type               (low)  | *
  6 | (2-byte result)        (high) |
    +---+---+---+---+---+---+---+---+
  7 | storage_type  (1-byte result) |
    +---+---+---+---+---+---+---+---+
  8 | blocks used            (low)  | *
  9 | (2-byte result)        (high) |
    +---+---+---+---+---+---+---+---+
  A | mod_date             (byte 0) |
  B | (2-byte result)      (byte 1) |
    +---+---+---+---+---+---+---+---+
  C | mod_time             (byte 0) |
  D | (2-byte result)      (byte 1) |
    +---+---+---+---+---+---+---+---+
  E | create_date          (byte 0) |
  F | (2-byte result)      (byte 1) |
    +---+---+---+---+---+---+---+---+
 10 | create_time          (byte 0) |
 11 | (2-byte result)      (byte 1) |
    +---+---+---+---+---+---+---+---+
```

* When file information about a volume directory is requested, the total number of blocks on the volume is returned in the aux_type field and the total blocks for all files is returned in blocks_used.

GET_FILE_INFO returns the information that is stored in the specified file's entry field. This call can be performed whether the file is open or closed. If the SET_FILE_INFO call is used to change the access while the file is open, the change does not take effect until the file has been closed and reopened.

Parameters

param_count
(1-byte value)

Parameter count: $A for this call.

pathname
(2-byte pointer)

Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the prefix is attached to the front to make a full pathname.

access
(1-byte result)

Access permitted: This byte determines how the file may be accessed. Its format is:

```
  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+
|D |RN|B |Reserved|W |R |
+--+--+--+--+--+--+--+--+

D:   Destroy enable bit
RN:  Rename enable bit
B:   Backup needed bit
W:   Write enable bit
R:   Read enable bit
```

For all bits, 1 = enabled, 0 = disabled. Bits 2 through 4 are not used. Usually access should be set to $C3.
If the file is destroy, rename, and write enabled it is **unlocked**. If all three are disabled, it is **locked**. Any other combination of access bits is called **restricted access**.

file_type
(1-byte result)

File type: This byte describes the contents of a file. The currently defined file types are listed below.

File Type	Preferred Use
$00	Typeless file (SOS and ProDOS)
$01	Bad block file
$02 *	Pascal code file
$03 *	Pascal text file
$04	ASCII text file (SOS and ProDOS)
$05 *	Pascal data file
$06	General binary file (SOS and ProDOS)
$07 *	Font file
$08	Graphics screen file
$09 *	Business BASIC program file
$0A *	Business BASIC data file
$0B *	Word Processor file
$0C *	SOS system file
$0D,$0E *	SOS reserved
$0F	Directory file (SOS and ProDOS)
$10 *	RPS data file
$11 *	RPS index file
$12 *	AppleFile discard file
$13 *	AppleFile model file
$14 *	AppleFile report format file
$15 *	Screen library file
$16-$18 *	SOS reserved
$19	AppleWorks Data Base file
$1A	AppleWorks Word Processor file
$1B	AppleWorks Spreadsheet file
$1C-$EE	Reserved
$EF	Pascal area
$F0	ProDOS added command file
$F1-$F8	ProDOS user defined files 1-8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS system file

Note: The file types marked with a * in the above list apply to Apple III SOS only; they are not used by ProDOS. For the file_types used by Apple III SOS only, refer to the *SOS Reference Manual*.

---

aux_type
(2-byte result)

Auxiliary type: This two-byte field is used by the system program. The BASIC system program uses it (low byte first) to store record size or load address, depending on the file_type. If this call is used on a volume directory file, aux_type will contain the total number of blocks on the volume.

storage_type

(1-byte result)

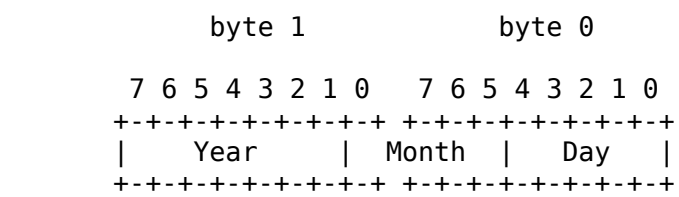
File kind: This byte describes the physical organization of the file. storage_type = $0F is a volume directory file; storage_type = $0D is a directory file; storage_type = $01, $02, and $03 are seedling, sapling, and tree files, respectively (see Appendix B). All other values are reserved for future use.

blocks_used
(2-byte result)

Blocks used by the file: These two bytes contain the total number of blocks used by the file, as stored in the blocks_used parameter of the file's entry. If this call is used on a volume directory file blocks_used contains the total number of blocks used by all the files on the volume.

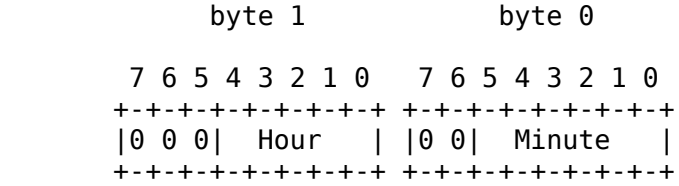
mod_date
(2-byte result)

This 2-byte field returns the date on which the file was last modified. It has this format:



mod_time
(2-byte result)

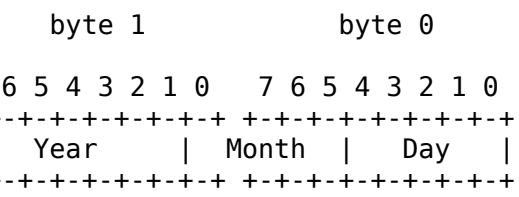
This 2-byte field returns the time at which the file was last modified. It has this format:



---

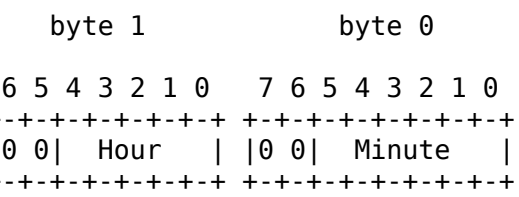
create_date
(2-byte result)

This 2-byte field returns the date on which the file was created. It has this format:



create_time
(2-byte result)

This 2-byte field returns the time at which the file was created. It has this format:

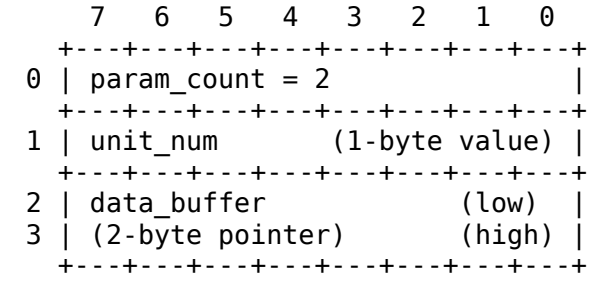


See Chapter 6 for information about the use of ProDOS with a clock/calendar card.

Possible Errors

$27 - I/O error
$40 - Invalid pathname syntax
$44 - Path not found
$45 - Volume directory not found
$46 - File not found
$4A - Incompatible file format
$4B - Unsupported storage_type
$53 - Invalid value in parameter list
$5A - Bit map address is impossible

4.4.6 - ON_LINE ($C5)



This command can be used to determine the names of all ProDOS (or SOS) volumes that are currently mounted (such as disks in disk drives), or it can be used to determine the name of a disk in a specified slot and drive.

When unit_num is 0, this command places a list of the volume names, slot numbers, and drive numbers of all mounted disks into the 256 byte buffer pointed to by data_buffer. When a specific unit_num is requested, only 16 bytes need be set aside for the buffer. The format of the returned information is described below.

The volume names are placed in the list in volume search order, as described in section 3.2.

Parameters

param_count
(1-byte value)

Parameter count: Must be 2 for this call.

unit_num
(1-byte value)

Device slot and drive number: This one-byte value specifies the hardware slot location of a disk device. The format is:

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

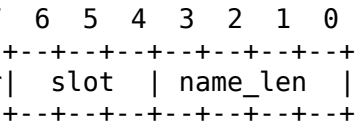
For drive 1, Dr = 0; for drive 2, Dr = 1. Slot specifies the device's slot number (1-7). If unit_num is 0, all mounted disks are scanned. Here are possible values for unit_num:

Slot:	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Drive 1:	70	60	50	40	30	20	10
Drive 2:	F0	E0	D0	C0	B0	A0	90

data_buffer
(2-byte pointer)

Data address pointer: This two-byte address (low byte first) points to a buffer for returned data, which is organized into 16 byte records. If unit_num is 0, the buffer should be 256 bytes long, otherwise 16 bytes is enough.

The first byte of a record identifies the device and the length of its volume name:



Bit 7 specifies drive 1 (Dr = 0) or drive 2 (Dr = 1). Bits 6-4 specify the slot number (1 through 7). Bits 3-0 specify a valid name_length if nonzero. The next 15 bytes of the record are for a volume name.

If name_length = 0, then an error was detected in the specified slot and drive. The error code is present in the second byte of the record. If error $57 (duplicate volume) is encountered, the third byte contains the unit number of the duplicate. When multiple records are returned, the last valid record is followed by one that has unit_num and name_length set to 0.

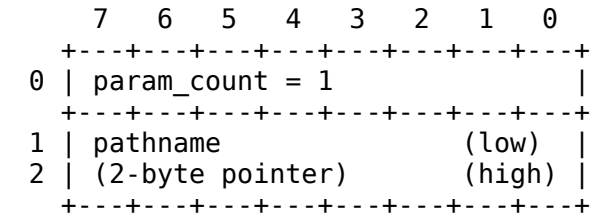
Remember: ON_LINE returns volume names that are not preceded by slashes. Remember to put a slash in front of the name before you use it in a pathname.

Possible Errors

$27 - I/O error
$28 - Device not connected
$2E - Disk switched: File still open on other disk
$45 - Volume directory not found
$52 - Not a ProDOS disk
$55 - Volume Control Block full
$56 - Bad buffer address
$57 - Duplicate volume

When an error pertains to a specific drive, the error code is returned in the second byte of the record corresponding to that drive, as described above. In such cases, the call completes with the accumulator set to 0, and the carry flag clear. Only errors $55 and $56 are not drive specific.

4.4.7 - SET_PREFIX ($C6)



Sets the system prefix to the indicated directory. The pathname may be a full pathname or a partial pathname. The system prefix can be set to null by indicating a pathname with a count of zero. The prefix must be no longer than 64 characters. When ProDOS is started up, the system prefix is set to the name of the volume in the startup drive.

The MLI verifies that the requested prefix directory is on an on-line volume before accepting it.

Parameters

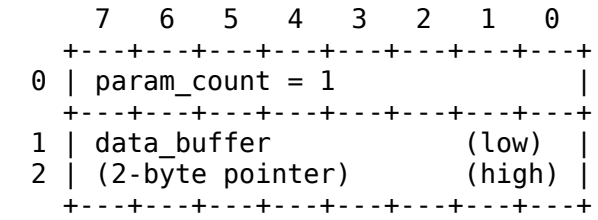
param_count
(1-byte value)
: Parameter count: 1 for this call.

pathname
(2-byte pointer)
: Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the current prefix is attached to the front to make a full pathname. A slash at the end of the pathname is optional.

Possible Errors

$27 - I/O error
$40 - Invalid pathname syntax
$44 - Path not found
$45 - Volume directory not found
$46 - File not found
$4A - Incompatible file format
$4B - Unsupported storage_type
$5A - Bit map disk address is impossible

4.4.8 - GET_PREFIX ($C7)



Returns the current system prefix. If the system prefix is set to null (no prefix), then a count of 0 is returned. Otherwise the returned prefix is preceded by a length byte and bracketed by slashes. Examples are $7/APPLE/ and $D/APPLE/BYTES/. Each character in the prefix is returned with its high bit cleared.

The buffer pointed to by data_buffer is assumed to be 64 bytes long.

Parameters

param_count
(1-byte value)
: Parameter count: Must be 1 for this call.

data_buffer
(2-byte pointer)
: Data address pointer: This two-byte address (low byte first) points to the buffer into which the prefix should be placed. It should be at least
64 bytes long.

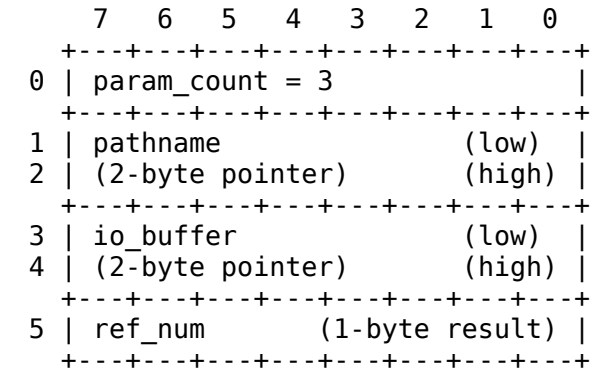
Possible Error

$56 - Bad buffer address

4.5 - Filing Calls

Each of the following sections contains a description of a filing command, including its parameters and the possible errors that may be returned.

4.5.1 - OPEN ($C8)



OPEN prepares a file to be read or written. It creates a file control block that keeps track of the current (open) characteristics of the file specified by pathname, it sets the current position in the file to zero, and it returns a reference number by which the other commands in this section must refer to the file.

The I/O buffer is used by the system for the entire time the file is open. It contains information about the file's structure on the disk, and it contains the current 512-byte block being read or written. It is used until the file is closed, and therefore should not be modified directly by the user. A maximum of eight files can be open at a time.

When a file is opened it is assigned a **level**, from 0 to $F, depending on the current value of the LEVEL location ($BF94) in the system global page. When the CLOSE command is issued with a ref_num of 0, all files at or above the current level are closed. Thus, a CLOSE with a ref_num of 0 and a level of 0 will close all open files.

Refer to Section 2.1.7, "File Levels," for an example of the use of level.

Warning
Once a file has been opened, that file's disk must not be removed from its drive and replaced by another. The system does not check the identity of a volume before writing on it. A system program should check a volume's identity before writing to it.

Parameters

param_count
(1-byte value)

Parameter count: 3 for this call.

pathname
(2-byte pointer)

Pathname pointer: A two-byte address (low byte first) that points to an ASCII string. The string consists of a count byte, followed by the pathname (up to 64 characters). If the pathname begins with a slash (/), it is treated as a full pathname. If not, it is treated as a partial pathname and the prefix is attached to the front to make a full pathname.

io_buffer
(2-byte pointer)

Buffer address pointer: This two byte-address (low byte first) indicates the starting address of a 1024-byte input/output buffer. The buffer must start on a page boundary (a multiple of $100) that is not already used by the system.
If a standard file is being accessed, the first 512 bytes of io_buffer contain the current block of data being read or written; the second 512 bytes contain the current index block, if there is one. If a directory file is being accessed, the first 512 bytes contain the current directory file block; the rest are unused.

ref_num
(2-byte result)

Reference number: When a file is opened, the filing system assigns this one-byte value. All subsequent commands to the open file use this reference number.

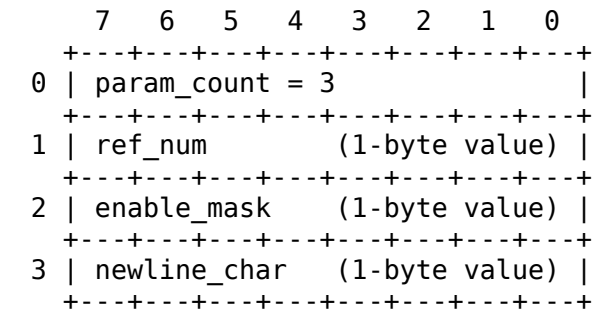
Refer to Appendix B for more information on directory file blocks, index blocks, and data blocks.

Possible Errors

$27 - I/O error
$40 - Invalid pathname syntax
$42 - File Control Block table full
$44 - Path not found

$45 - Volume directory not found
$46 - File not found
$4B - Unsupported storage_type
$50 - File is open
$53 - Invalid value in parameter list
$56 - Bad buffer address
$5A - Bit map disk address is impossible

4.5.2 - NEWLINE ($C9)



This call allows you to enable or disable newline read mode for any open file. When newline is disabled, a read request terminates when the requested number of characters has been read, or when the end of file is encountered. When newline is enabled, a read request will also terminate if the newline character (newline_char) is read.

Each character read is first transferred to the user's data buffer. Next it is ANDed with the enable_mask and compared to the newline_char. If there is a match, the read is terminated. For example, if enable_mask is $7F and newline_char is $0D (ASCII CR), either a $0D or $8D matches and terminates input. This process does not change the character.

Parameters

param_count
(1-byte value)
: Parameter count: 3 for this call.

ref_num
(1-byte value)
: Reference number: This is the filing reference number that was assigned to the file when it was opened.

enable_mask
(1-byte value)
: Newline enable and mask: A value of $00 disables newline mode; a nonzero value enables it. When the mode is enabled, each incoming byte is ANDed with this byte before it is compared to newline_char (below). A match causes the read request to terminate. A value of $FF makes all bits significant, a value of $7F causes only bits 0 through 6 to be tested, etc.

newline_char
(1-byte value)
: Newline character: When newline is enabled, a read request terminates if the input character, having been ANDed with the enable_mask equals this value.

Possible Error

$43 - Invalid reference number

4.5.3 - READ ($CA)

```
     7   6   5   4   3   2   1   0
   +---+---+---+---+---+---+---+---+
 0 | param_count = 4                |
   +---+---+---+---+---+---+---+---+
 1 | ref_num        (1-byte value) |
   +---+---+---+---+---+---+---+---+
 2 | data_buffer            (low)  |
 3 | (2-byte pointer)       (high) |
   +---+---+---+---+---+---+---+---+
 4 | request_count          (low)  |
 5 | (2-byte value)         (high) |
   +---+---+---+---+---+---+---+---+
 6 | trans_count            (low)  |
 7 | (2-byte result)        (high) |
   +---+---+---+---+---+---+---+---+
```

Tries to transfer the requested number of bytes (request_count), starting at the current position (MARK) of the file specified by ref_num to the buffer pointed to by data_buffer. The number of bytes actually transferred is returned in trans_count.

If newline read mode is enabled and a newline character is

encountered before request_count bytes have been read, then the trans_count parameter is set to the number of bytes transferred, including the newline byte.

If the end of file is encountered before request_count bytes have been read, then trans_count is set to the number of bytes transferred. The end of file error ($4C) is returned if and only if zero bytes were transferred (trans_count = 0).

Parameters

param_count
: Parameter count: 4 for this call.

ref_num
(1-byte value)
: Reference number: This is the filing reference number that was assigned to the file when it was opened.

data_buffer
(2-byte pointer)
: Data address pointer: This two-byte address (low byte first) points to the destination for the data to be read from the file.

request_count
(2-byte value)
: Transfer request count: This two-byte value (low byte first) specifies the maximum number of bytes to be transferred to the data buffer pointed to by data_buffer. The maximum value is limited to the number of bytes between the start of data_buffer and the nearest used page of memory.

trans_count
(2-byte result)
: Transferred: This two-byte value (low byte first) indicates the number of bytes actually read. It will be less than request_count only if EOF was encountered, if the newline character was read

while newline mode was enabled, or if some other error occurred during the request.

Possible Errors

$27 - I/O error
$43 - Invalid reference number
$4C - End of file has been encountered
$4E - Access error: file not read enabled
$56 - Bad buffer address
$5A - Bit map address is impossible

4.5.4 - WRITE ($CB)

```
     7   6   5   4   3   2   1   0
   +---+---+---+---+---+---+---+---+
 0 | param_count = 4               |
   +---+---+---+---+---+---+---+---+
 1 | ref_num        (1-byte value) |
   +---+---+---+---+---+---+---+---+
 2 | data_buffer            (low)  |
 3 | (2-byte pointer)       (high) |
   +---+---+---+---+---+---+---+---+
 4 | request_count          (low)  |
 5 | (2-byte value)         (high) |
   +---+---+---+---+---+---+---+---+
 6 | trans_count            (low)  |
 7 | (2-byte result)        (high) |
   +---+---+---+---+---+---+---+---+
```

Tries to transfer a specified number of bytes (request_count) from the buffer pointed to by data_buffer to the file specified by ref_num starting at the current position (MARK) in the file. The actual number of bytes transferred is returned in trans_count.

The file position is updated to position + trans_count. If necessary, additional data and index blocks are allocated to the file, and EOF is extended.

See Appendix B for an explanation of data and index blocks.

Parameters

param_count
(1-byte value)
Parameter count: 4 for this call.

ref_num
(1-byte value)
Reference number: This is the filing reference number that was assigned when the file was opened.

data_buffer
(2-byte pointer)
Data address pointer: This two-byte address (low byte first) points to the beginning of the data to be transferred to the file.

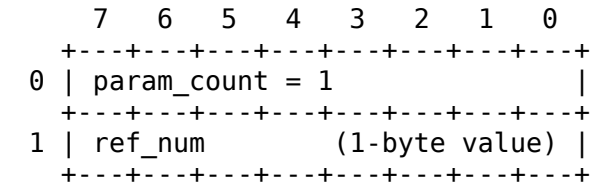
request_count
(2-byte value)
Transfer request count: This two-byte value (low byte first) specifies the maximum number of bytes to be transferred from the buffer pointed to by data_buffer to the file.

trans_count
(2-byte result)
Bytes transferred: This two-byte value (low byte first), indicates the number of bytes actually transferred. If no error occurs, this value should always be equal to request_count.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$43 - Invalid reference number
$48 - Overrun error: not enough disk space
$4E - Access error: file not write enabled
$56 - Bad buffer address
$5A - Bit map disk address is impossible

4.5.5 - CLOSE ($CC)



This call is used to release all resources used by an open file. The file control block is released. If necessary, the file's buffer (io_buffer) is emptied to the file and the directory entry for the file is updated. Until that ref_num is assigned to another open file, subsequent filing calls using that ref_num will fail.

If ref_num equals zero ($0), all open files at or above the current level are closed. For example, if you open files at levels 0, 1, and 2, set the level to 1, and then use CLOSE with ref_num set to 0, the files at level 1 and 2 are closed, but the ones at level 0 are not.

The level is a value from 0 to $F that is stored in the LEVEL location ($BFD8) of the system global page. It is only changed by system programs, and it is used by OPEN and CLOSE.

This call causes the backup bit to be set.

Parameters

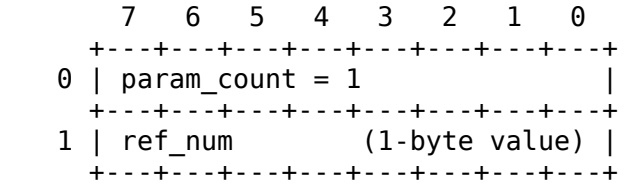
param_count
(1-byte value)
: Parameter count: 1 for this call.

ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened. CLOSE releases this reference number. If ref_num = 0, all open files at or above the current level are closed.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$43 - Invalid reference number
$5A - Bit map disk address is impossible

4.5.6 - FLUSH ($CD)



The file's write buffer (io_buffer) is emptied to the file, and the file's directory is updated. If ref_num equals zero ($0), then all open files at or above the current level are flushed.

The backup bit is set by this call.

FLUSH is further explained in Chapter 2, section "Closing and Flushing Files."

Parameters

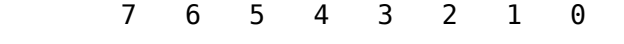
param_count
(1-byte value)
: Parameter count: 1 for this call.

ref_num
(1-byte value)
: Reference number: This is the filing reference number that was assigned to the file when it was opened. If ref_num = 0 all open files at or above the current level are flushed.

Possible Errors

$27 - I/O error
$2B - Disk write protected
$43 - Invalid reference number
$5A - Bit map disk address is impossible

4.5.7 - SET_MARK ($CE)



```
    +---+---+---+---+---+---+---+---+
  0 | param_count = 2               |
    +---+---+---+---+---+---+---+---+
  1 | ref_num        (1-byte value) |
    +---+---+---+---+---+---+---+---+
  2 |                         (low)  |
  3 | position       (3-byte value) |
  4 |                         (high) |
    +---+---+---+---+---+---+---+---+
```

Changes the current position (MARK) in the file to that specified by the position parameter. Position may not exceed the end of file (EOF) value.

See the example of SET_MARK in
Chapter 2, section "The EOF and MARK."

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

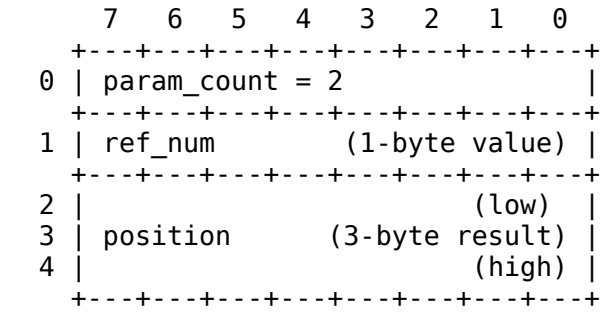
ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened.

position
(3-byte value)
: File position: This three-byte value (low bytes first) specifies to the File Manager the absolute position in the file at which the next read or write should begin (the MARK). The file position cannot exceed the file's EOF.

Possible Errors

$27 - I/O error
$43 - Invalid reference number
$4D - Position out of range
$5A - Bit map disk address is impossible

4.5.8 - GET_MARK ($CF)



Returns the current position (MARK) in an open file.

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

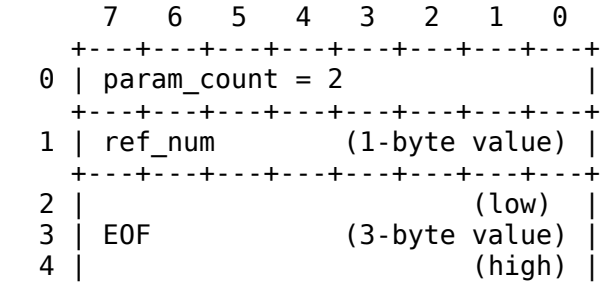
ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened.

position
(3-byte result)
: File position: This three-byte value (low bytes first) is the absolute position in the file at which the next read or write will begin, unless it is changed by a subsequent SET_MARK call.

Possible Error

$43 - Invalid reference number

4.5.9 - SET_EOF ($D0)



```
+---+---+---+---+---+---+---+---+
```

Sets the logical size of the file specified by ref_num to EOF. If the new EOF is less than the current EOF, then blocks past the new EOF are released to the system. If the new EOF is greater than or equal to the current EOF, no blocks are allocated. If the new EOF is less than the current position, the value of the position is set to the EOF. The EOF cannot be changed unless the file is write enabled.

The **logical size** of a file is the number of bytes that can be read from it.

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened.

EOF
(3-byte value)
: End Of File: This three-byte value (low bytes first) represents the logical end of a file. It can be greater or less than the current value of EOF. If it is less, blocks past the new EOF are released to the system.

Possible Errors

$27 - I/O error
$43 - Invalid reference number
$4D - Position out of range
$4E - Access error: File not write enabled
$5A - Bit map disk address is impossible

4.5.10 - GET_EOF ($D1)

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

```
  0 | param_count = 2               |
    +---+---+---+---+---+---+---+---+
  1 | ref_num        (1-byte value) |
    +---+---+---+---+---+---+---+---+
  2 |                         (low)  |
  3 | EOF           (3-byte result) |
  4 |                         (high) |
    +---+---+---+---+---+---+---+---+
```

Returns the number of bytes that can be read from the open file.

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened.

EOF
(3-byte result)
: End Of File: This three-byte result (low bytes first) contains the value of the logical end of file. This value is the maximum number of bytes that can be read from the file.

Possible Error

$43 - Invalid reference number

4.5.11 - SET_BUF ($D2)

```
     7   6   5   4   3   2   1   0
    +---+---+---+---+---+---+---+---+
  0 | param_count = 2               |
    +---+---+---+---+---+---+---+---+
  1 | ref_num        (1-byte value) |
    +---+---+---+---+---+---+---+---+
  2 | io_buffer               (low)  |
  3 | (2-byte pointer)        (high) |
    +---+---+---+---+---+---+---+---+
```

This call allows you to reassign the address of the input/output buffer that is used by the file specified by ref_num (assigned when the file

was opened). The MLI checks to see that the specified buffer is not already used by the system, then it moves the contents of the old buffer into the new buffer.

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

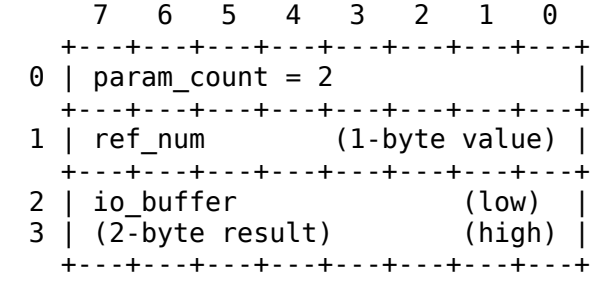
ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened.

io_buffer
(2-byte pointer)
: Buffer address pointer: This two-byte address (low byte first) indicates the starting address of a 1024 byte I/O buffer. The buffer must start on a page boundary (multiple of $100) and it must not already be used by the system.

Possible Errors

$43 - Invalid reference number
$56 - Bad buffer address

4.5.12 - GET_BUF ($D3)



Returns the address of the input/output buffer currently being used by the file specified by ref_num.

Parameters

param_count
: Parameter count: 2 for this call.

ref_num
(1-byte value)
: Reference number: The filing reference number that was assigned to the file when it was opened.

io_buffer
(2-byte result)
: Buffer address pointer: This two-byte address (low byte first) indicates the starting address of a 1024 byte I/O buffer. The buffer starts on a page boundary (multiple of $100).

Possible Error

$43 - Invalid reference number

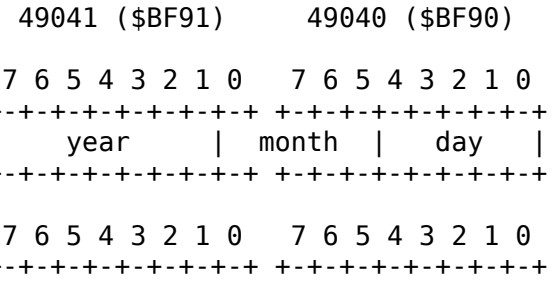
4.6 - System Calls

Each of the following sections describes a system command, including any parameters and possible errors.

4.6.1 - GET_TIME ($82)

This call has no parameter list, and it cannot generate an error. It calls a clock/calendar routine, if there is one, which returns the current date and time to the system date and time locations ($BF90-BF93). If there is no clock/calendar routine, the system date and time locations are left unchanged.

Here is the layout of the four bytes that make up the system date and time.



```
TIME:  |    hour       | |    minute     |
       +-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+

         49043 ($BF93)     49042 ($BF92)
```

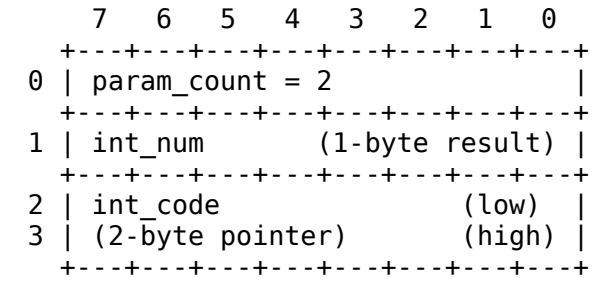
When ProDOS starts up, it looks for a clock/calendar card in one of the Apple II's slots. If it recognizes one, ProDOS installs a routine that can read the date and time from the card and place them in the system date and time locations. Otherwise, no routine is installed.

Note that the GET_TIME call number for ProDOS is different from the GET_TIME call number for SOS.

Chapter 5 explains the use of the date and time locations by the system.

Chapter 6 explains the installation of clock/calendar routines.

4.6.2 - ALLOC_INTERRUPT ($40)



This call places the address of an interrupt receiving routine int_code into the interrupt vector table. It should be made before you enable the hardware that could cause this interrupt. It is your responsibility to make sure that the routine is installed at the proper location and that it follows interrupt conventions.

The int_num that is returned gives an indication of what priority the interrupt is given (1, 2, 3, or 4). Routines that are installed first are given the highest priority. You must use this number when you remove the routine from the system.

Interrupt receiving routines are described in Chapter 6.

Parameters

param_count
(1-byte value)
: Parameter count: 2 for this call.

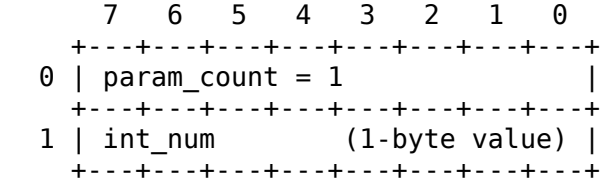
int_num
(1-byte result)
: Interrupt vector number: This value, from 1 to 4 is assigned by the MLI to int_num when this call is made. This number must be retained by the calling routine and used when removing an interrupt routine.

int_code
(2-byte pointer)
: Interrupt handler code entry address: This is a pointer (low byte first) to the first byte of a routine that is to be called when the system is polling in response to an interrupt.

Possible Errors

$25 - Interrupt vector table full
$53 - Invalid parameter

4.6.3 - DEALLOC_INTERRUPT ($41)



This call clears the entry for int_num from the interrupt vector table. You must disable interrupt hardware before you make this call. If you don't, and the hardware interrupts after the vector table has been updated, a **SYSTEM FAILURE** will occur (see Section 4.8.1).

Interrupt receiving routines are described in Chapter 6.

Parameters

param_count

(1-byte value)
: Parameter count: 1 for this call.

int_num
(1-byte value)
: Interrupt vector number: A value from 1 to 4 that was assigned by the MLI to int_num when ALLOC_INTERRUPT was called.

Possible Error

$53 - Invalid parameter

4.7 - Direct Disk Access Calls

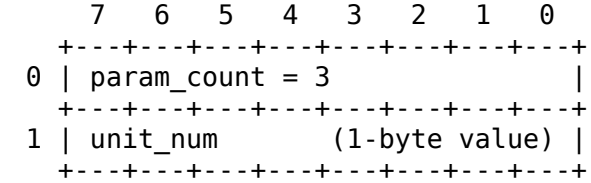
The direct disk access commands READ_BLOCK and WRITE_BLOCK, allow you to read from or write to any logical block on a disk. They are intended to be used by utility (such as copying) and diagnostic programs.

Warning
Application programs should *not* use these commands: they can very easily damage the data integrity of the ProDOS file structure. All necessary functions can be performed without these calls.

These calls will also read and write blocks (not tracks and sectors) from DOS 3.3 disks. A mapping of tracks and sectors on a DOS 3.3 disk to blocks read or written by ProDOS is given in Section B.5.

ProDOS BLOCK_READ and
BL0CK_WRITE calls can access DOS 3.3
disks: see Appendix B, Section "DOS 3.3
Disk Organization."

4.7.1 - READ_BLOCK ($80)



```
  2 | data_buffer            (low)  |
  3 | (2-byte pointer)       (high) |
    +---+---+---+---+---+---+---+---+
  4 | block_num              (low)  |
  5 | (2-byte value)         (high) |
    +---+---+---+---+---+---+---+---+
```

This call reads one block from the disk device specified by unit_num into memory starting at the address indicated by data_buffer. The buffer must be 512 or more bytes in length.

Parameters

param_count
(1-byte value)
: Parameter count: 3 for this call.

unit_num
(1-byte value)
: Device slot and drive number: This one-byte value specifies the hardware slot location of a disk device. The format is:

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

: The Dr bit specifies either drive 1 (Dr = 0) or drive 2 (Dr = 1). Slot must contain a slot number between 1 and 7, inclusive.

data_buffer
(2-byte pointer)
: Data address pointer: This two-byte address (low byte first) points to the destination for data. The buffer must be at least 512 bytes long.

---

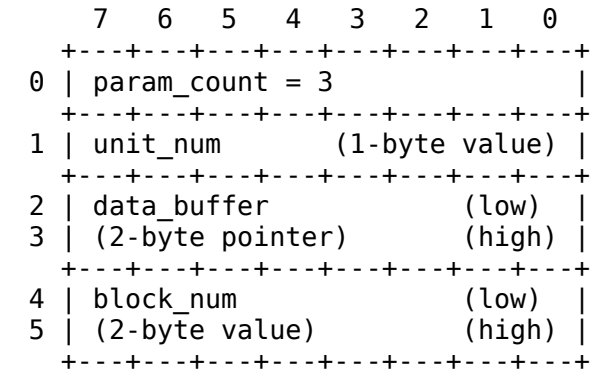
block_num
(2-byte value)
: Logical block number: This two-byte value (low byte first) specifies the logical address of a block of data to be read. Disk II's, for example, have block addresses ranging from $0 to $117. There is no general connection between block numbers and the layout of tracks and sectors on the disk.

The translation from logical to physical block is done by the device driver.

Possible Errors

$27 - I/O error
$28 - No device connected

4.7.2 - WRITE_BLOCK ($81)

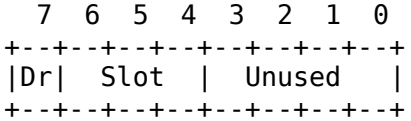


This call transfers one block of data from the memory buffer indicated by data_buffer to the disk device specified by unit_num. The block of data is placed in the logical block specified by block_num. It is assumed that the data buffer is at least 512 bytes long.

Parameters

param_count
(1-byte value)
Parameter count: 3 for this call.

unit_num
(1-byte value)
Device slot and drive number: This one-byte value specifies the hardware slot location of a disk device. The format is:



The Dr bit specifies either drive 1 (Dr = 0) or drive 2 (Dr = 1). Slot must contain a slot number

between 1 and 7, inclusive.

data_buffer
(2-byte pointer)
Data address pointer: This two-byte address (low byte first) points to the source of data to be transferred. It is assumed that the data buffer is at least 512 bytes in length.

block_num
(2-byte value)
Logical block number: This two-byte value (low byte first) specifies the logical address on a disk of the block to be written. Disk II's, for example, have block addresses ranging from $0 to $117. There is no general connection between block numbers and the layout of tracks and sectors on the disk. The translation from logical to physical block is done by the device driver.
An out-of-range block_num returns an I/O error. The number of blocks on a volume is returned in the aux_type field of the GET_FILE_INFO call of a volume directory file.

Possible Errors

$27 - I/O error
$28 - No device connected
$2B - Disk write protected

4.8 - MLI Error Codes

This is a summary of the ProDOS error codes. If there is no error, the C-flag is clear, and the Accumulator contains $00. If there is an error, the C-flag is set, and the Accumulator contains the error code.

$00 - No error.

$01 - Bad system call number. A non-existent command was issued.

$04 - Bad system call parameter count. This error will occur only if the call parameter list is not properly constructed.

$25 - Interrupt vector table full. Only four routines can be activated for interrupt processing at a time. One must be deactivated before another one may be enabled.

$27 - I/O error. This catch-all error is reported when some hardware failure prevents proper transfer of data to/from the disk device.

$28 - No device detected/connected. Will occur if, for example, drive 2 is specified for Disk II when only one drive is connected.

$2B - Disk write protected. Hardware write-inhibit is enabled, write request cannot be processed.

$2E - Disk switched: A WRITE, FLUSH, or CLOSE operation cannot be accomplished because a disk containing an open file has been removed from its drive.

$40 - Invalid pathname syntax. The pathname contains illegal characters.

$42 - File Control Block table full. The FCB can contain a maximum of eight entries. Thus, a maximum of eight files can be open concurrently.

$43 - Invalid reference number. The value parameter given as a reference number does not match the reference number of any currently open file.

$44 - Path not found. A filename in the specified pathname (which refers to a subdirectory) does not exist. The pathname's syntax is legal.

$45 - Volume directory not found. The volume name in the specified pathname does not exist. The pathname's syntax is otherwise legal.

$46 - File not found. The last filename of the pathname does not exist. The syntax of the pathname is legal.

$47 - Duplicate filename. An attempt was made to create a file that already exists or to rename a file with an already used name.

$48 - Overrun error. An attempt to allocate blocks on a block device during a CREATE or WRITE operation failed due to lack of space on the device. This error also is returned on an invalid EOF parameter. Data is written until the disk is full, but you will always be able to close the file.

$49 - Volume directory full. No more entries are left in the volume directory. In ProDOS 1.0, a volume directory can hold no more than 51 entries. No more files can be added (using CREATE) in this directory until others are destroyed.

$4A - Incompatible file format. The file is not backward compatible with this version of ProDOS. Storage_type is recognized, but the File Manager may not support that storage_type in a fully compatible fashion. This error is likely to occur when data written by a future version of the BFM is read back using an earlier version of the BFM.

$4B - Unsupported storage_type. File is of an organization unknown to the executing File Manager. This error may be reported if the directory is tampered with by the user. This error is also returned if you attempt to set the prefix to a nondirectory file.

$4C - End of file has been encountered. This error is returned after a READ call when the file position is equal to EOF and no data can be read.

$4D - Position out of range. Returned when the position parameter is greater than current EOF.

$4E - Access error. The file's access attribute forbids the RENAME, DESTROY, READ or WRITE operation that was attempted.

$50 - File is open. An attempt was made to OPEN, RENAME or DESTROY an open file.

$51 - Directory count error: The number of entries indicated in the directory header does not match the number of entries actually found in the file.

$52 - Not a ProDOS disk. The specified disk does not contain a ProDOS (or SOS) directory format.

$53 - Invalid parameter. The value of one or more parameters in the parameter list is out of range.

$55 - Volume Control Block table full. More than eight volumes on line. The VCB table can contain a maximum of eight entries. This error occurs only if eight files, on eight volumes, are open and the ON_LINE command is requested for a device having no open files.

$56 - Bad buffer address. The data_buffer or io_buffer specified conflicts with memory currently in use by the MLI.

$57 - Duplicate volume. This is a warning that two or more volume directory names are the same.

$5A - Bit map disk address is impossible. The volume bit map indicates that the volume contains blocks beyond the block count for that volume.

> **Note:** *System failure* errors should never occur. They indicate that the system has encountered a situation that should not have happened, and it has no available means of recovery.
>
> Possible causes include
>
> - bad RAM

- disk failure
- operating system bug
- unclaimed interrupt.

Chapter 5
Writing a ProDOS System Program

This chapter is about writing system programs that use the ProDOS MLI. It first explains the things that a program must do to qualify as a system program. Next it discusses some of the things that a system program must be aware of, particularly how it should use memory. The end of the chapter contains several programming hints.

5.1 - System Program Requirements

A ProDOS system program is any program that makes calls to the ProDOS MLI and that adheres to a set of standard system program rules. Each system program must have

- code to move the program from its load position to its final execution location, if necessary
- a version number in the system global page
- the ability to switch to another system program.

All other aspects of the system program are up to you.

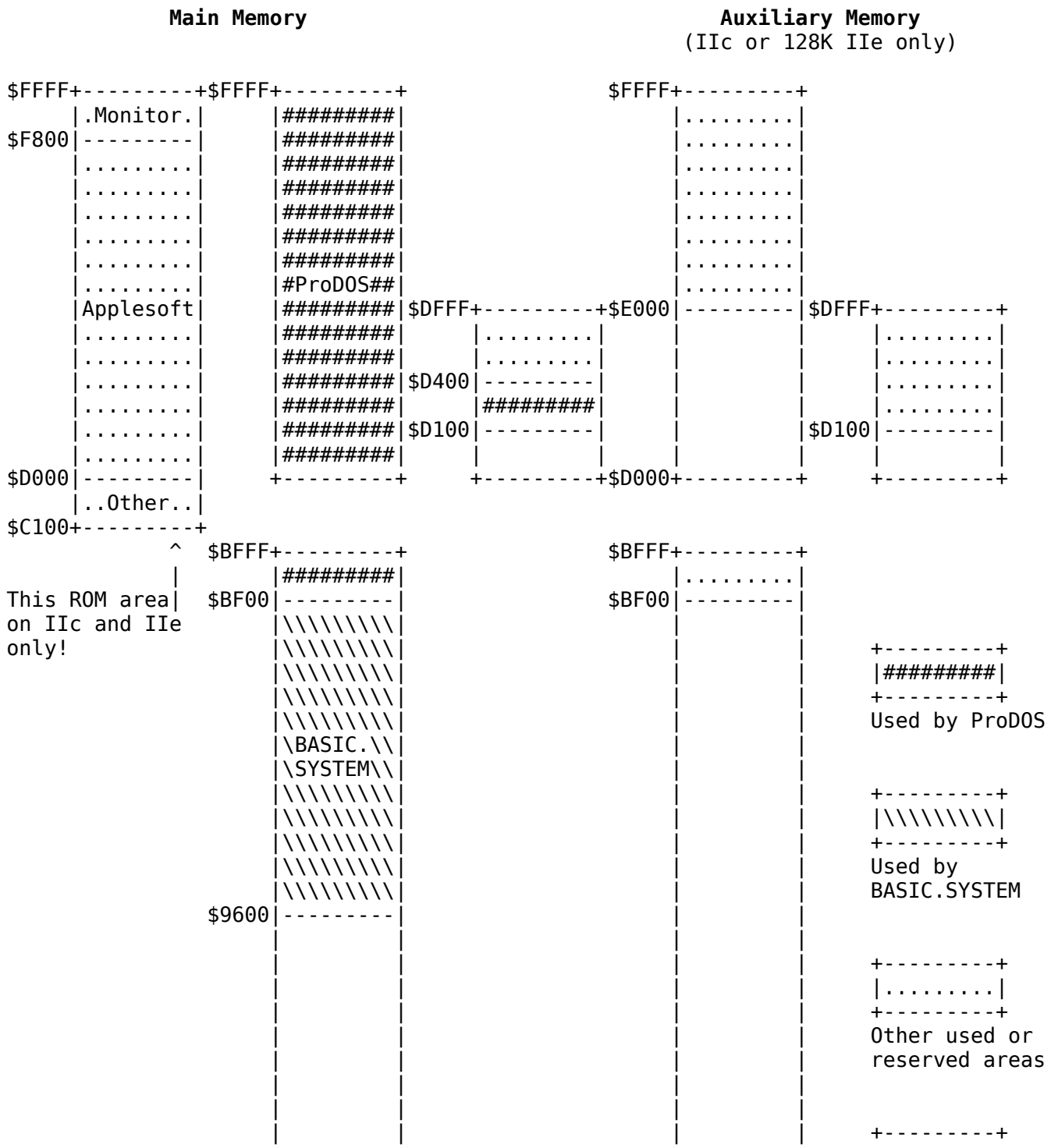
5.1.1 - Placement in Memory

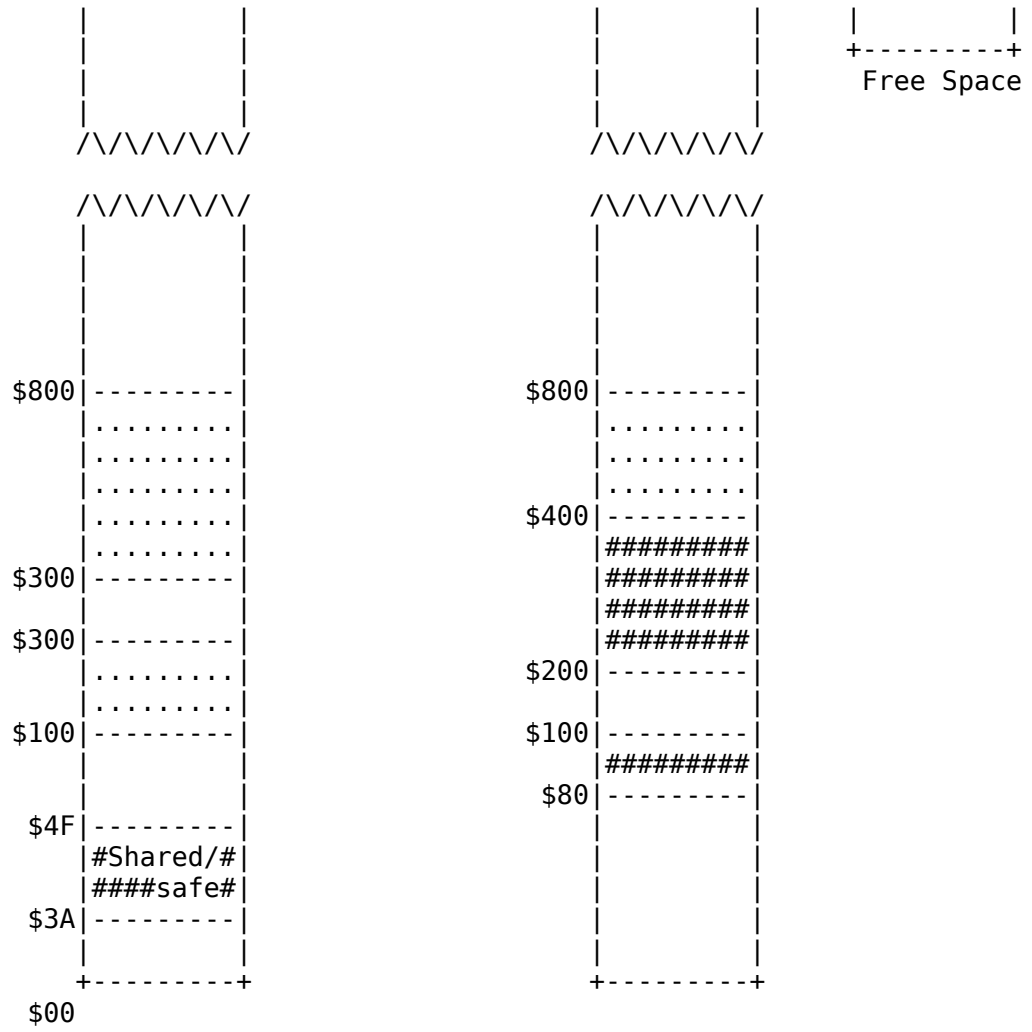
System programs are always loaded into memory starting at location $2000. When the system is first started up, the system program used is the first file on the startup disk with the name XXX.SYSTEM, and the $FF filetype. When one system program switches to another, it can load any file of type $FF.

Figure 5-1 shows the portions of memory that are available to system programs. If BASIC is not being used, the area assigned to BASIC.SYSTEM (the BASIC command interpreter) is also available.

A system program as large as $8F00 (36608) bytes can be loaded. The total space available to a system program is $B700 (46848) bytes.

Figure 5-1. Memory Map





5.1.2 - Relocating the Code

The final execution location(s) to which you can relocate your code depends on your system configuration. The memory locations $0800 through $BEFF are available to system programs.

5.1.3 - Updating the System Global Page

The MLI global page resides in locations $BF00 through $BFFF. These are the locations whose values you must set:

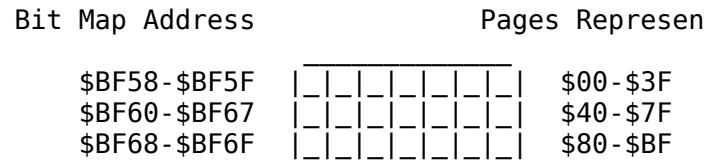
$BF58-$BF6F - The system bit map.
$BFFD - The version number of your system program.

In addition, there is other information in the global page that your program might find useful. These values are documented in the section "The System Global Page."

5.1.4 - The System Bit Map

The system bit map occupies bytes $BF58 through $BF6F in the system global page and it represents the status of each 256-byte page of memory from $0000 through $BFFF, as shown in Figure 5-2.

Figure 5-2. Memory Representation in the System Bit Map



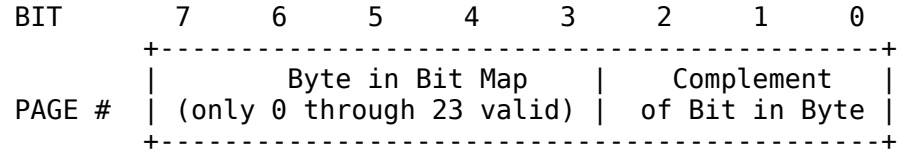
Within each byte, the bits are used in reverse order. Thus, bit 7 of byte $BF58 represents the first 256 bytes of memory, and bit 0 of byte $BF6F represents the last page before $C000.

You may have noticed that neither the Language Card area of memory nor the extended memory of an Apple IIe or Apple IIc is included in this map. This is because these regions of memory cannot be directly accessed by the MLI. You cannot read data into or out of these areas, and you cannot execute MLI calls from them. More information is given in this chapter in the sections "Using the Language Card" and "Using the Alternate 64K RAM Bank."

5.1.4.1 - Using the Bit Map

There are twenty-four bytes in the bit map: the high five bits of an address select which of these bytes contains a given page. Each byte represents eight 256-byte pages; the next three bits of an address form the complement of the bit number within that byte. Thus for page $00 in memory, the high five bits are zero: byte 0 of the bit map contains that page. The next three bits are zero, the complement of 000 (binary) is 111 (binary): bit 7 within byte zero contains that page. Figure 5-3 shows this relationship.

Figure 5-3. Page Number to Bit-Map Bit Conversion



Here is a short routine that accepts the high byte of an address in the Accumulator. It returns with the carry clear if the memory page is free; the carry is set if the page is already used (or if the page is in the Language Card). It destroys the values in the A, X, and Y registers.

```
-------------------------------------------------------------------------

SOURCE   FILE #01 =>PFREE
0000:        BF58    1 BITMAP  EQU  $BF58     ;the system bit map
0000:                2 *
0000:        0000    3 PFREE   EQU  *
0000:C9 C0           4         CMP  #$C0      ;in language card?
0002:B0 17   001B    5         BCS  NOTFREE   ;yes, it's protected
0004:AA              6         TAX            ;save page for bit in page
0005:4A              7         LSR  A         ;move byte number to right
0006:4A              8         LSR  A
0007:4A              9         LSR  A
0008:A8             10         TAY            ;save byte number
0009:8A             11         TXA            ;get bit in byte
000A:29 07          12         AND  #$7       ;mask off byte number
000C:AA             13         TAX            ;and save bit in byte
000D:A9 80          14         LDA  #$80      ;bit 7 set for bit 0 in byte
000F:CA             15 LOOP    DEX            ;done shifting?
0010:30 04   0016   16         BMI  CHKBIT    ;yes, check bit value
0012:4A             17         LSR  A         ;else shift again
0013:4C 0F 00       18         JMP  LOOP      ;and continue
0016:39 58 BF       19 CHKBIT  AND  BITMAP,Y  ;is selected bit set?
0019:F0 02   001D   20         BEQ  ISFREE    ;nope, page is free
001B:38             21 NOTFREE SEC            ;flag page not free
001C:60             22         RTS
001D:18             23 ISFREE  CLC            ;page is free
001E:60             24         RTS

-------------------------------------------------------------------------
```

5.1.5 - Switching System Programs

All system programs must use a standard way of starting and quitting.

5.1.5.1 - Starting System Programs

System programs are started in one of two ways:

- The disk containing ProDOS and the system program is started up; ProDOS loads and runs the first XXX.SYSTEM file of type SYS($FF). The order of search is determined by the file entries in the startup volume directory.
- The program is loaded by another program (such as the ProDOS FILER or the BASIC.SYSTEM) or by a program dispatcher or selector.

The system program is loaded and jumped to at $2000. The complete or partial pathname of the system program is stored at $280, starting with a length byte. The string is a full pathname if it starts with a slash. It is a partial pathname if it starts with a letter.

This pathname allows a system program to determine the directory where other needed files may reside. The program should never assume that the files are in a specific directory or subdirectory.

There is a way to pass a second pathname to interpreters -- for example, to language interpreters -- that like to run startup programs. The ProDOS dispatcher does not support this mechanism but other more sophisticated program selectors may. It requires that the interpreter start a certain way:

$2000 is a jump instruction. **$2003** and **$2004** are **$EE**.

If the interpreter starts this way, byte $2005 is assumed to indicate the length of a buffer that starts at $2006 and holds the pathname (starting with a length byte) of the startup file.

Interpreters that support this mechanism should supply their own default string, which should be a standard choice for a startup program or a flag not to run a startup program.

Once gaining control, the system program sets the reset vector and fixes the power-up byte. Never assume the state of the machine to be anything that is not clearly documented.

Important!
If your interpreter uses any location in the range $D100-$DFFF (the dispatcher/selector area) in the second 4K bank of RAM, be sure that the area is initially saved and then restored on exit.

5.1.5.2 - Quitting System Programs

Here is how to quit system programs:

1. Do normal housekeeping. Close files, reinstall /RAM if you have disconnected it, and so on.
2. Invalidate the power-up byte at $3F4. The simplest way is either to increment or to decrement it, which will always make it an invalid

check of the $3F2 vector.

3. Execute a ProDOS system call number $65 as follows:

```
EXIT       JSR  PRODOS        ;Call the MLI ($BF00)
           DFB  $65           ;CALL TYPE = QUIT
           DW   PARMTABLE     ;Pointer to parameter table
PARMTABLE  DFB  4             ;Number of parameters is 4
           DFB  0             ;0 is the only quit type
           DW   0000          ;Pointer reserved for future use
           DFB  0             ;Byte reserved for future use
           DW   0000          ;Pointer reserved for future use
```

Even though most of the parameter table is reserved for future use it must all be present. It must consist of seven bytes: $04 followed by six nulls ($00).

ProDOS MLI call $65, the QUIT call, moves addresses $D100 through $D3FF from the second 4K bank of RAM of the language card to $1000, and executes a JMP to $1000. What initially resides in that area is Apple's dispatcher code.

The dispatcher, once executed, does the following:

1. Allows the user to enter the prefix and filename of the system program (interpreter) to be executed.
2. Stores the system program name at $280, starting with a length byte. Once the system program executes, it can find from where it was starred, and locate any files it needs for processing.
3. Closes any open files.
4. Clears the bit map, and protects the zero, stack, text, and ProDOS global pages.
5. Reads in the system file at $2000, and executes a JMP to $2000.

To install your own QUIT code that loads your own selector program, you must, at some point, store the system program name at $280, close open files, clear the bit map, and protect the zero, stack, text, and ProDOS global pages, as described above. In addition, the $D100 byte must be a CLD ($D8) instruction, so that programs can tell whether selector code or the ProDOS dispatcher code is resident.

In addition to just leaving the pathname at $280 for the interpreter's use, a method to enable a selector program to specify an accompanying startup program has been defined. Once active, an interpreter can immediately run that program. This involves reserving an area in the system file, which a selector program overwrites with the startup

program's name. The interpreter then loads and executes that specified program.

Here is how the procedure works: the selector program looks at the first byte of the interpreter at $2000. If it is a JMP ($4C) instruction, and bytes $2003 and $2004 are both $EE, then byte $2005 is interpreted as a buffer size indicator with the buffer starting at $2006. The string at $2006 would be the normal ProDOS pathname or partial pathname, starting with a length byte.

```
Byte            Content

$2000-$2002     JMP CONT
$2003           $EE
$2004           $EE
$2005           $41
$2006           $07
$2007-$200D     Startup Code
.
.
.
$2047           CONT
.
.
.
```

The two $EEs let the selector program know that this particular interpreter can run a startup program. The interpreters that support this feature will supply their own default string, which may be a startup program or a flag of your choice.

5.2 - Managing System Resources

This section describes the interaction between ProDOS and the various parts of memory.

5.2.1 - Using the Stack

In the Apple II, the stack is stored in page $01 of memory, from the high byte of the page going down. When an interrupt occurs, the interrupt handler saves the low 16 bytes of the stack, but only if the stack is more than 3/4 full. For maximum interrupt efficiency, a system program should not use more than the upper 3/4 of the stack.

System programs should set the stack pointer to $FF at the warm-start

entry point.

5.2.2 - Using the Alternate 64K RAM Bank

When ProDOS is started up, it checks its environment. If it finds 128K of memory (Apple IIe with Extended 80-column Text card, or Apple IIc), the auxiliary 64K bank of memory is configured as a RAM disk named /RAM. Because the memory on the 80-column card is in slot 3, /RAM appears as slot 3 drive 2. Its unit number, as entered in the ProDOS global page's device list, is $BF.

Before using the auxiliary memory for any other purpose, you must protect your code from /RAM. The routines described here are examples only.

Note: These routines are examples; they are not being specified as suitable for any particular purpose.

5.2.2.1 - Protecting Auxiliary Bank Hi-Res Graphics Pages

If your use involves hi-res graphics, you may protect those areas of auxiliary memory. If you save a dummy 8K file as the first entry in /RAM, it will always be saved at $2000 to $3FFF. If you then immediately save a second dummy 8K file to /RAM, it will be saved at $4000 to $5FFF. This protects the hi-res pages in auxiliary memory while maintaining /RAM as an online storage device.

There is no formula for determining where the blocks of /RAM physically reside in memory. Further, the logical blocks are not physically contiguous. There is no guaranteed way to protect any other fixed portions of auxiliary memory by the dummy file method.

5.2.2.2 - Disconnecting /RAM

To protect all of the auxiliary memory that has not been reserved for use by Apple, you must disconnect /RAM. Note these three areas of the system global page:

- $BF10-$BF2F contains the disk device driver addresses.
- $BF31 contains the number of devices minus one.
- $BF32-$BF3F contains the list of disk device numbers.

Here is how to disconnect /RAM. It is suggested that you read block two on /RAM and check the FILE_COUNT field in the directory. If there are any files on /RAM, prompt the user either to continue with the disconnect or to cancel the process.

Check the MACHID byte at $BF96 to see if you have 128K. If not, there will be no /RAM to disconnect.

The slot 0 drive 1 disk-driver vector ($BF10) will point to the "No Device Connected" routine. The slot 0 vectors $BF10 and $BF20 are reserved for Apple's use: you cannot use these vectors if this convention is to work. If the slot 3 drive 2 vector also points to the same address, then /RAM is already disconnected.

If /RAM is on line, you are ready to remove it. (Note that the following steps can be adapted to disconnecting any device.)

1. Retrieve the slot 3 drive 2 device number you find in DEVLST, and save it.
2. Move any remaining device numbers forward in the DEVLST.
3. Retrieve the slot 3 drive 2 driver vector, and save it for later reinstallation.
4. Replicate the "No Device Connected" vector in slot 0 drive 1 into slot 3 drive 2.
5. Decrement the device count (DEVCNT).

/RAM is now disconnected. You are free to use the unreserved areas of auxiliary memory.

Note: If ProDOS has just been started up, /RAM is the last *disk device* installed. However, if the user has manually installed another device(s), the device number for /RAM will not be the last entry in the device list (DEVLST).

5.2.2.3 - How to Treat RAM Disks With More Than 64K

If there is a device in slot 3 drive 2 that is not /RAM, or is a RAM disk with a capacity of more than 64K, the following routine prevents it from being disconnected.

```
 ORG $1000
 DEVCNT EQU $BF31       ; GLOBAL PAGE DEVICE COUNT
 DEVLST EQU $BF32       ; GLOBAL PAGE DEVICE LIST
```

```
MACHID EQU $BF98       ; GLOBAL PAGE MACHINE ID BYTE
RAMSLOT EQU $BF26      ; SLOT 3, DRIVE 2 IS /RAM'S DRIVER VECTOR
*
* NODEV IS THE GLOBAL PAGE SLOT ZERO, DRIVE 1 DISK DRIVE VECTOR.
* IT IS RESERVED FOR USE AS THE "NO DEVICE CONNECTED" VECTOR.
*
NODEV EQU $BF10
*
*
RAMOUT PHP             ; SAVE STATUS AND
 SEI                   ; MAKE SURE INTERRUPTS ARE OFF!
*
* FIRST THING TO DO IS TO SEE IF THERE IS A /RAM TO DISCONNECT!
*
 LDA MACHID            ; LOAD THE MACHINE ID BYTE
 AND #$30              ; TO CHECK FOR A 128k SYSTEM
 CMP #$30              ; IS IT 128k?
 BNE DONE              ; IF NOT THEN BRANCH SINCE NO /RAM!
*
 LDA RAMSLOT           ; IT IS 128K; IS A DEVICE THERE?
 CMP NODEV             ; COMPARE WITH LOW BYTE OF NODEV
 BNE CONT              ; BRANCH IF NOT EQUAL, DEVICE IS CONNECTED
 LDA RAMSLOT+1         ; CHECK HI BYTE FOR MATCH
 CMP NODEV+1           ; ARE WE CONNECTED?
 BEQ DONE              ; BRANCH, NO WORK TO DO; DEVICE NOT THERE
*
* AT THIS POINT /RAM (OR SOME OTHER DEVICE) IS CONNECTED IN
* THE SLOT 3, DRIVE 2 VECTOR.  NOW WE MUST GO THRU THE DEVICE
* LIST AND FIND THE SLOT 3, DRIVE 2 UNIT NUMBER OF /RAM ($BF).
* THE ACTUAL UNIT NUMBERS, (THAT IS TO SAY 'DEVICES') THAT WILL
* BE REMOVED WILL BE $BF, $BB, $B7, $B3.  /RAM'S DEVICE NUMBER
* IS $BF.  THUS THIS CONVENTION WILL ALLOW OTHER DEVICES THAT
* DO NOT NECESSARILY RESEMBLE (OR IN FACT, ARE COMPLETELY DIFFERENT
* FROM) /RAM TO REMAIN INTACT IN THE SYSTEM.
*
*
CONT LDY DEVCNT        ; GET THE NUMBER OF DEVICES ONLINE
LOOP LDA DEVLST,Y      ; START LOOKING FOR /RAM OR FACSIMILE
 AND #$F3              ; LOOKING FOR $BF, $BB, $B7, $B3
 CMP #$B3              ; IS DEVICE NUMBER IN {$BF,$BB,$B7,$B3}?
 BEQ FOUND             ; BRANCH IF FOUND..
 DEY                   ; OTHERWISE CHECK OUT THE NEXT UNIT #.
 BPL LOOP              ; BRANCH UNLESS YOU'VE RUN OUT OF UNITS.
 BMI DONE              ; SINCE YOU HAVE RUN OUT OF UNITS TO
FOUND LDA DEVLST,Y     ; GET THE ORIGINAL UNIT NUMBER BACK
 STA RAMUNITID         ; AND SAVE IT OFF FOR LATER RESTORATION.
*
* NOW WE MUST REMOVE THE UNIT FROM THE DEVICE LIST BY BUBBLING
* UP THE TRAILING UNITS.
*
GETLOOP LDA DEVLST+1,Y ; GET THE NEXT UNIT NUMBER
 STA DEVLST,Y          ; AND MOVE IT UP.
 BEQ EXIT              ; BRANCH WHEN DONE(ZEROS TRAIL THE DEVLST)
 INY                   ; CONTINUE TO THE NEXT UNIT NUMBER...
 BNE GETLOOP           ; BRANCH ALWAYS.
*
EXIT LDA RAMSLOT       ; SAVE SLOT 3, DRIVE 2 DEVICE ADDRESS.
 STA ADDRESS           ; SAVE OFF LOW BYTE OF /RAM DRIVER ADDRESS
 LDA RAMSLOT+1         ; SAVE OFF HI BYTE
 STA ADDRESS+1         ;
*
 LDA NODEV             ; FINALLY COPY THE 'NO DEVICE CONNECTED'
 STA RAMSLOT           ; INTO THE SLOT 3, DRIVE 2 VECTOR AND
```

```
 LDA NODEV+1          ;
 STA RAMSLOT+1        ;
 DEC DEVCNT           ; DECREMENT THE DEVICE COUNT.
*
DONE PLP              ; RESTORE STATUS
*
 RTS                  ; AND RETURN
*
ADDRESS DW $0000      ; STORE THE DEVICE DRIVER ADDRESS HERE
RAMUNITID DFB $00     ; STORE THE DEVICE'S UNIT NUMBER HERE
*
```

5.2.2.4 - Reinstalling /RAM

Part of your exit procedure should include code to reinstall /RAM, making it available to the next application. Be sure /RAM has been disconnected before you reinstall it. Applications should not begin by reinstalling /RAM, because this would preclude passing files from one application to the next in /RAM.

Here is how to reinstall /RAM (or any general device):

1. Reinstall the device driver address you retrieved and saved as the slot 3 drive 2 vector.
2. Increment the device count (DEVCNT).
3. Reinstall the device number in the device list (DEVLST). It may be best to reinstall the device number as the first entry in the list. If the user has manually installed a disk driver, he may assume that because it was the last thing installed that it is still the last one in the list. It is recommended that you move all the entries in the list down one, and reinstall the /RAM device number as the first entry.
4. Set up the parameters for a format request and JSR to the device driver address you have reinstalled. The /RAM driver will set up a new directory and bit map.

The following is an example of what the reinstallation code might look like. These routines deal specifically with /RAM but can easily be adapted to any disk driver routines.

```
*
* THIS IS THE EXAMPLE /RAM INSTALL ROUTINE
*
RAMIN PHP              ; SAVE STATUS
 SEI                   ; AND MAKE SURE INTERRUPTS ARE OFF!
```

```
*
 LDY DEVCNT            ; GET THE NUMBER OF DEVICES - 1.
LOOP1 LDA DEVLST,Y     ; LOAD THE UNIT NUMBER
 AND #$F0              ; CHECK FOR SLOT 3, DRIVE 2 UNIT.
 CMP #$B0              ; IS IT THE SLOT 3, DRIVE 2 UNIT?
 BEQ DONE1             ; IF SO BRANCH.
 DEY                   ; OTHERWISE SEARCH ON...
 BPL LOOP1             ; LOOP UNTIL DEVLST SEARCH IS COMPLETED
 LDA ADDRESS           ; RESTORE THE DEVICE DRIVER ADDRESS
 STA RAMSLOT           ; LOW BYTE..
 LDA ADDRESS+1         ; NOW THE
 STA RAMSLOT+1         ; HI BYTE.
 INC DEVCNT            ; AFTER INSTALLING DEVICE, INC DEVICE COUNT
 LDY DEVCNT            ; USE Y FOR LOOP COUNTER..
LOOP2 LDA DEVLST-1,Y   ; BUBBLE DOWN THE ENTRIES IN DEVICE LIST
 STA DEVLST,Y          ;
 DEY                   ; NEXT
 BNE LOOP2             ; LOOP UNTIL ALL ENTRIES MOVED DOWN.
*
* NOW SET UP A /RAM FORMAT REQUEST
*
 LDA #3                ; LOAD ACC WITH FORMAT REQUEST NUMBER.
 STA $42               ; STORE REQUEST NUMBER IN PROPER PLACE.
*
 LDA RAMUNITID         ; RESTORE THE DEVICE
 STA DEVLST            ; UNIT NUMBER IN THE DEVICE LIST
 AND #$F0              ; STRIP THE DEVICE ID (ZERO LOW NIBBLE)
 STA $43               ; AND STORE THE UNIT NUMBER IN $43.
*
 LDA #$00              ; LOAD LOW BYTE OF BUFFER POINTER
 STA $44               ; AND STORE IT.
 LDA #$20              ; LOAD HI BYTE OF BUFFER POINTER
 STA $45               ; AND STORE IT.
*
 LDA $C08B             ; READ & WRITE ENABLE
 LDA $C08B             ; THE LANGUAGE CARD WITH BANK 1 ON.
*
* NOTE HOW THE DRIVER IS CALLED.  YOU JSR TO AN INDIRECT JMP SO
* CONTROL IS RETURNED BY THE DRIVER TO THE INSTRUCTION AFTER THE JSR.
*
 JSR DRIVER            ; NOW LET DRIVER CARRY OUT CALL.
 BIT $C082             ; NOW PUT ROM BACK ON LINE.
*
 BCC DONE1             ; IF THE CARRY IS CLEAR --> NO ERROR
 JSR ERROR             ; GO PROCESS THE ERROR
*
DONE1 PLP              ; RESTORE STATUS
 RTS                   ; THAT'S ALL
*
DRIVER JMP (RAMSLOT)   ; CALL THE /RAM DRIVER
*
ERROR BRK              ; YOUR ERROR HANDLER CODE WOULD GO HERE
 RTS                   ;
```

5.2.3 - The System Global Page

The $BF page of memory, addresses $BF00 through $BFFF, contains

the system's global variables. Some of them, such as the system bit map and the date and time locations, can be set and used by system programs. Others, such as the machine identification byte, are informational but are not to be changed. Still others are for internal use of the system only. Follow the rules described below.

The DFB assembler directive assigns a value to the current memory location. The DW directive assigns a two-byte address, low byte first, to the current location.

5.2.4 - Rules for Using the System Global Page

MLI entry point. This is the only address in the global page that you should ever call:

```
BF00:         BF00    2           ORG   GLOBALS
BF00:                 3 *
BF00:4C 4B BF         4 ENTRY     JMP   MLIENT1     ;MLI CALL ENTRY POINT
```

Other entry points. Do not use these:

```
BF03:4C F6 BF         5 JSPARE    JMP   SYS.RTS     ;Jump Vector to cold
                                                    ;start, selector program,
                                                    ;etc.
BF06:60 42 D7         6 DATETIME  DFB   $60,$42,$D7 ;CLOCK CALENDAR ROUTINE.
BF09:4C F8 DF         7 SYSERR    JMP   SYSERR1     ;ERROR REPORTING HOOK.
BF0C:4C 04 E0         8 SYSDEATH  JMP   SYSDEATH1   ;SYSTEM FAILURE HOOK.
BF0F:00               9 SERR      DFB   $00         ;ERR CODE, 0=NO ERROR.
```

Disk device driver vectors:

```
BF10:                11 *
BF10:                12 * DEVICE DRIVER VECTORS.
BF10:                13 *
BF10:AB DE           14 DEVADR01  DW    GNODEV      ;SLOT ZERO RESERVED
BF12:AB DE           15           DW    GNODEV      ;SLOT 1, DRIVE 1
BF14:AB DE           16           DW    GNODEV      ;SLOT 2, DRIVE 1
BF16:AB DE           17           DW    GNODEV      ;SLOT 3, DRIVE 1
BF18:AB DE           18           DW    GNODEV      ;SLOT 4, DRIVE 1
BF1A:AB DE           19           DW    GNODEV      ;SLOT 5, DRIVE 1
BF1C:AB DE           20           DW    GNODEV      ;SLOT 6, DRIVE 1
BF1E:AB DE           21           DW    GNODEV      ;SLOT 7, DRIVE 1
BF20:AB DE           22           DW    GNODEV      ;SLOT ZERO RESERVED
BF22:AB DE           23           DW    GNODEV      ;SLOT 1, DRIVE 2
BF24:AB DE           24           DW    GNODEV      ;SLOT 2, DRIVE 2
BF26:AB DE           25           DW    GNODEV      ;SLOT 3, DRIVE 2
BF28:AB DE           26           DW    GNODEV      ;SLOT 4, DRIVE 2
BF2A:AB DE           27           DW    GNODEV      ;SLOT 5, DRIVE 2
BF2C:AB DE           28           DW    GNODEV      ;SLOT 6, DRIVE 2
BF2E:AB DE           29           DW    GNODEV      ;SLOT 7, DRIVE 2
```

List of all active disk devices by unit number. When access to an unrecognized volume is requested, devices are searched from the end of the list to the beginning. See also Sections 3.1, 3.2, and 4.4.6. The lower half of each byte in **DEVLST** is a device identification: 0 = Disk II, 4 = ProFile, $F = /RAM.

```
BF30:               31 *
BF30:               32 * CONFIGURED DEVICE LIST BY DEVICE NUMBER
BF30:               33 * ACCESS ORDER IS LAST IN LIST FIRST.
BF30:               34 *
BF30:00             35 DEVNUM    DFB   $00         ;MOST RECENT ACCESSED
                                                   ;DEVICE.
BF31:FF             36 DEVCNT    DFB   $FF         ;NUMBER OF ON-LINE DEVICES
                                                   ;(MINUS 1).
BF32:00 00 00 00    37 DEVLST    DFB   $0,0,0,0    ;UP TO 14 UNITS MAY BE
                                                   ;ACTIVE.
BF36:00 00 00 00    38           DFB   0,0,0,0,0
BF3B:00 00 00 00    39           DFB   0,0,0,0,0

BF40:28 43 29 41    41           ASC   "(C)APPLE'83"
```

Routines reserved for MLI and subject to change.

```
BF4B:08             42 MLIENT1   PHP
BF4C:78             43           SEI
BF4D:4C B7 BF       44           JMP   MLICONT
BF50:8D 8B C0       45 AFTIRQ    STA   RAMIN
BF53:4C D8 FF       46           JMP   FIX45       ;Restore $45 after
                                                   ;Interrupt in Lang Card
BF56:00             47 OLD45     DFB   0
BF57:00             48 AFBANK    DFB   0
```

Memory map of the lower 48K. Each bit represents one page (256 bytes) of memory. Protected areas are marked with a **1**, uprotected with a **0**. ProDOS disallows reading into or io_buffer allocation in protected areas. See Section 5.1.

```
BF58:C0 00 00 00    56 MEMTABL   DFB   $C0,$00,$00,$00,$00,$00,$00,$00
BF60:00 00 00 00    57           DFB   $00,$00,$00,$00,$00,$00,$00,$00
BF68:00 00 00 00    58           DFB   $00,$00,$00,$00,$00,$00,$00,$01
```

The addresses in this table are buffer addresses for open files. These are informational only; they should not be changed except using the MLI call SET_BUF.

```
BF70:00 00          66 GL.BUFF   DW    $0000       ;FILE NUMBER 1
BF72:00 00          67           DW    $0000       ;FILE NUMBER 2
BF74:00 00          68           DW    $0000       ;FILE NUMBER 3
BF76:00 00          69           DW    $0000       ;FILE NUMBER 4
BF78:00 00          70           DW    $0000       ;FILE NUMBER 5
BF7A:00 00          71           DW    $0000       ;FILE NUMBER 6
BF7C:00 00          72           DW    $0000       ;FILE NUMBER 7
BF7E:00 00          73           DW    $0000       ;FILE NUMBER 8
```

Interrupt vectors are stored here. Again, these are informational and should be changed only by a call to the MLI using ALLOC_INTERRUPT. **Values of the A, X, Y, stack, and status registers at the time of the most recent interrupt are also stored here.** In addition, the address interrupted is preserved. These may be used for performance studies and debugging, but should not be changed by the user. The routines are polled in ascending order. See Section 6.2.

```
 BF80:00 00          85 INTRUPT1  DW    $0000       ;INTERRUPT ROUTINE 1
 BF82:00 00          86 INTRUPT2  DW    $0000       ;INTERRUPT ROUTINE 2
 BF84:00 00          87 INTRUPT3  DW    $0000       ;INTERRUPT ROUTINE 3
 BF86:00 00          88 INTRUPT4  DW    $0000       ;INTERRUPT ROUTINE 4
 BF88:00             89 INTAREG   DFB   $00         ;A-REGISTER
 BF89:00             90 INTXREG   DFB   $00         ;X-REGISTER
 BF8A:00             91 INTYREG   DFB   $00         ;Y-REGISTER
 BF8B:00             92 INTSREG   DFB   $00         ;STACK REGISTER
 BF8C:00             93 INTPREG   DFB   $00         ;STATUS REGISTER
 BF8D:01             94 INTBANKID DFB   $01         ;ROM, RAM1, OR RAM2 ($D000 IN LC)
 BF8E:00 00          95 INTADDR   DW    $0000       ;PROGRAM COUNTER RETN ADDR
```

The following options can be changed before calls to the MLI:

```
 BF90:00 00         101 DATELO    DW    $0000       ;BITS 15-9=YR, 8-5=MO, 4-0=DAY
 BF92:00 00         102 TIMELO    DW    $0000       ;BITS 12-8=HR, 5-0=MIN; LOW-HI
FORMAT.
 BF94:00            103 LEVEL     DFB   $00         ;FILE LEVEL: USED IN OPEN, FLUSH,
CLOSE.
 BF95:00            104 BUBIT     DFB   $00         ;BACKUP BIT DISABLE, SETFILEINFO
ONLY.
 BF96:00 00         105 SPARE1    DFB   $00,$00     ;RESERVED FOR MLI USE
```

The definition of MACHID at $BF98 is:

```
 BF98:              107 *
 BF98:              108 * The following are informational only.  MACHID
 BF98:              109 * identifies the System Attributes:
 BF98:              110 * (Bit 3 off) BITS 7,6-  00=II  01=II+   10=IIe   11=///
EMULATION
 BF98:              111 * (Bit 3 on)  BITS 7,6-  00=NA  01=NA    10=//c   11=NA
 BF98:              112 *              BITS 5,4-  00=NA  01=48K   10=64K   11=128K
 BF98:              113 *              BIT  3  -  Modifier for MACHID Bits 7,6.
 BF98:              114 *              BIT  2  -  RESERVED FOR FUTURE DEFINITION.
 BF98:              115 *              BIT  1=1-  80 Column card
 BF98:              116 *              BIT  0=1-  Recognizable Clock Card
 BF98:              117 *
 BF98:              118 * SLTBYT indicates which slots are determined to have
 BF98:              119 * ROMS. PFIXPTR indicates an active PREFIX if it is
 BF98:              120 * non-zero. MLIACTV indicates an MLI call in progress
 BF98:              121 * if it is non-zero. CMDADR is the address of the last
 BF98:              122 * MLI call's parameter list. SAVX and SAVY are the
 BF98:              123 * values of X and Y when the MLI was last called.
 BF98:              124 *
 BF98:00            125 MACHID    DFB   $00         ;MACHINE IDENTIFICATION.
 BF99:00            126 SLTBYT    DFB   $00         ;'1' BITS INDICATE ROM IN
```

```
SLOT(BIT#)
 BF9A:00            127 PFIXPTR   DFB   $00         ;IF = 0, NO PREFIX ACTIVE..
 BF9B:00            128 MLIACTV   DFB   $00         ;IF <> 0, MLI call in progress
 BF9C:00 00         129 CMDADR    DW    $0000       ;RETURN ADDRESS OF LAST CALL TO
MLI.
 BF9E:00            130 SAVEX     DFB   $00         ;X-REG ON ENTRY TO MLI
 BF9F:00            131 SAVEY     DFB   $00         ;Y-REG ON ENTRY TO MLI
```

The following space is reserved for Language Card bank-switching routines. All routines and addresses are subject to change at any time without notice and will, in fact, vary with system configuration. The routines presented here are for 64K systems only:

```
 BFA0:4D 00 E0      141 EXIT      EOR   $E000       ;TEST FOR ROM ENABLE.
 BFA3:F0 05   BFAA  142           BEQ   EXIT1       ;BRANCH IF RAM ENABLED.
 BFA5:8D 82 C0      143           STA   ROMIN       ;ELSE ENABLE ROM & RETURN.
 BFA8:D0 0B   BFB5  144           BNE   EXIT2       ;BRANCH ALWAYS
 BFAA:              145 **
 BFAA:AD F5 BF      146 EXIT1     LDA   BNKBYT2     ;FOR ALT RAM (MOD BY MLIENT1)
 BFAD:4D 00 D0      147           EOR   $D000       ;ENABLE.
 BFB0:F0 03   BFB5  148           BEQ   EXIT2       ;BRANCH IF NOT ALT RAM.
 BFB2:AD 83 C0      149           LDA   ALTRAM      ;ELSE ENABLE ALT $D000
 BFB5:68            150 EXIT2     PLA               ;RESTORE RETURN CODE.
 BFB6:40            151           RTI               ;RE-ENABLE INTERRUPTS & RETURN
 BFB7:              152 **
 BFB7:38            153 MLICONT   SEC
 BFB8:6E 9B BF      154           ROR   MLIACTV     ;INDICATE TO INTERRUPT ROUTINES MLI
ACTIVE.
 BFBB:AD 00 E0      155           LDA   $E000       ;PRESERVE LANGUAGE CARD / ROM
 BFBE:8D F4 BF      156           STA   BNKBYT1     ; ORIENTATION FOR PROPER
 BFC1:AD 00 D0      157           LDA   $D000       ; RESTORATION WHEN MLI EXITS...
 BFC4:8D F5 BF      158           STA   BNKBYT2
 BFC7:AD 8B C0      159           LDA   RAMIN       ;NOW FORCE RAM CARD ON
 BFCA:AD 8B C0      160           LDA   RAMIN       ; WITH RAM WRITE ALLOWED.
 BFCD:4C 00 DE      161           JMP   ENTRYMLI
```

Interrupt exit and entry routines:

```
 BFD0:              163 *
 BFD0:              164 * INTERRUPT EXIT/ENTRY ROUTINES
 BFD0:              165 *

 BFD0:AD 8D BF      167 IRQXIT    LDA   INTBANKID   ;DETERMINE STATE OF RAM CARD
 BFD3:F0 0D   BFE2  168 IRQXIT0   BEQ   IRQXIT2     ; IF ANY.  BRANCH IF ENABLED.
 BFD5:30 08   BFDF  169           BMI   IRQXIT1     ;BRANCH IF ALTERNATE $D000 ENABLED.
 BFD7:4A            170           LSR   A           ;DETERMINE IF NO RAM CARD PRESENT.
 BFD8:90 0D   BFE7  171           BCC   ROMXIT      ;BRANCH IF ROM ONLY SYSTEM.
 BFDA:AD 81 C0      172           LDA   ROMIN1      ;ELSE ENABLE ROM FIRST.
 BFDD:B0 08   BFE7  173           BCS   ROMXIT      ;BRANCH ALWAYS TAKEN...
 BFDF:AD 83 C0      174 IRQXIT1   LDA   ALTRAM      ;ENABLE ALTERNATE $D000.
 BFE2:A9 01         175 IRQXIT2   LDA   #1          ;PRESET BANKID FOR ROM.
 BFE4:8D 8D BF      176           STA   INTBANKID   ;(RESET IF RAM CARD INTERRUPT)
 BFE7:AD 88 BF      177 ROMXIT    LDA   INTAREG     ;RESTORE ACCUMULATOR...
 BFEA:40            178           RTI               ; AND EXIT!

 BFEB:2C 8B C0      180 IRQENT    BIT   RAMIN       ;THIS ENTRY ONLY USED WHEN ROM
```

```
 BFEE:2C 8B C0      181           BIT   RAMIN       ; WAS ENABLED AT TIME OF INTERRUT.
 BFF1:4C 4D DF      182           JMP   IRQRECEV    ; A-REG IS STORED AT $45 IN ZPAGE.
 BFF4:00            183 BNKBYT1   DFB   $00
 BFF5:00            184 BNKBYT2   DFB   $00
 BFF6:              185 **
 BFF6:2C 8B C0      186 SYS.RTS   BIT   RAMIN       ;Make certain Language card is
switched in
 BFF9:4C 02 E0      187           JMP   SYS.END     ;Or anywhere else we need to go
```

Each system program should set IVERSION to its own current version number. ProDOS sets KVERSION to its current version number.

```
 BFFC:00            188 IBAKVER   DFB   $00         ;UNDEFINED: Reserved for future use
 BFFD:00            189 IVERSION  DFB   $00         ;Version # of currently running
Interpreter
 BFFE:00            191 KBAKVER   DFB   $00         ;UNDEFINED: Reserved for future use
 BFFF:02            192 KVERSION  DFB   $2          ;VERSION NO. (RELEASE ID)
```

5.3 - General Techniques

The first part of this chapter discusses the things that a system program must do. This section of the manual describes some of the things that system programs commonly do, and it gives some techniques for implementing them.

5.3.1 - Determining Machine Configuration

It is often useful for a system program to know what type of Apple II it is running on. The MACHID byte in the system global page identifies the machine type, the amount of memory, and whether an 80-column text card or clock/calendar card was detected.

MACHID byte: see Section 5.2.3.

5.3.1.1 - Machine Type

Two bits distinguish an Apple II, an Apple II Plus, an Apple IIe, an Apple IIc, or an Apple III in Apple II emulation mode. This distinction is most useful for two reasons:

1. The Apple IIe and IIc always have lowercase available. Screen messages can be coded using uppercase and lowercase, and then made all uppercase if the machine is not an Apple IIe or IIc (or if

it is a Apple II without an 80-column text card).

2. The Apple IIe and IIc have keys that are not available on earlier versions of the Apple II (most notably [UP], [DOWN], [OA], [SA], and [DELETE]). Software should be coded to use the keys most convenient for the system it is running on, and the screen messages should be adjusted accordingly.

5.3.1.2 - Memory Size

The possible memory sizes are 64K and 128K. A system program can use these values when deciding where to relocate itself. Recall that the alternate 64K bank cannot contain code that makes calls to the MLI and it cannot be used for system buffers.

5.3.1.3 - 80-Column Text Card

This bit is always set in the Apple IIc. It is set in an Apple IIe if an 80-column text card that follows the defined protocol is in slot 3 or in the auxiliary slot. This protocol guarantees that the features of the card can be turned on by a JSR to $C300, the beginning of the ROM on the card (note that this disconnects BASIC.SYSTEM).

80-column text cards -- and other Apple IIe features -- can be turned off using the following sequence of instructions:

```
LDA #$15     ;Character that turns off video firmware
JSR $C300   ;Print it to the video firmware
```

5.3.2 - Using the Date

A system program often has reason to use the current date: to mark files with a modification date, to use as identification on a listing, or just for display on the screen. Whatever the use, it is usually desirable to obtain the most current setting.

Save the system date and time locations ($BF90-BF93) for possible future use, and then clear them. Next use the GET_TIME call. If there is a clock/calendar card with an installed clock routine, then the system date and time locations will become nonzero. This is the date and time you should use. If the GET_TIME call has no effect, then you should either use the values that were previously in the date and time locations, or prompt the user for the current date and time. Since

the date and time locations are set to 0 when the system is started (unless ProDOS recognizes a clock/calendar card), it is reasonable to use nonzero values of the date and time locations as a default date and time.

If there is no system time, and the call to GET_TIME returns nothing an alternative is to use the GET_FILE_INFO call and to use the last modified date and time as a default. If the user updates the time, and you place these values in the system date and time locations, a SET_FILE_INFO call will update the time for the next GET_FILE_INFO.

The system updates the date and time at every CREATE, DESTROY, RENAME SET_FILE_INFO CLOSE, and FLUSH operation.

Refer to the GET_TIME call in Chapter 4, and to the description of clock/calender routines in Chapter 6 for more details.

5.3.3 - System Program Defaults

Each file entry in a directory has a two-byte aux_type field. This field contains information such as load address for BASIC programs or binary files, and record length for text files; for system files it is unused. If your system program has a small amount of default information that you would like to preserve from one execution of the program to the next, this field is a good place to store it.

To alter the contents of this field, use the GET_FILE_INFO call to read the current contents of the file's entry, change the values in the aux_id field, then use the SET_FILE_INFO call with the same parameter list to save the modified values in the file's entry.

5.3.4 - Finding a Volume

Since one does not always know the names of all the online volumes, it is sometimes necessary to allow users to specify volumes by slot and drive instead of by volume name. Before the slot and drive information can be used to access ProDOS files, it must be converted to a volume name. To convert slot and drive numbers to volume names, you can

use the following steps:

1. Make the slot and drive numbers into a unit_num. This number is used to specify the desired device to the ON_LINE call. The format of a unit_num is given in Section 4.4.6.
2. Use the unit_num in the ON_LINE call. This call will return a count byte followed by the volume name. This volume name is not preceded by a slash. You must increase the count by one and insert a slash preceding the volume name before using this name in other ProDOS calls.

5.3.5 - Using the RESET Vector

In the Apple II, pressing [CONTROL]-[RESET] causes an unconditional jump to the RESET vector (at $3F2 in memory). Because the user can press [CONTROL]-[RESET] at any time -- including while files are open -- ProDOS cannot take responsibility for disk integrity after [RESET] has been pressed: the system program must do it.

Your program should place in the RESET vector the address of a routine that displays a message advising that it will be closing any open files, and then close the files. Once this is done, the program may take any action required by the application. It is preferable either to jump back to the beginning of the program or to jump directly to the quit routine.

5.4 - ProDOS System Program Conventions

For the sake of consistency from one piece of software to the next follow the conventions used in this manual:

- Use the same terminology whenever possible. If your application implements any of the functions used by the BASIC system program, the Filer, the Convert program, or the Editor/Assembler, try to use the same wording.
- Use the same catalog format in all software that displays a list of files. It is not necessary to implement both the 40- and 80-column formats (see the CAT and CATALOG commands of the BASIC system program).

  If you choose to implement your own version of this command,

recognize the file types and display the three-letter abbreviations that are shown in the quick reference card of this manual.

- The standard Apple II "Air-raid" bell has been replaced with a gentler tone. Use it to give users some aural feedback that they are using a ProDOS program. The code for it follows.

Page 101

```
SPKR      EQU   $C030         ;this clicks the speaker
*
LENGTH    DS    1             ;duration of tone
*
* This is the wait routine from the Monitor ROM.
*
WAIT      SEC
WAIT2     PHA
WAIT3     SBC   #1
          BNE   WAIT3
          PLA
          SBC   #1
          BNE   WAIT2
          RTS
*
* Generate a nice little tone
* Exits with Z-flag set (BEQ) for branching
* Destroys the contents of the accumulator
*
BELL      LDA   #$20          ;duration of tone
          STA   LENGTH
BELL1     LDA   #$2           ;short delay...click
          JSR   WAIT
          STA   SPKR
          LDA   #$20          ;long delay...click
          JSR   WAIT
          STA   SPKR
          DEC   LENGTH
          BNE   BELL1         ;repeat LENGTH times
          RTS
```

Page 102

Chapter 6
Adding Routines to ProDOS

Page 103

This chapter explains device-handling routines that can be used with the ProDOS MLI. Because such routines are connected to and interact with the MLI, they are essentially invisible to the BASIC system

program described in Appendix A of this manual and in *BASIC Programming With ProDOS*.

Appendix A explains the rules for installing routines when the BASIC system program is active.

The types of routines described in this chapter are:

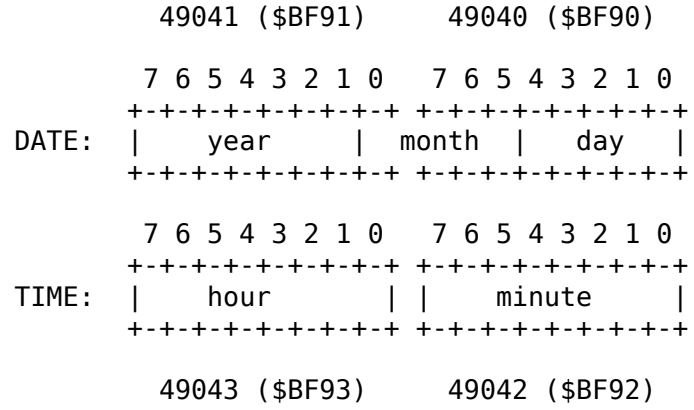
- clock/calendar routines
- interrupt handling routines
- disk driver routines.

Note: These routines must all begin with a CLD instruction and end with an RTS.

6.1 - Clock/Calendar Routines

ProDOS has a built-in clock driver that queries a clock/calendar card for the date and time. After the routine stores that information in the ProDOS Global Page ($BF90-$BF93), either ProDOS or your own application programs can use it. See Figure 6-1.

Figure 6-1. ProDOS Date and Time Locations



You can cause ProDOS to call the clock driver and to update the date and time by issuing a GET_TIME call (see Section 4.6.1).

ProDOS calls the clock driver routine for every call that might need the date and time: CREATE, DESTROY, RENAME SET_FILE_INFO CLOSE, and FLUSH.

The ProDOS clock driver expects the clock card's firmware to return information in a certain way. The ROM on the clock card must also follow Apple's identification convention if it is to be recognized by ProDOS at startup.

The ProDOS clock driver expects the clock card to send an ASCII string to the GETLN input buffer ($200). This string must have the following format (including the commas):

mo,da,dt,hr,mn

where

mo is the month (01 = January...12 = December)
da is the day of the week (00 = Sunday...06 = Saturday)
dt is the date (00 through 31)
hr is the hour (00 through 23)
mn is the minute (00 through 59)

For example:

07,04,14,22,46

would represent Thursday, July 14, 10:46 p.m. The year is looked up in a table in the clock driver.

When the ProDOS system file is executed, it installs the address of the clock routine at $BF07, $BF08 -- whether there is a recognized clock card or not.

ProDOS recognizes a clock card if the following bytes are present in the Cn00 ROM:

$Cn00 = $08
$Cn02 = $28
$Cn04 = $58
$Cn06 = $70

The address is preceded by a $4C (JMP) if a clock card is recognized, or by a $60 (RTS) if not.

The ProDOS clock driver uses the following addresses for its I/O to the clock:

Cn08 - READ entry point
Cn0B - WRITE entry point

The accumulator is loaded with an #A3 before the JSR to the WRITE entry point. This value could be used to let the clock card's firmware know in what format to leave the time.

The ProDOS driver takes the ASCII values sent by the clock, converts them to binary, and stores them in the ProDOS Global Page.

The driver uses zero page locations $3A through $3E. It also saves and restores the peripheral RAM card location $F8+n, where *n* is the slot where the card resides.

6.1.1 - Other Clock/Calendars

To support clock cards that do not follow the ProDOS protocol defined above, you can locate your code in a number of places. The cleanest solution is to replace the ProDOS routines with your own, if they fit.

If you look at $BF07,$BF08, you will find the location to put your code. There is room for 125 bytes.

To install your code, simply write-enable the language card area, and move your code. Your relocation code must justify the absolute addresses as part of the relocation procedure. Finally, restore any soft switches you have changed. (There is no guarantee as to the absolute location of the clock-driver code on future revisions of ProDOS, only that its location can be found by examining the global page.)

All that your code needs to do is get the time from the clock card, convert it to the ProDOS format, and store it in the date and time locations in the global page.

Your installation routine can be called either from an application program, or as part of the STARTUP program.

6.2 - Interrupt Handling Routines

To aid the development of software that can handle interrupts, the MLI provides a convention for interfacing interrupt driven devices.

To use interrupts, you must install from one to four interrupt receiving routines somewhere in memory. It is up to you to check and update

the system bit map to be sure that the routines do not conflict with ProDOS or other concurrently executing programs.

Once a routine is installed, you must use the ALLOC_INTERRUPT call to inform the MLI of the starting address of the receiving routine. After this call has been successfully completed, you may enable the hardware for interrupts.

When an interrupt occurs, the MLI's interrupt handler preserves the 6502's registers, zero page locations $FA thru $ff, and, if the stack is more than 3/4 full, 16 bytes of the stack. Then it calls each receiving routine (via JSR), one by one, in the order in which they were installed. Each installed routine must begin with a CLD instruction.

When the routine that can process the interrupt is called, it should carry out its task, clear the interrupt on the hardware, and return (via an RTS) with the carry flag clear. When a routine that cannot process the interrupt is called, it should return (via an RTS) with the carry flag set so that the MLI knows to call the next routine in the list.

As mentioned above, all 6502 registers, locations $FA thru $FF and if the stack is more than 3/4 full, 16 bytes of the stack, are preserved. The interrupt routine may use these resources freely for temporary data storage.

Note: There is no general way for an interrupt routine to identify whether or not its device was the source of the interrupt. This task depends on the specific characteristics of the device; in fact, some devices provide no mechanism for interrupt verification. It is necessary to service such a device after all others have been polled.

If no installed and allocated routine claims a pending interrupt, a **SYSTEM FAILURE** message will be displayed and program execution will be halted.

When finished with a interrupt driven device, a DEALLOC_INTERRUPT call should be made, but only after the device itself is disabled.

Warning
This warning does not apply to the Apple IIc nor to Apple IIe's with enhanced ROMs. Because the Apple II Monitor program relies

on a zero-page location ($45) that is overwritten when an interrupt occurs, you should disable interrupts while you are using the Monitor program. The system also uses location $7F8 to store the I/O slot location that was in use before an interrupt occurred; do not use this location.

6.2.1 - Interrupts During MLI Calls

The preceding section does not discuss what a program should do if an interrupt were to occur during the execution of an MLI call and your interrupt handling routine itself makes calls to the MLI.

The interrupt routine must allow the MLI to complete its current call before initiating a new call to the MLI. The mechanism for doing this consists of changing the globals so that the MLI completes its call and returns to your routine rather than to the the routine that originally called it. Then your routine can use the MLI as needed. When it is finished, it must restore the 6502 registers to the state they would have been in at completion of the MLI call had the interrupt not occurred, and then jump back to the proper address in the original routine.

To do this, the interrupt handling routine should first check the status of the MLI. If the flag **MLIACTV** ($BF9B) has the high bit set, then the MLI was in the middle of a call. Your routine should then:

1. Save the return address of the original caller (**CMDADR**, $BF9C), replacing it with the address to which the MLI should return on completion of the current call.
2. Claim the interrupt by disabling interrupts on the hardware, and clearing the carry flag.
3. RTS

   The MLI's interrupt handler believes that the interrupt has been processed, so it completes the current MLI call and returns to the address in **CMDADR**, which is actually in your routine. Your routine should now do this:

4. Save the A, X, Y, and P registers as the return state for the routine whose call just completed.
5. Use the MLI as needed.
6. Restore the A, X, Y, and P registers.

7. Jump to the original **CMDADR**.

The original program sees only that its MLI call was successfully completed, and it continues execution.

6.2.2 - Sample Interrupt Routine

Here is a sample interrupt routine that reads the date from a clock/calender card, and displays it in the upper-right corner of the screen once per second. It assumes the card is in slot 2.

```
SOURCE   FILE #01 =>SHOWTIME
----- NEXT OBJECT FILE NAME IS SHOWTIME.0
0300:        0300    1           ORG   $300
0300:        C20B    2 WTTCP     EQU   $C20B     ;CLOCK WRITE ENTRY PT (SLOT 2)
0300:        C208    3 RDTCP     EQU   $C208     ;CLOCK READ ENTRY PT (SLOT 2)
0300;        C080    4 TCICR     EQU   $C080     ;INTERRUPT CONTROL REG (SLOT 2)
0300:        C088    5 TCMR      EQU   $C088     ;MYSTERY REGISTER (SLOT 2)
0300:                6 *
0300:        0200    7 IN        EQU   $200      ;WHERE CLOCK LEAVES THE TIME
0300:                8 *
0300:        0412    9 UPRIGHT   EQU   $412      ;THE UPPER RIGHT OF THE SCREEN
0300:        047A   10 INTONI    EQU   $47A      ;LEAVE INTERRUPTS ON (SLOT 2)
0300:        07FA   11 INTON2    EQU   $7FA      ;LEAVE INTERRUPTS ON (SLOT 2)
0300:                12 *
0300:        BF00   13 MLI       EQU   $BF00     ;ENTRY POINT TO THE PRODOS MLI
0300:                14 *
0300:                15 * CALLING INTERRUPTS, CALLING INTERRUPTS
0300:                16 *
0300:20 7E 03        17           JSR   ALLOC.INT ;HAVE MLI INSTALL INT ROUTINE
0303:60              18           RTS             ;THAT'S ALL FOLKS
0304:                19 *
0304:                20 *
0304:        0304    21 SHOWTIME  EQU   *
0304:D8              22           CLD
0305:08              23           PHP
0306:78              24           SEI             ;DISABLE INTERRUPTS
0307:A0 20           25           LDY   #$20      ; FOR SLOT 2
O3O9;B9 80 C0        26           LDA   TCICR,Y   ;GET VAL OF INT CONTROL REG
03OC:29 20           27           AND   #$20      ;CHK BIT 5 - IS INT FROM CLK?
030E:F0 3C   034C    28           BEQ   NOTCLK    ;IF BIT 5 OFF, INT NOT FROM CLK
0310:B9 88 C0        29           LDA   TCMR,Y    ;CLEAR MYSTERY REGISTER
0313:B9 80 C0        30           LDA   TCICR,Y   ;CLEAR INTERRUPT ON HARDWARE
0316:CE 4F 03        31           DEC   COUNTER   ;ONLY PRINT TIME EVERY SECOND
0319:D0 2E   0349    32           BNE   EXITCLK   ; NOT TIME TO PRINT YET
031B:                33 *
031B:A2 27           34           LDX   #39       ;SAVE THE INPUT BUFFER
031D:BD 00 02        35 DOIN      LDA   IN,X      ; SINCE THE CLOCK WRITES OVER
0320:9D 56 03        36           STA   INBUF,X   ; IT WHEN IT IS CALLED
0323:CA              37           DEX             ;
0324:10 F7   031D    38           BPL   DOIN      ;
0326:                39
0326:A9 A5           40           LDA   #$A5      ;SET APPLESOFT STRING INPUT
0328:20 0B C2        41           JSR   WTTCP     ; MODE & SEND IT TO THE CARD
```

```
032B:20 08 C2       42           JSR   RDTCP     ;READ TIME INTO INPUT BUFFER
032E:               43
032E:A2 15          44           LDX   #21
0330:BD 01 02       45 GETNEXT   LDA   IN+1,X    ;PRINT TIME TO SCREEN
0333:9D 12 04       46           STA   UPRIGHT,X ;CHARS 0-22 OF INPUT BUFFER
0336:CA             47           DEX               ;
0337:10 F7   0330   48           BPL   GETNEXT     ;
0339:               49
0339:A9 40          50 SETCNTR   LDA   #64         ;SET UP COUNTER FOR NEXT TIME
```

```
033B:8D 4F 03       51           STA   COUNTER     ;
033E:               52
033E:A2 27          53           LDX   #39         ;RESTORE THE INPUT BUFFER
0340:BD 56 03       54 DOIN2     LDA   INBUF,X     ;
0343:9D 00 02       55           STA   IN,X        ;
0346:CA             56           DEX               ;
0347:10 F7   0340   57           BPI   DOIN2       ;
0349:               58 *
0349:28             59 EXITCLK   PLP
034A:18             60           CLC               ;TELL MLI INT WAS PROCESSED
034B:60             61           RTS
034C:28             62 NOTCLK    PLP
034D:38             63           SEC               ;TELL MLI IT ISN'T OURS
034E:60             64           RTS
034F:               65 *
034F:        0001   66 COUNTER    DS    1,0         ;
0350;               67 *
0350:02 00          68 AIPARMS   DFB   2,0         ;PUT ALLOCATE AND DEALLOCATE
0352:04 03          69           DW    SHOWTIME    ; INTERRUPT PARAMETERS HERE,
0354:               70 *
0354:01 00          71 DIPARMS   DFB   1,0         ; SO BOTH ROUTINES CAN USE THEM
0356:               72 *
0356:        0028   73 INBUF     DS    40,0        ;SAVE 40 BYTES IN HERE
037E:               74 *                           ; FOR INPUT BUFFER SAVE/RESTORE
```

Note the important features of this routine:

1. The routine begins with a CLD instruction (line 22).
2. The routine checks to see if the IRQ interrupt is being caused by the clock/calendar card (lines 25-28). If not, it returns with the carry set (lines 62-64).
3. If the interrupt belongs to the clock/calendar card, it clears the interrupt hardware (lines 29-30).
4. When it is done with the interrupt task, it returns with carry clear (lines 59-61).

The following routine adds the interrupt routine to ProDOS using the ALLOC_INTERRUPT call. Having done this, it then activates interrupts on the clock/calendar card. Then a CLI instruction is executed to allow

the 6502 to process interrupts.

```
03A0:A9 00          94 DEALLOC.INT LDA #0        ;DISABLE INTERRUPTS
03A2:8D 7A 04       95             STA   INTON1  ; IN THE THUNDERCLOCK
03A5:8D FA 07       96             STA   INTON2
03A8:Ao 20          97             LDY   #$20
03AA;99 80 C0       98             STA   TCICR,Y
03AD:               99 *
03AD:AD 51 03      100             LDA   AIPARMS+1 ;GET INT_NUM
03B0:8D 55 03      101             STA   DIPARMS+1 ; FOR DEALLOCATION
03B3:20 00 BF      102             JSR   MLI       ;CALL THE MLI TO
03B6:41            103             DFB   $41       ; DEALLOCATE INT ROUTINE
03B7:54 03         104             DW    DIPARMS
03B9:D0 01   03BC  105             BNE   OOPS2     ;BREAK ON ERROR
03BB:60            106             RTS             ;DONE
03BC:              107 *
03BC:00            108 OOPS2       BRK             ;BREAK ON ERROR
```

The next routine disables interrupts on the clock/calendar card before removing the interrupt routine from ProDOS with a DEALLOC_INTERRUPT call.

```
037E:               75
037E:20 00 BF       76 ALLOC.INT JSR   MLI       ;CALL THE MLI TO
0381:40             77           DFB   $40       ; ALLOCATE THE INTERRUPT
0382:50 03          78           DW    AIPARMS   ;
0384:D0 19   039F   79           BNE   OOPS      ;BREAK ON ERROR
0386:               80 *
0386:A0 20          81           LDY   #$20
0388:A9 AC          82           LDA   #$AC      ;SET 64HZ INTERRUPT RATE
038A:20 0B C2       83           JSR   WTTCP     ; BY WRITING A ',' To CLOCK
038D:A9 40          84           LDA   #$40      ;NOW ENABLE THE SOFTWARE
038F:8D 7A 04       85           STA   INTON1    ; AND TELL IT NOT TO DISABLE
0392:8D FA 07       86           STA   INTON2    ; INTERRUPTS AFTER READS
0395:99 80 C0       87           STA   TCICR,Y
0398:A9 01          88           LDA   #1        ;PRINT TIME IMMEDIATELY
039A:8D 4F 03       89           STA   COUNTER   ; ONCE PER SECOND LATER
039D:58             90           CLI             ;ALLOW THE 6502 TO SEE THE
039E:60             91           RTS             ; INTERRUPTS
039F:               92 *
039F:00             93 OOPS      BRK             ;BREAK ON ERROR
```

6.3 - Disk Driver Routines

If a disk drive supplied by another manufacturer is to work with ProDOS, it must look and act just like a disk drive supplied by Apple Computer, Inc. Its boot ROM must have certain things in certain locations, and its driver routine must use certain zero-page locations for its call parameters.

6.3.1 - ROM Code Conventions

During startup, ProDOS searches for block storage devices. If it finds the following three bytes in the ROM of a particular slot, ProDOS assumes it has found a disk drive (*n* represents slot number):

$Cn01 = $20
$Cn03 = $00
$Cn05 = $03

If $CnFF = $00, ProDOS assumes it has found a Disk II with 16-sector ROMs and marks the device driver table in the ProDOS global page with the address of the Disk II driver routines. The Disk II driver routines support any drive that emulates Apple's 16-sector Disk II (280 blocks, single volume, and so on).

If $CnFF = $FF, ProDOS assumes it has found a Disk II with 13-sector ROMs, which ProDOS does not support.

If ProDOS finds a value other than $00 or $FF at $CnFF, it assumes it has found an intelligent disk controller. If the STATUS byte at $CnFE indicates that the device supports READ and STATUS requests, ProDOS marks the global page with a device-driver address whose high-byte is equal to $Cn and whose low-byte is equal to the value found at $CnFF.

The only calls to the disk driver are STATUS, READ, WRITE, and FORMAT. The STATUS call should perform a check to verify that the device is ready for a READ or WRITE. If it is not, the carry should be set and the appropriate error code returned in the accumulator. If the device is ready for a READ or WRITE, then the driver should clear the carry, place a zero in the accumulator, and return the number of blocks on the device in the X-register (low-byte) and Y-register (high-byte).

If you wish to interface a disk controller card with more than two drives (or a device with more than two volumes), additional device driver vectors for disk controllers plugged into slot 5 or 6 may be installed in slot 1 or 2 locations. There will be no conflict with character devices physically present in these slots.

Device numbers for four drives in slot 5 or 6 are listed below.

Physical Slot Five:
S5,D1 = $50
S5,D2 = $D0
S1,D1 = $10
S1,D2 = $90

Physical Slot Six:
S6,D1 = $60
S6,D2 = $E0
S2,D1 = $20
S2,D2 = $A0

The special locations in the ROM code are:

$CnFC-
$CnFD
: The total number of blocks on the device. Used for writing the disk's bit map and directory header after formatting. (If this location is $0000, it indicates that the number of blocks must be obtained by making a STATUS request.)

$CnFE
: The status byte (bits 0 and 1 must be set for ProDOS to install the driver vector.)
bit 7 - Medium is removable.
bit 6 - Device is interruptable.
bit 5-4 - Number of volumes on the device (0-3).
bit 3 - The device supports formatting.
bit 2 - The device can be written to.
bit 1 - The device can be read from (must be on).
bit 0 - The device's status can be read
-- (must be on).

$CnFF
: The low-byte of entry to the driver routines. ProDOS will place $Cn + this byte in the global page.

6.3.2 - Call Parameters

parameters are passed to the driver are:

$42
Command:

0 = STATUS request
1 = READ request
2 = WRITE request
3 = FORMAT request

Note: The FORMAT code in the driver need only lay down address marks if required. The calling routine should write the virgin directory and bit map.

$43
Unit Number:

```
  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+
|DR|  SLOT  | NOT USED  |
+--+--+--+--+--+--+--+--+
```

Note: The UNIT_NUMBER that appears in the device list (DEVLST) in the system globals will include the high nibble of the status byte ($CnFE) as an ID in its low nibble.

$44-$45
Buffer Pointer:

Indicates the start of a 512-byte memory buffer for data transfer.

$46-$47
Block Number:

Indicates the block on the disk for data transfer.

The device driver should report errors by setting the carry flag and loading the error code into the accumulator. The error codes that should be implemented are:

$27 - I/O error
$28 - No device connected
$2B - Write protected

Appendix A
The ProDOS BASIC System Program

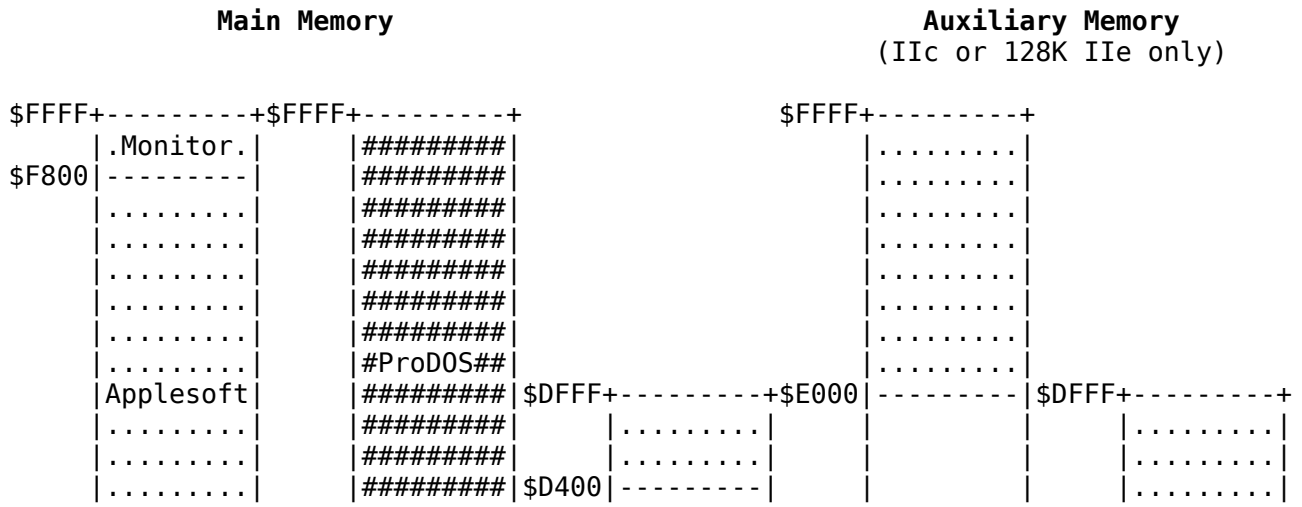
This appendix explains aspects of the BASIC system program (BASIC.SYSTEM) that are beyond the scope of the manual *BASIC Programming With ProDOS*. The primary subjects discussed in this appendix are

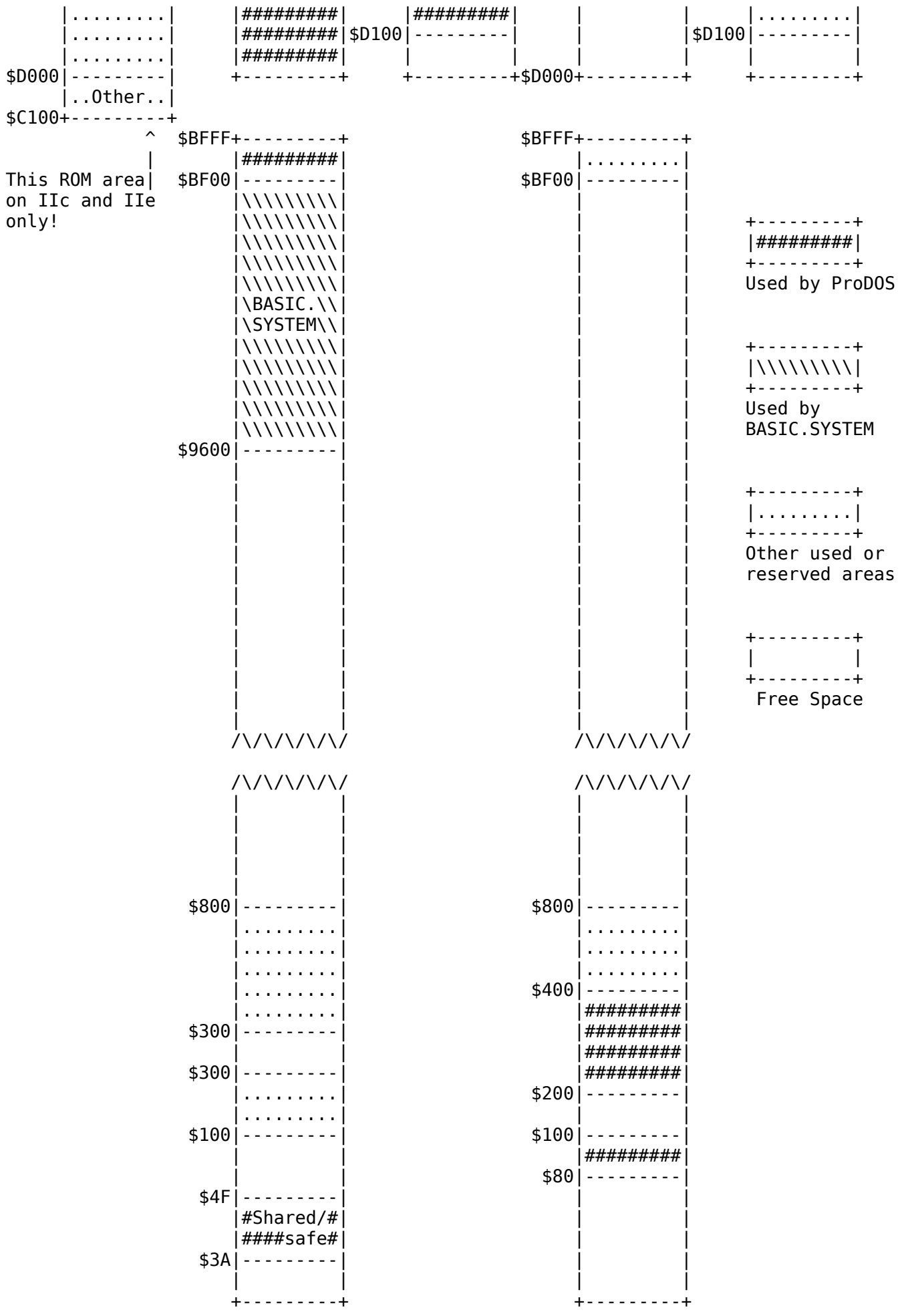
- how the BASIC system program uses memory
- how a machine-language program can make calls to the BASIC system program
- useful locations in the BASIC system program
- how you can add commands to the BASIC system program.

A.1 - Memory Map

The arrangement of ProDOS in memory is decided when the system is started up, and it depends on your particular system configuration. Figure A-1 shows the memory organization for an Apple IIe (64K or 128K) or Apple IIc (128K).

Figure A-1. Memory Map




$D000
..Other..
$C100
This ROM area on IIc and IIe only!
$BFFF
$BF00
$9600
\BASIC.\\
\SYSTEM\\
$800
$400
$300
$200
$100
$4F
#Shared/#
####safe#
$3A
$80
Used by ProDOS
Used by BASIC.SYSTEM
Other used or reserved areas
Free Space

A.2 - HIMEM

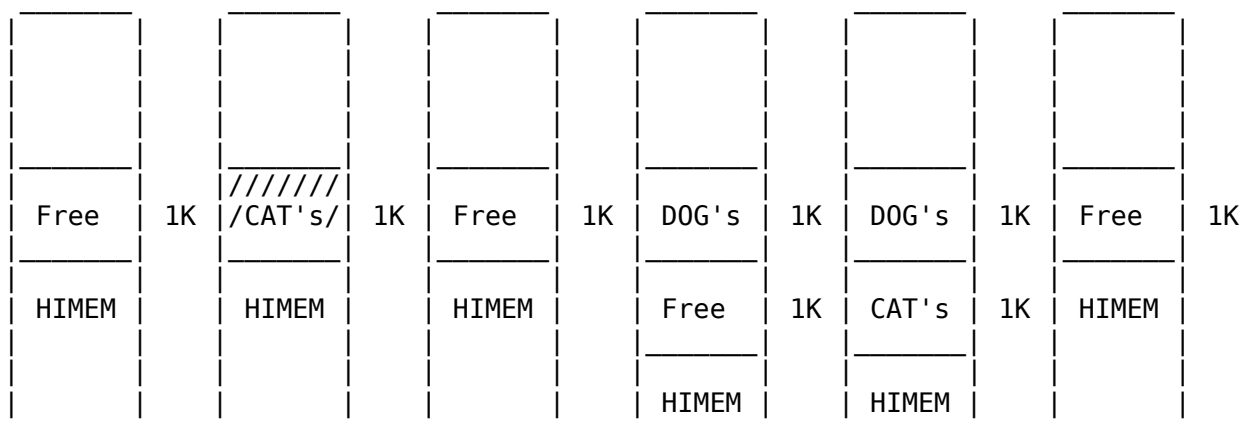
When ProDOS starts up the BASIC system program, it loads all the necessary programs and data into memory as shown in Figure A-1, leaves a 1K buffer on the highest available 1K boundary, and then sets HIMEM right below this buffer. This buffer is used as the file buffer for commands, such as CATALOG, that only need a temporary buffer. Table A-1 shows the possible settings of HIMEM, and the maximum number of bytes available to a program running under such a system configuration.

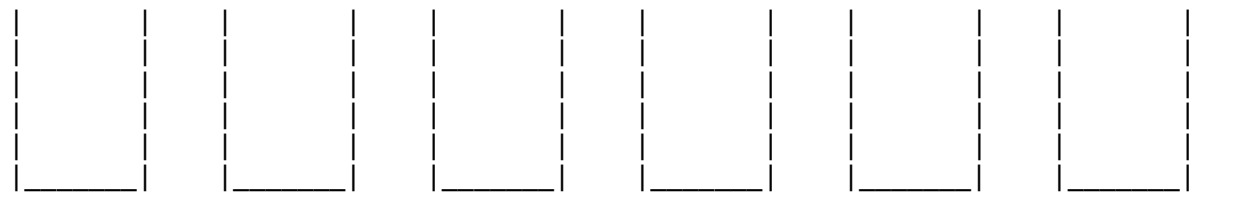
Table A-1. HIMEM and Program Workspace

System Configuration	HIMEM	Bytes Available to Programs
64K Applesoft in ROM	38400 ($9600)	36352 ($8E00)

These settings are in effect immediately after you boot the BASIC system program. While a program is running, however, these figures may change. Each time a file is opened, ProDOS lowers HIMEM by 1K ($400), keeping the 1K temporary command buffer immediately above it, and places a buffer for the file where the old temporary buffer was. When a file is closed, ProDOS releases the file's buffer, and raises HIMEM by 1K. Figure A-2 illustrates this process.

Figure A-2. The Movement of HIMEM





A.2.1 - Buffer Management

There are many times when you might want machine-language routines to coexist with ProDOS; for example, when using interrupt-driven devices, when using input/output devices that have no ROM, or when using commands that you have added to ProDOS.

BASIC.SYSTEM provides buffer management for file I/O. Those facilities can also be utilized from machine-language modules operating in the ProDOS/Applesoft environment to provide protected areas for code, data, and so on.

BASIC.SYSTEM resides from $9A00 upward, with a general-purpose buffer from $9600 (HIMEM) to $99FF. When a file is opened, BASIC.SYSTEM does garbage collection if needed, moves the general-purpose buffer down to $9200, and installs a file I/O buffer at $9600. When a second file is opened, the general-purpose buffer is moved down to $8E00 and a second file I/O buffer is installed at $9200. If an EXEC file is opened, it is always installed as the highest file I/O buffer at $9600, and all the other buffers are moved down. Additional regular file I/O buffers are installed by moving the general-purpose buffer down and installing it below the lowest file I/O buffer. All file I/O buffers, including the general-purpose buffer, are 1K (1024 bytes) and begin on a page boundary.

BASIC.SYSTEM may be called from machine language to allocate any number of pages (256 bytes) as a buffer, located above HIMEM and protected from Applesoft BASIC programs. The ProDOS bit map is not altered, so that files can be loaded into the area without an error from the ProDOS Kernel. If you subsequently alter the bit map to protect the area, you must mark the area as free when you are finished -- BASIC.SYSTEM will not do it for you.

To allocate a buffer, simply place the number of desired pages in the

accumulator and use **JSR GETBUFR** ($BEF5). If the carry flag returns clear, the allocation was successful and the accumulator will return the high byte of the buffer address. If the carry flag returns set, an error has occurred and the accumulator will return the error code. Note that the X and Y registers are not preserved.

The first buffer is installed as the highest buffer, just below BASIC.SYSTEM from $99FF downward, regardless of the number and type of file I/O buffers that are open. If a second allocation is requested, it is installed immediately below the first. Thus, it is possible to assemble code to run at known addresses-relocatable modules are not needed.

To de-allocate the buffers created by the above call and move the file buffers back up, just use **JSR FREEBUFR** ($BEF8). Although more than one buffer may be allocated by this call, they may not be selectively de-allocated.

Important!
All routines that are to be called by BASIC.SYSTEM should begin with the CLD instruction. This includes I/O routines accessed by **PR#** and **IN#** and clock/calendar routines. This allows ProDOS to spot accidental calls.

For tips on raising LOMEM to provide more memory for assembly-language routines, and protecting high-res graphics pages, see the *Applesoft BASIC Programmer's Reference Manual*.

A.3 - The BASIC Global Page

The BASIC system program has a specific area of memory, its global page, in which it keeps its current status. This page lies in the address range $BE00 through $BEFF (48640-48895). When BASIC.SYSTEM is active, its fields are defined as follows:

```
BE00:  CI.ENTRY   JMP WARMDOS     ;Reenter ProDOS/Applesoft
BE03:  DOSCMD     JMP SYNTAX      ;External entry for command string
BE06:  EXTRNCMD   JMP XRETURN     ;Called for added CMD syntaxing
BE09:  ERROUT     JMP ERROR       ;Handles ONERR or prints error
BE0C:  PRINTERR   JMP PRTERR      ;Prints error message
                                  ;Number is in accumulator
BE0F:  ERRCODE    DFB 0           ;ProDOS error code stored here
                                  ;and $DE for Applesoft
```

Default I/O vectors. These may be changed by the user to remap slots for nondisk devices. When the system is booted, all slots not containing a ROM are considered *not connected* and the default vector is left to point at the appropriate error handling routine.

```
BE10:  OUTVECT0  DW  COUT1       ;Monitor video output routine
BE12:  OUTVECT1  DW  NODEVERR    ;Default $C100 when ROM present
BE14:  OUTVECT2  DW  NODEVERR    ;Default $C200 when ROM present
BE16:  OUTVECT3  DW  NODEVERR    ;Default $C300 when ROM present
BE18:  OUTVECT4  DW  NODEVERR    ;Default $C400 when ROM present
BE1A:  OUTVECT5  DW  NODEVERR    ;Default $C500 when ROM present
BE1C:  OUTVECT6  DW  NODEVERR    ;Default $C600 when ROM present
BE1E:  OUTVECT7  DW  NODEVERR    ;Default $C700 when ROM present
BE20:  INVECT0   DW  CHIN1       ;Monitor keyboard input routine
BE22:  INVECT1   DW  NODEVERR    ;Default $C100 when ROM present
BE24:  INVECT2   DW  NODEVERR    ;Default $C200 when ROM present
BE26:  INVECT3   DW  NODEVERR    ;Default $C300 when ROM present
BE28:  INVECT4   DW  NODEVERR    ;Default $C400 when ROM present
BE2A:  INVECT5   DW  NODEVERR    ;Default $C500 when ROM present
BE2C:  INVECT6   DW  NODEVERR    ;Default $C600 when ROM present
BE2E:  INVECT7   DW  NODEVERR    ;Default $C700 when ROM present
BE30:  VECTOUT   DW  COUT1       ;Current character output routine
BE32:  VECTIN    DW  CHIN1       ;Current character input routine
BE34:  VDOSIO    DW  DOSOUT      ;ProDOS char out intercept routine
```



```
BE36:            DW  DOSINP      ;ProDOS char in intercept routine
BE38:  VSYSIO    DW  0,0         ;Internal redirection of I/O
BE3C:  DEFSLT    DFB $06         ;Default slot, set by 'S' parm
BE3D:  DEFDRV    DFB $01         ;Default drive, set by 'D' parm
BE3E:  PREGA     DFB 0           ;Register save area
BE3F:  PREGX     DFB 0
BE40:  PREGY     DFB 0
BE41:  DTRACE    DFB 0           ;Applesoft trace enable
BE42:  STATE     DFB 0           ;0=Imm, >0=Def modes
BE43:  EXACTV    DFB 0           ;EXEC file active if bit 7 on
BE44:  IFILACTV  DFB 0           ;Input file active if bit 7 on
BE45:  OFILACTV  DFB 0           ;Output file active if bit 7 on
BE46:  PFXACTV   DFB 0           ;Prefix input active if bit 7 on
BE47:  DIRFLG    DFB 0           ;File being accessed is directory
BE48:  EDIRFLG   DFB 0           ;End of directory encountered
BE49:  STRINGS   DFB 0           ;Counter for free string space
BE4A:  TBUFPTR   DFB 0           ;Temporory buffered char count (WRITE)
BE4B:  INPTR     DFB 0           ;Input char count during kbd input
BE4C:  CHRLAST   DFB 0           ;Last character output (for error detect)
BE4D:  OPENCNT   DFB $00         ;Number of open file (except EXEC file)
BE4E:  EXFILE    DFB $00         ;Flag to indicate EXEC file being closed
BE4F:  CATFLAG   DFB $00         ;File being input is (translated) dir
BE50:  XTRNADDR  DW  0           ;Execution address of external cmd (0)
BE52:  XLEN      DFB 0           ;Length of command string-1, ('HELP'=3)
BE53:  XCNUM     DFB 0           ;BASIC cmd number (external cmd if =0)
```

Command parameter **PBITS/FBITS** bit definitions:

```
BE54:  PFIX      EQU $80         ;Prefix needs fetching, pathname optional
BE54:  SLOT      EQU $40         ;No parameters to be processed
BE54:  RRUN      EQU $20         ;Command only valid during program
BE54:  FNOPT     EQU $10         ;Filename is optional
BE54:  CRFLG     EQU $08         ;CREATE allowed
BE54:  T         EQU $04         ;File type
BE54:  FN2       EQU $02         ;Filename '2' for RENAME
BE54:  FN1       EQU $01         ;Filename expected
```

And for **PBITS+1/FBITS+1** definitions:

```
BE54:  AD        EQU $80         ;Address
BE54:  B         EQU $40         ;Byte
BE54:  E         EQU $20         ;End address
BE54:  L         EQU $10         ;Length
BE54:  LINE      EQU $08         ;'@' line number
BE54:  SD        EQU $04         ;Slot and drive numbers
BE54:  F         EQU $02         ;Field
BE54:  R         EQU $01         ;Record
BE54:  V         EQU $00         ;Volume number ignored
```

When the BASIC system program recognizes one of its commands, it sets up **PBITS** to indicate which parameters (#S, #D, and so on) may be used with that command. Then it parses the command string, marking the found parameters in **FBITS**, and placing their values in locations $BE58-$BE6B. The meanings of the bit within **PBITS** and **FBITS** are discussed in the section "Adding Commands to the BASIC System Program."

```
BE54:  PBITS     DW  0           ;Allowed parameter bits
BE56:  FBITS     DW  0           ;Found parameter bits
```

The following locations hold the values of the parameters for the BASIC commands. As the BASIC system program parses command options, it sets the value of the corresponding command parameters. Previously set parameters do not change.

```
BE58:  PVALS     EQU *
BE58:  VADDR     DW  0           ;Parameter value for 'A' parm
BE5A:  VBYTE     DFB 0,0,0       ;Parameter value for 'B' parm
BE5D:  VENDA     DW  0           ;Parameter value for 'E' parm
BE5F:  VLNTH     DW  0           ;Parameter value for 'L' parm
BE61:  VSLOT     DFB 0           ;Parameter value for 'S' parm
BE62:  VDRIV     DFB 0           ;Parameter value for 'D' parm
BE63:  VFELD     DW  0           ;Parameter value for 'F' parm
BE65:  VRECD     DW  0           ;Parameter value for 'R' parm
BE67:  VVOLM     DFB 0           ;Parameter value for 'V' parm
BE68:  VLINE     DW  0           ;Parameter value for '@' parm
BE6A:  PTYPE     EQU *-PVALS
BE6A:  VTYPE     DFB 0           ;Parameter value for 'T' parm
```

```
BE6B:  PIOSLT    EQU *-PVALS
BE6B:  VIOSLT    DFB 0           ;Parameter value for IN# or PR#
BE6C:  VPATH1    DW  TXBUF-1     ;Pathname 1 buffer
BE6E:  VPATH2    DW  TXBUF2      ;Pathname 2 buffer (RENAME)
```

`GOSYSTEM` is used to make all MLI calls since errors must be translated before returning to the calling routine. On entry the Accumulator should contain the call number. The address of the parameter table is looked up and set based on the call number. Only file management calls can be made using this routine: $C0-$D3. The original implementation of this BASIC system program contains only these calls.

```
BE70:  GOSYSTEM  STA SYSCALL     ;Save call number
BE73:            STX CALLX       ;Preserve X register
BE76:            AND #$1F        ;Strip high bits of call number
BE78:            TAX             ; and use as lookup index
BE79:            LDA SYSCTBL,X   ;Get low address of parm table
BE7C:            STA SYSPARM
BE7F:            LDX CALLX       ;Restore X before calling
BE82:            JSR MLIENTRY    ;Call ProDOS MLI to execute request
BE85:  SYSCALL   DFB 0
BE86:  SYSPARM   DW  *           ;(High address should be same
                                 ; as parameter tables)
BE88:            BCS BADCALL     ;Branch if error encountered
BE8A:            RTS
```

`BADCALL` converts MLI errors into BASIC system program error equivalents. Routines should be entered with error number in the Accumulator. The **`BADCALL`** routine should be used whenever a ProDOS MLI call returns an error and BASIC.SYSTEM will be used to print the error message. Returns BASIC system program error number in Accumulator. All unrecognized errors are mapped to I/O error. X register is restored to its value before the call is made. Carry is set.

```
BE8B:  BADCALL   LDA #12         ;19 errors are mapped to
BE8D:  MLIERR1   CMP MLIERTBL,X  ; other than I/O error
BE90:            BEQ MLIERR2
BE92:            DEX
BE93:            BPL MLIERR1
BE95:            LDX #$13        ;If not recognized, make it I/O error
BE97:  MLIERR2   LDA BIERRTBL,X  ;return error in Accumulator
BE9A:            LDX CALLX       ;Restore X register
BE9D:            SEC             ;Set Carry to indicate error
BE9E:  XRETURN   RTS
BE9F:  CISPARE1  DFB $00
```

The following are the system-call parameter tables. These tables

must reside within the same page of memory. Only those parameters that are subject to alterations have been labeled. **SYSCTBL** below contains the low-order addresses of each parameter table. **SYSCTBL** is used by **GOSYSTEM** to set up the address of the parameter table for each call. (See **GOSYSTEM**.)

```
BEA0:  SCREATE   DFB $07
BEA1:            DW  TXBUF-1     ;Pointer to pathname
BEA3:  CRACESS   DFB $C3         ;$C1 if directory create
BEA4:  CRFILID   DFB $00
BEA5:  CRAUXID   DW  $0000
BEA7:  CRFKIND   DFB 0
BEA8:            DW  0           ;No predetermined date/time
BEAA:            DW  0
BEAC:  SSGPRFX   EQU *
BEAC:  SDSTROY   DFB $01
BEAD:            DW  TXBUF-1     ;This call requires no modifications
BEAF:  SRECNAME  DFB $02
BEB0:            DW  TXBUF-1     ;No modifications needed
BEB2:            DW  TXBUF2
BEB4:  SSGINFO   DFB $00         ;P.CNT=7 if SET_FILE_INFO
                                 ;P.CNT=A if GET_FILE_INFO
BEB5:            DW  TXBUF-1
BEB7:  FIACESS   DFB $00         ;Access used by lock/unlock
BEB8:  FIFILID   DFB $00         ;FILE ID is type specifier
BEB9:  FIAUXID   DW  $0000       ;Aux_id is used for load addr
                                 ; and record length
BEBB:  FIFKIND   DFB $00         ;Identifies trees vs. directories
BEBC:  FIBLOKS   DW  $0000       ;Used by CAT commands for root dir
BEBE:  FIMDATE   DW  $0000       ;Modification date & time
BEC0:            DW  $0000       ;should always be zeroed before call
BEC2:            DW  $0000       ;Create date and time ignored
BEC4:            DW  $0000
```

```
BEC6:  SONLINE   EQU *
BEC6:  SSETMRK   EQU *
BEC6:  SGETMRK   EQU *
BEC6:  SSETEOF   EQU *
BEC6:  SGETEOF   EQU *
BEC6:  SSETBUF   EQU *
BEC6:  SGETBUF   EQU *
BEC6:            DFB $02         ;Parameter count
BEC7:  SBUFREF   EQU *
BEC7:  SREFNUM   EQU *
BEC7:  SUNITNUM  EQU *
BEC7:            DFB 0           ;Unit or reference number
BEC8:  SDATPTR   EQU *
BEC8:  SMARK     EQU *
BEC8:  SEOF      EQU *
BEC8:  SBUFADR   EQU *
BEC8:            DFB 0,0,0       ;Some calls only use 2 bytes
                                 ;MRK & EOF use 3 bytes
BECB:  SOPEN     DFB $03
BECC:            DW  TXBUF-1
BECE:  OSYSBUF   DW  $0000
BED0:  OREFNUM   DFB 0
BED1:  SNEWLIN   DFB $03
```

```
BED2:  NEWLREF   DFB $00         ;Reference number
BED3:  NLINEBL   DFB $7F         ;Newline character is always CR
BED4:            DFB $0D         ; both $0D and $8D are recognized
BED5:  SREAD     EQU *
BED5:  SWRITE    EQU *
BED5:            DFB $04
BED6:  RWREFNUM  DFB $00
BED7:  RWDATA    DW  $0000       ;Pointer to data to be read/written
BED9:  RWCOUNT   DW  $0000       ;Number of bytes to be read/written
BEDB:  RWTRANS   DW  $0000       ;returned # of bytes read/written
```

```
BEDD:  SCLOSE    EQU *
BEDD:  SFLUSH    EQU *
BEDD:            DFB $01
BEDE:  CFREFNUM  DFB $00
BEDF:  CCCSPARE  DFB $00
BEE0:            ASC 'COPYRIGHT APPLE, 1983'
BEF5:  GETBUFR   JMP GETPAGES
BEF8:  FREBUFR   JMP FREPAGES
BEF8:  RSHIMEM   DFB 0, 0, 0, 0, 0
```

A.3.1 - BASIC.SYSTEM Commands From Assembly Language

There are times when a routine wants to perform functions that are already implemented by the BASIC system program -- deleting and renaming files, displaying a directory, and so on. The **DOSCMD** vector serves just this function.

First a routine should place the desired BASIC command in the input buffer ($200). It should be an ASCII string with the high bits set, followed by a carriage return ($8D), exactly as the Monitor **GETLN** routine would leave a string. Next the routine should do a JSR to the **DOSCMD** entry point ($BE03).

BASIC.SYSTEM will parse the command, set up all the parameters, (as explained in Section A.3.3), and then execute the command. If there is an error, it will return the error code in the accumulator with the carry set. If it is 0, there was no error. Otherwise it contains a BASIC system program error number.

Note: The **JSR DOSCMD** must be executed in deferred mode (from a BASIC program), rather than in immediate mode. This applies also to the Monitor program: from the Monitor, you can't do a **$xxxxG** to execute the code that contains the **JSR DOSCMD**. This is because BASIC.SYSTEM checks certain state flags, which are set correctly only while in deferred mode.

There are certain commands that do *not* work as expected when initiated via **DOSCMD**: RUN -(dash command), LOAD, CHAIN, READ, WRITE, APPEND, and EXEC. Use them this way at your own risk.

The commands that *do* work correctly are: CATALOG, CAT, PREFIX, CREATE, RENAME, DELETE, LOCK, UNLOCK, SAVE, STORE, RESTORE, PR#, IN#, FRE, OPEN, CLOSE, FLUSH, POSITION, BRUN, BLOAD, and BSAVE.

The following are:

1. An example of a BASIC program that uses the BLOAD command to load an assembly-language routine that exercises the **DOSCMD** routine.
2. A listing of that assembly-language routine.

You should review them before writing your own routine.

```
 10 REM YOU MUST CALL THE ROUTINE FROM INSIDE A BASIC PROGRAM
 11 REM
 12 REM
 20 PRINT CHR$(4)"BLOAD/P/PROGRAMS/CMD.0"
 30 CALL 4096
 40 PRINT "BACK TO THE WONDERFUL WORLD OF BASIC!"
 50 END
```

```
 1000:         1000    1           ORG   $1000
 1000:         FD6F    2 GETLN1    EQU   $FD6F         ; MONITORS INPUT ROUTINE
 1000:         BE03    3 DOSCMD    EQU   $BE03         ; BASIC.SYSTEM GLBL PG DOS CMD
ENTRY
 1000:         FDED    4 COUT      EQU   $FDED         ; MONITORS CHAR OUT ROUTINE
 1000:         BE0C    5 PRERR     EQU   $BE0C         ; PRINT THE ERROR
 1000:                 6 *
 1000:                 7 *
 1000:                 8 *
 1000:A2 00            9 START     LDX   #0            ; DISPLAY PROMPT...
 1002:BD 1F 10        10 L1        LDA   PROMPT,X      ;
 1005:F0 06   100D    11           BEQ   CONT          ; BRANCH IF END OF STRING
 1007:20 ED FD        12           JSR   COUT          ;
 100A:E8              13           INX                 ;
 100B:D0 F5   1002    14           BNE   L1            ; LOOP UNTIL NULL TERMINATOR HIT
 100D:                15 *
 100D:20 6F FD        16 CONT      JSR   GETLN1        ; ACCEPT COMMAND FROM KB
 1010:20 03 BE        17           JSR   DOSCMD        ; AND EXECUTE COMMAND
 1013:2C 10 C0        18           BIT   $C010         ; CLEAR STROBE
 1016:B0 02   101A    19           BCS   ERROR         ; BRANCH IF ERROR DETECTED
 1018:90 E6   1000    20           BCC   START         ; OTHERWISE RESTART
 101A:                21 *
 101A:                22 *
```

```
101A:               23 * NOTE: AFTER HANDLING YOUR ERROR YOU MUST CLEAR THE CARRY
101A:               24 *       BEFORE RETURNING TO BASIC OR BASIC WILL DO
101A:               25 *       STRANGE TO YOU.
101A:               26 *
101A:20 0C BE       27 ERROR     JSR   PRERR         ; PRINT 'ERR'
101D:18             28           CLC                ;
101E:60             29           RTS                ; RETURN TO BASIC
101F:               30 *
101F:               31           MSB   ON
101F:               32 *
101F:8D             33 PROMPT    DB    $8D          ; OUTPUT A RETURN FIRST
1020:C5 CE D4 C5    34           ASC   'ENTER       BASIC.SYSTEM COMMAND --> '
103F:00             35           DB    0
```

DOSCMD is merely a way to perform some BASIC.SYSTEM commands from assembly language, and is not a substitute for performing the commands in BASIC. Keep in mind the consequences of the command you are executing. For example, when doing a BRUN or BLOAD, make sure the code is loaded at proper addresses.

After you call **DOSCMD**, the carry bit will be set if an error has occurred. The accumulator will have the error number.

There are three ways to handle **DOSCMD** errors:

- Do a **JSR ERROUT** ($BE09). This returns control to your **BASIC ONERR** routine, where you can handle the error.
- Do a **JSR PRINTERR** ($BE0C). This prints Out the error and returns control to the point just after the **JSR**.
- Handle the error yourself. Be sure to clear the carry (CLC) before returning control to BASIC.SYSTEM. If you don't, an error will be assumed, and the results are unpredictable.

A.3.2 - Adding Commands to the BASIC System Program

The **EXTRNCMD** location in the global page allows you to add your own commands to the ProDOS command set. Once you attach a command, it is treated as if it were one of the BASIC.SYSTEM commands, except that the original commands have preference. To execute your command in immediate mode, just enter it. To execute it in deferred mode, preface it with **PRINT CHR$(4)**.

Whenever BASIC.SYSTEM receives a command, it first checks its command list for a match. If the command is not recognized, BASIC.SYSTEM sends the command to the external command handlers, if any are connected. If no external command handler claims the

command, BASIC.SYSTEM passes control to Applesoft, which returns an error if the command is not recognized.

If you have frequent need for special commands, you can write your own command handler and attach it to BASIC.SYSTEM through the EXTRNCMD jump vector. First, save the current EXTRNCMD vector (to JMP to if the command is not yours), and install the address of your routine in EXTRNCMD+1 and +2 (low byte first). Your routine must do three things:

- It must check for the presence of your command(s) by inpecting the GETLN input buffer. If the command is not yours, you must set the carry (SEC) and JMP to the initial EXTRNCMD vector you saved to continue the search.
- If the command is yours, you must zero XCNUM ($BE53) to indicate an external command, and set XLEN ($BE52) equal to the length of your command string minus one.

  If there are no associated parameters (such as slot, drive, A$, and so on) to parse, or if you're going to parse them yourself, you must set all 16 parameter bits in PBITS ($BE54,$BE55) to zero. And, if you're going to handle everything yourself before returning control to BASIC.SYSTEM, you must point XTRNADDR ($BE50,$BE51) at an RTS instruction. XRETURN ($BE9E) is a good location. Now, just fall through to your execution routines.

  If there are parameters to parse, it is easiest to let BASIC.SYSTEM parse them for you (unless you want to use some undefined parameters). By setting up the bits in PBITS ($BE54,$BE55), and setting XTRNADDR ($BE50,$BE51) equal to the location where execution of your command begins, you can return control to BASIC.SYSTEM, with an RTS, and let it parse and verify the parameters and return them to you in the global page.

- It must execute the instructions expected of the command, and it should RTS with the carry cleared.

Note: Having BASIC.SYSTEM parse your external command parameters was initially intended only for its own use. As it happens, not all parameters can be parsed separately. The low byte of PBITS ($BE54) must have a nonzero value to have BASIC.SYSTEM parse parameters. This means that regardless of the parameters you need parsed, you

must also elect to parse some parameter specified by the low byte of PBITS. For example, set PBITS to $10, filename optional (this parameter need not be known by the user).

The following are two sample routines, BEEP and BEEPSLOT. They can reside together as external commands. BEEP handles everything itself, while BEEPSLOT lets you pass a slot and drive parameter (,S#,D#) where the drive is ignored.

A.3.2.1 - BEEP Example

```
*************************************************************
*                                                           *
*  BRUN BEEP.0 TO INSTALL THE ROUTINE'S ADDRESS IN EXTRNCMD. *
*  THEN TYPE BEEP AS AN IMMEDIATE COMMAND OR USE PRINT      *
*  CHR$(4);"BEEP" IN A PROGRAM.                             *
*                                                           *
*************************************************************
*
*
            ORG  $300
INBUF       EQU  $200     ;GETLN input buffer.
WAIT        EQU  $FCA8    ;Monitor wait routine.
BELL        EQU  $FF3A    ;Monitor bell routine.
EXTRNCMD    EQU  $BE06    ;External cmd JMP vector.
XTRNADDR    EQU  $BE50    ;Ext cmd implementation addr.
XLEN        EQU  $BE52    ;length of command string-1.
XCNUM       EQU  $BE53    ;CI cmd no. (ext cmd - 0).
PBITS       EQU  $BE54    ;Command parameter bits.
XRETURN     EQU  $BE9E    ;Known RTS instruction.
            MSB  ON       ;Set high bit on ASCII
*
* FIRST SAVE THE EXTERNAL COMMAND ADDRESS SO YOU WON'T
* DISCONNECT ANY PREVIOUSLY CONNECTED COMMAND.
*
            LDA  EXTRNCMD+1
            STA  NXTCMD
            LDA  EXTRNCMD+2
            STA  NXTCMD+1
*
            LDA  #>BEEP      ;Install the address of our
            STA  EXTRNCMD+1  ; command handler in the
            LDA  #<BEEP      ; external command JMP
            STA  EXTRNCMD+2  ; vector.
            RTS
*
BEEP        LDX  #0          ;Check for our command.
NXTCHR      LDA  INBUF,X     ;Get first character.
            CMP  CMD,X       ;Does it match?
            BNE  NOTOURS     ;No, back to CI.
            INX              ;Next character
            CPX  #CMDLEN     ;All characters yet?
            BNE  NXTCHR      ;No, read next one.
*
```

```
            LDA  #CMDLEN-1   ;Our cmd! Put cmd length-1
            STA  XLEN        ; in CI global XLEN.
            LDA  #>XRETURN   ;Point XTRNADDR to a known
            STA  XTRNADDR    ; RTS since we'll handle
            LDA  #<XRETURN   ; at the time we intercept
```



```
            STA  XTRNADDR+1  ; our command.
            LDA  #0          ;Mark the cmd number as
            STA  XCNUM       ; zero (external).
            STA  PBITS       ;And indicate no parameters
            STA  PBITS+1     ; to be parsed.
*
            LDX  #5          ;Number of desired beeps.
NXTBEEP     JSR  BELL        ;Else, beep once.
            LDA  #$80        ;Set up the delay
            JSR  WAIT        ; and wait.
            DEX              ;Decrement index and
            BNE  NXTBEEP     ; repeat until X = 0.
*
            CLC              ;All done successfully.
            RTS              ; RETURN WITH THE CARRY CLEAR.
*
NOTOURS     SEC              ; ALWAYS SET CARRY IF NOT YOUR
            JMP  (NXTCMD)    ; CMD AND LET NEXT COMMAND TRY
*                            ; TO CLAIM IT.
CMD         ASC  "BEEP"      ;Our command
CMDLEN      EQU  *-CMD       ;Our command length
*
NXTCMD      DW   0           ; STORE THE NEXT EXT CMD'S
                             ; ADDRESS HERE.
```


A.3.2.2 - BEEPSLOT Example

```
*************************************************************
*                                                           *
* BRUN BEEPSLOT.0 TO INSTALL THE ROUTINE'S ADDRESS IN       *
* EXTRNCMD.  THEN ENTER BEEPSLOT,S(n),D(n).  ONLY A LEGAL   *
* SLOT AND DRIVE NUMBERS ARE ACCEPTABLE.  IF NO SLOT NUMBER *
* IT WILL USE THE DEFAULT SLOT NUMBER.  ANY DRIVE NUMBER IS *
* SIMPLY IGNORED.  THE COMMAND MAY ALSO BE USED IN A        *
* PROGRAM PRINT CHR$(4) STATEMENT.                          *
*                                                           *
*************************************************************
*
*
            ORG  $2000
INBUF       EQU  $200       ;GETLN input buffer.
WAIT        EQU  $FCA8      ;Monitor wait routine.
BELL        EQU  $FF3A      ;Monitor bell routine
EXTRNCMD    EQU  $BE06      ;External cmd JMP vector.
XTRNADDR    EQU  $BE50      ;Ext cmd implementation addr.
XLEN        EQU  $BE52      ;Length of command string-1.
XCNUM       EQU  $BE53      ;CI cmd no. (ext cmd = 0).
PBITS       EQU  $BE54      ;Command parameter bits.
```

```
VSLOT      EQU  $BE61      ;Verified slot parameter.
           MSB  ON         ;Set high bit on ASCII.
*
* REMEMBER TO SAVE THE PREVIOUS COMMAND ADDRESS.
*
           LDA  EXTRNCMD+1
           STA  NXTCMD
           LDA  EXTRNCMD+2
           STA  NXTCMD+1
*
           LDA  #>BEEPSLOT ;Install the address of our
           STA  EXTRNCMD+1 ; command handler in the
           LDA  #<BEEPSLOT ; external command JMP
           STA  EXTRNCMD+2 ; vector.
           RTS
*
BEEPSLOT   LDX  #0         ;Check for our command.
NXTCHR     LDA  INBUF,X    ;Get first character.
           CMP  CMD,X      ;Does it match?
           BNE  NOTOURS    ;NO, SO CONTINUE WITH NEXT CMD.
           INX             ;Next character
           CPX  #CMDLEN    ;All characters yet?
           BNE  NXTCHR     ;No, read next one.
*
           LDA  #CMDLEN-1  ;Our cmd! Put cmd length-1
           STA  XLEN       ; in CI global XLEN.
           LDA  #>EXECUTE  ;Point XTRNADDR to our
```

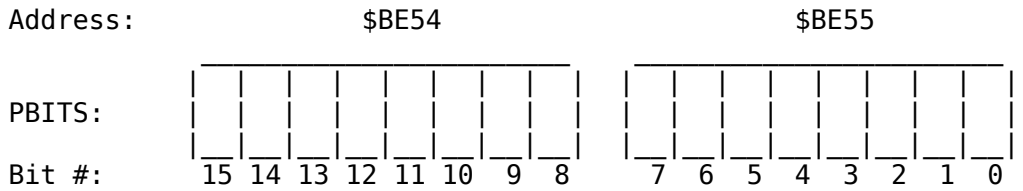


```
           STA  XTRNADDR   ; command execution
           LDA  #<EXECUTE  ; routine
           STA  XTRNADDR+1
           LDA  #0         ;Mark the cmd number as
           STA  XCNUM      ; zero (external).
*
           LDA  #%00010000 ;Set at least one bit
           STA  PBITS      ; in PBITS low byte!
*
           LDA  #%00000100 ;And mark PBITS high byte
           STA  PBITS+1    ; that slot & drive are legal.
           CLC             ;Everything is OK.
           RTS             ;Return to BASIC.SYSTEM
*
EXECUTE    LDA  VSLOT      ;Get slot parameter.
           TAX             ;Transfer to index reg.
NXTBEEP    JSR  BELL       ;Else, beep once.
           LDA  #$80       ;Set up the delay
           JSR  WAIT       ; and wait.
           DEX             ;decrement index and
           BNE  NXTBEEP    ; repeat until x = 0.
           CLC             ;All done successfully.
           RTS             ;Back to BASIC.SYSTEM.
*
* IT'S NOT OUR COMMAND SO MAKE SURE YOU LET BASIC
* CHECK WHETER OR NOT IT'S THE NEXT COMMAND.
*
NOTOURS    SEC             ;SET CARRY AND LET
           JMP  (NXTCMD)   ; NEXT EXT CMD GO FOR IT.
*
CMD        ASC  "BEEPSLOT" ;Our command
```

```
CMDLEN     EQU  *-CMD      ;Our command length
NXTCMD     DW   0          ; STORE THE NEXT COMMAND'S
                           ; ADDRESS HERE.
```


A.3.3 - Command String Parsing

First, the external command must tell the BASIC system program which parameters are allowed for the command. It does this by assigning the appropriate values to the two **PBITS** bytes, which have the following meanings:



Bit # - Meaning

15 - Prefix needs fetching. Pathname is optional
14 - No parameters to be processed
13 - Command only valid during program execution
12 - Filename is optional
11 - Create allowed if file doesn't exist
10 - File type (Ttype) optional
9 - A second filename expected
8 - A first filename expected
7 - Address (A#) allowed
6 - Byte (B#) allowed
5 - End address (E#) allowed
4 - Length (L#) allowed
3 - Line number (@#) allowed
2 - Slot and Drive (S# and D#) allowed
1 - Field (F#) allowed
0 - Record (R#) allowed

Having done this, the routine should place the length of the recognized command word minus one into **XLEN** ($BE52). It should also place a $00 into **XCNUM** ($BE53), indicating that an external command was found, and it should place the address within the routine at which further processing of the parsed command will take place into **XTRNADDR** ($BE50). Then it should RTS back to the BASIC system program.

The BASIC system program will see that the command was recognized, and it will parse the string according to **PBITS**. For each parameter that was used in the command, it will set the corresponding bit in **FBITS** ($BE56) and update the value of that parameter in the global page. Finally, it will do a JSR to the location indicated in **XTRNADDR** ($BE50).

The routine can now process the command. All parameters are stored in the global page except the filenames which are stored in the locations indicated by **VPATH1** and **VPATH2**.

The HELP command is such a routine. When you type **-HELP**, the help command is loaded into memory at $2000, it moves HIMEM down and places itself above HIMEM, then it marks itself in the bit map. Finally it places the start address of the routine in the **EXTRNCMD** vector. The BASIC system program now *recognizes* a series of HELP commands as well as the NOHELP command.

The NOHELP command removes the help routine's address from the **EXTRNCMD** vector, unmarks the routine from the bit map, and moves HIMEM back up.

A.4 - Zero Page

Figure A-3 is a memory map that shows the locations used by the Monitor, Applesoft, the Device Drivers, and the ProDOS MLI. The owner of each location is shown by a letter: M, A, D, or P.

Figure A-3. Zero Page Memory Map

Use by the Monitor (M), Applesoft (A), Disk Drivers (D), and ProDOS MLI (P) is shown.

Decimal	Hex	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
0	$00	DA	DA	A	A	A	A					A	A	A	A	A	A
16	$10	A	A	A	A	A	A	A	A	A							A
32	$20	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
48	$30	M	M	M	M	M	M	M	M	M	M	PMD	PMD	PMD	PMD	PMD	DM
64	$40	PMD	PMD	PMD	PMD	PMD	PMD	PMD	PM	PM	PM	P	P	P	P	PM	M
80	$50	MA	MA	MA	MA	MA	MA	A	A	A	A	A	A	A	A	A	A
96	$60	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

```
112 $70   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A
128 $80   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A
144 $90   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A
160 $A0   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A
176 $B0   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A
192 $C0   A   A   A   A   A   A   A   A   A   A   A   A   A   A
208 $D0   A   A   A   A   A   A           A   A   A   A   A   A   A   A
224 $E0   A   A   A       A   A   A   A   A   A   A
240 $F0   A   A   A   A   A   A   A   A   A   A
```

If you need many zero-page locations for your routines, choose a region of already-used locations, save them at the beginning of the routine, and then restore them at the end.

A.5 - The Extended 80-Column Text Card

The Apple IIe computer can optionally contain an Extended 80-Column Text Card, giving the computer access to an additional 64K of RAM. (The Apple IIc has the equivalent of such a card built in.) ProDOS uses this extra RAM as a volume, just like a small disk volume. This volume is initially given the name /RAM, but it can be renamed.

The 64K of RAM on the card is logically partitioned into 127 512-byte blocks of information. The contents of these blocks are:

Blocks 00-01 - Unavailable
Block 02 - Volume directory
Block 03 - Volume bit map
Blocks 04-07 - Unavailable
Blocks 08-126 - Directories and files

A detailed description of the way these blocks are used on a disk volume is in Appendix B. The major differences between a disk volume and /RAM are:

- On a disk volume, blocks 0 and 1 are used for the loader program. Since /RAM is not a bootable volume, these blocks are not used.
- On a disk volume, there are usually four blocks reserved for the volume directory, with a maximum capacity of 51 files in the volume directory. On /RAM, there is only one block of volume directory: it can hold 12 files (any or all of them can be subdirectory files).
- Normal disk devices are associated with a given slot and drive. /RAM is placed in the device list as slot 3, drive 2.

This arrangement gives you a total of 119 blocks of file storage.

Appendix B
File Organization

This appendix contains a detailed description of the way that ProDOS stores files on disks. For most system program applications, the MLI insulates you from this level of detail. However, you must use this information if you want

- to list the files in a directory
- to copy a sparse file without increasing the file's size
- to compare two sparse files.

This appendix first explains the organization of information on volumes. Next, it shows the storage of volume directories, directories, and the various stages of standard files. Finally it presents a set of diagrams that summarize all the material in this appendix. You can refer to these diagrams as you read the appendix. They will become your most valuable tool for working with file organization.

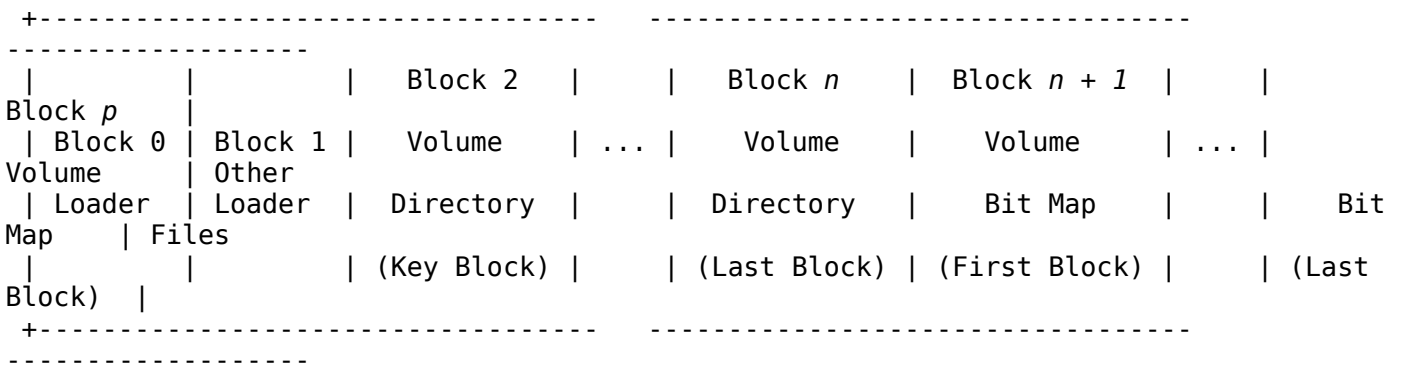
B.1 - Format of Information on a Volume

When a volume is formatted for use with ProDOS, its surface is partitioned into an array of tracks and sectors. In accessing a volume, ProDOS requests not a track and sector, but a logical block from the device corresponding to that volume. That device's driver translates the requested block number into the proper track and sector number; the physical location of information on a volume is unimportant to ProDOS and to a system program that uses ProDOS. This appendix discusses the organization of information on a volume in terms of logical blocks, numbered starting with zero, not tracks and sectors.

When the volume is formatted, information needed by ProDOS is

placed in specific logical blocks. A **loader program** is placed in blocks 0 and 1 of the volume. This program enables ProDOS to be booted from the volume. Block 2 of the volume is the **key block** (the first block) of the **volume directory file**; it contains descriptions of (and pointers to) all the files in the volume directory. The volume directory occupies a number of consecutive blocks, typically four, and 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 subdirectory file information, standard file information, or are empty. The first blocks of a volume look something like Figure B-1.

Figure B-1. Blocks on a Volume

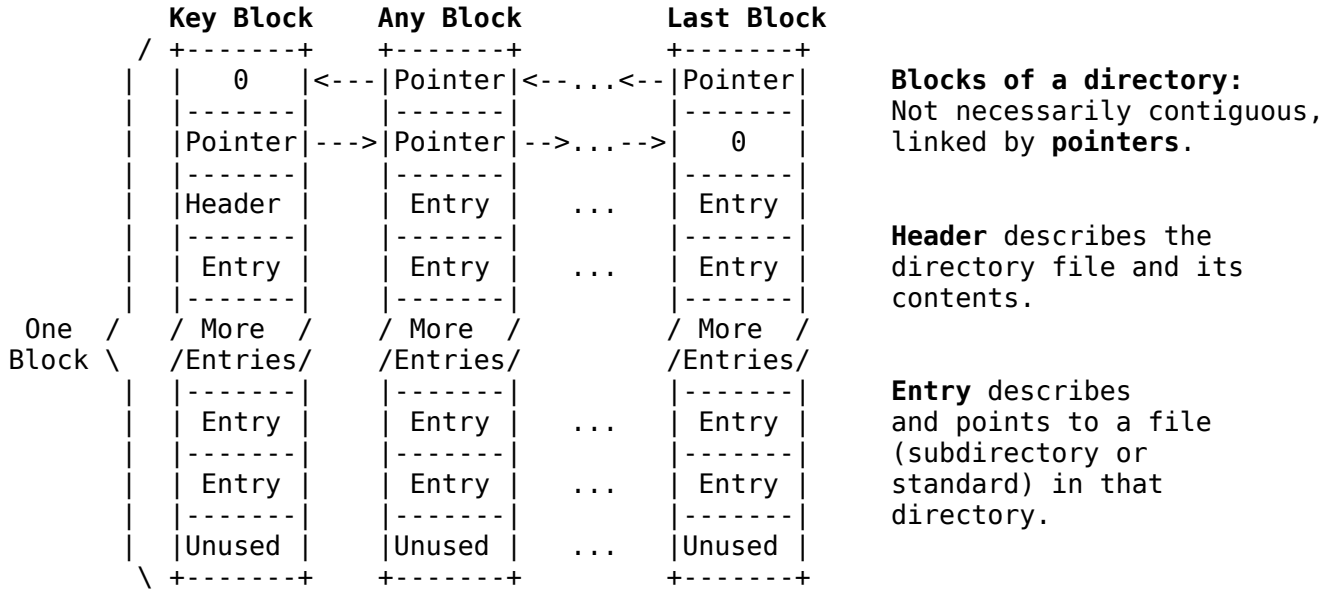


The precise format of the volume directory, volume bit map, subdirectory files and standard files are explained in the following sections.

B.2 - Format of Directory Files

The format of the information contained in volume directory and subdirectory files is quite similar. Each consists of a **key block** followed by zero or more blocks of additional directory information. The fields in a directory's key block are: a pointer to the next block in the drectory; a header entry that describes the directory; a number of file entries describing, and pointing to, the files in that directory; and zero or more unused bytes. The fields in subsequent (non-key) blocks in a directory are: a number of entries describing, and pointing to, the files in that directory; and zero or more unused bytes. The format of a directory file is represented in Figure B-2.

Figure B-2. Directory File Format



The header entry is the same length as all other entries. The only organizational difference between a volume directory file and a subdirectory file is in the header.

See the sections "Volume Directory Headers" and "Subdirectory Headers."

B.2.1 Pointer Fields

The first four bytes of each block used by a directory file contain pointers to the preceding and succeeding blocks in the directory file, respectively. 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.

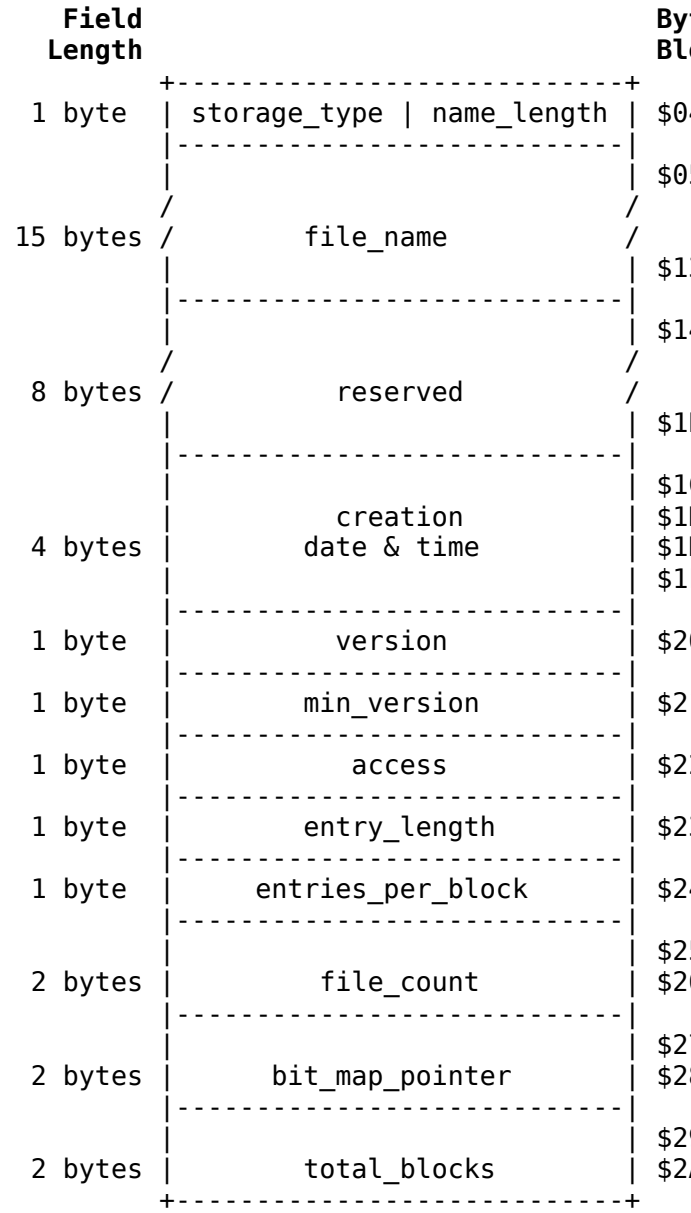
By the Way: All block pointers used by ProDOS have the same format: low byte first, high byte second.

B.2.2 - Volume Directory Headers

Block 2 of a volume is the key block of that volume's directory file. The volume directory header is at byte position $0004 of the key block, immediately following the block's two pointers. Thirteen fields are

currently defined to be in a volume directory header: they contain all the vital information about that volume. Figure B-3 illustrates the structure of a volume directory header. Following Figure B-3 is a description of each of its fields.

Figure B-3. The Volume Directory Header



storage_type and **name_length** (1 byte): Two four-bit fields are packed into this byte. A value of $F in the high four bits (the storage_type) identifies the current block as the key block of a volume directory file. The low four bits contain the length of the volume's

name (see the file_name field, below). The name_length can be changed by a RENAME call.

file_name (15 bytes): The first n bytes of this field, where n is specified by name_length, contain the volume's name. This name must conform to the filename (volume name) syntax explained in Chapter 2. 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): Reserved for future expansion of the file system.

creation (4 bytes): The date and time at which this volume was initialized. The format of these bytes is described in Section B.4.2.2.

version (1 byte): The version number of ProDOS under which this volume was initialized. This byte allows newer versions of ProDOS to determine the format of the volume, and adjust their directory interpretation to conform to older volume formats. In ProDOS 1.0, version = 0.

min_version: Reserved for future use. In ProDOS 1.0, it is 0.

access (1 byte): Determines whether this volume directory can be read written, destroyed, and renamed. The format of this field is described in Section B.4.2.3.

entry_length (1 byte): The length in bytes of each entry in this directory. The volume directory header itself is of this length. entry_length = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. entries_per_block = $0D.

file_count (2 bytes): The number of active file entries in this directory file. An active file is one whose storage_type is not 0. See Section B.2.4 for a description of file entries.

bit_map_pointer (2 bytes): 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 using the total_blocks field, 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. If the number of blocks used by all files on the volume is not the same as the number recorded in the bit map, the directory structure of the volume has been damaged.

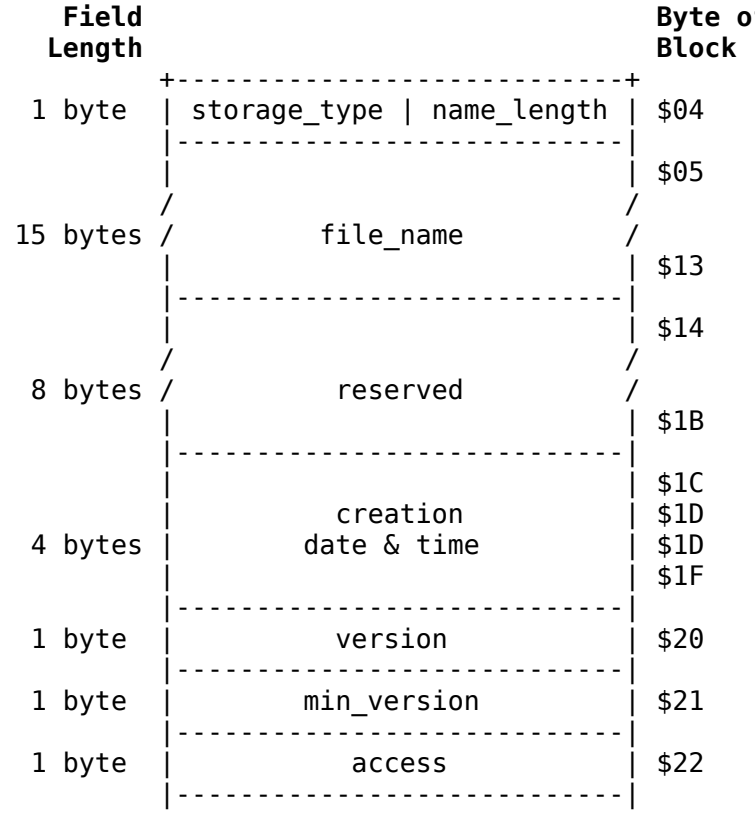
total_blocks (2 bytes): The total number of blocks on the volume.

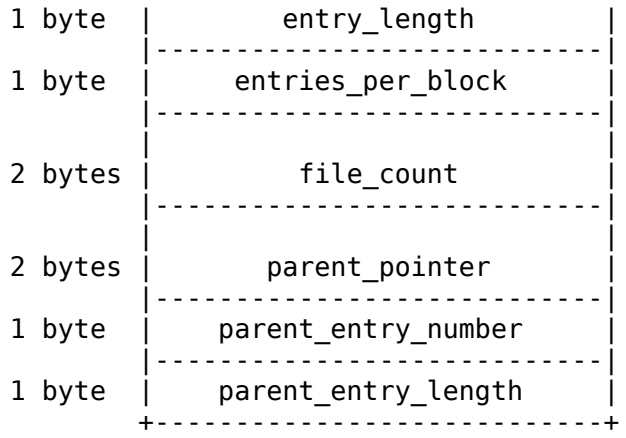
B.2.3 - Subdirectory Headers

The key block of every subdirectory file is pointed to by an entry in a parent directory; for example, by an entry in a volume directory (explained in Section B.2). A subdirectory's header begins at byte position $0004 of the key block of that subdirectory file, immediately following the two pointers.

Its internal structure is quite similar to that of a volume directory header. Fourteen fields are currently defined to be in a subdirectory. Figure B-4 illustrates the structure of a subdirectory header. A description of all the fields in a subdirectory header follows Figure B-4.

Figure B-4. The Subdirectory Header





---

storage_type and **name_length** (1 byte): Two four-bit fields are packed into this byte. A value of $E in the high four bits (the storage_type) identifies the current block as the key block of a subdirectory file. The low four bits contain the length of the subdirectory's name (see the file_name field, below). The name_length can be changed by a RENAME call.

file_name (15 bytes): The first name_length bytes of this field contain the subdirectory's name. This name must conform to the filename syntax explained in Chapter 2. This field can be changed by the RENAME call.

reserved (8 bytes): Reserved for future expansion of the file system.

creation (4 bytes): The date and time at which this subdirectory was created. The format of these bytes is described in Section B.4.2.2.

version (1 byte): The version number of ProDOS under which this subdirectory was created. This byte allows newer versions of ProDOS to determine the format of the subdirectory, and to adjust their directory interpretations accordingly. ProDOS 1.0: version = 0.

min_version (1 byte): The minimum version number of ProDOS that can access the information in this subdirectory. This byte allows older versions of ProDOS to determine whether they can access newer subdirectories. min_version = 0.

access (1 byte): Determines whether this subdirectory can be read, written, destroyed, and renamed, and whether the file needs to be backed up. The format of this field is described in Section B.4.2.3. A subdirectory's access byte can be changed by the SET_FILE_INFO

call.

entry_length (1 byte): The length in bytes of each entry in this subdirectory. The subdirectory header itself is of this length. entry_length = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. entries_per_block = $0D.

file_count (2 bytes): The number of active file entries in this subdirectory file. An active file is one whose storage_type is not 0. See Section "File Entries" for more information about file entries.

parent_pointer (2 bytes): 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): The entry number for this subdirectory within the block indicated by parent_pointer.

parent_entry_length (1 byte): The entry_length for the directory that owns this subdirectory file. Note that with these last three fields you can calculate the precise position on a volume of this subdirectory's file entry. parent_entry_length = $27.

B.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.

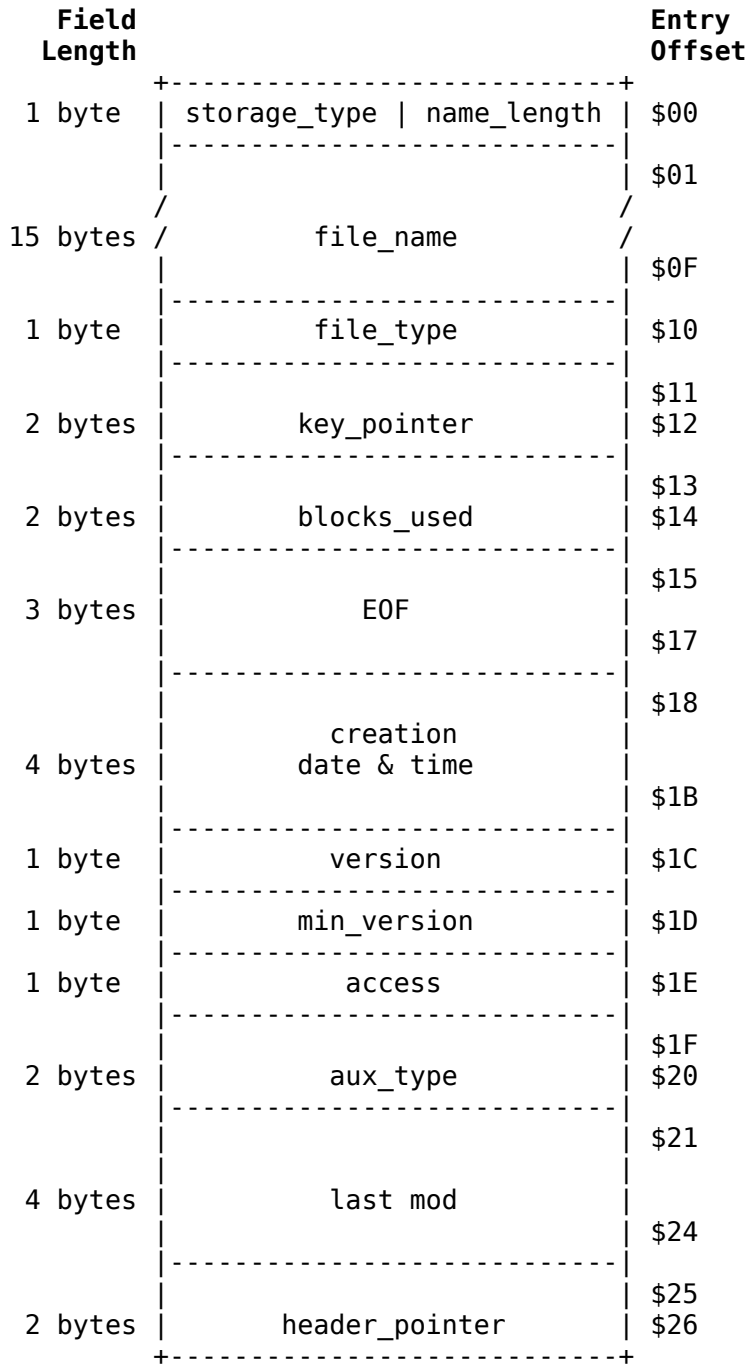
An entry in a directory file may be active or inactive; that is, it may or may not describe a file currently in the directory. If it is inactive, the first byte of the entry (storage_type and name_length) has the value zero.

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 B-5 describes the format of a file entry.

Figure B-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 pointed to by this file entry:

$1 = Seeding file
$2 = Sapling file
$3 = Tree file
$4 = Pascal area
$D = Subdirectory

Seedling, sapling, and tree files, the three forms of a standard file, are described in Section B.3. The low four bits contain the length of the file'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 file's name. This name must conform to the filename syntax explained in Chapter 2. This field can be changed by the RENAME call.

file_type (1 byte): A descriptor of the internal structure of the file. Section B.4.2.4 contains a list of the currently defined values of this byte.

key_pointer (2 bytes): The block address of the master index block if a tree file, of the index block if a sapling file, and of the block if a seedling file.

blocks_used (2 bytes): 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 B.3 for more information on standard files.

EOF (3 bytes): 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 in Section B.3.6, EOF may be greater than the number of bytes actually allocated on the disk.

creation (4 bytes): The date and time at which the file pointed to by this entry was created. The format of these bytes is described in Section B.4.2.2.

version (1 byte): The version number of ProDOS under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS to determine the format of the file, and adjust their

interpretation processes accordingly. In ProDOS 1.0, version = 0.

min_version (1 byte): The minimum version number of ProDOS that can access the information in this file. This byte allows older versions of ProDOS to determine whether they can access newer files. In ProDOS 1.0, min_version = 0.

access (1 byte): Determines whether this file can be read, written, destroyed, and renamed, and whether the file needs to be backed up. The format of this field is described in Section B.4.2.3. The value of this field can be changed by the SET_FILE_INFO call. You cannot delete a subdirectory that contains any files.

aux_type (2 bytes): A general-purpose field in which a system program can store additional information about the internal format of a file. For example, the ProDOS BASIC system program uses this field to record the load address of a BASIC program or binary file, or the record length of a text file.

last_mod (4 bytes): 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 B.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.

B.2.5 - Reading a Directory File

This section deals with the techniques of reading from directory files, not with the specifics. The ProDOS calls with which these techniques can be implemented are explained in Chapter 4.

Before you can read from a directory, you must know the directory's pathname. With the directory's pathname, you can open the directory file, and obtain a reference number (**`RefNum`**) for that open file. Before you can process the entries in the directory, you must read three values from the directory header:

- the length of each entry in the directory (entry_length)
- the number of entries in each block of the directory

(entries_per_block)
- the total number of files in the directory (file_count).

Using the reference number to identify the file, read the first 512 bytes from the file, and into a buffer (**ThisBlock**). The buffer contains two two-byte pointers, followed by the entries; the first entry is the directory header. The three values are at positions $1F through $22 in the header (positions $23 through $26 in the buffer). In the example below, these values are assigned to the variables **EntryLength**, **EntriesPerBlock**, and **FileCount**.

```
Open(DirPathname, Refnum);               {Get reference number    }
ThisBlock       := Read512Bytes(RefNum); {Read a block into buffer}
EntryLength     := ThisBlock[$23];       {Get directory info      }
EntriesPerBlock := ThisBlock[$24];
FileCount       := ThisBlock[$25] + (256 * ThisBlock[$26]);
```

Once these values are known, a system program can scan through the entries in the buffer, using a pointer to the beginning of the current entry **EntryPointer**, a counter **BlockEntries** that indicates the number of entries that have been examined in the current block, and a second counter **ActiveEntries** that indicates the number of active entries that have been processed.

An entry is active and is processed only if its first byte, the storage_type and name_length, is nonzero. All entries have been processed when **ActiveEntries** is equal to **FileCount**. If all the entries in the buffer have been processed, and **ActiveEntries** doesn't equal **FileCount**, then the next block of the directory is read into the buffer.

```
EntryPoint      := EntryLength + $04;         {Skip header entry}
BlockEntries    := $02;            {Prepare to process entry two}
ActiveEntries   := $00;            {No active entries found yet }

while ActiveEntries < FileCount do begin
     if ThisBlock[EntryPointer] <> $00 then begin  {Active entry}
          ProcessEntry(ThisBlock[EntryPointer]);
          ActiveEntries := ActiveEntries + $01
     end;
     if ActiveEntries < FileCount then  {More entries to process}
          if BlockEntries = EntriesPerBlock
               then begin           {ThisBlock done. Do next one}
                    ThisBlock    := Read512Bytes(RefNum);
                    BlockEntries := $01;
                    EntryPointer := $04
               end
               else begin           {Do next entry in ThisBlock }
                    EntryPointer := EntryPointer + EntryLength;
```

```
                    BlockEntries := BlockEntries + $01
                end
 end;
 Close(RefNum);
```


This algorithm processes entries until all expected active entries have been found. If the directory structure is damaged, and the end of the directory file is reached before the proper number of active entries has been found, the algorithm fails.

B.3 - Format of Standard Files

Each active entry in a directory file points to the key block (the first block) of a file. As shown below, the key block of a standard file may have several types of information in it. The storage_type field in that file's entry must be used to determine the contents of the key block. This section explains the structure of the three stages of standard file: seedling, sapling, and tree. These are the files in which all programs and data are stored.

B.3.1 - Growing a Tree File

The following scenario demonstrates the *growth* of a tree file on a volume. This scenario is based on the block allocation scheme used by ProDOS 1.0 on a 280-block flexible disk that contains four blocks of volume directory, and one block of volume bit map. Larger capacity volumes might have more blocks in the volume bit map, but the process would be identical.

A formatted, but otherwise empty, ProDOS volume is used like this:

Blocks 0-1 - Loader
Blocks 2-5 - Volume directory
Block 6 - Volume bit map
Blocks 7-279 - Unused

If you open a new file of a nondirectory type, 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 key block is indicated below by an arrow.

The volume now looks like this:

```
Data Block 0
    Blocks 0-1      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:

```
Index Block 0
Data Block 0
Data Block 1
    Blocks 0-1      Loader
    Blocks 2-5      Volume directory
    Block 6         Volume bit map
    Block 7         Data block 0
--> Block 8         Index block 0
    Block 9         Data block 1
    Blocks 10-279   Unused
```

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

Here's a new picture of the volume:

```
Master Index Block
Index Block 0
Index Block 1
Data Block 0
Data Block 255
Data Block 256
    Blocks 0-1      Loader
    Blocks 2-5      Volume directory
    Block 6         Volume bit map
    Block 7         Data block 0
    Block 8         Index block 0
```

```
    Blocks 9-263    Data blocks 1-255
--> Block 264       Master index block
    Block 265       Index block 1
    Block 266       Data block 256
    Blocks 267-279  Unused
```

As data is written to this file, additional data blocks and index blocks are allocated as needed, up to a maximum of 129 index blocks (one a master index block), and 32,768 data blocks, for a maximum capacity of 16,777,215 bytes of data in a file. If you did the multiplication, you probably noticed that a byte was lost somewhere. The last byte of the last block of the largest possible file cannot be used because **EOF** cannot exceed 16,777,216. If you are wondering how such a large file might fit on a small volume such as a flexible disk, refer to Section B.3.6 on sparse files.

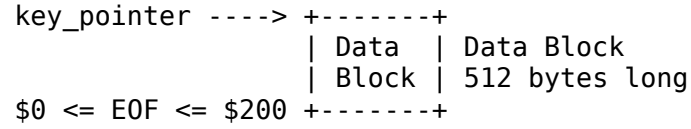
This scenario shows the growth of a single file on an otherwise empty volume. The process is a bit more confusing when several files are growing -- or being deleted -- simultaneously. However, the block allocation scheme is always the same: when a new block is needed ProDOS always allocates the first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

B.3.2 Seedling Files

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

The structure of such a seedling file appears in Figure B-6.

Figure B-6. Structure of a Seedling File



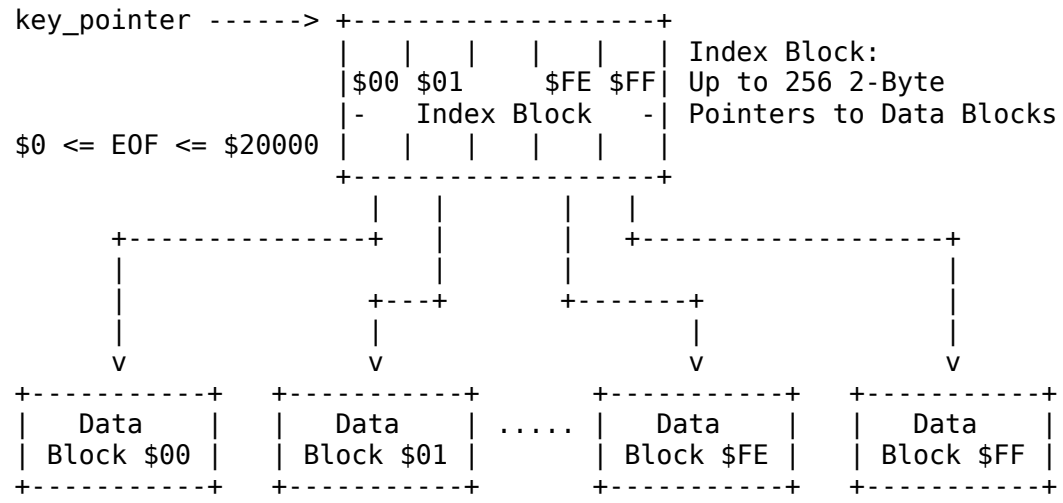
The file is called a seedling file because, if more than 512 data bytes are written to it, it grows into a sapling file, and thence into a tree file.

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

B.3.3 - Sapling Files

A **sapling file** 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. See Figure B-7.

Figure B-7. Structure of a Sapling File



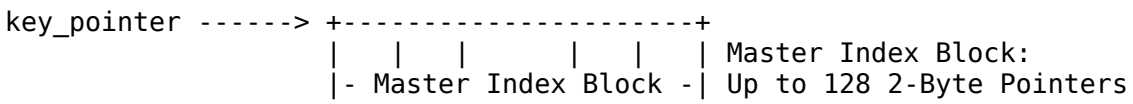
The key block of a sapling file is its index block. ProDOS retrieves data blocks in the file by first retrieving their addresses in the index block.

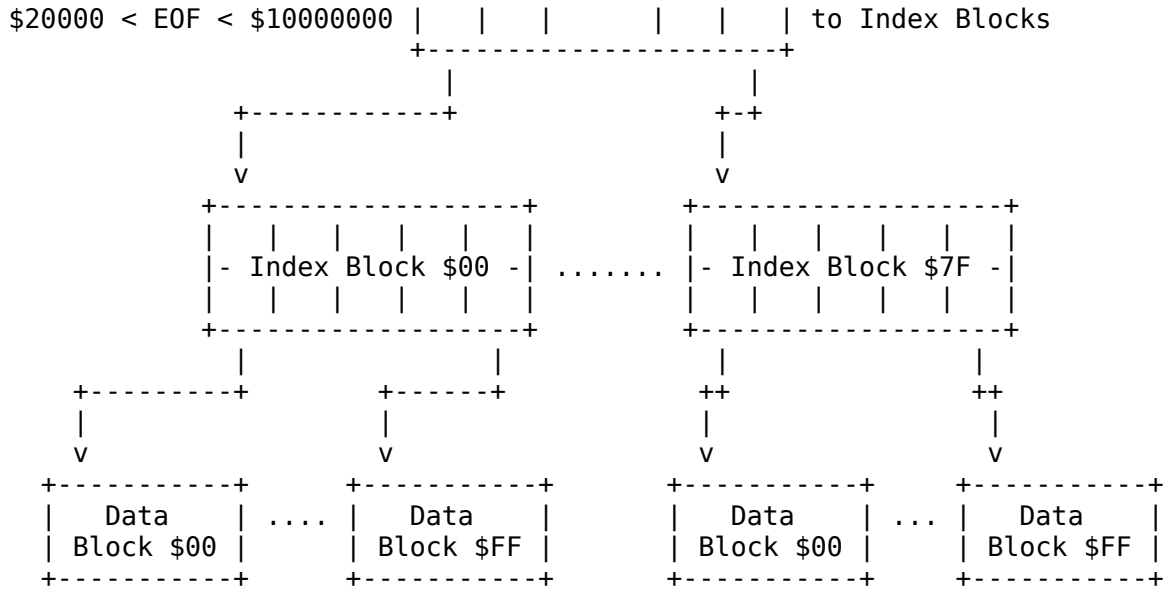
The **storage_type** field of an entry that points to a sapling file has the value $2.

B.3.4 - Tree Files

A **tree file** contains more than 128K bytes, and less than 16M bytes ($20000 < **EOF** < $1000000). A tree file consists of a master index block, 1 to 128 index blocks, and 1 to 32,768 data blocks. The master index block contains the addresses of the index blocks, and each index block contains the addresses of up to 256 data blocks. The structure of a tree file is shown in Figure B-8.

Figure B-8. The Structure of a Tree File





The key block of a tree file is the master index block. By looking at the master index block, ProDOS can find the addresses of all the index 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.

B.3.5 - Using Standard Files

A system program or application program operates the same on all three types of standard files, although the **storage_type** in the file's entry can be used to distinguish between the three. A program rarely reads index blocks or allocates blocks on a volume: ProDOS does that. The program need only be concerned with the data stored in the file, not with how they are stored.

All types of standard files are read as a sequence of bytes, numbered from 0 to **EOF**-1, as explained in Chapter 4.

B.3.6 - 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. ProDOS implements sparse files by allocating only those data blocks that have

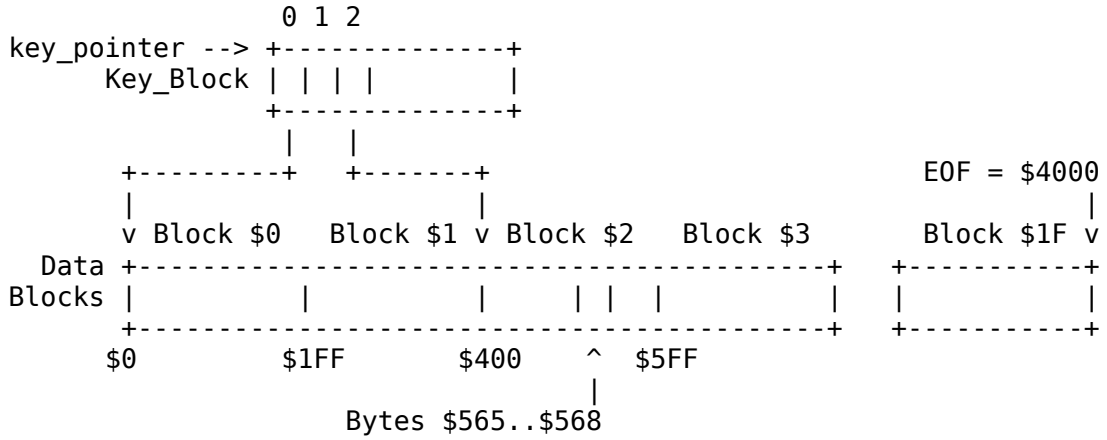
had data written to them, as well as the index blocks needed to point to them.

For example, you 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. If you create a file with an **EOF** of $0, ProDOS allocates only the key block (a data block) for a seedling file, and fills it with null characters (ASCII $00).

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

If you now 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 B-9 shows how the file is organized.

Figure B-9. A Sparse File



Thus ProDOS 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), ProDOS allocates an index block and 32 data blocks for a sapling file, and fills the data blocks with nulls.

By the Way: The first data block of a standard file, be it a seedling, sapling, or tree file, is always allocated. Thus there is always a data block to be read in when the file is opened.

B.3.7 - Locating a Byte in a File

The algorithm for finding a specific byte within a standard file is given below.

The MARK is a three-byte value that indicates an absolute byte position within a file.

```
Byte #        Byte 2             Byte 1            Byte 0

bit #      7             0   7             0   7             0
          +-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+
MARK      |Index Number |Data Block Number|   Byte of Block   |
          +-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+
Used by:    Tree only    Tree and sapling      All three
```

If the file is a tree file, then the high seven bits of the MARK determine the number (0 to 127) of the index block that points to the byte. The value of the seven bits indicate the location of the low byte of the index block address within the master index block. The location of the high byte of the index block address is indicated by the value of these seven bits plus 256.

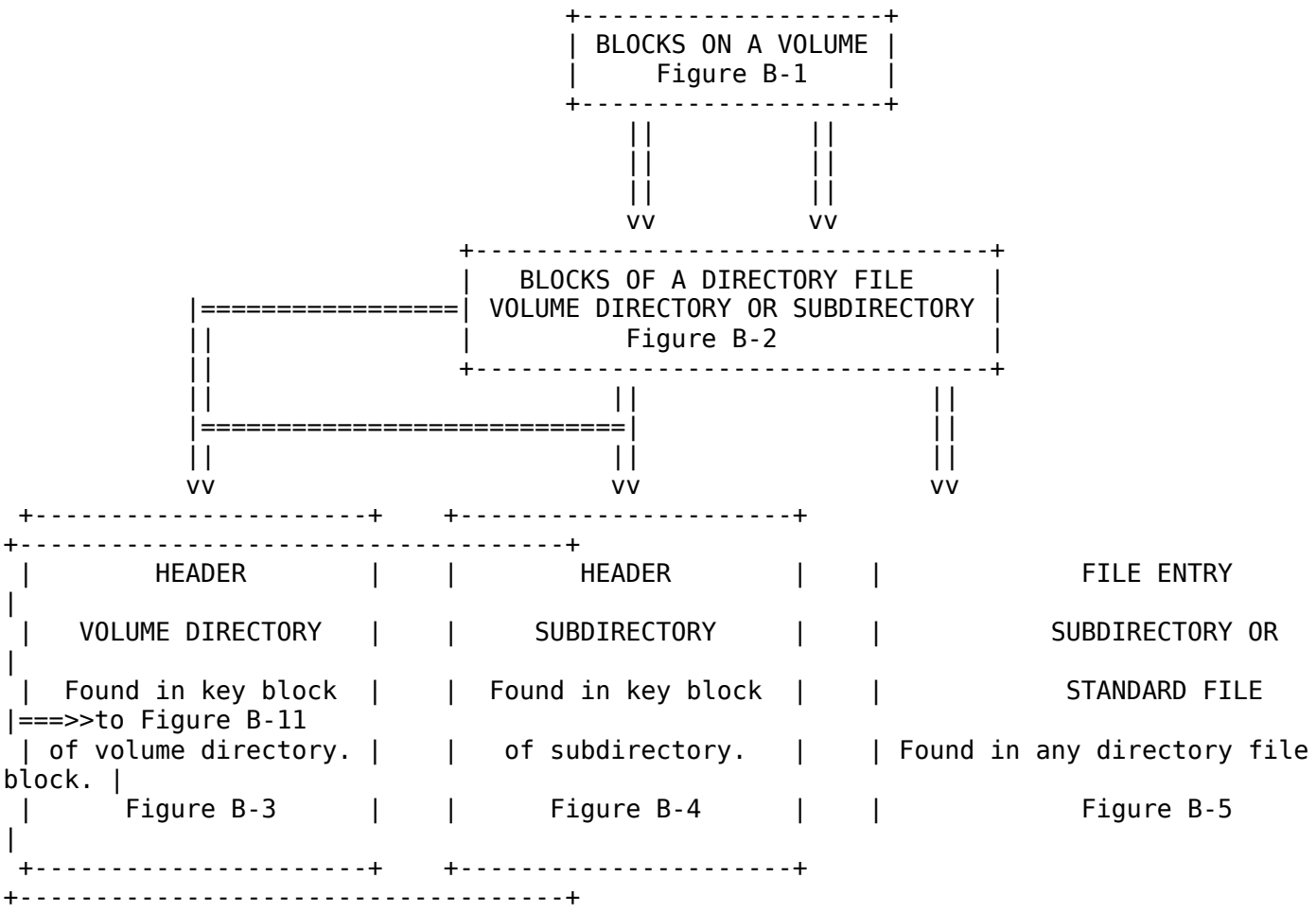
If the file is a tree file or a sapling file, then the next eight bits of the MARK determine the number (0-255) of the data block pointed to by the indicated index block. This 8-bit value indicates the location of the low byte of the data block address within the index block. The high byte of the index block address is found at this offset plus 256.

For tree, sapling, and seedling files, the low nine bits of the MARK are the absolute position of the byte within the selected data block.

B.4 - Disk Organization

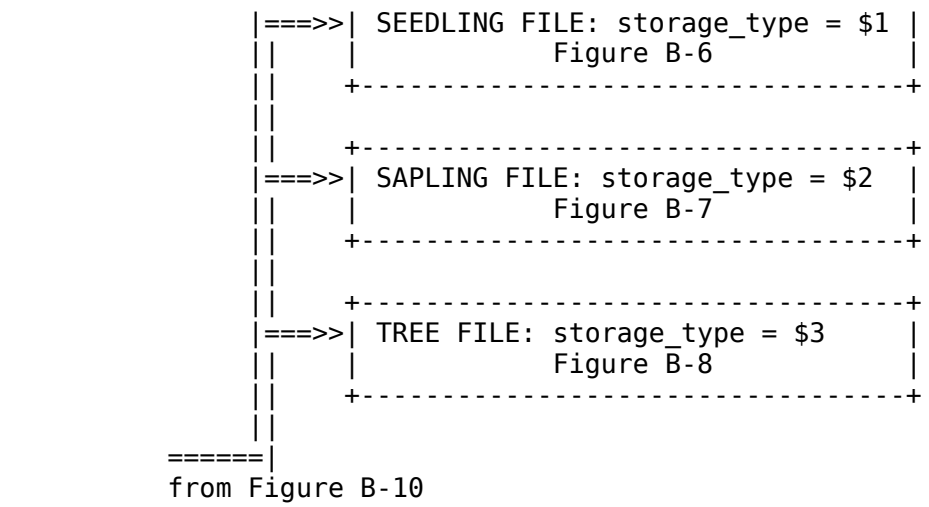
Figure B-10 presents an overall view of block organization on a volume. Figure B-11 shows the complete structures of the three standard files types. Figure B-12 is a summary of header and entry field information.

Figure B-10. Disk Organization



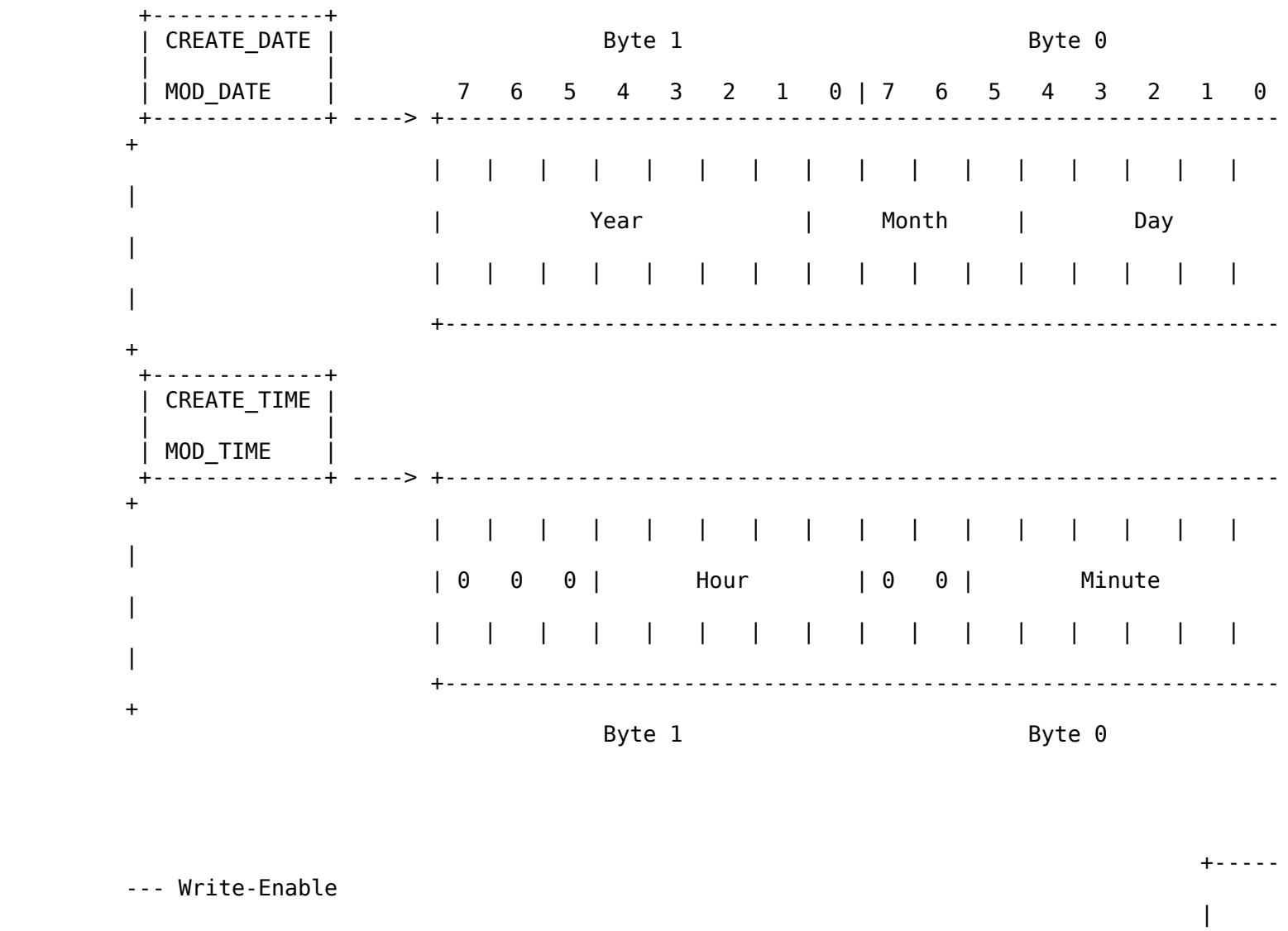
B.4.1 - Standard Files

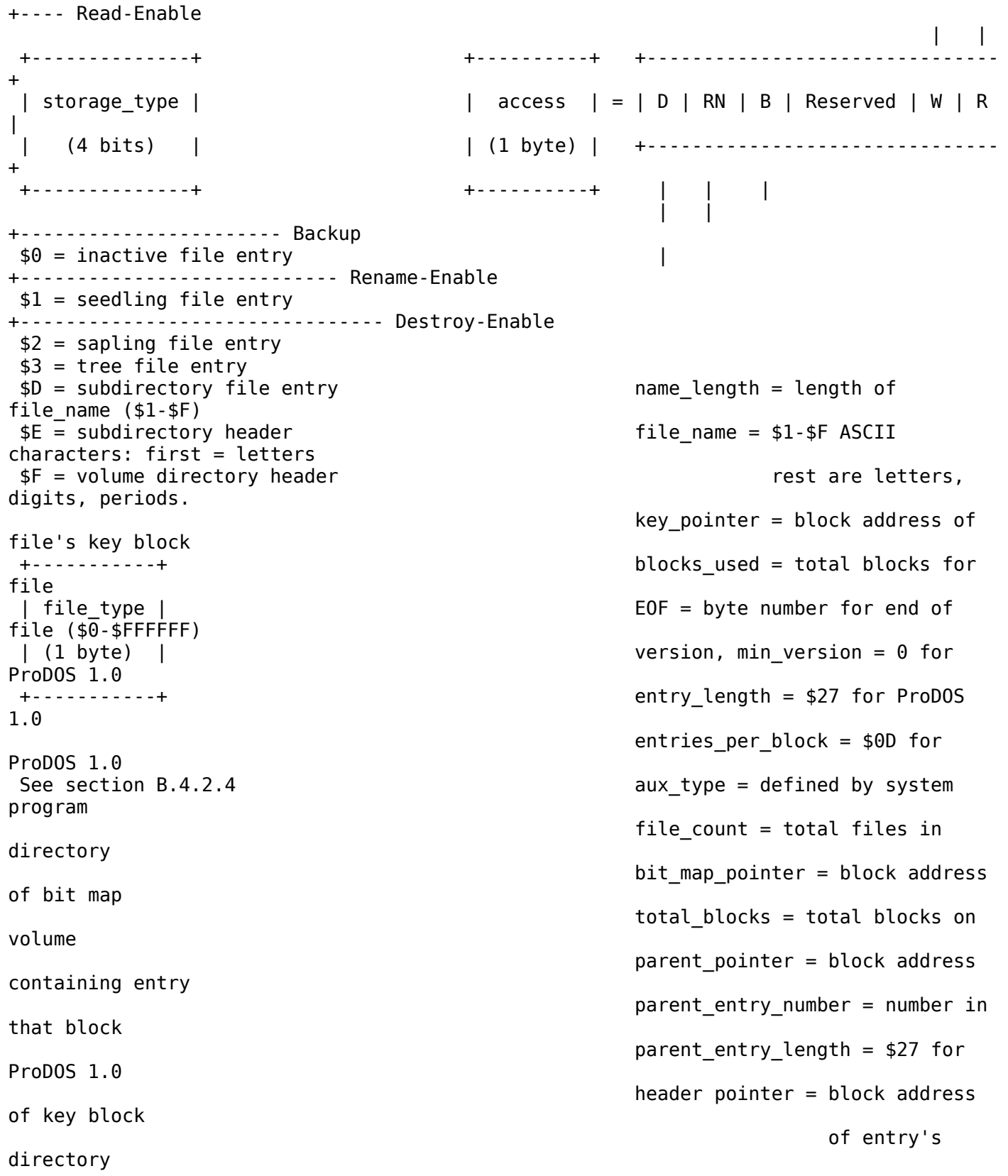
Figure B-11. Standard Files



B.4.2 - Header and Entry Fields

Figure B-12. Header and Entry Fields





B.4.2.1 - The storage_type Attribute

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 (EOF <= 256 bytes)
$2 indicates a sapling file entry (256 < EOF <= 128K bytes)
$3 indicates a tree file entry (128K < EOF < 16M bytes)
$4 indicates Pascal area
$D indicates a subdirectory file entry
$E indicates a subdirectory header
$F indicates a volume directory header

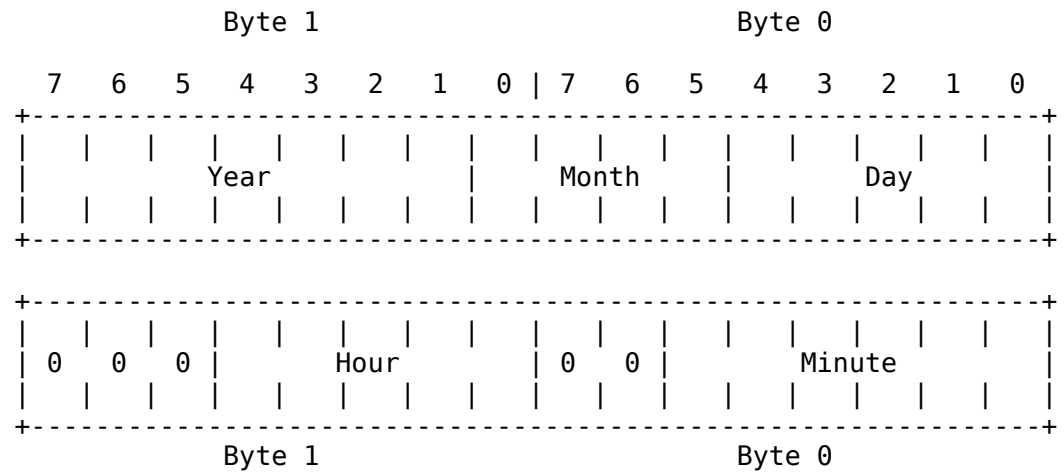
The name_length, the low-order four bits of the first byte, specifies the number of characters in the file_name field.

ProDOS 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, ProDOS changes a tree file to a sapling file and a sapling file to a seedling file.

B.4.2.2 - The creation and last_mod Fields

The date and time of the creation and last modification of each file and directory is stored as two four-byte values, as shown in Figure B-13.

Figure B-13. Date and Time Format

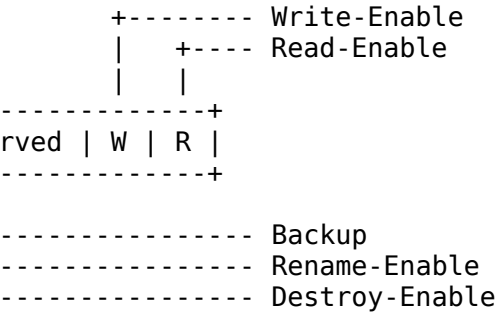


The values for the year, month, day, hour, and minute are stored as binary integers, and may be unpacked for analysis.

B.4.2.3 - The access Attribute

The access attribute field (Figure B-14) determines whether the file can be read from, written to, deleted, or renamed. It also contains a bit that can be used to indicate whether a backup copy of the file has been made since the file's last modification.

Figure B-14. The access Attribute Field



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

ProDOS sets bit 5, the **backup bit**, of the access field to 1 whenever the file is changed (that is, after a CREATE, RENAME, CLOSE after WRITE, or SET_FILE_INFO operation). This bit should be reset to 0 whenever the file is duplicated by a backup program.

Note: Only ProDOS may change bits 2-4; only backup programs should clear bit 5, using SET_FILE_INFO.

B.4.2.4 - The file_type Attribute

The file_type attribute within an entry field identifies the type of file described by that entry. This field should be used by system programs to guarantee file compatibility from one system program to the next. The values of this byte are shown in Table B-1.

Table B-1. The ProDOS File_Types

*The file types marked with a * apply to Apple III only; they are not ProDOS compatible. For the file types used by Apple III SOS only, refer to the SOS Reference Manual.*

File Type	Preferred Use
$00	Typeless file (SOS and ProDOS)
$01	Bad block file
$02 *	Pascal code file
$03 *	Pascal text file
$04	ASCII text file (SOS and ProDOS)
$05 *	Pascal data file
$06	General binary file (SOS and ProDOS)
$07 *	Font file
$08	Graphics screen file
$09 *	Business BASIC program file
$0A *	Business BASIC data file
$0B *	Word Processor file
$0C *	SOS system file
$0D,$0E *	SOS reserved
$0F	Directory file (SOS and ProDOS)
$10 *	RPS data file
$11 *	RPS index file
$12 *	AppleFile discard file
$13 *	AppleFile model file
$14 *	AppleFile report format file
$15 *	Screen Library file
$16-$18 *	SOS reserved
$19	AppleWorks Data Base file
$1A	AppleWorks Word Processor file
$1B	AppleWorks Spreadsheet file
$1C-$EE	Reserved
$EF	Pascal area
$F0	ProDOS CI added command file
$F1-$F8	ProDOS user defined files 1-8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS system file

B.5 - DOS 3.3 Disk Organization

Both DOS 3.3 and ProDOS 140K flexible disks are formatted using the same 16-sector layout. As a consequence, the ProDOS READ_BLOCK and WRITE_BLOCK calls are able to access DOS 3.3 disks too. These calls know nothing about the organization of files on either type of disk.

When using READ_BLOCK and WRITE_BLOCK, you specify a 512-byte block on the disk. When using RWTS (the DOS 3.3 counterpart to READ_BLOCK and WRITE_BLOCK), you specify the track and sector of a 256-byte chunk of data, as explained in the *DOS Programmer's Manual*. You use READ_BLOCK and WRITE_BLOCK to access DOS 3.3 disks, you must know what 512-byte block

corresponds to the track and sector you want.

Figure B-15 shows how to determine a block number from a given track and sector. First multiply the track number by 8, then add the Sector Offset that corresponds to the sector number. The half of the block in which the sector resides is determined by the Half-of-Block line (1 is the first half; 2 is the second).

Figure B-15. Tracks and Sectors to Blocks

```
       Block = (8 * Track) + Sector Offset

        Sector : 0 1 2 3 4 5 6 7 8 9 A B C D E F
 Sector Offset : 0 7 6 6 5 5 4 4 3 3 2 2 1 1 0 7
  Half of Block: 1 1 2 1 2 1 2 1 2 1 2 1 2 1 2 2
```

Refer to the *DOS Programmer's Manual* for a description of the file organization of DOS 3.3 disks.

Appendix C
ProDOS, the Apple III, and SOS

This appendix explains the relationships between ProDOS, the Apple III, and SOS. It should be helpful to those already familiar with SOS and to those thinking about developing assembly-language programs concurrently for SOS and ProDOS.

C.1 - ProDOS, the Apple III, and SOS

As explained earlier in the manual, blocks 0 and 1 of a **ProDOS-formatted** disk contain the boot code -- the code that reads the operating system from the disk and runs it. Not explained was that this boot code runs on either an Apple II or an Apple III.

When you start up either an Apple II or an Apple III system with a ProDOS disk, the boot code is loaded at $800, and executed. The first thing it does is look to see whether it is running on an Apple II or Apple III. If it is running on an Apple II, it tries to load in the file PRODOS. If it is running on an Apple III, it tries to load in the file

SOS.KERNEL. In either case, if the proper file is not found, it displays the appropriate error message.

This means that two versions of an application could be written, one for the Apple II, the other for the Apple III, and packaged together on the same disk. This single disk could be sold to both Apple II and Apple III owners.

C.2 - File Compatibility

SOS and ProDOS use the same directory structure: no exceptions. Every file on a ProDOS disk can be read by a SOS program and vice versa.

The file types that are used by both systems are directory files, text files, and binary files. These three types are adequate for the sharing of data between SOS and ProDOS versions of the same program.

File types that are intended for one system, but encountered on the other (as when you CATALOG a ProDOS disk using Business BASIC) are not inherently different from recognized file types; they just might cause a number to be displayed as their type instead of a name. The ProDOS BASIC system program, Filer, Conversion program, and Editor/Assembler all recognize and display names for all currently defined SOS file types. The abbreviations displayed when Apple III file types are encountered using ProDOS are shown in the quick reference section of this manual.

C.3 - Operating System Compatibility

Because of the larger amount of memory available to SOS, it is a much more complete operating system than is ProDOS. SOS has a complete and well defined file manager, device manager, memory manager, and interrupt and event handler. ProDOS has a file manager and simplified interrupt and memory calls.

C.3.1 - Comparison of Input/Output

SOS communicates with all devices -- the console, printers, disk drives, and so on -- by making open, read, write, and close calls to the

appropriate device; writing to one device is essentially the same as writing to another. ProDOS can perform these operations on files only. Apple II peripherals generally have their driver code in ROM on the peripheral card. There is no consistent method for communicating with them. Thus the protocol for using any particular device must be known by the system program that is currently running.

C.3.2 - Comparison of Filing Calls

The set of calls to the ProDOS operating system is essentially a subset of the calls to SOS. All filing calls shared by the two systems have the same call number and nearly identical sets of parameters. Some differences are:

- With ProDOS you don't specify the file size when you create a file. Files are automatically extended when necessary.
- With SOS the GET_FILE_INFO call returns the size of the file in bytes (the EOF). In ProDOS you must OPEN the file and then use the GET_EOF call.

- The SOS VOLUME command corresponds to the ProDOS ON_LINE command. When given a device name, VOLUME returns the volume name for that device. When given a unit number (derived from the slot and drive), ON_LINE returns the volume name.
- For SOS, SET_MARK and SET_EOF can use a displacement from the current position. ProDOS uses only an absolute position in the file.

C.3.3 - Memory Handling Techniques

SOS has a fairly sophisticated memory manager: a system program requests memory from SOS, either by location or by amount needed. If the request can be satisfied, SOS grants it. That portion of memory is then the sole responsibility of the requestor until it is released.

A ProDOS system program is responsible for its own memory management. It must find free memory, and then allocate it by marking it off in a memory bit map. If a page of memory is marked in the bit map, ProDOS will not write data into that page. ProDOS can thus prevent users from destroying protected areas of memory (presumably all data is brought into memory using the ProDOS READ

call).

C.3.4 - Comparison of Interrupts

In SOS, any device capable of generating an interrupt must have a device driver capable of handling the interrupt; the device driver and the interrupt handler are inseparable. ProDOS does not have device drivers; thus, interrupt handling routines are installed separately using the ALLOC_INTERRUPT call. Also, whereas SOS has a distinct interrupt priority for each device in the system, ProDOS must poll the routines one by one until someone claims the interrupt.

Appendix D
The ProDOS Machine Language Exerciser

The ProDOS Exerciser program is a menu-driven program that allows you to practice calls to the ProDOS Machine Language Interface without writing a system program. It is useful for learning how the various ProDOS MLI calls work. Using it, you can test the behavior of a ProDOS-based program before writing any code.

D.1 - How to Use It

To start up the Exerciser program from BASIC, type

-/EXERCISER/EXER.SYSTEM

and press [RETURN].

This causes the Exerciser (which is a machine-language program, but not a system program) to be loaded at $2000, and then relocated to the highest available spot in memory. On a 64K system, it occupies memory from $7400 on.

The Exerciser main menu displays all the MLI calls and their call

numbers, as well as a few other commands. To select an MLI call, simply type the call number followed by [RETURN]. To select one of the other commands, type the displayed letter followed by [RETURN].

When you select either a call or a command, a list of parameters for that call is displayed. The parameters for each MLI call are displayed almost exactly as they would have to be coded in a ProDOS-based application. The only difference is that a true parameter list would contain a two-byte pointer to a pathname, whereas the Exerciser displays the pathname itself. The meanings of the parameters for each ProDOS call are described in Chapter 4 in the section describing that call.

The default values for each of the parameters are displayed. The cursor pauses at each of the parameters that requires a value to be entered. You may accept the default value by pressing [RETURN] or change the value by typing the new value followed by [RETURN]. All values are displayed and entered in hexadecimal.

When you have entered values for all required parameters, press [RETURN]. The call is executed, values returned by the call are displayed, and an error message is displayed. If error **$00** is indicated the call was successful. If the call was unsuccessful, the Apple II beeps as it displays the error message.

Errors are discussed at the end of
Chapter 4.

D.2 - Modify Buffer

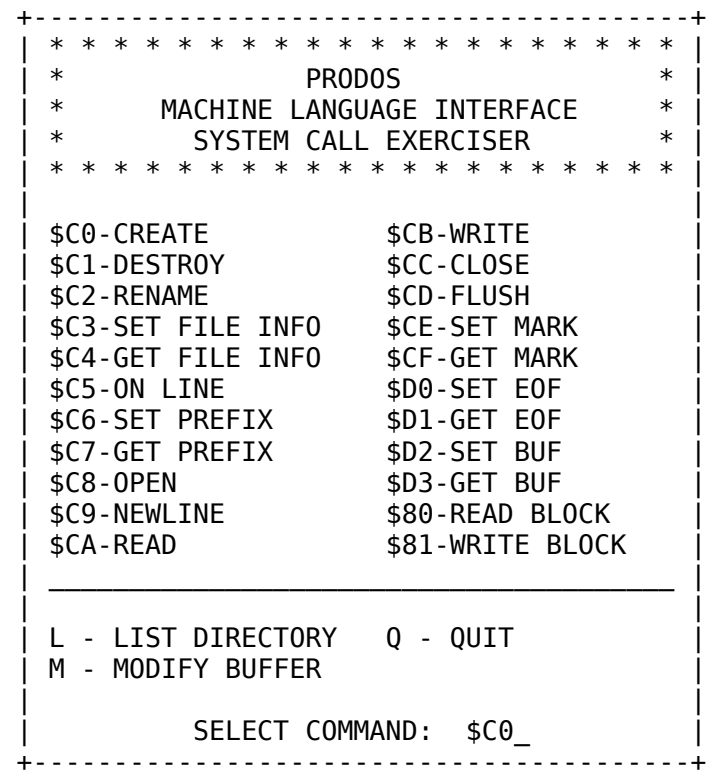
The Modify Buffer command can be used to examine or edit the Contents of memory. It asks you for a data buffer address; this is the address at which you wish to start editing. You can then page forward or backward through memory using [>] and [<], respectively.

Each screen displays the values of 256 consecutive bytes, arranged in 16 rows of eight bytes each. The ASCII characters associated with these bytes are displayed at the right of the screen (as printed with the high bits set). On a standard Apple II, lowercase ASCII codes are converted to the corresponding uppercase codes. Each row is preceded by the address of the first byte in that row (just like the LIST command in the Apple II Monitor).

To move the cursor to a different byte on the screen, use [I], [J], [K], and [M], or the arrow keys. To change a byte of memory, simply type the new value right over the old one. The value is updated in memory as well as on the screen. The Modify Buffer command remembers the original values of the last 16 bytes you changed. To restore up to sixteen changed bytes, press **U** (for Undo) once for each value to be restored.

If a memory page is marked in the system bit map as used by the system, the editor displays the message **MEMORY PAGE PROTECTED** and it does not allow you to change a value in that page.

screen shot from front cover



Index

A

A register ... 96
access ... 150, 153, 157
byte ... 13
accumulator ... 29, 77, 85
`Active Entries` ... 158
ALLOC_INTERRUPT call ... 35, 170, 111, 178
alternate 64K RAM bank ... 89
APPEND command ... 131
Apple II ... xvi, 98
Apple II Plus ... 98
Apple II SOS ... 176
Apple IIc ... 98, 143
Apple IIe ... 98, 143
-- with extended 80-column text card ... 89
Apple III ... 98
file types ... 176
Applesoft ... 121, 134, 142
assembly language ... 131
aux_type ... 39, 46, 50, 100, 157
auxiliary bank hi-res graphics pages ... 89

B

backup bit ... 63, 64, 172
`BADCALL` ... 128
bank-switching routines ... 97
BASIC.SYSTEM ... xv, 82, 121, 124, 176
BEEP example ... 136
BEEPSLOT example ... 138
binary files ... 176
bit map ... 84, 150
BLOAD command ... 132
`Block Entries` ... 158
Block File Manager (BFM) ... 7, 28, 31
block number ... 115, 146
blocks ... 18
blocks_used ... 50, 156
boot code ... 176

boot ROM ... 22
-- disk drives ... 112
booting ... 22
BRUN command ... 132
BSAVE command ... 132
buffer ... 15
-- allocation ... 25
-- pointer ... 115
byte, locating a specific ... 166

C

C-flag 29, 77
calender card ... *See* clock/calender card
calls
-- filing ... 33, 56
-- housekeeping ... 32
-- system ... 35
carry flag ... 122
CAT command ... 132
CATALOG command ... 132
catalog format ... 101
CHAIN command ... 131
clock/calender card ... 2,6,71,99
CLOSE call ... 13, 16, 17, 26, 34, 99, 104, 132
CMDADR address ... 108
Command Dispatcher ... 7,28
command list ... 134
commands, adding ... 134
CONVERT.program ... 3, 176
CREATE call ... 13, 32, 99, 104, 132
create_date ... 39, 51
create_time ... 39, 51
creation ... 150, 153, 156
-- date ... 171
-- time ... 171
creation_date ... 13
creation_time ... 13

D

dash (-) command ... 131
data blocks ... 19

data_buffer ... 15, 52, 55
data files ... 18
date and time, system ... 71
DEALLOC INTERRUPT call ... 35, 107, 112
defaults (system program) ... 100
DELETE call ... 132
DESTROY call ... 13, 32, 99, 104
device drivers ... 142
directory files ... 3,17,18,176
-- reading ... 157
-- structure ... 18
disconnecting /RAM ... 90
disk
-- access ... 16
-- controller card ... 113
-- device driver vectors ... 94
-- devices ... 95
-- driver routines ... 28
-- operating system ... xv, 2
-- RAM ... 91
-- volume ... 143
Disk II driver ... 113
disk-drive controller card ... 22
dispatcher code ... 87
DOS 3.3 ... 174
-- disks ... 73
DOS ProDOS Conversion program ... xv, 3
DOSCMD vector ... 131, 134

E

80-column text card ... 99
emulation mode ... 98
enable_mask ... 58
endtry_length ... 154
entries (directory file) ... 17
Entries Per Block ... 150, 153, 154, 158
entry field ... 43, 47
Entry Length variable ... 158
Entry Pointer variable ... 158
entry_length ... 150, 153

entry points ... 94
EOF ... 15, 20, 67, 156, 164, 171
-- *See also* individual calls
error codes (ProDOS) ... 77
EXEC file ... 17, 131
EXERCISER program ... 31, 180
EXTRNCMD location ... 134

F

FBITS ... 126, 141
fields, pointer ... 148
file(s)
-- binary ... 176
-- buffer ... 26
-- closing ... 14, 16
-- control block ... 14, 56
-- creating ... 13
-- data ... 19
-- directory ... 18, 176
-- flushing ... 16
-- logical size ... 67
-- naming ... 10
-- opening ... 13
file_count ... 150, 153, 154 158
file_name ... 150, 150, 153, 156
file_type ... 13
filename ... 10
Filer, ProDOS ... 176
Filer Program ... xv
filing calls ... 3, 5
-- ProDOS vs. SOS ... 177
FLUSH ... 16, 17, 34, 99, 104, 132
FORMAT call ... 113
FRE call ... 132

G

GET_BUF call ... 26, 34
GET_EOF call ... 15, 34, 177
GET_FILE_INFO call ... 32, 43, 99, 100, 177
GET_MARK call ... 15, 34
GET_PREFIX call ... 11,33

GET_TIME call ... 35, 99, 104
GETLN input buffer ... 105, 135
global page ... 84, 104, 141
global variables ... 25
GOSYSTEM ... 127, 129

H

header entry ... 147
header_pointer ... 157
headers (subdirectory) ... 151
HELP command ... 141
hi-res graphics ... 89
HIMEM command ... 141
housekeeping calls ... 3, 32, 36-54

I

I/O buffer ... 14, 69
I/O vectors ... 123
IN# command ... 22, 132
index blocks ... 19, 160, 162, 163
input/output
-- buffer ... 14, 69
-- vectors ... 123
-- ProDOS vs. SOS ... 177
int_num ... 72, 73
interrupt(s) ... 2, 72
-- routines ... 97
exit routines ... 97
handler ... 28
handling calls ... 3
Receiver/Dispatcher ... 7
vector(s) ... 96
-- table ... 72
interrupt-driven devices ... 121
io_buffer ... 16, 33
-- *See also* individual calls
IVERSION ... 97

J

jump to subroutine (JSR) ... 29

K

key block ... 146, 147, 151, 159, 162, 164
key_pointer ... 156
key_pointer field ... 36
KVERSION ... 97

L

language card area ... 106
last_mod ... 157
level ... 56
linked list ... 36
LOAD command ... 131
loader program ... 22, 146
LOCK command ... 132
logical block ... 146
LOMEM command ... 122

M

MACHID byte ... 96, 98
machine configuration ... 98
Machine Language Interface (MLI) ... 3
machine language routines ... xv, 121
MARK ... 14, 15, 20, 65, 66, 164, 166
master index block ... 19, 160, 163
memory ... 98
-- calls ... 3
-- handling (ProDOS vs. SOS) ... 178
-- management ... 2
-- map ... 24, 95
-- page ... 181
min_version ... 150, 153, 157
MLI (Machine Language Interface) ... 3, 5, 15, 22, 23, 25, 108, 180

-- entry point ... 94
-- issuing calls to ... 29
MLIATV flag ... 108
mod_date ... 46
mod_time ... 46, 50
Modify Buffer command ... 181
monitor ... 142

N

name_length ... 150, 153, 154, 156, 158
new_pathname ... 42
NEWLINE call ... 15, 33
newline_char ... 58
NOHELP command ... 141
null prefix ... 11
null_field ... 46

O

ON_LINE command ... 33, 178
OPEN call ... 26, 31, 33, 132, 177

P

pages ... 5
param_count ... *See* individual calls
parameter count ... 31
parent_entry_length ... 154
parent_entry_number ... 154
parent_pointer ... 153
parsing command ... 140
partial pathnames ... 10, 11
Pascal area ... 156
pathname ... 10, 11, 13
PBITS ... 126, 135, 141
peripheral cards ... xvii
pointer ... 18, 31
POSITION command ... 132
PR# command ... 22, 132
prefix ... 11, 132
ProDOS BASIC Programming Examples disk ... 3

ProDOS ... xv
-- Editor/Assembler ... 176
-- error codes ... 77
-- Filer ... 3, 20
-- Machine Language Interface ... 5, 142, 180
PRODOS program ... 22
ProDOS User's Disk ... 3
ProFile ... 4
program selectors ... 86

Q

QUIT call ... 87

R

/RAM ... 23, 89, 143
-- alternate 64K RAM bank ... 89
-- disconnecting ... 90
-- reinstalling ... 92
RAM disks ... 91
READ call ... 15, 33, 113, 131
READ_BLOCK call ... 35, 73, 174
ref_num ... 13
reference number ... 15, 16
register, stack ... 96
RENAME call ... 13, 32, 99, 104, 132, 150, 153, 156
request_count ... 62
-- *See also* individual calls
RESET vector ... 101
RESTORE command ... 132
result command ... 31
RUN command ... 131
RWTS (DOS 3.3) ... 174

S

sapling file ... 19, 156, 160, 164, 171
SAVE command ... 132
search order, volume ... 23
sectors ... 146
seedling file ... 19, 156, 160, 161

SET_BUF call ... 26
SET_EOF call ... 15, 34, 178

SET_FILE_INFO call ... 13, 32, 47, 99, 100, 104, 157, 172
SET_MARK call ... 15, 34, 66, 178
SET_PREFIX call ... 11, 33
SHOWTIME program ... 109-112
16-sector ROMs ... 113
6502 machine language ... xv, xvi
6502 registers ... 107, 108
slot(s) ... xvii
-- and drive ... 100
-- 5 ... 113
-- 6 ... 113
soft switches ... 106
SOS file ... 177
SOS KERNEL file ... 176
SOS volume command ... 178
sparse files ... 161
stack ... 25, 89, 107
register ... 96
standard files ... 17, 19, 159-166
starting up ... 22
startup disk ... 22
startup volume ... 23
STATUS call ... 113
status register ... 96
storage_type ... 13, 36, 39, 50, 150, 153, 154, 156, 158, 159, 162, 163
STORE command ... 132
strings ... 140
subdirectory ... 4
-- files ... 147
SYSCTBL ... 129
system
-- bit map ... 5
-- date and time ... 71, 99
-- failure ... 79
-- global page ... 22
-- level ... 16
-- prefix ... 55

-- programs ... 2,3,25,82
---- quitting ... 87
---- starting ... 86

T

13-sector ROMs ... 113
tone, warning ... 101
total_blocks ... 151
tracks ... 146
trans_count ... 62
-- *See also* individual calls
tree files ... 19, 156, 159, 160, 164, 171
tree structure ... 19, 36

U

unit_num ... 52
UNLOCK command ... 132

V

value ... 31
variables (global) ... 25
version ... 150, 153, 156
volume(s) ... 146
-- bit map ... 146
-- directory ... 4, 147
-- directory file ... 146
-- finding ... 100
-- names ... 10, 51
-- search order ... 23
VPATH1 ... 141
VPATH2 ... 141

W

WRITE command ... 15, 34, 113, 131
write buffer ... 64
WRITE_BLOCK call ... 35, 73, 174

X

X register ... 96, 122
XCNUM ... 135, 141
XLEN ... 135, 141
XRETURN ... 135
XTRNADDR ... 135, 141
XXX.SYSTEM ... 22, 82

Y

Y register ... 96, 122

Z

zero page ... 107

Tell Apple

Apple uses comments and suggestions from Apple computer owners like you to improve existing products and develop new and better products. Now that you've used this product, we want to know your thoughts and suggestions about your experience. Please use this form to tell Apple what you think.

Rest of card omitted

ProDOS Technical Reference Manual Quick Reference Card

ASCII Tables

Dec	ASCII	Hex	Binary 76543210
0	NUL	00	00000000

1	SOH	01	00000001
2	STX	02	00000010
3	ETX	03	00000011
4	EOT	04	00000100
5	ENQ	05	00000101
6	ACK	06	00000110
7	BEL	07	00000111
8	BS	08	00001000
9	HT	09	00001001
10	LF	0A	00001010
11	VT	0B	00001011
12	FF	0C	00001100
13	CR	0D	00001101
14	50	0E	00001110
15	SI	0F	00001111
16	DLE	10	00010000
17	DC1	11	00010001
18	DC2	12	00010010
19	003	13	00010011
20	004	14	00010100
21	NAK	15	00010101
22	SYN	16	00010110
23	ETB	17	00010111
24	CAN	18	00011000
25	EM	19	00011001
26	SUB	1A	00011010
27	ESC	1B	00011011
28	FS	1C	00011100
29	GS	1D	00011101
30	RS	1E	00011110
31	US	1F	00011111

Dec	**ASCII**	**Hex**	**Binary 76543210**
32	SP	20	00100000
33	!	21	00100001
34	"	22	00100010
35	#	23	00100011
36	$	24	00100100
37	%	25	00100101
38	&	26	00100110
39	'	27	00100111
40	(	28	00101000
41	)	29	00101001
42	*	2A	00101010
43	+	2B	00101011
44	,	2C	00101100
45	-	2D	00101101
46	.	2E	00101110
47	/	2F	00101111
48	0	30	00110000
49	1	31	00110001
50	2	32	00110010
51	3	33	00110011
52	4	34	00110100
53	5	35	00110101
54	6	36	00110110
55	7	37	00110111
56	8	38	00111000

57	9	39	00111001
58	.	3A	00111010
59	;	3B	00111011
60	<	3C	00111100
61	=	3D	00111101
62	>	3E	00111110
63	?	3F	00111111

Dec	ASCII	Hex	Binary 76543210
64	@	40	01000000
65	A	41	01000001
66	B	42	01000010
67	C	43	01000011
68	D	44	01000100
69	E	45	01000101
70	F	46	01000110
71	G	47	01000111
72	H	48	01001000
73	I	49	01001001
74	J	4A	01001010
75	K	4B	01001011
76	L	4C	01001100
77	M	4D	01001101
78	N	4E	01001110
79	0	4F	01001111
80	P	50	01010000
81	Q	51	01010001
82	R	52	01010010
83	S	53	01010011
84	T	54	01010100
85	U	55	01010101
86	V	56	01010110
87	W	57	01010111
88	X	58	01011000
89	Y	59	01011001
90	Z	5A	01011010
91	[	5B	01011011
92	/	5C	01011100
93	]	5D	01011101
94	^	5E	01011110
95	_	5F	01011111

Dec	ASCII	Hex	Binary 76543210
96	`	60	01100000
97	a	61	01100001
98	b	62	01100010
99	C	63	01100011
100	d	64	01100100
101	e	65	01100101
102	f	66	01100110
103	g	67	01100111
104	h	68	01101000
105	i	69	01101001
106	j	6A	01101010
107	k	6B	01101011

108	I	6C	01101100
109	m	6D	01101101
110	n	6E	01101110
111	a	6F	01101111
112	p	70	01110000
113	q	71	01110001
114	r	72	01110010
115	s	73	01110011
116	t	74	01110100
117	u	75	01110101
118	v	76	01110110
119	w	77	01110111
120	x	78	01111000
121	y	79	01111001
122	z	7A	01111010
123	{	7B	01111011
124	|	7C	01111100
125	}	7D	01111101
126		7E	01111110
127	DEL	7F	01111111

File Types

file_type	Preferred Use
$00	Typeless file (SOS and ProDOS)
$01	Bad block file
$02 *	Pascal code file
$03 *	Pascal text file
$04	ASCII text file (SOS and ProDOS)
$05 *	Pascal data file
$06	General binary file (SOS and ProDOS)
$07 *	Font file
$08	Graphics screen file
$09 *	Business BASIC program file
$0A *	Business BASIC data file
$0B *	Word Processor file
$0C *	SOS system file
$0D,$0E *	SOS reserved
$0F	Directory file (SOS and ProDOS)
$10 *	RPS data file
$11 *	RPS index file
$12 *	AppleFile discard file
$13 *	AppleFile model file
$14 *	AppleFile report format file
$15 *	Screen library file
$16-$18 *	SOS reserved
$19	AppleWorks Data Base file
$1A	AppleWorks Word Processor file
$1B	AppleWorks Spreadsheet file
$1C-$EE	Reserved
$EF	Pascal area
$F0	ProDOS added command file
$F1-$F8	ProDOS user defined files 1-8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file

```
$FC             Applesoft program file
$FD             Applesoft variables file
$FE             Relocatable code file (EDASM)
$FF             ProDOS system file
```

* Apple III SOS only; not used by ProDOS.

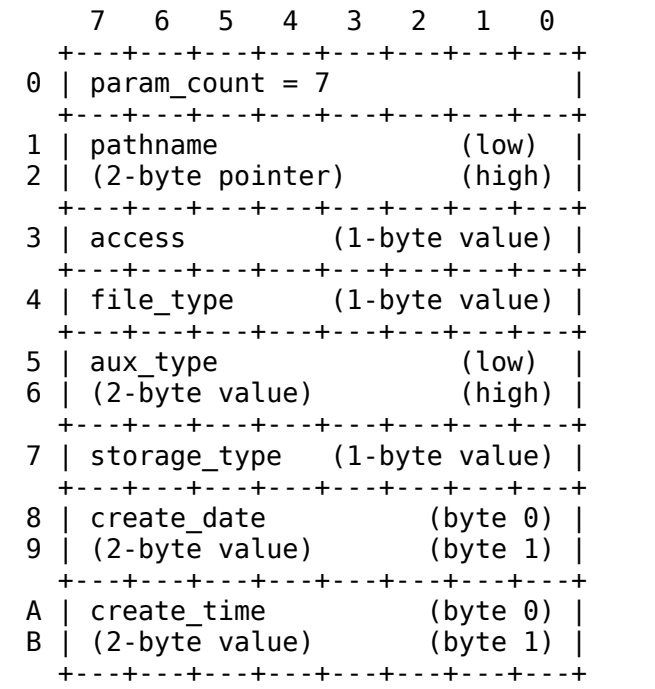
For the file_types used by Apple III SOS only, refer to the *SOS Reference Manual*.

MLI Error Codes

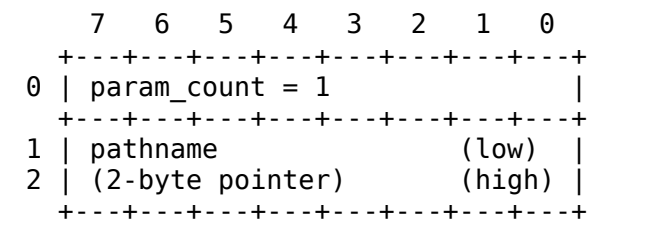
```
$00:    No error
$01:    Bad system call number
$04:    Bad system call parameter count
$25:    Interrupt table full
$27:    I/O error
$28:    No device connected
$2B:    Disk write protected
$2E:    Disk switched
$40:    Invalid pathname
$42:    Maximum number of files open
$43:    Invalid reference number
$44:    Directory not found
$45:    Volume not found
$46:    File not found
$47:    Duplicate filename
$48:    Volume full
$49:    Volume directory full
$4A:    Incompatible file format, also a ProDOS directory
$4B:    Unsupported storage_type
$4C:    End of file encountered
$4D:    Position out of range
$4E:    File access error, also file locked
$50:    File is open
$51:    Directory structure damaged
$52:    Not a ProDOS volume
$53:    Invalid system call parameter
$55:    Volume Control Block table full
$56:    Bad buffer address
$57:    Duplicate volume
$5A:    File structure damaged
```

Refer to Section 4.8 for a more detailed description of these error codes.

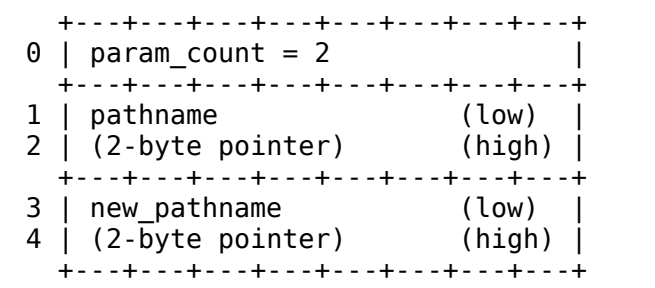
ProDOS MLI Calls



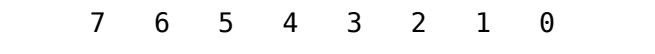
4.4.2 DESTROY ($C1)

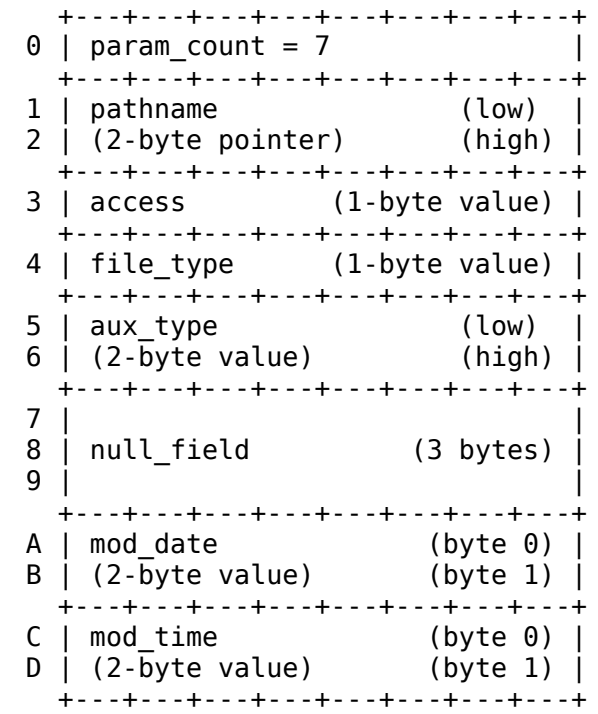


4.4.3 RENAME ($C2)

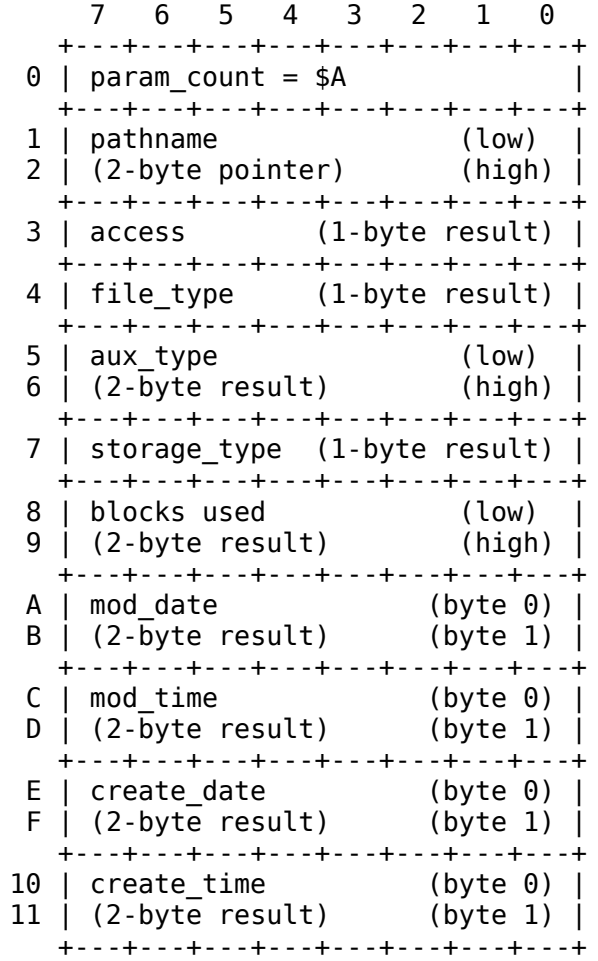


4.4.4 SET_FILE_INFO ($C3)





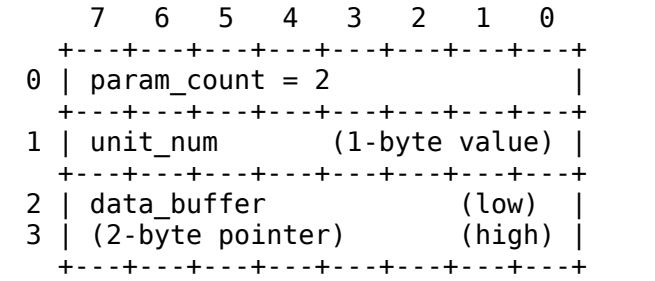
4.4.5 GET_FILE_INFO ($C4)



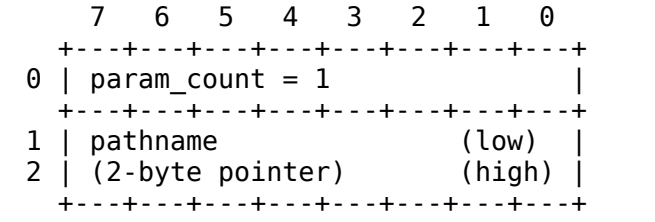
* When file information about a

volume directory is requested, the total number of blocks on the volume is returned in the aux_type field and the total blocks for all files is returned in blocks_used.

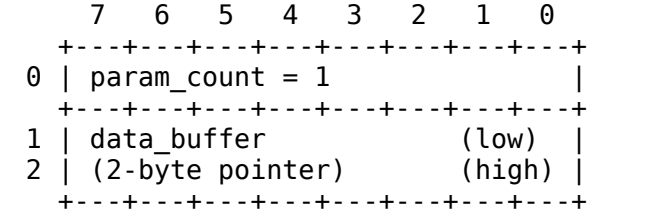
4.4.6 ON_LINE ($C5)



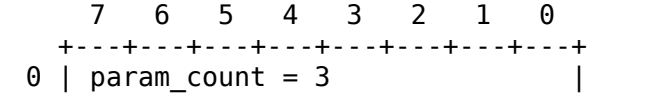
4.4.7 SET_PREFIX ($C6)

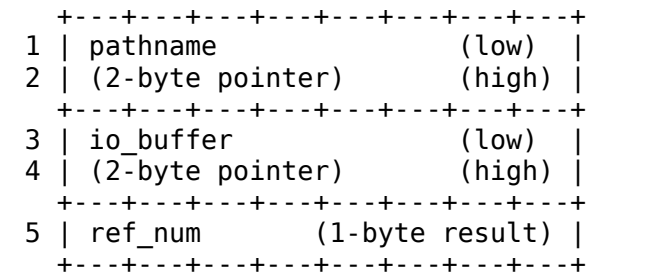


4.4.8 GET_PREFIX ($C7)

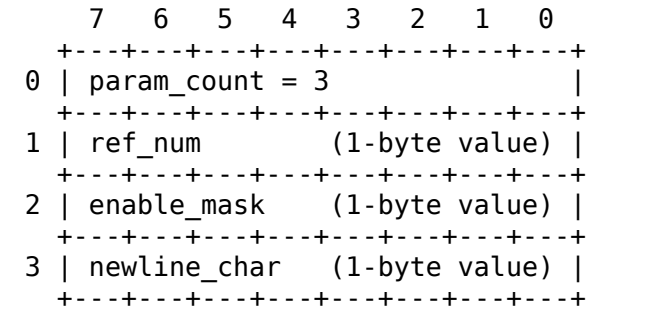


4.5.1 OPEN ($C8)

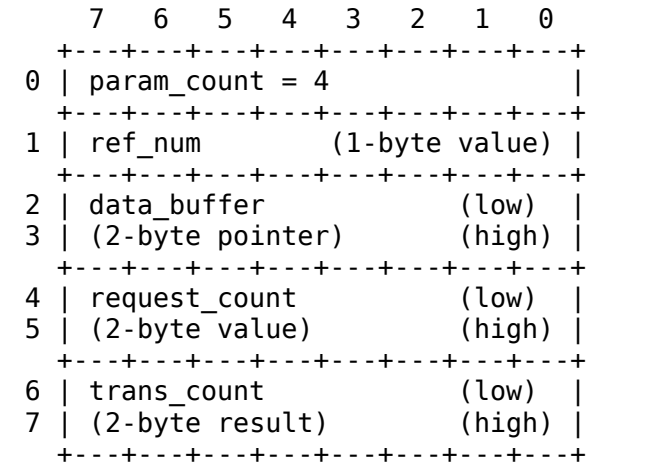




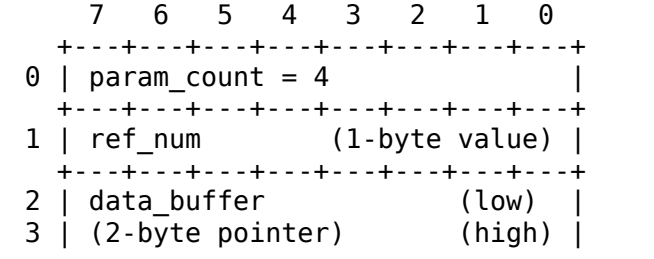
4.5.2 NEWLINE ($C9)

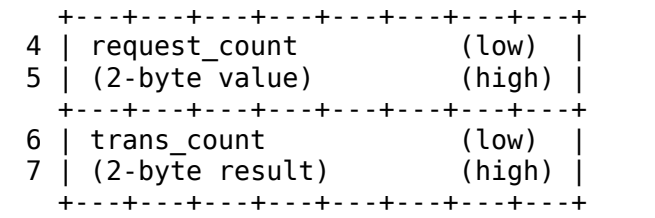


4.5.3 READ ($CA)

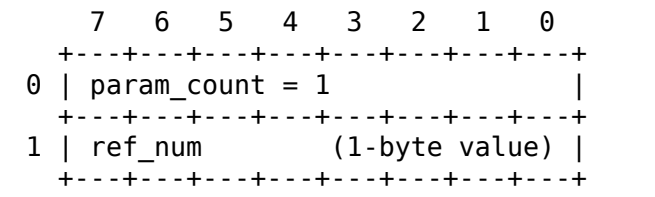


4.5.4 WRITE ($CB)

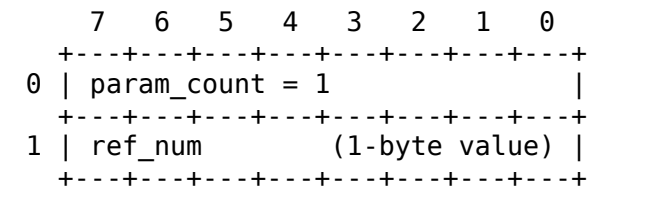




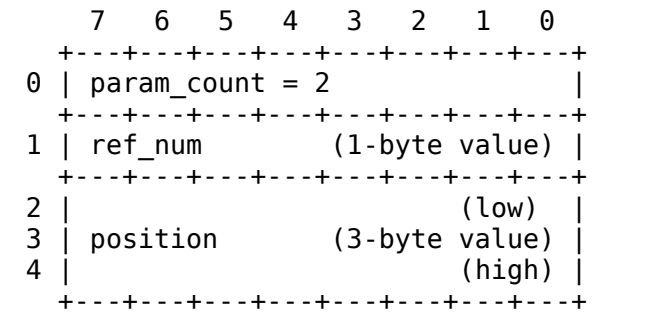
4.5.5 CLOSE ($CC)



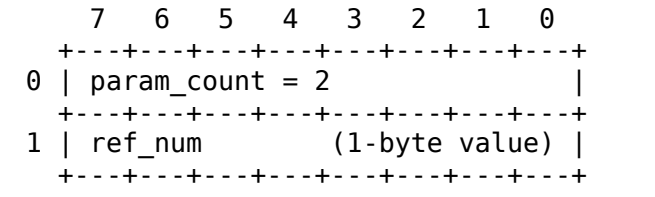
4.5.6 FLUSH ($CD)



4.5.7 SET_MARK ($CE)



4.5.8 GET_MARK ($CF)



```
 2 |                          (low)  |
 3 | position      (3-byte result) |
 4 |                          (high) |
   +---+---+---+---+---+---+---+---+
```

```
     7   6   5   4   3   2   1   0
   +---+---+---+---+---+---+---+---+
 0 | param_count = 2               |
   +---+---+---+---+---+---+---+---+
 1 | ref_num        (1-byte value) |
   +---+---+---+---+---+---+---+---+
 2 |                          (low)  |
 3 | EOF            (3-byte value) |
 4 |                          (high) |
   +---+---+---+---+---+---+---+---+
```

4.5.9 SET_EOF ($D0)

```
     7   6   5   4   3   2   1   0
   +---+---+---+---+---+---+---+---+
 0 | param_count = 2               |
   +---+---+---+---+---+---+---+---+
 1 | ref_num        (1-byte value) |
   +---+---+---+---+---+---+---+---+
 2 |                          (low)  |
 3 | EOF            (3-byte value) |
 4 |                          (high) |
   +---+---+---+---+---+---+---+---+
```

4.5.10 GET_EOF ($D1)

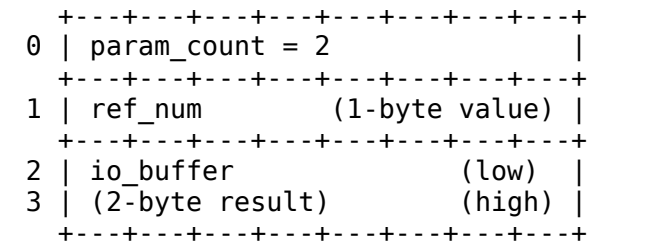
```
     7   6   5   4   3   2   1   0
   +---+---+---+---+---+---+---+---+
 0 | param_count = 2               |
   +---+---+---+---+---+---+---+---+
 1 | ref_num        (1-byte value) |
   +---+---+---+---+---+---+---+---+
 2 |                          (low)  |
 3 | EOF           (3-byte result) |
 4 |                          (high) |
   +---+---+---+---+---+---+---+---+
```

4.5.11 SET_BUF ($D2)

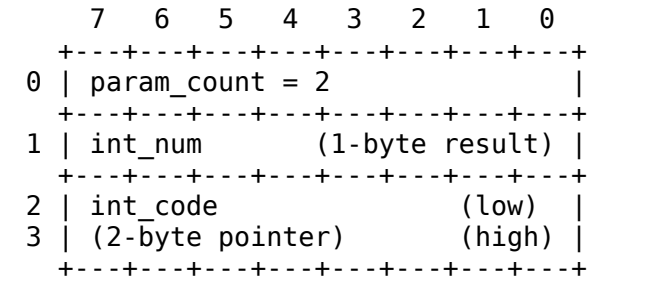
```
     7   6   5   4   3   2   1   0
   +---+---+---+---+---+---+---+---+
 0 | param_count = 2               |
   +---+---+---+---+---+---+---+---+
 1 | ref_num        (1-byte value) |
   +---+---+---+---+---+---+---+---+
 2 | io_buffer                (low)  |
 3 | (2-byte pointer)         (high) |
   +---+---+---+---+---+---+---+---+
```

4.5.12 GET_BUF ($D3)

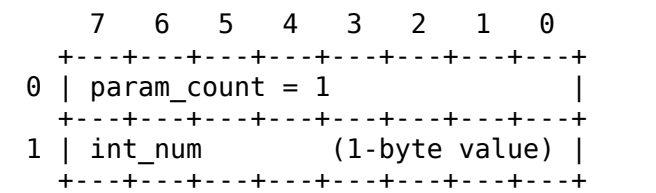
```
     7   6   5   4   3   2   1   0
```



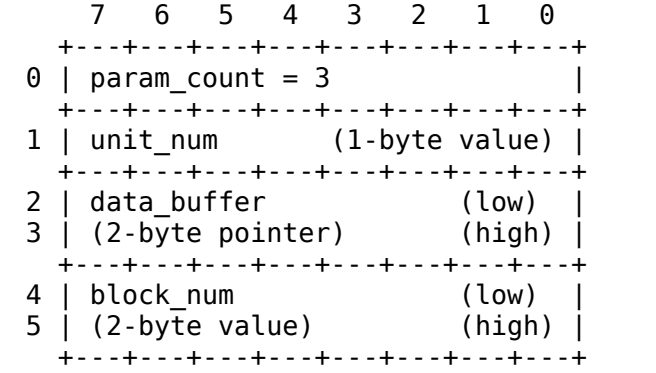
4.6.2 ALLOC_INTERRUPT ($40)



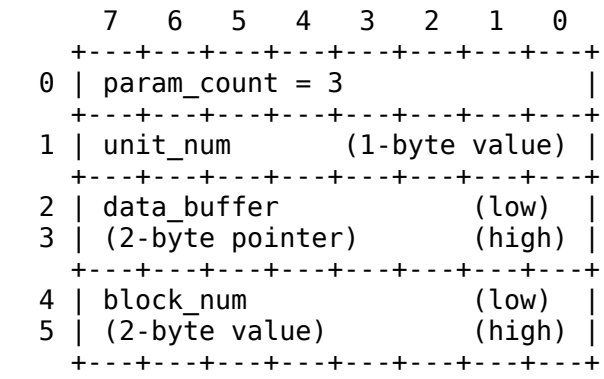
4.6.3 DEALLOC_INTERRUPT ($41)



4.7.1 READ_BLOCK ($80)



4.7.2 WRITE_BLOCK ($81)



Errors in this manual

The following errors were noted in
this manual and faithfully reproduced:

- page xi: two consecutive sections labeled B.4.2.3
- page 2: the caption for Figure 1-1 is missing
- page 24: memory map lists $300 twice
- page 28: "management" misspelled as "mangagement"
- page 60: param_count is missing "(1-byte value)"
- page 70: param_count is missing "(1-byte value)"
- page 83: memory map lists $300 twice
- page 95: "unprotected" misspelled as "uprotected"
- page 99: "calendar" misspelled as "calender"
- page 108: "the the" instead of "the"
- page 109: "calendar" misspelled as "calender"
- page 111: the two routines are in each other's position
- page 114: "interruptible" misspelled as "interruptable"
- page 114: some text appears to be missing after 6.3.2
- page 119: memory map lists $300 twice
- page 125: "Temporary" misspelled as "Temporory"
- page 131: address of RSHIMEM is BEF8 and should be BEFB
- page 135: "inspecting" misspelled as "inpecting"
- page 147: "directory" misspelled as "drectory"
- page 183: "calendar" misspelled as "calender" three times
- page 184: both "endtry_length" and "entry_length" with different page numbers
- Quick Reference Card: tilde (~) missing from ASCII table

If you discover other errors in this manual, either in the paper version or
in this online version, then please post that information on USENET in the
comp.sys.apple2.programmer newsgroup.