From The Editors of COMPUTE! Magazine and Optimized Systems Software, Inc.

INSIDE ATARI DOS

Compiled by Bill Wilkinson,
Optimized Systems Software, Inc.

Published by **COMPUTE! Books,**
A Division of Small System Services, Inc.,
Greensboro, North Carolina

COMPUTE! Books is a division of Small System Services, Inc.,
Publishers of **COMPUTE!** Magazine
~~Editorial Offices are located at:~~
~~625 Fulton Street, Greensboro, NC 27403 USA. (919)275-9809~~

~~Optimized Systems Software, Inc., is located at:~~
~~10379 Lansdale Avenue, Cupertino, CA 95014 USA. (408)446-3099.~~

Return to Table of Contents | Previous Chapter | Next Chapter

Preface

This book contains the only complete and official listings for the disk File Manager System (FMS) commonly known as "Atari DOS 2.0S." You will note that we have clearly stated that the purchase of this book does not entitle you to make, sell, give, or otherwise distribute copies of either the original Atari DOS 2.0S or any modified version you may produce as a result of using this book.

By way of information, should you desire to produce and distribute a modified version of this product (e.g., to support a new disk drive), you must sign a contract and licensing agreement with the party who owns the rights to grant such licenses for non-exclusive uses. Currently, Optimized Systems Software is the only entity able to grant such licenses.

Some of you may find it strange that the publishers of **COMPUTE!** magazine are publishing this book. You might wonder why Atari, Inc., hasn't released this information before. Why can you only obtain distribution rights from Optimized Systems Software? For the answers to these and other questions we present the following Introduction, an historical perspective on the development of the systems software for the Atari Home Computers.

Notes on the Web Edition of Inside Atari DOS

Because different companies hold the copyright on the text in this book and the Atari DOS source code, this online version of Inside Atari DOS includes the full text but does not include the actual source code.
-Kevin Savetz, November 27 2002

Printed in the United States of America

ISBN 0-942386-02-7

10 9 8 7 6 5 4 3 2

Return to Table of Contents | Previous Chapter | Next Chapter

Introduction
BEING A HISTORY OF TWO BIRTHS "COLEEN" AND "CANDY"

I don't know exactly when the concept of the Atari Computer was developed within the corporate mind of Atari, Inc., nor do I know all of the people responsible for nursing that concept into reality. The following history covers the relationship with Atari, Inc., during the evolution of the system software.

Sometime in early 1978, when the Atari 800 and 400 were still called "Coleen" and "Candy" and were still in the breadboard stages, Atari bought a copy of the source for Microsoft 8K BASIC. This version of BASIC was fundamentally the same product that was implemented by Commodore in the early PETs, was used by OSI, and was a close ancestor of Applesoft. Six months and many, many Atari man-hours later, that 8K BASIC was almost functioning properly on the Atari prototypes. But buying source for a program buys you just that: source. Generally, you also receive little documentation, sometimes obscure code, no guide to modification, and no real support. What to do? The products were due to be shown in early January, 1979, at the Consumer Electronics Show (CES) in Las Vegas, Nevada.

Enter Shepardson Microsystems, Inc. (SMI), my employer at that time. Though little known by the microcomputer public, SMI had already produced some very successful, private labeled microcomputer software. Among our better-known efforts were the original Apple DOS, Cromemco 16K Extended BASIC, and Cromemco 32K Structured BASIC (just being completed at that time). Also, we had done some work for Atari on a custom game processor. (Which used a 12-bit ROM and 5-bit RAM configuration and was well received at Atari, hut never produced.)

Coincidentally, about that same time SMI had also purchased source for Microsoft 6502 BASIC. After producing Apple's DOS, we had the bright idea of mating the Apple II peripheral bus with the KIM/SYM/AIM system bus (and it still seems like a good idea to us, but ...). The idea was to provide a disk system (Apple's) to the Single Board Computer market. Needing a BASIC to sell with the system, we plunked down a few grand and purchased Microsoft's. Though it looked to us like it would be difficult to modify, we were intending to resell it with a minimum of changes, so it seemed appropriate.

A New BASIC?

Re-enter Atari, some time in the late summer of 1978, asking if SMI could help them. With Microsoft BASIC? Well ... we really didn't want to, but ... Could we propose a new BASIC? We talked. And had meetings, and a study contract, and more meetings, and finally we wrote a specification for a 10K, ROM-based BASIC. (I still have a copy of that spec, and it's amazing how little the final version deviated from that original.)

Of course, in the middle of all these discussions, Atari naturally divulged how their (truly superb) ROM-based Operating System would interface both with BASIC and with various devices. Somewhere in here, my memory of the sequence of events and discussions becomes a little unclear, but suffice it to say that we found ourselves making a bid on producing not only a BASIC for Atari, hut also the File Manager (disk device driver) which would change Atari OS to Atari DOS.

Sometime in late September, 1978, the final proposal was made to Atari, and it was accepted by them shortly thereafter. In mid-October, 1978, we received the go-ahead. The project leader was Paul Laughton, author of Apple DOS. The bulk of the work ended up being done by Paul and Kathleen O'Brien. Though I was still involved in the finishing touches on Cromemco BASIC, I take credit for designing the floating point scheme used in Atari BASIC. Paul Krasno implemented the math library routines following guidelines supplied to us by Fred Ruckdeschel (author of the acclaimed text, BASIC Scientific Subroutines). And, of course, much credit must go to Mike Peters, our combination keypuncher/computer operator/junior programmer/troubleshooter.

Since we obviously couldn't have the Atari machines to work on (they hadn't been built yet), the first step was to bring up an emulator for Atari's CIO ("Central Input-Output," the true heart of Atari's OS) on our Apple II systems. With Paul Laughton leading the way (and doing a lion's share of the work), the pieces fell

together quickly. Little things had to be overcome: the cross-assembler was modified to handle the syntax table pseudo-ops, the 256-byte Apple disk sectors had to be made to look like 128-byte Atari sectors, the BASIC interpreter seemed to function, but was waiting for the floating point routines. And there are funny things to tell of, also. Like our cross-assembler, running on an IMP-16P (a 1973 vintage, 16-bit, bit-sliced PMOS microprocessor) that used keypunched cards for input, a floppy disk (with no DOS) as temporary storage, and a paper tape punch as output.

Somehow, Kathleen and Paul guided the two programs unerringly toward completion. On December 28, 1978, Atari's purchasing department at last delivered a signed copy of the final purchase order. It called for delivery of both products by April 6, 1979. There was a clause which provided for a $1,000 per week incentive (if we finished early) and penalty (if we finished late). What is especially humorous about that December 28th date is that the first working versions of both BASIC and FMS had already been delivered to Atari over a week before! That is fast work.

Fortunately, then, Atari took their new Atari BASIC to CES. Unfortunately, there was a limit on the amount of incentive money collectible. Oh, well.

In the months that followed, SMI fixed bugs, proofread manuals, and worked on other projects (including the Atari Assembler/Editor, which was mostly Kathleen's effort). The nastiest bugs in BASIC were fixed by December, 1979, but it was too late: Atari had already ordered tens of thousands of BASIC ROMs. The FMS bugs were easier to get fixed, since DOS is distributed on disk.

In mid-1980, Paul Laughton once again tore into FMS. This time, he modified it to handle the ill-fated 815 double-density disk drive and added "burst I/O" (and there will be much more about both these subjects in the technical discussion that follows).

In late 1980, and early 1981, Bob Shepardson, owner of Shepardson Microsystems, Inc., decided that the pain and trouble of having employees wasn't justified by the amount of extra income (if any) that he derived. Though we still occasionally function in a loose, cooperative arrangement, the halcyon days of SMI seem to be over.

A New Beginning

I negotiated with Bob Shepardson for his rights to the Atari products (FMS, BASIC, and the Assembler/Editor) and their Apple II counterparts. Thankfully, Atari had purchased from SMI only a non-exclusive right to distribute these products. SMI had retained the rights to license other users on a similar non-exclusive basis (and, indeed, SMI sold a version for the Apple II during most of 1980).

So now it was frantic time again: this was February 25, 1981, and the West Coast Computer Faire was April 3rd. But our brand new company, Optimized Systems Software, arrived on time, bringing with it BASIC A +, OS/A + and EASMD. All three were enhanced, disk-based versions of the original Atari programs (and, in fact, derived some of their enhancements from the previous OSS Apple II products).

The products have been well received by the Atari user community, in part due to the fact that they are truly compatible, yet enhanced, versions of standard Atari software.

Why This Book?

The decision to publish these listings was not an easy one to make; and it is, in its own way, an historic occasion. After all, have you ever seen anyone offering source or listings of CP/M, the most popular of all computer operating systems? Since Atari, to their credit, has honored the original agreement with SMI and not released either source or listings without permission, the responsibility for doing SO seemed to rest with OSS.

But Atari has set a powerful precedent by publishing the listings of DUP (their portion of DOS 2.0S) and the OS ROMs. The clamor from Atari users for the source for FMS finally even reached us, so we have bowed to the inevitable, and honored the same commitment that Atari has made: to release as much information and aid as possible to the user community.

We hope that the users will appreciate these efforts and, in turn, respect our rights and Copyrights. As long as there is a mutual respect and benefit, you, the user, can expect continued support.

About This Book

With the release of this book, the dedicated Atari enthusiast can examine all the inner workings of Atari DOS and modify his (or her) system to his heart's delight. Rather than simply publish listings, we have chosen also to provide a complete guide to the workings of FMS.

Although the listing itself is relatively clear and commented, all but the most expert would have trouble plowing through some of the tortuous logic necessary in such a program. The guide included here describes all aspects of the FMS, including the external view, the charts and tables, the various interfaces, and (in copious detail) the functions of the individual subroutines (including complete entry and exit parameters).

There is much of value here even for the person who never intends to modify Atari DOS. We feel that FMS is a

fairly well-structured, relatively sophisticated, system level assembly language program. We hope that most users will gain by the insights presented here.

We would welcome any notes you would care to send pointing out errors either in the DOS or in this book.

Bill Wilkinson
Optimized Systems Software
Cupertino, California
February, 1982

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter One
ATARI DOS OVERVIEW

The standard Atari Disk Operating System, DOS 2.0S, consists of four separate elements, ranked as follows in order of their visibility to the average DOS user.

1. DUP – Disk Utility Package
2. CIO – Central Input/Output
3. FMS – File Management System
4. SIO – Serial Input/Output

It is helpful to understand the entire Input/Output (I/O) process. While this book is intended to give detailed information on the workings of FMS, this overview will attempt to at least show how the four elements of DOS are connected. To this end, we would first call your attention to Figure 1. This figure is, itself, an overview of the entire Atari I/O system, including indications as to how and where data and control flows between the various elements thereof. Figures 1-1 through 1-4 show close-ups of portions of this diagram as they relate to the four elements of DOS.

In these figures, the rectangular boxes represent system elements, and are appropriately labeled. The wide, lettered arrows represent the flow of data (via buffers, control blocks, or even registers) between the various elements. The narrow, numbered arrows show how and where control, and control information, is transferred.

1-1. Disk Utility Package

DUP (which shows as DUP.SYS in a disk directory listing) is the most obvious and visible element of Atari DOS. DUP's function is to provide the user with keyboard access to the various file management functions in FMS. It does so via the menu which is displayed when, for example, the user keys DOS from BASIC. Actually, the menu offers several options which are not directly a part of the FMS (e.g., copy and duplicate files). Refer to the Atari *Disk Operating System II Reference Manual* (part number C016347) for more information.

DUP is not an integral part of FMS. DUP may be relatively easily replaced with a program of the user's choice. In fact, our own OS/A + does exactly that: instead of a menu, the user is given a command-driven keyboard interface to the other elements of DOS.

DUP is not even a privileged portion of DOS (excepting, perhaps, for needing to know a little of the internals of FMS when it performs a Duplicate Disk function). Any user application program (and that includes Atari BASIC, BASIC A +, EASMD, and many, many more) interacts the same way DUP does. Figure 1-1 shows the proper flow of control in DOS. Note that DUP transfers control only to CIO, which, in turn, transfers control to FMS and thence to SIO. An application program which maintains this protocol should he able to perform correctly in any Atari system, regardless of the revision of the OS ROMs and/or FMS.

Of course, control is not the only thing which DUP must transfer. It must also tell CIO where its data is and what to do with it. Refer to Figure 1-2 for a diagram of the complete application/CIO interface (again, it is labeled in this way because DUP is just another application program as far as the rest of DOS is concerned). CIO always expects an Input/Output Control Block (IOCB) and usually (i.e., for all but the simplest operations) needs a buffer into or out of which it may perform its operations.

1-2. Central Input/Output

CIO is actually the heart of the entire Atari Computer. It is less than 800 bytes long and yet serves to handle virtually all the input and output which takes place in the computer. CIO is a part of the Atari OS ROMs, the 10K byte package which also houses the floating point routines, the default character set, the interrupt handlers, and several device drivers.

The entire set of operations summarized in Figure 1-2 is covered in detail in the Atari OS Manual (C01655) and will he covered only briefly here. Readers of **COMPUTE!** will also find some helpful material on this subject in issues #18 through #21 (November, 1981, through February, 1982) in the INSIGHT: ATARI columns.

In order to allow easy control and data flow, CIO is written to expect and provide for eight Input/Output Control Blocks (IOCBs) which are used to pass the information needed to process the various kinds of I/O requests. An application places the necessary command and control information in an IOCB which it selects (data path A). If a buffer is required, the application must provide one (data path C) and place its address into the IOCB. When ready to execute the I/O command, the application places the IOCB number (times 16) in the 6502's X-register (data path C) and executes a JSR call to CIO (control path 1). Note that a few command variations may pass data via the 6502's A-register, but we may consider that simply a special case location of the user's buffer.

When CIO receives control, it examines the information in the IOCB (and, for some operations, in the user buffer) to determine what actions it is to perform. Generally, this action requires the execution of a device handler routine.

A device handler (interchangeably known as a device driver) is a system routine that performs I/O operations for a specific device (or class of devices). Examples of device handlers include the "P:" driver (the printer) and the "E:" driver (the screen/keyboard editor). Figure 1-3 illustrates the interface between CIO and the various device handlers. Note that FMS is simply another device handler as far as CIO is concerned, having been given the name "D:".

All device drivers are required to contain a table of address pointers (known as the Device Vector Table) to various specific routines within themselves, including a device OPEN routine, GET CHARACTER routine, etc. The name of a device and the address of this table is placed in CIO's Device Handler Table. When an application program makes an I/O request to CIO for a specific device, CIO searches the Device Handler Table for the given name and corresponding Device Vector Table address. With the thus-located vector table, CIO can then call the appropriate device handler routine (via a JSR, along control path two of Figure 1-3).

1-3. File Management System

As stated above, FMS is actually simply another device driver as far as CIO is concerned. The control and data flows shown in Figure 1-3 are equally valid for all device drivers in the Atari system. Note that many of the drivers in the default ("as-shipped") system reside entirely within the so-called OS ROMs. Although it resides in RAM, what is somewhat unique about FMS is that the Atari system initialization code contains a segment of "boot" code which loads FMS into memory upon power-on.

FMS is the system device handler for all I/O operations that specify the device name "D" (including "D1:", "D2:", etc.). In order to perform its functions, FMS examines the data in the specified IOCB (data path F). It may also examine, read, or write data to or from the user-supplied buffer (data path I). Data path H is used to pass the IOCB-designator (again, via the X-register) and single-byte transfer data (via the A-register).

FMS is called upon to perform a variety of tasks, including all disk I/O, file renaming, protecting, deleting, etc. Since the rest of this book consists of a listing of FMS along with detailed explanations of all sections thereof, we will not now dwell on the inner workings of FMS.

However, we do need to note that, in order to perform its work, FMS must transfer data to and from the disk. FMS accesses the disk drive via SIO, the fourth element of DOS.

1-4. Serial Input/Output

SIO is the name given to the component of DOS which drives and controls the Atari serial I/O bus and the various peripherals (disk, printer, modem, etc.) which are placed on that bus. Figure 1-4 illustrates the interface between FMS and SIO, but it could just as well serve to show (for example) how the printer driver talks to the various Atari printers.

The SIO is primarily driven by a request placed in SIO's Device Control Block (DCB) by the device handler (data path K) followed by a transfer of control (control path three) via a JSR. SIO uses the information in the DCB (data path M) to determine what it needs to do. If the DCB specifies a serial bus data transfer (as opposed to, for example, a status request), then the address of the data buffer must also be passed (via a field in the DCB). For example, the FMS buffer shown is accessed via data paths J (from FMS) and L (from SIO).

Although SIO only understands the single system DCB, the buffer specified may be located anywhere in memory. FMS takes advantage of this to implement "burst I/O" (discussed in section 12), which has SIO transferring data directly to or from the user's buffer (data path E).

Since the actual disk data transfer occurs in fact within the 810 disk drive and, since SIO communicates to the drive via data path N, one might reasonably argue that the disk drive constitutes a fifth component of DOS. However, because the disk drive functions are preprogrammed in ROM, and because SIO implements the only method of accessing the disk (as well as most other peripherals), then, for all practical purposes, even machine language software may treat SIO as the last link in the I/O chain on the Atari Computers.

Once again, we remind you to study Figure 1. In the following dissertation and dissection of FMS, we shall refer to this chart often.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Two
DISK ORGANIZATION

The purpose of FMS is to organize the 720 data sectors available on an 810 diskette into a system of named data files. FMS has three primary data structures that it uses to organize the disk: the Volume Table of Contents, the Directory, and Data Sectors. The Volume Table of Contents is a single disk sector which keeps track of which disk sectors are available for use in data files. The Directory consists of directory sectors. It is used to associate file names with the location of the files' sectors on the disk. Each Directory entry contains a file name, a pointer to the first data sector in the file, and some miscellaneous information. The Data sectors contain the actual data and some control information that link one data sector to the next data sector in the file. Figure 2-1 illustrates the relation between the Directory and the Data files.

Disk Directory

The Directory starts at disk sector $169 and continues for eight contiguous sectors, ending with sector $170. These sectors were chosen for the directory because they are in the center of the disk and therefore have the minimum average seek time from any place else on the disk. Each directory sector has space for eight file entries. Thus, it is possible to have up to 64 files on one disk.

A Directory entry is 16 bytes in size, as illustrated by Figure 2-2. The directory entry flag field gives specific status information about the current entry. The directory count field is used to store the number of sectors currently used by the file. The last eleven bytes of the entry are the actual file name. The primary name is left justified in the primary name field. The name extension is left justified in the extension field. Unused filename characters are blanks ($20). The Start Sector Number field points to the first sector of the data file.

Data Sectors

A Data Sector is used to contain the file's data bytes. Each 128 byte data sector is organized to hold 125 bytes of data and three bytes of control information as shown in Figure 2-3. The data bytes start with the first byte (byte 0) in the sector and run contiguously up to, and including, byte 124. The control information starts at byte 125.

The sector byte count is contained in byte 127. This value is the actual number of data bytes in this particular sector. The value may range from zero (no data) to 125 (a full sector). Any data sector in a file may be a short sector (contain less than 125 data bytes).

The left six bits of byte 125 contain the file number of the file. This number corresponds to the location of the file's entry in the Directory. Directory entry zero in Directory sector $169 has the file number of zero. Entry one in Directory sector $169 has the file number one - and so forth. The file number value may range from zero to 63 ($3F). The file number is used to insure that the sectors of one file do not get mixed up with the sectors of another file.

The right two bits of byte 125 (and all eight bits of byte 126) are used to point to the next data sector in the file. The ten bit number contains the actual disk sector number of the next sector. Its value ranges from zero to 719 ($2CF). If the value is zero, then there are no more sectors in the file sector chain. The last sector in the file sector chain is the End-Of-File sector. The End-Of-File sector may or may not contain data, depending upon the value of the sector byte count field.

Volume Table Of Contents (VTOC)

The VTOC sector is used to keep track of which disk sectors are available for data file usage. The VTOC sector is located at sector $168. Figure 2-4 illustrates the organization of the VTOC sector. The most important part of the VTOC is the sector bit map.

The sector bit map is a contiguous string of 90 bytes, each of which contains eight bits. There are a total of 720 (90 x 8) bits in the bit map – one for each possible sector on an 810 diskette. The 90 bytes of bit map start at VTOC byte ten ($0A). The leftmost bit ($80 bit) of byte $0A represents sector zero. The bit just to the right of the leftmost bit ($40 bit) represents sector one. The rightmost bit (bit $01) of byte $63 represents sector 719.

The fact that FMS interprets the bit map as representing sectors zero through 719 is a bug. The Atari 810 disk drive will not accept commands for sector zero. It will accept commands for sector 720. In other words, the bit map is skewed by one. The problem cannot be fixed now because there are already tens of thousands of diskettes whose bit maps are to be interpreted as representing sectors zero through 719, and because some savvy applications writers have taken advantage of this feature. (A bug which generates useful side effects is known in the programming profession as a feature.) Sector 720 can never be used by FMS and is therefore available for miscellaneous purposes.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Three
FMS FILE CONTROL BLOCKS (FCB)

The FMS File Control Blocks are used to store information about files that are currently being processed. Each file that is being processed concurrently by FMS requires one FCB. Since the Atari system has eight IOCB's, FMS must be prepared to handle up to eight files concurrently, thus there are eight FCBs. The FCBs were designed to have a one-to-one correspondence with the IOCBs. When a file is to he processed with IOCB number three, FMS will use FCB number three for that file. When a file is to be processed with IOCB number five, FMS will use FCB number five for that file. Each FCB is the same size as an IOCB (16 bytes). The FCBs are located in a contiguous RAM area just like the IOCBs When CIO calls FMS, the X register contains the displacement (IOCB number times 16) to the IOCB making the request. The FMS uses this displacement value to access both the IOCB information and the FCB information. Please refer to the listing at location $1381 for the following discussion about the FCBs.

FCBFNO

The file number of the file currently being processed. The value (zero to 63) is shifted left two bits. When a file has been opened for reading, this value will be used to check for a file number mismatch in the data sectors. When a file is opened for write, this value will he placed in the file number field of the data sectors.

FCBOTC

Open Type Code. This value is used as a flag to indicate which mode the file has been opened for:

Input is $04.
Output is $08.
Update is $0C.
Append is $01.
Directory read is $02.

FCBSLT

This is a flag used to indicate that the file being processed was created by DOS 1 rather than DOS 2. The Data Sector length byte has a different interpretation under DOS 1.

FCBFLG

This field is a working flag. If the value is $80, then the file is eligible to acquire new data sectors. Files that are opened for Output or Append are eligible to acquire new data sectors. If the value is $40, then the sector currently is in a memory buffer, has been modified, and needs to be written back to the disk.

FCBMNL

If the file is opened for Output or Append, this value will be either 125 or 253 depending upon the drive type. The 253 value is meant for the Atari 815 dual density drive. If the file is opened for Read or Update, then this value represents the number of data bytes that are in the data sector currently in a buffer. This value is obtained from the Data Sector data length field (byte 125 of the data sector.)

FCBDLN

This value points to the next data byte to be operated on in a data sector. If the file is opened for Output or Append, this value points to the next available (unused) data byte in the current data sector. If the file is opened for Update, then this value points to the next data sector byte to be either read or modified. If the file is opened for Input, then this value points to the next byte to he read.

FCBBUF
This value is an index into the sector buffer table. The sector buffer table is a list of buffer addresses. When a file is being processed, a sector buffer is required to hold data sectors. This field tells FMS which FMS buffer has been allocated to the file.

FCBCSN
The sector number of the sector currently in the buffer is stored in this field.

FCBLSN
The sector number of the next sector in the tile chain is stored in this field.

FCBSSN
If the file has been Opened for Append, then this field contains the sector number of the start of the sectors to be appended to the file when the append file is closed.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Four
FMS IN ITIALIZATION

DUP gets control whenever the system is booted or the RESET key is pressed. DUP will call the FMS initialization routine, DINIT at $7E0.

DINIT

Functions:

1) Determine how many (and what type of) disk drives will be used.
2) Set up a drive table and allocate a drive buffer for each drive.
3) Allocate sector buffers and build the sector buffer table.
4) Clear the FCBs to zero.
5) Set MEMLO.
6) Enter the D: device into the Device Handler Table.
7) Exit to caller via RTS.

Drive Determination

The DRVBYT byte at $70A is used to tell FMS how many disk drives will he used and what the drive number of the drives will be. The rightmost bit (bit $01) indicates drive 1. The next left bit ($02) indicated drive 2 [illegible] and so forth. If the bit is one, then the drive is to be used. If the drive is zero then the drive is not to be used. The code will allocate up the eight drives, even though the 810 hardware only has switches for drives 1,2,3 and 4.

If DRVBYT indicates that a drive is to be used, then FMS issues a status command to that drive to determine if it is active and what type (810 or 815) of drive it is.

Drive Allocations

The drive determination process sets up two tables (Figure 4-1). The first table is the DRVTBL. This table is indexed into by the drive number (minus one). If the value in the table is zero then the drive is not to be used. If the value is one, then the drive is an active 810 and requires one drive buffer. If the value is two, then the drive is an 815 and requires two 128 byte buffers.

The second table is the drive buffer table. The drive buffer table contains the address of the drive buffer to be used for each drive. This Drive Buffer will be used to hold the VTOC sector on the diskette in the drive. The table is separated into two sections: DBUFAL contains the least significant address byte and DBUFAH which contains the most significant address byte. The drive buffer table is also accessed by the drive number (minus one).

When a file is being processed, the Drive number is obtained from the IOCB Device Number field, ICDNO. The obtained value is decremented by one and is then used as an index into the Drive Tables. The Drive Type is copied from the DRVTBL entry to DRVTYP ($12FE) for easy access by FMS. The Drive Buffer address is copied from the DBUFAL and DBUFAH table entries to the zero page drive buffer pointer, ZDRVA ($45).

Sector Butter Allocations

The SABYTE at location $709 is used to inform FMS about the number of 128 areas to he allocated as sector buffers. One 128 buffer is required for each file which is to he processed concurrently on 810 drives. Two 128 byte buffers are required for each file which is to be processed concurrently on 815 drives.

The Sector Buffer Allocation table, SECTBL at $1319, is used to indicate if a buffer is available for allocation to a file (Figure 4-2). If a buffer is available, the entry is set to zero. If the buffer is not available, the entry is a minus value. The table is 16 bytes in size and therefore can be used to allocate up to sixteen 128 byte buffers. During the initialization process, entries which are to be unused are set to a minus value.

The Sector Buffer Address Table is a table of addresses which point to the individual sector buffers. The table is divided into two parts: SABUFL contains the least significant address byte, SABUFH contains the most significant address byte.

When a file is being processed, an available buffer number is found in SECTBL by search for a zero valued

entry. The located buffer is allocated to the file by entering a minus value ($80) into the table and placing the corresponding buffer number into the DCB buffer number field, FCBBUF. When the file processing is done, the buffer is deallocated by setting the SECTBL entry to zero.

Setting MEMLO

The Atari MEMLO location ($2E7) is set after the FMS buffers have been allocated. The address of the last sector buffer allocated is incremented by 128. This value is then placed into MEMLO.

Device Handler Table Entry

The Device Handler Table ($31A) is searched for a "D" entry or the first (from the top) empty entry. When an appropriate entry is found, FMS inserts (or reenters) "D" as a DEVICE NAME and sets the DEVICE vector entry to point to the FMS Device Vector table at DFMSDH ($7CB).

Chapter Five
FMS ENTRY

The Device Vector Table for FMS is located at DFMSDH ($7CB). The address of this table is placed in the Device Handler Table by the FMS Initialization routine. When CIO needs to call an FMS function (Figure 1, control path 2), it will locate the address of the function via the table at DFMSDH. This table is the standard Atari Device Handler Vector Table. The six entries are for:

Open
Close
Get Byte
Put Byte
Status
Device Dependent (XIO) Commands

Each of the six FMS entry points starts with a subroutine call to the FMS SETUP routine. SETUP ($1164) prepares FMS parameters to deal with the particular task to he performed.

SETUP

Address ◆$1164
Entry Registers ◆ A = Possible ◆Put Data' data byte.
X = IOCB number times 16.
Y = Don't Care.
Exit Registers ◆ A = Unknown.
X = IOCB number times 16.
Y = Sector Buffer Index

Functions:

1) Initialize ERRNO to $9F. This value will be used in the FMS exit routines to form a FMS error number in the event of error.
2) Save the X Register in CURFCB. This value will be used as an index to the proper IOCB and the proper FCB for the current operation.
3) Save the value of the stack register as it was upon entry to FMS. This value will be used in the FMS exit routine.
4) Set up drive information values from the drive number contained in the zero page IOCB field ICDNOZ.
5) Allocate a sector buffer to the FCB if one is not already allocated.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Six
FMS EXIT

There are two types of FMS exits: the normal exit and the error exits. Both of these exit types end up calling the RETURN routine.

RETURN

Address ? $12D3

Entry Registers ? A = Return Code
X = Don't Care.
Y = Don't Care

Exit Registers ? A = Possible ?Get Byte' data byte.
X = IOCB number times 16.
Y = Return Code

Functions:

1) The X register is loaded with the current IOCB number times 16 from CURFCB.
2) The return code is placed in the IOCB status field (ICSTA).
3) The stack register is restored to point to the stack displacement at FMS entry from the value saved in ENTSTK.
4) The possible ?Get Data? data byte is loaded into the A register.
5) The Y register is loaded with the return code.
6) The caller (CIO) is returned to via the RTS instruction.

GREAT And FGREAT

GREAT and FGREAT are the exit points used by FMS when the operation has terminated normally. FGREAT is located at $12EA and is used to free the sector buffer that has been allocated to the FCB. The FRESBUF routine is used to free the buffer. FGREAT exits directly to GREAT ($12F0). The GREAT exit point loads the normal return code ($01) into the A register and goes to RETURN.

Error Exits

The ERREOF exit is called when an end of file condition is found. ERREOF loads the end-of-file condition code ($88) in the A register and goes to RETURN.

The ERRIO exit is called if an error occurs during an I/O operation (Figure 1, control flow 3). The error code from the DCB (control path K) is loaded into the A register as the FMS return code and control is passed to RETURN.

All other errors exits are at the ERxxx labels starting at $12B5. The error code is developed by means of a series of 6502 INC instructions which increment the ERRNO (which was initialized to $9F at FMS entry). The final instruction at the end of the INC chain loads the final ERRNO value into the A register and control is passed directly to RETURN.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Seven
DEVICE DEPENDENT COMMANDS

A Device Dependent Command is any command which is not Open, Close, Get Byte, Put Byte, or Status. When the command value in the IOCB is greater than 15 ($OF), CIO will call the Device Handler Device Dependent Command routine. The Device Handler must determine if the command is a valid command for that device. The Device Dependent Commands that for FMS are:

Rename
Delete
Lock
Unlock
Point
Note
Format
The FMS Device Dependent Command routine starts at DFMDDC.

DFMDDC

Address ? $BA7
Entry Registers ? A = Don't Care
X = IOCB number times 16
Y = Don't Care.
Exit Registers ? A = Unknown.
X = IOCB and FCB number times 16
Y = Unknown.

Function:

1) Call SETUP
2) If the command is Format (254), then go to the Format routine, XFORMAT at $D18.
3) If the command is not Format, then check that the command value is $20 through $26. If the command value is not in this range then exit via the ERDVDC (Command Error) routine.
4) If the command is valid, go to the command via the DCDCVT vector table.

XFORMAT

The XFORMAT routine executes the FORMAT Device Dependent Command.

Address ? $D18
Entry Registers ? A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers ? A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) Issue the format I/O command to the drive. This will cause the drive to perform the physical formating of the disk. If the command returns with good status and there were no bad sectors reported, then continue with the logical format operations. In the event of physical format errors, exit via the ERDBAD error exit.

2) Clear the drive buffer to zero.
3) Set the sector count values into the DVDMSN (VTOC displacement one) and the DVDNSA (VTOC displacement three) fields.
4) Set all 90 sector bit map bits to one (available).
5) Deallocate the first four sectors for the hoot sectors.
6) Deallocate the middle nine sectors for the VTOC and the Directory.
7) Write the VTOC to the Disk.
8) Clear the eight directory sectors to zero.
9) Exit via the FOREAT exit.

XDELETE

The XDELETE routine executes the DELETE Device Dependent Command.

Address ➔ $C32
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16
Y = Don't Care
Exit Parameters ➔ A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) The filename is decoded via the FNDCODE routine.
2) The first filename is searched for via the SFDIR routine.
3) The file, if found, is deleted via the XDELO routine.
4) If the file just deleted was DOS.SYS then the boot record is re-written via the DELDOS routine.
5) The directory is searched for the next matching entry. If an entry is found then the process repeats at step three. If no further matching directory entries are found, then exit via FGREAT.

XDELO

The XDELO routine is used to delete the file whose directory entry is indicated by the CDIRD (current Directory Displacement) byte ($1305).

Address ➔ $C53
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care
Exit Registers ➔ A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) The OPVTOC routine is called to insure that the disk is not write protected.
2) The TSTLOCK routine is called to insure that the file is not locked.
3) The file deleted bit is set in the directory entry flag and the directory sector is written back to the disk.
4) The VTOC sector bit map bits for the sectors in the file are set to one to make them eligible for reuse. This process is achieved by reading each sector in the file sector chain and calling the FRESECT routine to change the VTOC bit map.
5) The VTOC Write Required Bit is set so that the VTOC will be written back to the disk.

XRENAME

The XRENAME routine executes the RENAME Device Dependent Command.

Address ➔ $BD9
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.

Exit Registers → A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) The filename is decoded via the FNDCODE routine.
2) The directory is searched for the first entry to be renamed. If no entry is found then the ERFNF (File not found) exit is taken.
3) The TSTLOCK routine is called to insure that the file is not locked.
4) If TSTDOS determines that the old filename is DOS.SYS then the boot record is rewritten via the DELDOS routine.
5) If new filename is DOS.SYS, then the boot record is rewritten via the SETDOS routine.
6) The filename in the directory is changed to the new filename.
7) The directory sector is rewritten.
8) The directory is searched for the next filename match. If a match is found, then the process repeats at step three. If no further match is found then, exit via FGREAT.

XLOCK And XUNLOCK

The XLOCK routine executes the LOCK Device Dependent Command. The XUNLOCK routine executes the UNLOCK Device Dependent Command.

Address → $C7C
Entry Registers → A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers → A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) The XLOCK entry sets the LOCK bit value, DFDLOC ($20), into TEMP4. The XUNLOCK entry sets a zero value into TEMP4. Both routines then go to XLCOM.
2) The filename is decoded via the FNDCODE routine.
3) The directory is searched for the first file entry match. If no match is found, the ERFNF (file not found) exit is taken.
4) The files directory flag is modified to either LOCKED or UNLOCKED by means of the value previously set into TEMP4.
5) The Directory sector is written back to the disk.
6) The CSFDIR routine is called to find the next filename match. If a match is found, then the process repeats at step four. If no match is found, then exit via FGREAT.

XPOINT

The XPOINT routine executes the POINT Device Dependent Command.

Address → $CBA
Entry Registers → A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers → A = Unknown.
X = Unknown.
Y = Unknown

.

Functions:

1) If the FCBFLG indicates that the file can acquire sectors (Opened for Output or Append), then exit via the ERRPOT (point error) exit.
2) If the current sector is not the same as the sector POINTed to by the IOCB AUX3 and AUX4 fields, then write the current sector back to the disk (if it has been changed).
3) Read the POINTed to sector into the sector buffer.

4) Set the FCB next byte pointer, FCBDLN, to the value indicated by the user Point data in the IOCB AUX5 field.
5) Exit to FGREAT.

XNOTE

The XNOTE routine executes the NOTE Device Dependent Command.

Address ➔ $D03
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers ➔ A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) The current sector number and data displacement into the sector is moved to the appropriate IOCB fields, ICAUX3, ICAUX4, ICAUX5.
2) Exit via GREAT.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Eight

FMS OPEN ROUTINES

The FMS Open routine, DFMOPN, is called directly by CIO via the FMS Device Vector Table, DFMSDH at $7CB.

DFMOPN

The DFMOPN routine is the FMS file open routine.

Address — $8AB
Entry Registers — A = Don't Care
X = IOCB number times 16.
Y = Don't Care.
Exit Registers — A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) Initialize for this operation by calling SETUP.
2) Decode the filename via FNDCODE.
3) Examine the open code in ICAUX1 for the open-for-directory-read command. If this is a directory read command, go to LISTDIR.
4) If not a directory read, then search the directory for the first match on the file name and save the resulting search condition on the stack.
5) Determine the exact type of Open operation to he performed by examining the IOCB ACUXI field. If INPUT, go to DFOIN. If Output, go to DFOUT. If Update, go to DFOUPD. If Append, go to DFOAPN. If none of the above, exit via the ERDVDC (device command error) exit.

DFOIN

DFOIN ($8D8) is entered when opening a file for Input. The routine pops the stack to determine if the directory search for the file name was successful. If the file name was found in the directory, then go to DFOUI. If the search was not successful, then exit to ERFNF (file not found).

DFOUPD

DFOUPD ($8DD) is entered when opening a file for Update (Input and Output). The routine pops the stack to determine if the file name was found in the directory. If the file was not found, then exit to ERFNF (file not found). If the file was found, insure that the file is not Locked by calling TSTLOCK. If the file is unlocked, then continue at DFOUI.

DFOUI

DFOUI ($8E3) is entered to finish opening a file for Input or Update. The read setup routine, DFRDSU, is called. FMS then exits via the GREAT exit.

DFDRDSU

DFDRDSU ($9AE) is entered to set up a data file for reading. It begins by calling SETFCB to set some standard file information into the FCB. It continues by setting up the FCB with various other parameters to read the first data sector in the file. This sector is read via the RDNSO routine. When the sector has been read into the sector buffer, the code returns to the caller.

DFOAPN

DFOAPN ($BEC) is entered to open a file for Append.

1) Pop the stack to determine if the file has been found in the directory. If the file was not found exit via ERFNF.
2) If the file was created by DOS 1, then exit via ERAPO.
3) Insure the file is not locked by calling TSTLOCK.
4) Insure the diskette is not write protected by calling OPVTOC.
5) Allocate a new sector for the start of the Append chain by calling GETSECTOR.
6) Save the sector number of the sector obtained in FCBSSN so that it will be available when the file is closed.
7) Continue opening the file as it it were being opened for Output at DHFOX2.

DFOOUT

The DFOOUT ($911) routine is entered when opening a file for Output.

1) Pop the stack to determine if the file was found in the directory.
2) If the file was found, then delete it via the XDELO ($C53) routine.
3) If the file was not found, then make a new entry in the directory via the code at DFOXI ($91D).
4) Allocate a data sector for the file via the GETSECTOR routine.
5) Put the necessary information about the file into the directory and write the directory sector hack to the disk.
6) Continue at DHFOX2.

DHFOX2

DHFOX2 ($97C) is entered to finish the Open process for files that are being opened for Output or Append.

1) Finish initializing the FCB via SETFCB.
2) If the TSTDOS routine determines that the file name being opened is DOS.SYS, then write out DOS via the WRTDOS routine.
3) Exit via GREAT.

SETFCB

The SETFCB ($995) routine is used in the various Open file routines to place certain common data into the FCB.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Nine
FMS CLOSE ROUTINES

The FMS close routine is called directly by CIO via the FMS Device Vector Table, DFMSDH at $7CB.

DFMCLS

Address • $B15
Entry Registers • A = Don't Care.
X = IOCB number times 16.
Y = Don't Care.
Exit Registers • A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) Initialize via call to SETUP.
2) If the file was not opened for some form of output (Output, Update or Append) then clear the FCB open flag, FCBOTC and exit via FGREAT.
3) If the FCBFLG indicates that the file has not acquired sectors, then continue at CLUPDT to close the Update file.
4) Write the last data sector via WRTLSEC.
5) Read the file's directory sector into the directory buffer via the RRDIR routine.
6) Get the sector count from the directory.
7) If the file was opened for Output (i.e. it is not open for Append), then continue at CLOUT.
8) Read all the data sector of the file until the end-of-file sector is found.
9) Place the sector address of the start of the Append chain into the link sector field of the (old) end-of-file sector.
10) Continue at CLOUT.

CLOUT

The CLOUT ($B50) routine is entered to finish closing a file that had been opened for Output or Append.

1) The sector count field of the directory is updated.
2) The open for output flag is turned off.
3) The file in use flag is set.
4) The directory sector is written back to the disk by the DRTDIR routine.
5) The VTOC sector is written back to the disk by the WRTVTOC routine.
6) The FCB open code flag, FCBOTC, is cleared to zero.
7) Exit via FGREAT.

CLUPDT

The CLUPDT ($B75) is called to finish the closing of a file that had been opened for Update.

1) If the current sector in the sector buffer has been modified then write it back to the disk via the WRCSIO routine.
2) Clear the FCB open flag, FCBOTC, to zero.
3) Exit via FGREAT.

Chapter Ten
GET BYTE ROUTINE

The FMS GET BYTE routine, DFMGET, is called directly by CIO via the FMS Device Vector Table, DFMSDH at $7GB. The GET BYTE routine's function is to get and return the next sequential data byte to CIO.

DFMGET

Address ? $ABF
Entry Registers ? A = Don't Care
Y = IOCB number times 16.
X = Don't Care.
Exit Registers ? A = Unknown.
Y = Unknown.
X = Unknown.

Functions:

1) Initialize via the SETUP routine.
2) If the FCB is opened for Directory read, then go to GDCHAR.
3) If the current sector is empty, attempt burst I/O (see Burst I/O section), then continue with number four.
4) Read the next sector via the RDNXTS routine. If the read sector operation did not return an end-of-file condition, then continue at step three, else exit via ERREOF (end-of-file error).
5) Get the data byte from the sector and place it in SVDBYT for the exit routines.
6) If the next byte in the file is the end-of-file byte, exit via RETURN with the impending end-of-file condition code ($03), else exit via GREAT.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Eleven
PUT BYTE ROUTINE

The FMS PUT BYTE routine, DFMPUT, is called directly by CIO via the FMS Device Vector Table, DFMSDH at $7CB. The PUT BYTE routine's function is to place the single data byte transmitted by CIO into the data sector.

DFMPUT

Address • $99C
Entry Registers • A = The •put data• data byte.
X = The IOCB number times 16.
Y = Don't Care.
Exit Registers • A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1)The data byte in the A register is saved in SVDBYT.
2) SETUP is called to initialize for this operation.
3) If the caller was not CIO, then prevent a burst I/O operation from occurring.
4) If the file was not opened for output, then exit via ERDVDC (device command error).
5) If the current data sector is full, write the sector via WRTNXS, then attempt burst I/O (see BURST I/O section). If a burst I/O operation did take place, then get the next byte after the area just written by burst I/O and place it into the SVDBYT cell.
6) Increment the sector data byte count.
7) Move the data byte from SVDBYT to the next available data byte in the sector.
8) Set the sector modified flag in the FCB.
9) Exit via GREAT.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Twelve
BURST I/O

The CIO is designed to fill or empty a large user buffer with data bytes sent to or received from a device handler, a byte at a time. To fill a thousand-byte buffer, CIO would have to call FMS one thousand times in rapid succession. While the process is simple and easy to implement by both CIO and the Device Handlers, it can be very slow. This is particularly true in the case of FMS which has a great deal of overhead code to go through each time it is called. FMS circumvents most of the CIO/FMS calls for large data transfers via the BURST I/O routines.

Burst I/O operates by reading or writing data sectors directly into the user buffer (Figure 1, data path I). There are a number of tests that must he passed before a burst I/O operation can take place. If any of the tests fail, then the CIOIFMS data transfer reverts to the normal mode of operation.

When the PUT BYTE routine is called, it will call the WTBUR ($A1F) routine when it is ready to start filling a new data sector. WTBUR will not allow a burst I/O operation to happen if the file has been opened for Update. If the file has not been opened for Update, then WTBUR goes to the common read/write burst I/O test routine, TBURST at $A28. If the file has been opened for Update, then exit Burst I/O indicating that a Burst I/O did not happen. When WTBUR calls TBURST, it has the A register set to non-zero to indicate that it is write.

When the GET BYTE routine is called, it will call the RTBUR ($A26) routine when it is ready to read a new data sector. RTBUR indicates that it is read by setting the A register to zero and then enters TBURST.

TBURST

1) Save the A register in BURTYP. This value will indicate if the burst operation is a read or a write.
2) If the I/O command in the IOCB is for text I/O (a transfer that is to end when the Atari end-of-line ($9B) character is transferred), then TBURST will exit indicating (carry set) that a burst I/O operation did not occur.
3) If the user buffer length in the IOCB is not at least a full sector in size, then exit without doing a burst I/O.
4) If all the above tests pass, then perform a burst I/O operation. The first step in the burst I/O operation is to change the zero page sector buffer pointer, ZXBA ($47) from the FMS sector buffer address to the user buffer address.
5) If the operation is read, then read the next sector via RDNXTS. If the read sector operation produced an end-of-file, then go to BUREOF, else go to BBINC.
6) If the operation is write, then the area in the user buffer, where the three bytes of data sector control information is to be placed, will be saved. The data will be written via the WRTNXS routine. The saved user data will then be copied back into the user buffer. The code then continues at BBINC.

BBINC

The BBINC routine is entered after a single burst I/O sector has been read or written. BBINC updates data counters in the FCB and in the IOCB and tests for the end of the Burst I/O.

1) The zero page sector buffer pointer is incremented by the length of data in a sector (125 or 253).
2) The user buffer length is decremented by the length of data in a sector.
3) The TBLEN routine is called to determine if there is enough room left in the user buffer to read or write another full sector (128 or 256 bytes). If another sector can he read or written, then the process repeats at NXTBUR ($A3E).

4) If there is not enough room in the user buffer to perform another full sector read or write, then BUREOF is entered.

BUREOF

1) The final address in the zero page sector pointer, ZSBA ($47), is moved to the IOCB buffer address field.
2) The value in the zero page sector buffer pointer is restored by the SSBA routine.
3) The caller is returned to with the carry cleared to indicate that a burst I/O operation has happened.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Thirteen
READING THE DIRECTORY AS A FILE

A formatted subset of the data in the Directory can be read as if the Directory were a disk file. This is accomplished by using the open directory code ($02) in the IOCB ICAUX1 byte. When FMS recognizes this code in the Open routine (at $8B1), it will go directly to the LISTDIR routine. The LISTDIR routine prepares the FCB for reading the directory as a file. The GET BYTE routine will recognize the read directory condition from information stored in the FCBOTC field (see $AC2) and go directly to the directory read character I/O routine GDCHAR.

LISTDIP

Address ➔ $DAD
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers ➔ A = Unknown.
X = Unknown.
Y = Unknown.

Functions:

1) The TEMP4 byte is used to count the characters that have been transmitted by GDBYTE from the formatted line buffer. LISTDIR sets this value to zero to indicate the start of a new formatted line.
2) The SFDIR routine is called to start a wild card search for the file name in the directory.
3) If a match is found then FDENT is called to format the entry and prepare for the GDBYTE calls. Exit via GREAT.
4) If a match is not found, then LDCNT is called to prepare to send the xxx FREE SECTORS line.

GDCHAR

GDCHAR ($DB9) is entered from GET BYTE to get a single data byte from a formatted directory line.

1) The TEMP4 flag is tested. If the value is negative, then all formatted information has been transmitted. Exit is via the ERREOF (end-of-file error) exit.
2) The value in TEMP4 is used as an index into the formatted line buffer to get the next character. The character is placed into SVDBYT for loading into the A register by the RETURN routine.
3) The character retrieved from the buffer is examined for the EOL ($9B) character.
4) If the character is not an EOL, then exit is via GREAT.
5) If the character was an EOL, then the line length is examined to see if the line was a directory entry line (i.e., if the length was 17) or the final xxx FREE SECTORS.
6) If the line was the final line, then TEMP4 is set to a negative value ($80) to indicate that all formatted lines have been sent. Exit is via GREAT.
7) If the line was not the final line, then CSFDIR is called to find the next matching file name.
8) If a file name match is found, then FDENT is called to format the found entry into the formatted line buffer. Exit is via GREAT.
9) If a file name match is not found, then go to LDCNT to format the final line.

LDCNT

LDCNT ($DE9) formats the final line of a directory read.

1) Read the VTOC.
2) Get the free sector count from the VTOC and convert it to ATASCII via the CVDX routine.
3) Move the FREE SECTORS message to the formatted line buffer.
4) EXIT is via FGREAT.

FDENT

The FDENT ($E21) routine formats the current directory entry into the formatted line buffer for subsequent reading by GDBYTE.

1) The directory flag is checked for the file locked condition. If the file is locked, then the "*" is placed in the formatted line.
2) The file name is moved from the directory entry to the formatted line.
3) The file sector count is converted to ATASCII and placed in the formatted line.
4) The EOL character is placed in the formatted line.
5) Exit is via the RTS instruction.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Fourteen
SECTOR I/O ROUTINES

The FMS performs sector I/O by calling the SIO routine in the OS ROM (Figure 1, control path 3). All sector I/O calls in the FMS occur from the BSIO routine. There are several other routines that are designed to set up information for BSIO. These routines deal with reading and writing sectors of a particular type such as data sectors, directory sectors, and the VTOC sector.

BSIO

Address ➔ $76C

Entry Registers ➔ A = Sector number most significant byte.
Y = Sector number least significant byte.
X = If 1, then 128 byte 1/0 (810 drive).
If 2, then 256 byte I/O (815 drive).

Exit Registers ➔ A = Status byte from DCB.
Y = Unknown.
X = IOCB and FCB number times 16.

Functions:

1) The sector number is stored in the DCB from the A,Y register pair. The DCB is the interface control block for SIO calls.
2) If the carry is clear, then the DCB is set up for read data. If the carry is set, then the DCB is set up for write data.
3) The serial bus ID for the disk, and the disk timeout values are placed into the DCB.
4) The error retry counter, RETRY, is set for four retries.
5) The I/O data length is set to 128 or 256 depending upon the data in the X register.
6) The serial I/O routine ($E459) is called to execute the I/O.
7) If the I/O operation was good, then the X register is loaded with the IOCB (and FCB) number times 16 from the CRFCB cell and the status byte from the DCB is loaded into the A register. Return is via the RTS instruction.
8) If the I/O operation was bad, then the retry counter is decremented. If the retry value is positive, then the I/O is retried. If the value is negative, then the routine is exited in the manner described in step seven.

DSIO

The DSIO routine is called to perform data sector I/O operations.

Address➔$11F7

Entry Registers ➔ A = Sector number most significant byte.
Y = Sector number least significant byte.
X = IOCB and FCB number times 16.

Exit Registers ➔ A = I/O condition code.
Y = Unknown.
X = IOCB and FCB number times 16.

Functions:

1) The sector buffer address is obtained from the zero page sector buffer pointer ZSBA ($47) and placed in the DCB buffer address field, DCBBUF.

2) The drive type byte is loaded into the X register from DRVTYP. If the drive is an 810, then the value will be one. If the drive is an 815, then the value will be two.
3) BSIO is called.
4) The DSIO caller is returned to via the RTS instruction.

RDDIR And WRTDIR

The RDDIR and the WRTDIR routines are used to perform Directory sector I/O operations. The RDDIR entry ($106E) sets the carry to indicate read. The WRTDIR entry ($1071) clears the carry to indicate write. Both of the routines continue at DIRIO.

DIRIO

1) Save the read/write flag (carry sense) on the stack.
2) Set the address of the directory buffer into the DCB buffer field, DCBBUF.
3) The CDIRS cell contains the number of the directory sector to he read or written. This value ranges from zero to seven. The DIRIO routine creates the actual sector number to read or write by adding $169 to the CDIRS value. The resulting sector number is placed in the A,Y register combination.
4) Continue at DSYSIO.

RDVTOC And WRTVTOC

The RDVTOC and WRTVTOC routine are called to initiate I/O to and from the VTOC sector. The RDVTOC routine ($108B) first checks the write required byte in the VTOC sector buffer. If the value of this byte is not zero, then the VTOC is already in the buffer (and has been changed). If the VTOC is already in the buffer, then the read does not have to be done; therefore, the RDVTOG routine will return to the caller. If the write-required byte is zero, then RDVTOC will clear the carry to indicate that the operation is read. The WRTVTOC routine ($1095) sets the write required byte to zero, then sets the carry to indicate a write operation. Both RDVTOC and WRTVTOC continue at VTIO.

VTIO

1) The read/write flag is pushed onto the stack.
2) The VTOC sector buffer address is moved from the zero page drive buffer address pointer ZDRVA ($45) to the DCB buffer pointer, DCBBUF.
3) The A,Y register combination is loaded with the VTOC sector number ($168).
4) Continue at DSYSIO.

DSYSIO

1) The read/write sense is popped from the stack.
2) The drive type value is loaded into the X register from DRVTYP.
3) BSIO is called.
4) If the I/O operation was good, then return to the caller via the RTS instruction.
5) If the I/O operation was bad, the exit via the ERRSYS exit (fatal system I/O error).

OPVTOC

The OPVTOC routine ($10BF) is used by various FMS routines to insure that the diskette is not write protected before executing functions that will write to the disk. This routine will read the VTOC via RDVTOC and then attempt to write the VTOC via WRTVTOC. If the diskette is write protected, the WRTVTOC will cause an I/O error exit (error number 144). If the diskette is not write protected, then the routine will return to the caller. When OPVTOC does return to the caller, the current disk VTOC is in the drive buffer.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Fifteen
FILE NAME DECODE ROUTINE

The FNDCODE routine is used to transform the user supplied file name into a form that is usable in FMS for wild card searching of the directory. The primary and extension parts of the user file name are padded with blanks and question marks as required. The following examples show the types of transform performed by FNDCODE:

User File Name	Transformed File Name
D:*.*	??????????
D1:GLOP.*	GLOP ???
D1:GLOP.BAS	GLOP BAS
D2:*.ASM	????????ASM
D:GL?P.S*	GL?P S??
D1:G*	G???????

FNDCODE

Address ➔ $E9E
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers ➔ A Unknown.
X = IOCB and FCB number times 16.
Y = Unknown.

Functions:

1) The user file name buffer is searched for the colon (:) delimiter. If the delimiter is not found within 256 characters then exit to ERRFN routine (file name error).
2) The FMS file name buffer, FNAME, is cleared to blanks.
3) The EXTSW byte is set to zero. When EXTSW is zero, the primary file name field is being processed. When EXTSW is minus, then the extension file name field is being processed.
4) The next character in the user file name buffer is examined.
5) If the character is an asterisk (*), then the field is padded with question mark characters to the end of the field.
6) If the character is a period and the extension field is being processed, then exit via the RTS instruction.
7) If the character is a period and the primary field is being processed, then switch to the extension field processing.
8) If the character is a question mark, then put it into the FNAME via FDSCHAR.
9) If the character is alphanumeric (A through Z, or 0 through 9), then put it into FNAME via FDSCHAR.
10) If the character is none of the above, then assume that end of the filename has been found and exit via the RTS instruction.
11) If a character was stored, then continue at step four.

FDSCHAR

1) If the character counter register, X, indicates that the primary field is full, then exit without storing the character.
2) If the character counter register, X, indicates that the extension field name is full, then exit without storing the character.
3) Store the character into FNAME indexed by the X register.
4) Increment the X register.
5) Return to caller via the RTS instruction.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Sixteen
DIRECTORY SEARCHING

The Directory search routine searches the directory entries for a file name that matches the name in FNAME. The routine has two entry points: SFDIR which is used to begin the search at the start of the directory, and CSFDIR, which is used to continue searching the directory at the entry just past the previously found matching entry.

The routines have five memory cells that they use for controlling the search operation: DHOLES, DHOLED, CDIRS, CDIRD and SFNUM. The CDIRS cell contains the current relative directory sector number (zero through seven). The CDIRD cell contains the displacement into the directory sector of the current entry. DHOLES gives the relative directory sector number (zero through seven) of the first hole or available entry in the directory. The DHOLED cell gives the displacement to the first available entry that is the hole. The SFNUM cell is used to contain the current file number of the entry being examined. The value in SFNUM will be from zero through 63.

If the value of DHOLES is $FF at the end of the search, then the directory is full.

The directory search routine will exit with the carry clear if a match was found. It will exit with the carry set if no matching entry was found.

SFDIR

The SFDIR routine ($F21) is called to start searching the directory at the start of the directory.

1) Initialize DHOLES, CDIRS, SFNUM to $FF.
2) Initialize CDIRD to $70.
3) Continue at CSFDIR.

CSFDIR

The CSFDIR routine ($F31) is called to continue searching the directory.

1) Increment the file number, SFNUM.
2) Increment CDIRD by the size of a directory entry (16).
3) If the CDIRD is now greater than, or equal to, 128 ($80) then increment CDIRS by one. If the value of CDIRD is now eight, then exit with the carry set to indicate that a match was not found. If CDIRD is less than eight, then read the next directory sector via RDDIR. Set CDIRD to zero.
4) If the directory entry flag field is zero then the end of the used portion of the directory has been reached. If a hole has not been found, then mark this entry as a hole. Exit with the carry set to indicate that the file was not found.
5) If the directory entry flag field indicates that the file is open for output, then skip this entry.
6) If the directory entry flag field indicates that the file has been deleted, and a hole has not been found, then mark this entry as a hole and continue searching the directory.
7) If the file is in use, then check the file name in the directory entry for a match with the name in FNAME. Wild card characters in FNAME (question marks) are assumed to match the corresponding characters in the directory entry file name.
8) If the names match, then exit with the carry clear to indicate that a match was found.
9) If a match was not found, then continue to search the directory.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Seventeen
WRITE NEXT SECTOR

The write next sector routine, WRTNXS, is used to write a data sector to disk.

Address • $F94
Entry Registers • A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers • A = Unknown.
X = IOCB and FCB number times 16.
Y = Unknown.

Functions:

1) If the file has been opened for update, and the sector has not been modified, then do not write the sector. Read the next data sector and then return to caller.
2) If the file has been opened for update, and the sector has been modified, then write the current sector. Read the next data sector into the sector buffer and return to the caller.
3) It the file is not opened for update, then allocate a new sector to the file by calling GETSECTOR.
4) Move the sector byte count from the FCB FCBDLN field to the data sector byte count field.
5) Move the address of the newly acquired sector from the FCB FCBLSN field into the link field of the current data sector.
6) Write the current sector to the disk via WRCSIO.
7) If the I/O was bad, mark the FCB by placing a zero value into FCBOTC as closed and exit via RETURN with the I/O error number as the return code.
8) If the I/O was good, then increment the FCB sector counter field, FCBCNT.
9) Call MVLSN to move the sector number of the link sector number field of the FCB, FCBLSN, to the current sector number field of the FCB, FCBCSN.
10) Set the current data length field of the FCB, FCBDLN, to zero.
11) Set the maximum data length field of the FCB, FCBMLN, to 125 (if 810 drive) or 253 (if 815 drive).
12) Return to user via the RTS instruction.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Eighteen
READ NEXT SECTOR

The read next sector routine, RDNXTS, reads the next sector in the file sector chain into the sector buffer. If there are no more sectors in the chain, then the routine returns with the carry set to indicate end-of-file. If the routine returns with the carry clear, then the next sector has been read.

RDNXTS

Address ? $100F
Entry Registers ? A = Don?t Care.
X = IOCB and FCB number times 16.
Y = Don?t Care.
Exit Registers ? A = Unknown.
X = IOCB and FCB number times 16.
Y = Unknown.

Functions:

1) If the file has been opened for Update, then WRTNXS is called to write the current sector if it has been modified.
2) If the FCB link sector number field, FBCLSN, is zero then there are no further sectors to read. Return to the caller with the carry set to indicate that the end-of-file has been reached.
3) Call MVLSN to move the FCB link sector number field, FCBLSN, the FCB current sector number field, FCBCSN.
4) Call RWCSIO with the carry set to read the next sector.
5) If the I/O operation was bad, exit via the ERRIO exit (I/O error).
6) Insure that the file number in the sector just read agrees with the file number in the FCB. If the file numbers are not the same, exit via the ERFNMM exit (file number mismatch). Note: if the routine was called by delete, return to delete indicating end-of-file.
7) Move the link sector number from the data sector to the FCB link sector field in the FCB, FCBLSN.
8) Move the sector data length information from the data sector to the FCB maximum data length field, FCBMLN.
9) Reset the FCB data length field, FDBDLN, to zero.
10) Return to the caller with the carry clear to indicate that a sector has been read.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Nineteen
GET AND FREE SECTOR ROUTINES

The get sector routine, GETSECTOR, is called when a new sector is needed. The routine searches the bit map in the VTOC for a free sector. The sector found is deallocated from the bit map and the sector number is returned to the caller. The free sector routine, FRESECT, is given a sector number to be freed. FRESECT locates the required bit map bit in the VTOC and turns it on (sets it to one). The sector is now eligible for reuse.

GETSECTOR

Address ➔$1106
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers ➔ A = Unknown.
X = IOCB and FCB number times 16.
Y = Unknown.

Functions:

1) The Y register is used as an index into the bit map bytes.
2) The bit bytes are examined sequentially from the first bit map byte to the last bit map byte until a non-zero byte is found. The displacement to this byte is saved in TEMP 1.
3) If no bits are found in the bit map, then the ERRNSA exit (no sectors available) is taken.
4) The number-of-sectors-available-field, in the VTOC, is decremented by one.
5) The VTOC write required byte in the VTOC is set to a non-zero value to indicate that the VTOC has been changed and must be written back to the disk.
6) The non-zero bit map byte that was found in the bit map search is retrieved. The bits in this byte are shifted left until a bit moves into the carry flag. The carry is then set clear and the bits shifted back to their original position. The byte with the newly allocated sector bit turned off is placed back into the bit map.
7) The number of bits shifted and the index to the bit map byte are used to develop the sector number represented by the bit.
8) The sector number is stored in the FCB link sector field, FCBLSN.
9) The user then returned to via the RTS instruction.

FRESECT

Address ➔ $10C5
Entry Registers ➔ A = Don't Care.
X = IOCB and FCB number times 16.
Y = Don't Care.
Exit Registers ➔ A = Unknown.
X = IOCB and FCB number times 16.
Y = Unknown.

Functions:

1) The sector to be freed is in the FCB current sector field, FCBCSN. If the sector number is zero, then FRESECT exits back to the user via the RTS instruction.
2) The sector number is divided by eight to determine the bit map byte which represents the sector. The remainder from this division represents the bit within the byte.
3) The byte is retrieved from the bit map, the bit is turned on, and the byte placed hack into the bit map.
4) The number of available sectors field in the VTOC is incremented by one.
5) The VTOC write required byte is set to non-zero to indicate that the VTOC has been changed and needs to be written back to the disk.
6) The caller is returned to via the RTS instruction.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Twenty
THE BOOT PROCESS

When the Atari computer is turned on, the routines in the OS ROM will (under certain conditions) read the first sector from the disk in drive one into memory. It will then examine certain specific locations in this record to decide how to boot the disk. In the following discussion, refer to Figure 20-1. The OS ROM code will load BRCNT consecutive sectors (starting with sector one) onto memory, starting at the address contained in BLDADR. When the OS ROM code has finished this task, it will make a JSR call to the code that is seven bytes into the start of the boot area. In the case of FMS, this is the IMP XBCONT instruction at $706. The XBCONT code will continue the boot load process. The XBCONT code examines the DFSFLG to see if a DOS.SYS file exists. If the file exists, then the sector number of the first sector in DOS.SYS will be in DFLINK. The routine will then read all the sectors in the chain starting at DFLINK into the memory area pointed to by DFLADR. When the entire DOS.SYS file is read into memory, XBCONT returns to the OS ROM code.
The OS ROM code will eventually vector through the BINTADR so that the FMS can initialize itself. In the DOS 2.0S system, BINTADR points into the DUP.SYS code. DUP.SYS then receives control from the OS ROM rather than the FMS. One of the tasks that DUP.SYS performs during its initialization is to call the FMS initialization routine.

XBCONT

The XBCONT routine ($714) is entered by the OS ROM code during the boot process to allow the boot process to continue in the manner best suited for the code being booted.

Functions:

1) If the DFSFLG indicates that a DOS.SYS file does not exist, then the OS ROM is returned to with the carry set to indicate that the boot has failed.
2) The address contained in DFLADR is moved to the zero page address pointer, ZBUFP, and to the DCB buffer pointer field, DCBBUF.
3) The sector number contained in DFLINK is loaded into the A,Y register pair, the carry is cleared to indicate read, and BSIO is called to read a DOS.SYS sector.
4) The next sector Link is obtained from the link field of the data sector just read.
5) If the sector link value is zero, then the DOS.SYS end-of-file has been reached. The OS ROM will be returned to with the carry clear to indicate that the boot read was good.
6) If the sector link value is not zero, then the zero page buffer pointer and the DCB buffer pointer are incremented by the amount of data in the sector (125 for 810 drives, 253 for 815 drives).
7) The process continues by reading the next sector into memory.

Return to Table of Contents | Previous Chapter | Next Chapter

Chapter Twenty-One
MAINTAINING THE BOOT RECORD

The boot record (sector 1) contains information about the DOS.SYS file. When DOS.SYS is opened for output, FMS will write all of FMS out to the disk as part of the open process. It will also modify sector zero to indicate that a DOS. SYS file exists and to indicate where on the disk it Is. If DOS. SYS is ever Deleted or Renamed (to something not DOS.SYS), then the boot record must be modified to indicate that a DOS.SYS file does not exist. If a file is ever renamed to DOS.SYS, then the boot record is modified to point to the new DOS. SYS file.

WRTDOS

The WRTDOS routine ($120A) is used to write a new DOS.SYS file to disk and to update the boot record to indicate that a DOS. SYS file exists.

Functions:

1) The sector number which is contained n the FCB sector number link field, FCRLSN, is used as the first sector of the DOS.SYS file. This sector number is placed in the boot record area in page seven along with the other necessary information.
2) Sectors one, two, and three are written from the memory area from $700 through $87F.
3) The FMS is written to the DOS. SYS via the WD0 routine.
4) Exit is via GREAT.

WD0

The WD0 routine ($1267) is used to write the FMS to the DOS.SYS file.
Functions:

1) The address contained in DFLADR is moved to the zero page buffer pointer, ZBUFP.
2) The FMS is copied from its area in memory to the file sector buffer in 125 byte chunks.
3) The buffers are written to disk by the WRTNXS routine.
4) The process continues until the entire FMS area has been written.
5) The caller is returned to via the RTS instruction.

DELDOS

The DELDOS routine ($1219) is used to modify the boot record to indicate that DOS. SYS does not exist.
Functions:

1) The DFSFLG is set to zero to indicate that DOS. SYS does not exist.
2) The area from $700 to $87F is written to sectors one, two, and three.
3) The caller is returned to via the RTS instruction.

Return to Table of Contents | Previous Chapter | Next Chapter

Atari DOS 2.0S

(The source code for Atari DOS is not available at atariarchives.org. Although we have permission to publish the text of the book, we do not have permission to publish the source.)

Return to Table of Contents | Previous Chapter | Next Chapter

Appendix A
AN INTERMEDIATE USER'S GUIDE TO THIS BOOK

If you are familiar with machine language, commented source code, and hexadecimal numbers, you probably won't need to read this appendix. On the other hand, if you don't know or are new to machine language – perhaps some of the information here will help.

A knowledge of machine language is important to grasping the sense of the DOS since it is written in machine language. However, we will briefly cover some of the fundamentals, as they relate to the book, in the hope that this might be a starting point. One of the functions of this book is to reveal the inner workings of Atari DOS. A benefit of knowing how it works is that you are able to change it to suit yourself, to customize it.

First we'll examine the meaning of the various fields of information which are in the source code (page 59 on). Then, after a brief look at how to deal with hexadecimal numbers, we can make a modification to DOS step-by-step to show how it's done.

The book is divided into two sections: roughly the first half is a series of descriptions of the major subroutines of the disk operating system. The latter half is a commented source code of the DOS. In order to better understand what you can accomplish with all this information, we can set up a problem and solve it using the book.

What's "Commented Source Code"?

We'll change the DOS so that we could type in a disk command using lowercase letters. Unfortunately, the D: must be in uppercase, the program which makes this decision is in ROM and we can't get at it and change it. The rest of the command can be in lowercase, though, after we make our change to the DOS in RAM. After fixing it, any routine that uses the disk will accept lowercase as in D: open.

Before getting into the details of the modification there is some important preliminary information. What, for example, is "commented source code?"

Machine language differs in several respects from BASIC. When you write a program in BASIC, you never see how it looks to the computer. Instead you see something like this:

```
10 FORI=1 TO 100
20 NEXT I
```

This delay loop just creates a brief pause in a program. If you RUN the above, the computer handles the problem of translating the BASIC words into machine language. Anything the computer does must be translated into machine language (ML). Translating (or interpreting) a BASIC program takes place during the RUN of the program – that's why BASIC is so slow compared to ML.

By contrast, ML is translated before it is RUN. Programming ML is done in two stages: 1. writing the source code and then 2. assembling it into object code. The computer does most of the drudgery of this because most ML is written by using a program called an assembler which handles many of the details. Some assemblers are so complex that using them can seem almost like programming in BASIC.

Here is how you might program the above example delay loop when using an assembler:

```
1000 LDY #64 ; SET COUNTER TO 100
1001 LOOP DEY
1002 BNE LOOP
```

Probably the most peculiar thing about this, to the beginner, is how 64 stands for 100 (it's hex, we'll get to it in a minute). The line numbers could be BASIC, but the instructions are 6502 mnemonics (memory aids). LDY means to load the Y register with 100 (decimal). The next line is named (labeled) "loop" because assemblers don't say

GOTO 1001. Instead, they use convenient names. In any event, the Y register is decremented by DEY, it's lowered by one. So each time the program cycles through the LOOP address, it will lower the counter one. Finally, the instruction at 1002 says, Branch if Not Equal (to zero). In other words, GOTO LOOP if Y hasn't yet counted down to zero. When Y reaches zero, the program will continue on, following whatever instruction is in line 1003.

After the above program is written, though, it still cannot be RUN. There is the second step, the creation of object code (executable), the assembly process.

You tell the assembler to assemble this program. The result of that is an additional two &#65533;fields&#65533; (zones). Above, we have five fields: line number, label, mnemonic (instruction), operand (the #64), and a comment field which is the equivalent of BASIC REM statements. There will soon be a total of seven fields.

After assembly, the two new fields are the addresses and the object code (expressed as hex bytes). By the way, BASIC always assigns its programs a starting address in memory, but, in ML, the programmer must make this known to the assembler. It's not the computer's decision. Assume the computer were told to assemble the above example at address $2000 (this would be 8192, in decimal). The dollar sign means that a number is a hex number. The labels, mnemonics, and operands would be translated into object code and put into the computer's memory. As you'll see in the second half of this book, a printout of completed assembly looks like this:

```
2000 A000  1000          LDY #64  ;SET COUNTER TO 100
2002 88    1001 LOOP     DEY
2003 DOFF  1002          BNE LOOP
```

Hex

Before concluding this brief overview of some fundamentals of machine language, we should explain how to read the numbers in the source code listings.

```
100 DIM H$(23),N$(9):OPEN#1,4,0,&#65533;K:&#65533;
130 GRAPHICS 0
140 PRINT &#65533;PLEASE CHOOSE:
150 PRINT &#65533;1 &#65533; Input HEX & get decimal back
    .&#65533;
160 PRINT &#65533;2 &#65533; Input DECIMAL to get hex bac
    k.&#65533;
170 PRINT:PRINT &#65533;==>&#65533;;:GET#1,K
180 IF K<49 OR K>50 THEN 170
190 PRINT CHR$(K):ON K&#65533;48 GOTO 300,400
300 H$=&#65533;@ABCDEFGHI!!!!!!!JKLMNO&#65533;
310 PRINT &#65533;HEX&#65533;;:INPUT N$:N=0
320 FOR 1=1 TO LEN(N$)
330 N=N*16+ASC(H$(ASC(N$(I))-47))-64:NEXT I
350 PRINT &#65533;$&#65533;;N$;&#65533;=&#65533;;N:PRINT:PRINT:GOTO 140
400 H$=&#65533;0123456789ABCDEF&#65533;
410 PRINT &#65533;DECIMAL&#65533;;:INPIJT N:M=4096
420 PRINT N;&#65533;$&#65533;;
430 FOR I=1 TO 4:J=INT(N/M)
440 PRINT H$(J+1,J+1);:N=N-M*J:M=M/16
450 NEXT I:PRINT:PRINT:GOTO 140
```

This program will turn a decimal number into hex or vice versa. Hexadecimal is a base 16 number system, where decimal is base ten. This means that you count from zero to fifteen before going to the next column. For example, you count up zero one two. . . until you reach nine in decimal. Then you go to the next column and have a one-zero (10) to show that there is one in the &#65533;ten's column&#65533; and zero in the &#65533;one's column.&#65533;

In hex, what was a &#65533;ten's column&#65533; becomes a &#65533;sixteen's column.&#65533; In other words, the symbol &#65533;10&#65533; means that there is one sixteen and zero &#65533;ones.&#65533; So, the decimal number 17 would be written in hex, as $11 (one sixteen plus one one). The decimal number 15 would, in hex, be $0F. After nine, we run out of digits, so the first few letters of the alphabet are used: A = 10, B = 11, C = 12, D = 13, E = 14, and F=15.

This explains how to &#65533;read&#65533; hex numbers if you don't want the program above to do it for you. The number $64 is decimal 100 because there are six 16's and four one's. 6 X 16+4= 100.

Addresses can be larger than two digits, up to a maximum of four. You might see an address such as $11F7 in the listings. The third column is the 256's and the fourth column is the 4096's. So to find out what this address is in decimal, you can multiply 7 X 1, 15 X 16, 1 X 256, and 1 X 4096. And add them all together.

A quicker way is to find out the first two, (15 x 16+7=247) and then multiply the second two by 256. It comes out the same. The second two would be $11 (17 in decimal) so 17 x 256 + 247 = 4599. It might be easier to just use the BASIC program to make the translations until hex becomes more familiar.

Making A Modification

Now that you have the entire source listing of DOS 2.0S, you can customize it to fit your needs.

You may have felt restricted by the limitations on file names. A file name can consist of eleven characters: up to eight characters plus an optional three-character extension. The first character must be from A-Z; subsequent characters can be from A-Z or 0-9. That's it. No punctuation. No imbedded spaces. No lowercase.

By changing only two locations in the file name decode section of DOS, many more characters are permitted. We will modify DOS to accept any ASCII characters in a file name except character graphics and inverse video. Additionally, the filename can start with a number (e.g. D:3-D). Unfortunately, there is no foolproof way to allow imbedded spaces such as D:TIME OUT.

The following fragment of code checks to see that a character of the file name falls in the range of A-Z. If the character is less than (carry clear) 65 [ASC(A)] or greater than or equal to (carry set) 91 ASC(Z) + 1], then the test fails. All we do is change the check for A to a check for ! (its number in the code is one greater than space), and the check for Z + 1 to z + 1 (lowercase z).

Included in this range of 90 characters are the numbers (48-57) and all punctuation. Since we start with 33, space is excluded. It is possible to permit imbedded spaces, but the file would then be inaccessible in certain situations where a space is used as a delimiter. You can allow it at your discretion, or even permit the entire (almost) ATASCII character set to be used by changing the limits to 0 and 255.

```
CMP  #A
BCC  FD5
CMP  #$5B
BCC  FD6
```

We change this to:

```
CMP  #!
BCC  FD5
CMP  #$7B
BCC  FD6
```

The changes can be made in BASIC with POKE 3818,33:POKE 3822,123 or change hex locations $0EEA to $21 and $0EEE to $7B. The section of code we're modifying is located between source line numbers 4072 through 4193. Remember to rewrite the modified DOS to disk with WRITE DOS FILES (Menu selection H) if you want your change to be permanent.

Other equally simple changes are also possible. You could change the wild-card character (*) to any other character by changing location $0EC7 to the desired character. A more ambitious task would be to increase the maximum file name length.

This brings up a final point software compatibility. For example, if you changed the wild card character to @,you couldn't run any previous programs that assume * as the wild card character. Our change is less dangerous if you allow lowercase file names, the unmodified DOS won't be able to access it, although it will look fine on the directory. This change has not been exhaustively tested for conflicts, so we can't guarantee its usage. Nevertheless, it seems quite useful and shows that some customizing can be accomplished with a few simple changes.

When experimenting, always keep a backup copy of your valuable disks in case something should go awry.

```
100 REM CHANGE DOS PROGRAM
110 REM FOR DOS 2.0S ONLY
120 REM CHANGE LOW RANGE CHECK FROM
130 REM 65 TO 33. THIS ALLOWS
140 REM ANY CHARACTER (EXCEPT
150 REM GRAPHICS AND INVERSE VIDEO)
160 REM TO START A FILENAME, INSTEAD
170 REM OF ONLY A THROUGH Z.
180 REM 0EE9 C941 CMP #'A
190 REM 0EE9 C921 CMP #'!
200 POKE 3818,33
210 REM CHANGE HIGH RANGE TO EXTEND
220 REM UP TO ASCII "Z"
230 REM (LOWERCASE Z)
240 REM 0EED C95B CMP #$5B
250 REM 0EED C97B CMP #$7B
260 REM POKE 3822,123
270 REM NO NEED TO CHANGE NUMERIC
280 REM CHECK SINCE IT IS NO
```

```
290 REM LONGER EXECUTED, THANKS
300 REM TO THE ABOVE CODE.
```

Some Cautions

Care is necessary when making customizations. Only make the changes to a copy of your DOS – not the original –system master.– (You shouldn't be able to do this anyway, since the disk is –write-protected,– but better safe than sorry.) Remember that any files SAVEd with your custom DOS will probably not be compatible with the original, unchanged DOS. Alternation of the DOS can have unpredictable effects; we urge caution and cannot accept any liability for software or hardware damage incurred through the use of this book.

Things To Look Out For

These modifications could make a customized DOS incompatible with the original, unmodified DOS 2.0S:

1) File name changes (such as allowing lowercase, or increasing the length)
2) Changes to DOS file structure (such as using a different –linking– system)
3) Removing error-checks. These built-in traps insure disk integrity and reliability. When you alter one, you could risk muddling one or more files. For example, if you allow an automatic –wild-card– feature, where an asterisk is assumed at the end of a file, it could cause havoc when performing a SCRATCH, RENAME, or UPDATE operation. Another example is removing some of the qualifications for –burst-I/O.– Remember that a lot of thought went into each design consideration.

Keeping these suggestions in mind, here are some ideas for modifications. You may need to type in and re-assemble (with your insertions) the entire DOS when making certain modifications.

1) Adding a STATUS check before a disk access. Have you ever noticed how long the drive will grind away when no disk is inserted? You can query the disk for its status, and even add a –Drive not ready– error message if the drive door is not closed or a disk is not inserted. Check your DOS manual for details.
2) Adding Disk Utility commands. These would be additional functions performed by the FMS, keyed to the –special command.– Some of the tasks performed by the Disk Utility Package could be a part of the DOS kernal, such as LOAD and SAVE binary files. You could even implement new commands such as –relative file– support, where you only give the DOS a –record number– to randomly access a file. The file could be divided into records of any length.
3) Allocate more sectors for the directory, thereby extending the maximum amount of directory entries.
4) Add a disk name and/or disk I.D. number (serial number?) to the disk (maybe on sector 720). It could even print out with the directory.
5) Given the extra –unused– bytes in the file name, add a byte for file type, such as program, data, object code, etc., and have it printed out with the directory, making it easy to identify files without having to use the extension. This would be hard to interface with software, however.

Remember that some of this is risky business. Keep backup disks for any disk you are –experimenting– with. That way, you should lose no important files.

The publishers and authors of this book disclaim any responsibility for errors or problems caused by modification of Atari DOS 2.0s.

Return to Table of Contents | Previous Chapter