FIRST AND FINEST

OS/A+

OS/A+
OS/A+
OS/A+
OS/A+

Systems Software for
Apple and Atari Computers

Optimized Systems Software, Inc.

a reference manual for

O S / A +

an Operating System for Atari Computers +
an Operating System for Apple Computers +
an Operating System for Advanced users

PREFACE

OS/A+ is the result of the efforts of several persons, and we believe that proper credit should be given. The original Apple version of the console processor (CP) and the original version ("version 2") of the File Manager System (which is, of course, identical with Atari's DOS 2.0S) were written by Paul Laughton, ex of Shepardson Microsystems, Inc., who also authored the original Apple DOS (version 3.1). The current versions of all other portions are primarily the work of Mark Rose, of OSS, with the collaboration of Bill Wilkinson and Mike Peters.

We realize that OS/A+ is not the most sophisticated, most complete, operating system for any and all microcomputers, but we believe that the inherent power and flexibility that it exhibits within its compact size are a good match for the size and features of the machines it is intended for.

TRADEMARKS

The following trademarked names are used in various places within this manual, and credit is hereby given:

OS/A+, BASIC A+, MAC/65, and C/65 are trademarks of Optimized Systems Software, Inc.

Apple, Apple II, and Apple Computer(s) are trademarks of Apple Computer, Inc., Cupertino, CA

Atari, Atari 400, Atari 800, Atari Home Computers, and Atari 850 Interface Module are trademarks of Atari, Inc., Sunnyvale, CA.

TABLE OF CONTENTS

Table of Contents (Continued)

Table of Contents (Continued)

CHAPTER 1: INTRODUCTION

OS/A+ was originally an accident, brought about by the fact that that we developed Atari's DOS and Atari's BASIC on an Apple II computer. To simulate Atari's indeed excellent OS ROMs, we wrote our own simple CIO (Central Input/Output) system. From there, it was only logical that we install a Console Processor similar to that of Digital Research's CP/M (their trademark). Then when we introduced BASIC A+, we moved the "CP" over to the Atari, and presto! There was born OS/A+ version 1.0 for the Atari.

This manual actually describes three products:

OS/A+ version 2 for Atari Computers
OS/A+ version 4 for Atari Computers
OS/A+ version 4 for Apple II Computers

1.1 SYSTEM REQUIREMENTS

Although both versions of OS/A+ for the Atari will run nicely in 32K bytes of RAM, it isn't realistic to use less than 40K or 48K and expect to do useful work with most languages and/or applications. The Apple II version requires 48K bytes of RAM, and we heartily recommend 64K. Obviously, with all versions at least one disk drive is required. Two disks are highly recommended. The Atari version 4 system requires (and runs on) only double density or larger disk drives.

1.2 GETTING STARTED

Anxious to try OS/A+? Can't wait to wade through all this? Familiar with CP/M or R/T-11 or similiar operating systems? If you can answer "yes" to all of these, we recommend you read Chapter 2, at least. Then you can refer to Chapters 3 and 4 to use more system features. Or move to Chapters 5 and 7 to start writing your own assembly language software.

Even if you are not skipping ahead, you may want to skim through much of this for now; but plan to come back later when you're ready to really understand the system.

Anyway, put your OS/A+ master disk (WITH write protect tab on, PLEASE!) into Drive One (Drive One, Slot 6 for Apple users), turn on your system, and try us!

P.S. Maybe the first command you want to learn is "DUPDSK" (section 4.9), to back up your valuable system master. Surprised? OS/A+ is NOT copy protected. We hope you are considerate of our rights so that we may continue to be considerate of your convenience.

1.3 OVERVIEW OF OS/A+

OS/A+ (and, naturally, Atari's OS) utilizes a software concept which is built around a structured and layered scheme. In particular, application programs are expected to make calls to the OS via the Central Input Ouptput routine ("CIO"). In turn, CIO is a dispatcher which examines the application program's request and routes the necessary subrequests to the appropriate device driver(s).

On the Atari, the device drivers may in turn call the SIO (Serial Input/Output) routines to perform the actual channel communications with devices on the serial bus (obvious exceptions include the screen and keyboard, which do not require serial bus service). Finally, the device (on the serial bus) receives the SIO request and performs the actual I/O needed. The diagram on the next page illustrates this process.

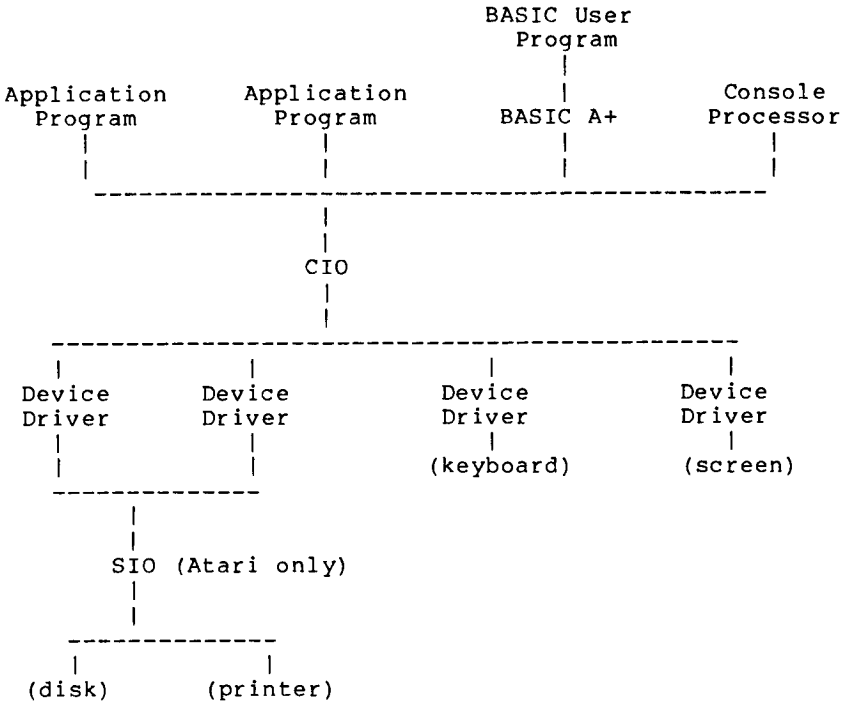

Figure 1-1
Overview of Hierarchy of OS/A+

The scheme used on the Apple II is identical excepting only that SIO is not needed nor used, and the various device drivers generally all communicate directly with their respective peripheral(s).

Generally speaking, there is no reason why any one or more portions of this hierarchical structure cannot be replaced with another, equivalent section of code. On the Atari computer, in fact, DOS (or, more properly, the FMS or File Management System) itself is "added" to the default structure only if a disk drive is present at power-on time. Several manufacturers, for example, have produced their own printer or screen drivers, replacing the Atari-supplied drivers with minimal effect.

Unfortunately, in the case of Atari computers, we cannot say that any given portion may be replaced with NO effect, simply because an unfortunately high portion of software written for the Atari violates the hierarchy (by direct calls to device routines, or worse). These violators are by no means in the majority, or we might have no hope of ever producing an improved Atari system. However, we should be aware of at least the most important of these (quite frankly) poorly written programs and maintain what compatibility that we can when we change the system.

Generally, the worst offenders are programs such as VISICALC and MICROSOFT BASIC, both of which make assumptions about memory layout and disk usage. However, these programs (and most others) are shipped with an operating system intact on the disk on which they reside. Thus, although we may not force them to take advantage of the expanded capabilities that our device drivers may offer, at least we need only maintain compatibility with a standard Atari 800 and 810 Disk Drive to allow their usage on otherwise improved products.

1.4 WHY TWO ATARI VERSIONS OF OS/A+ ?

Because we like to add to the confusion, of course. Seriously, when we originally produced Atari DOS, we wrote it to Atari specifications. There is more detail on this subject in Chapter 9, but suffice to say the real problem with Atari's FMS (and hence with OS/A+ version 2) is that it was never designed to handle disks larger than 256 Kilobytes. But now PERCOM DATA COMPANY has added double sided, double-density disks to their catalog, with capacities of nearly 400 Kilobytes.

Given that we need to access more than 256K bytes per disk and/or file, how can we expand on the Atari system? An obvious solution is to introduce the concept of "logical disks", wherein a larger drive might contain two or more disjoint segments, each wholly allocated to imitate an 810 (or 815...the difference is solely in the number of bytes per sector) disk system. Anyone who has tried the Corvus equivalent of this scheme will recognize the inadequacies of this solution.

So, given that we will no longer be compatible with Atari products, why not sieze this opportunity to "do it right"? Why not produce a wholly different file manager system that is not bound by the restrictions of Atari DOS? This is the path we have preferred to choose.

Thus we come to OS/A+ version 4, a mapped file system. Since we wrote not only Atari DOS but also Apple DOS, we naturally thought of an extension on the Apple scheme as the logical step up from Atari DOS and version 2 of OS/A+. We do not know if it might ever happen, but using our version 4 scheme would, presumably, enable a manufacturer to offer disk systems which were MEDIA and FILE COMPATIBLE on both Apple and Atari (and perhaps other 6502 systems).

For more information on the philosophy and structure of version 4, please see chapter 10.

Chapter 2 The OS/A+ Console Processor (CP)

As you might recall from Chapter 1, Figure 1-1, the Console Processor (or sometimes the "Command Program", but in either case always "CP") is NOT a priveleged part of the system. CP functions as just an easy-to-use interface between the human at the keyboard and the machine level of the CIO calls.

In Chapter 1, we mentioned that any portion of the OS/A+ system could be replaced without change to any of the other sections. This is perhaps most true of CP. For example, in a dedicated run-time environment, CP has no reason to exist. Others have written their equivalent of CP and placed it under the Atari DOS system, but we believe that the CP of OS/A+ is a very well-designed, well-executed human interface, especially considering that it occupies less than 800 bytes of your precious memory.

This chapter serves as an introduction to using the OS/A+ system, especially as it is viewed through the eyes of the Console Processor.

2.1 BOOTING UP (and returning to CP)

When an OS/A+ disk is booted, the CP is immediatly entered. On an Atari computer, if a cartridge has been inserted which works with the disk (such as the BASIC cartridge), then the cartridge will be entered upon bootup instead of the CP. Re-entry of OS/A+ from the cartridge is normally done through the use of the DOS command (e.g, the BASIC command for this is DOS). Some cartridges do not allow DOS-type exits and thus OS/A+ cannot be used with these cartridges.

In any case, on either Apple or Atari machines, when CP is entered it will clear the screen and display:

```
OSS OS/A+   ATARI (or apple) VERSION x.xx
D1:<cursor>
```

The D1: is the command prompt. It serves two purposes. First it tells the user it is ready to accept a command. Secondly, it is a reminder of the default disk drive. The default drive, in this case, being the familar file spec for drive 1.

2.2 Default Drive and File Specifiers

Most CP commands and parameters deal with files of one sort or another. OS/A+ requires files be specified with a filename of the form:

<device>: <optional-file-name>

The device for disk files is of the form Dn: where n=1,2,3,4,5,6,7,8. For example, D1: is the device name of the primary (boot) disk drive. Other types of devices are: Printer=P:, Cassette=C: (Atari only), Screen=S:, etc. The optional-file-name is used for named file accessing devices such as the disk units. To work with the disk file TEST.ORG on disk drive number 1, the operating system requires that the file spec D1:TEST.ORG be used. Having to always specify the D1: can be tedious, especialy if most of the user's file work is on a single drive.

CP is designed to prefix all filenames appearing in a CP command line with the default drive - if and only if a device has not been explictly specified. In the case of D1:TEST.ORG, the user could enter only TEST.ORG for a file name and allow CP to prefix it with the default drive. Thus TEST.ORG becomes D1:TEST.ORG in the OS/A+ system. If TEST.ORG happened to be on drive two and the default drive was drive one, the user could enter D2:TEST.ORG; and CP would see that the user has explicitily specified a <device> and would thus not append the default drive device to that file name.

If the user needs to work a great deal with files on drive two, he can change the default drive so as to avoid the now necessary D2: prefix typing. Where the system prompts D1:<cursor>, the user can respond with D2:<return> to change the default drive to the D2: device. The next CP prompt line will show D2:<cursor>. Now files accessed on drive one will require the explict D1: prefix typing, while files on drive two will not require prefix typing. Only devices of the form Dn: (where n = 0-9) are allowed as default drives. OS/A+ does not check to insure that the new default drive actually exists. The user's first indication of an invalid default drive will occur when OS/A+ attempts to access a file on the invalid device (via user command). The error message "INVALID DEVICE" will indicate the situation. The user should then set the default device to a valid disk unit. The default device change command is one of the many intrinsic CP commands.

2.3 CP Commands

CP has three general classes or groups of commands. The classes are intrinsic commands, extrinsic commands, and execute commands. Intrinsic commands are executed by means of resident code in the OS/A+ Console Processor monitor. Extrinsic commands are executed by means of loading and running programs. The execute subset of commands provide for the batch execution of CP commands from a file.

2.4 An Overview of Intrinsic Commands

The intrinsic commands are executed via code in the OS/A+ Console Processor monitor (CP). These commands do not require the loading of programs to perform their functions. The following is a summary of the most useful OS/A+ and CP intrinsic commands:

```
DIRECTORY - List Directory
PROTECT   - Protect a file (from change or erase)
UNPROTECT - Unprotect a file
ERASE     - Erase (delete) a file
RENAME    - Renames a file
LOAD      - Load a binary file
SAVE      - Save a binary file
RUN       - Execute a program at some address
CARTRIDGE - Run Atari cartridge in the "A"
            cartridge slot (Atari users only)
```

(The default drive change command Dn: is also considered an intrinsic command.)

All intrinsic commands may be abreviated with the first three characters. As a matter of fact, OS/A+ only looks at the first three characters while testing for an intrinsic command. Each of the commands will be covered in detail later in this manual; however, to give you a feel of the intrinsic commands, let's look at the DIRECTORY command. While looking at these examples, assume the D1: is the default device and has been placed on the screen by CP.

```
D1:DIRECTORY   list all files of disk on drive one
D1:DIRECT        "   "     "   "   "     "   "    "
D1:DIRTY         "   "     "   "   "     "   "    "
D1:DIR           "   "     "   "   "     "   "    "
D1:DIR *.*       "   "     "   "   "     "   "    "
D1:DIR D1:       "   "     "   "   "     "   "    "
D1:DIR D1:*.*    "   "     "   "   "     "   "    "
D1 DIR D2:     list all files of disk on drive two
D1:DIR D2:*.*    "   "     "   "   "     "   "    "
D1:DIR *.OBJ   files with extension .OBJ on drive one
D1:DIR D2:*.AS files with extension .ASM on drive two
```

note: under version 4 of OS/A+ the file spec necessary to refer to all files on a disk is just "*", not "*.*"

2.5 An Overview of Extrinsic Commands

The extrinsic commands are programs that are run by OS/A+. Any program file of the load file format and containing the .COM extension may be used as a OS/A+ extrinsic command. The OS/A+ COPY command is one such extrinsic command. If you LIST the OS/A+ DIRECTORY, you will see a file named COPY.COM . The program in the COPY.COM file is what is executed when the COPY command is entered. Assuming that D1: is the default device, the COPY command would look like:

D1:COPY <from-file-name> <to-file-name>

or

D1:COPY TEST.OBJ D2:TEST.OBJ

to copy TEST.OBJ from drive one to drive two. (COPY has many more options available. See the section on COPY later in this manual for more details.)

Whenever any command is given to OS/A+ it first compares the command entered (first three characters only) to its intrinsic command list. If the command is not in the intrinsic list, it is assumed to be extrinsic.
OS/A+ will process the extrinsic command by:

1) Prefix the command with the default device (if a device is not specified).
2) Attach the .COM extension to the command.
3) Open the generated file spec for input.
4) Test file for program of proper LOAD file format.
5) Load and execute the program.

The COPY command illustrated will execute only if the file COPY.COM exists on drive one and is of the Load file format. If any element of the procedure fails various error messages will result. Step 1 of the procedure implies that a device may be specified. If the default device was drive two and the COPY.COM

program was on drive one, our example COPY would look like:

```
D2:D1:COPY D1:TEST.OBJ TEST.OBJ
-------------------------------
```

which again copies TEST.OBJ from device one to device two. Never explictly specify the .COM extension as part of the command. The command COPY.COM will result in a file spec of D1:COPY.COM.COM, which is generally invalid. If the file is not of the proper LOAD format, the error message ADR RANGE ERROR will most likely appear.

Some extrinsic commands (such as COPY) are supplied by OSS. The number of possible extrinsic commands is not, however, limited to these few; commands may be written by the user to perform virtually any function. If you are intrested in writing your own extrinsic commands, see Chapter 7.

If an extrinsic command (i.e., a program running in RAM) has control, the program may generally be rerun or reentered by simply using the RUN command with no parameters.

2.6 An Overview of Batch Processing

The OS/A+ execute feature allows the user to execute one or many CP commands with a single command. Let's suppose that you wrote a set of BASIC programs that must be run in sequence. You could issue the CP extrinsic BASIC command (thus executing BASIC.COM), then from BASIC run each program one at a time. If the running time of the BASIC programs was very long you could sit at the key board for hours just to type RUN program name every once in awhile. OS/A+ allows you to create and execute an EXECUTE file which contains one or many OS/A+ commands. You would then enter one command that would free you from the keyboard for more important (or fun) things.

Any text file with the filename extension .EXC can be used as an OS/A+ execute file. The execution of the file is invoked much like the extrinsic commands, except the command is preceeded with a commercial "at" symbol ("@"). To execute the EXECUTE file DEMO.EXC on the D1: default device, type:

```
D1:@DEMO
   -----
```

CP will create the file spec D1:DEMO.EXC and then set up OS/A+ to read it line by line executing the CP commands just as if they were being entered from the keyboard.

CHAPTER 3: THE INTRINSIC COMMANDS DETAILED

Intrinsic Commands are those commands which may be given anytime the system is displaying the OS/A+ prompt (e.g., D1: or D2:). Since these commands reside within the Console Processor's memory, they do not need to be loaded from disk.

Any intrinsic command may be accessed by using just the first three characters of the command name (see section 2.4 for examples). A consequence of this is that no extrinsic command program may start with three characters which match any of the instrinsic commands. For example, a program named "PROCESS3.COM" could not be called by simply typing "PROCESS3", since OS/A+ would view that as the intrinsic command "PROtect". Solutions: (1) Rename the extrinsic command file. (2) Use "LOAD PROCESS3". If the program soes not then automatically run, simply type "RUN" and it will execute.

The intrinsic commands are detailed below, one to a page in alphabetic order.

Section 3.1

command: CAR

purpose: This command transfers control to a cartridge

users: Atari users only

usage: CAR

arguments: none

options: none

Description

The CAR command allows the user to enter a cartridge from OS/A+. The cartridge will retain control of the system until a DOS command is executed from the cartridge.

section 3.2

command: DIRectory

purpose: The command allows the user to view the disk directory

usage: DIR [Dn:][file-specifier]

arguments: optionally, a file specifier string

options: none

Description

The dir command searches the disk directory of the specified disk (or the current default drive, if Dn: is omitted) for all files matching the file-specifier. The names of all files matching the specifier are then printed to the screen, together with the length of the file (in sectors). An asterisk preceding the file's name indicates that the file is protected from erasure, writing, or renaming.

The file-specifier may be any valid file name (see sections on file structure) and may contain the "wild-card" characters '?' and '*'. A question mark ('?') will match any character in a file name, while an asterisk ('*') will match any string of zero or more characters. For example,

```
        DIR *AB.C??
```

will match and list

```
        XAB.CXX
        AB.CUR
        BEOBAB.CNN
          etc.
```

Section 3.3

command: END

purpose: Stop batch execution from within an execute file

usage: END

arguments: none

options: none

Description

The END command causes OS/A+ to stop reading commands from a batch file and to resume prompting the user for commmands. This command has no effect outside of a batch file.

Section 3.4

command: ERAse

purpose: This command removes files from a disk

usage: ERA [Dn:]file-specifier

arguments: a file specifier string

options: none

Description

The ERA command permanently removes files from a disk. All files matching the file-specifier string on the specified drive (or the current default drive, if Dn: is omitted) will be erased from the disk. These files will no longer be shown when a DIR command is issued, nor will they be available for any type of file access.

As this command causes the irreversible deletion of files from the disk, it should be used with care. Use the PRO command to guard files against accidental erasure.

Examples:

ERASE *.BAK
will erase all files with an extension of .BAK that are unprotected and that reside on the current default drive.

ERA D2:DUP.SYS
will erase the file named DUP.SYS from disk drive number two.

Notes:

If ERAse does not find any erasable files that match the specifier, it will return a file not found error.

Section 3.5

command: LOAd

purpose: Load disk files into memory

usage: LOAD file-name

arguments: a file name

options: none

Description
The LOAD command allows the user to load binary load image files into user memory. The files must be compatible with the normal binary object files used by the normal host computer operating system. That is:

For Atari users, each segment of the memory image file must be preceeded by two addresses, the starting and ending addresses in RAM memory of the segment. The entire file must be preceeded by two bytes with all bits on ($FF, $FF). This format is identical to that produced by Atari's Assembler/Editor Cartridge and most upgraded products (including EASMD and MAC/65 from OSS).

For Apple II users, each segment of the memory image file must be preceeded by a two-byte address (the starting address in RAM memory of the segment) and a two byte number which represents the length of the segment in bytes. This is identical with the file format produced by the "BSAVE" command in Apple Dos 3.3, excepting only that the concept has been extended to allow multiple segment in the file (as if two or more BSAVED images were concatenated).

Section 3.6

command: NOScreen

purpose: Turns off command echo to screen during batch

usage: NOS

arguments: none

options: none

Description
Normally, all commands encountered during batch execution are echoed to the screen as if they were typed in by the user. The NOS command can be used to prevent this echo. All commands within an execute file will then no longer be echoed until the execute file is stopped for any reason or a SCR command is encountered.

This command only effects commands encountered in batch mode.

Section 3.7

command: PROtect

purpose: This command protects files from accidental erasure, writing, or renaming

usage: PRO [Dn:]file-specifier

arguments: a file specifier string

options: none

Description

The PRO command allows the user to protect one or more files from any erasure, writing, or renaming. All files matching the file-specifier on the specified disk (or the current default drive, if Dn: is omitted) will be protected. These files will then be shown with a preceding asterisk when a DIR command is issued. The UNP command can be used to disable the protection, when desired.

Section 3.8

command: REMark

purpose: Facilitates remarks to screen during batch execution

usage: REM any characters

arguments: a string of zero or more characters

options: none

Description

The REM command performs no operation whatsoever. Its sole purpose is to provide a means of easily printing messages to the screen from an executing batch file (see section on batch execution). When encountered during batch execution, the command line containing the REM command will be echoed to the screen.

Section 3.9

command: REName

purpose: Rename a file to a new name

usage: REN from-file-name to-file-name

arguments: two file names

options: none

Description

The REN command will search the specified disk (or the default drive, if Dn: is not specified) for a file whose name matches the specified from-file-name. If the file is found, its name will be changed to the indicated to-file-name. An error occurs if the from-file is not found on the disk. The to file-name should NOT be preceeded by a disk drive specifier.

Section 3.10

command: RUN

purpose: This command transfers control to a address in memory

usage: RUN [hex-address]

arguments: an optional hexadecimal address

option: none

Description

The RUN command causes OS/A+ to immediately perform a jump to the indicated address (or to the address contained in the OS/A+ RUNLOC, if no address is given). The hex-address, if present, must consist of 3 or 4 hexadecimal digits.

The address in RUNLOC is set any time an extrinsic command is issued or a program is loaded using the LOAD command. Therefore, the RUN command may be used to reenter a program such as BASIC after leaving the program through a DOS command.

IMPORTANT NOTE:

Most standard OS/A+ interactive system programs will set RUNLOC to point to their warmstart entry point. Thus, for example, if the user returns to DOS in order to perform an INTRINSIC command, he/she may reenter the systems program by simply typing RUN. At the current writing, BASIC A+ and MAC/65 (for example) both follow this protocol: simply type RUN from CP to reenter at their warmstart points.

Section 3.11

command: SAVe

purpose: Save a portion of memory to a disk file

usage: SAVE file-name start-address end-address

arguments: a file name
a hexadecimal starting address
a hexadecimal ending address

options: none

Description

The SAVE command allows the user to write portions of memory to disk files in standard binary file format. The two addresses define the portion of memory to be written to disk; the second address must be greater than or equal to the first. A file which has been 'saved' may be later returned to memory using the LOAD command.

Example:

At the time of this writing, the BASIC A+ user with an Atari computer having 48K Bytes of RAM could patch the distribution copy of BASIC A+ and save the new patched version to disk via

```
SAVE NEWBASIC.COM 8400 BC00
```

(PLEASE verify these addresses in your BASIC A+ manual; they ARE subject to change.)

Section 3.12

command: SCReen

purpose: Cause batch commands to be echoed to the screen

usage: SCR

arguments: none

options: none

Description

The SCR command causes commands encountered during batch execution to be echoed to the screen. The NOS command may be used to turn off the echo of batch commands.

This command only effects commands encountered in batch mode.

Section 3.13

command: UNProtect

purpose: This command removes the protection caused by the PRO command

usage: UNP [Dn:]file-specifier

arguments: a file specifier string

options: none

Description
The UNP command allows the user to remove the write protection caused by the PRO command so that files may again be erased, renamed, or written to. All files matching the file-specifier on the specified drive (or the current default drive, if Dn: is omitted) will be affected. These files will no longer be shown with a preceding asterisk when the DIR command is used.

CHAPTER 4: EXTRINSIC COMMANDS DETAILED

This chapter gives a descrption of each extrinsic command supplied as a standard part of an OS/A+ system master diskette (except that some commands may be specific to particualar versions of packages).

Remember that any program which is (or may be made into) a DOS-compatible load file may be used as an extrinsic "command". The sole exceptions to this rule are programs whose names begin with three characters which match any of the three character intrinsic commands. For a list of the intrinsic commands and further information on this point, see Chapter 3.

As all extrinsic commands cause programs to be loaded, such commands may only be invoked when a disk containing the particular command program has been inserted in the appropriate drive.

Section 4.1

command: ADOS

purpose: This command allows access to version 2 and version 4 files at the same time

users: Atari version 4 users only

usage: ADOS

arguments: none

options: none

Description

This program installs OS/A+ version 2 along with version 4. This allows the user to access version 4 disks as Dn: while accessing version 2 disks as An:. The usage of this command does require more memory for the dos, so the low memory pointer ($2E7) will be moved up accordingly. In order to restore the system to its former state (i.e., version 4 only), press the system reset key.

The DUPDSK command must not be used while the ADOS command is in effect! This will result in a system crash.

After using ADOS to install the version 2 file system, you may use APCOPY to copy version 2 files to version 4 diskettes, or vice versa.

Section 4.2

command: BASIC

purpose: This command loads and executes the BASIC A+ language

users: BASIC A+ owners only

usage: BASIC [file-name]

arguments: optionally, the name of a saved BASIC A+ program

options: none

Description

This command loads and executes the file BASIC.COM, which is the BASIC A+ language. If the optional file name is specified, BASIC A+ will automatically load and execute that file. The file must have previously been 'SAVE'd from BASIC A+. Refer to the BASIC A+ manual for information specific to the language.

*** FOR MORE INFORMATION, SEE YOUR BASIC A+ MANUAL ***

Section 4.3

command: C65

purpose: this command loads and executes the OSS C/65 compiler

users: C/65 owners only

usage: C65 souurce-file destination-file [-T]

arguments: two file specifiers

option: -T include C/65 source Text in assembler output

Description:

This command loads and executes the file C65.COM, the OSS small C compilier. Two filenames are required. The first file given must be the name of a text file containing C/65 source code and statements. The second file specified will be created (or reused, if it already exists), and the compiler will write MAC/65-compatible assembly language to it.

Option

If the -T option is specified, the MAC/65 file will contain the user's C/65 text lines. Each source line precedes the assembly code it generates, if any.

*** FOR MORE INFORMATION, SEE YOUR C/65 MANUAL ***

Section 4.4

command: CONFIG

purpose: Allows the user to change the status of a configurable drive (e.g., PERCOM)

users: OS/A+ users with configurable drives

usage: CONFIG [parm1 parm2 ...] [-N]

arguments: an optional list of paramaters which define the desired status of drives in the system

options: -N no drive configuration table will be displayed

Description

If no parameters are given, this command simply reports the status of all drives currently attached to the Atari computer.

If one or more parameters are given, they are presumed to be requests to Percom-compatible disk drives to configure themselves. In particular, a parameter consists of a single numeric digit (in the range of 1 to 8) followed by one or two alpha characters (the "Mode"). The digit is presumed to be a disk drive number (corresponding to D1: through D8:). The legal character combinations usable as Modes are as follows:

Mode	Meaning
S	Configure this drive as a Single density, single sided drive.
D	Configure this drive as a Double density, single sided drive.
DD	Configure this drive as a Double density, Double sided drive.

Options

Normally, the CONFIG command will list out the current drive configuration. Using the -N option will cause this table to be omitted.

Example:

CONFIG 1D 2DD
requests that D1: be configured as double density, single sided, while D2: will become double density, double sided.

Notes:

If a configuration request is made, the file manager system is reinitialized and the system status is reported, as if the command CONFIG with no parameters had been given.

If a configuration request is invalid (e.g., if the drive is not capable of being configured via software), the command will report an error.

Section 4.5

command: COPY

purpose: This program copies files. Note the cautions listed below.

usage: COPY source-file destination-file [-fqsw]
or
COPY file [-FQSW]

arguments: one or two file specifiers

options:

-F	force overwrite of existing file
-Q	query before each file transfer
-S	single disk copy
-W	wait for user response before copying

Description

The copy program copies one or more files without changing the source file. In the first form, all files matching the source-file specifier would be copied to files indicated by the destination specifier, which may be on the same or a different disk. In the second form, the files indicated by the file name would be copied to files having the same name on the same drive. This enables the copying of files on a single disk system. The source and destination file specifiers should be of one of the following forms:

1) [Dn:]file-name
2) Dn:

In form 1, the drive specifier (Dn:) is optional; the current default drive will be assumed if no drive specifier is given. In the second form, all files from the indicated drive would be copied to or from another disk.

Options

The -f option causes the program to overwrite an existing file if it has the same name as a destination file to be copied. If this option is not specified, files whose destination names already exist will not be copied.

The -q option causes the program to ask the user whether to copy each file.

The -s option indicates to the program that it must perform the copy on a single drive. Copy will prompt the user to insert source and destination disks at the proper time.

The -w option indicates that the program must wait for the user to insert the proper disks before initiating the copy.

CAUTION:

Do NOT use COPY in conjuction with the ADOS command to copy FROM version 2 diskettes TO version 4 diskettes. Instead, use the COPY24 command utility found on your Version 4 Master Diskette.

COPY may be used to copy version 4 files to version 2 diskettes, but COPY24 may be a more advisable choice.

Examples:

COPY *.*

will copy all files on the current disk on the current drive to another disk on the same drive. The system will prompt the user when the diskette needs to be swapped. Generally, DUPDSK is a more effective and faster means of performing this function.

COPY *.COM D3: -F

will copy all files having an extension of ".COM" from the current disk drive to drive 3 (which could be the same as the current drive; caution). If the file(s) already exist on drive 3, they will be erased and rewritten.

COPY D2:C*.* D1: -Q

will ask the user if he wants to copy each file starting with the letter "C" from drive 2 to drive 1.

Section 4.6
---

command: COPY24

purpose: transfer files from version 2 to version 4 diskettes (or vice versa)

users: Atari owners with OS/A+ version 4 only

usage: COPY24 filename [,filename...] [-D][-W]

arguments: from one to eight filenames, specifying the files on the source disk which are to be transferred

options:

-D	the version 2 disk (whether source or destination) is a Double Density disk
-W	the version 2 disk is to be Written

description:

COPY24 is a valid command only when issued from the CP of version 4 OS/A+ and, even then, only after ADOS has been used to install the version 2 file manager.

Unless the "-D" or "-W" option(s) are specified, APCOPY will copy all the files specified in the command line from a standard Atari-format version 2 single density diskette to a properly initialized version 4 OS/A+ diskette, where both diskettes will be used on the same drive. The files are transferred one at a time, with an appropriate prompt given when necessary to change diskettes.

NOTE that there will be a prompt before the first copy can proceed.

NOTE the restriction that both diskettes be used on a single drive. This is done to insure that the drive in use can be given the proper commands to reconfigure itself.

(continued)

options:

The "-D" option may be used when it is desired to specify that the Version 2 diskette was formatted and written as a double density diskette (whether by OS/A+ version 2 or by Atari DOS 2.0 patched to enable double density).

The "-W" option simply reverses the direction of the copy. If you think of "COPY24" as meaning "COPY via read from version 2 and write to version 4", then "-W" may be thought of as meaning "Write to version 2 from version 4, instead".

Section 4.7

command: DO

purpose: This command allows the user to perform several operations with one command line

usage: DO command[;command;command...]
or
DO

arguments: optionally, a list of commands separated by semi-colons

options: none

Description

This DO command allows the user to issue several commands on one line. These commands are not restricted to OS/A+ intrinsic and extrinsic commands, however. For example, the following DO command would load the BASIC language, enter a previously 'list'ed program, and run the program:

```
DO BASIC;ENTER "D:PROGRAM";RUN
```

In the second form of the DO command, the DO program will prompt the user for a list of commands, one at a time, saving these away for use. The entry of just a carriage return when prompted for a command will cause the entire list of commands to be executed.

Section 4.8

command: DUPDBL

purpose: This program provides fast copying of entire double density diskettes

users: ONLY Atari owners using version 2 OS/A+ AND ONLY those using double density disks

usage: DUPDBL

arguments: none

options: none

Description

The DUPDBL program will prompt the user for source and destination drives, and will ask whether to format the destination disk. The entire source disk will then be copied to the destination disk in a manner somewhat faster than the copy utility would provide. The two disks, however, MUST be double density OS/A+ diskettes formatted under version 2 of OS/A+ (or Atari DOS 2.0S as patched for double density). IF the destination drive is the same as the source drive, the program will prompt the user to swap disks during the duplication process.

Section 4.9

command: DUPDSK

purpose: This program provides fast copying of entire floppy disks of the same size and type

users: Atari owners: all versions and diskettes EXCEPT double density disks under version 2 OS/A+
Apple owners: standard Apple disks only

usage: DUPDSK

arguments: none

options: none

Description

The dupdsk program will prompt the user for source and destination drives, and will ask whether to format the destination disk. The entire source disk will then be copied to the destination disk in a manner somewhat faster than the copy utility would provide. The two disks, however, must be of the same size and type. If the destination drive is the same as the source drive, the program will prompt the user to swap disks during the duplication process.

CAUTION: Do NOT attempt to use DUPDSK to duplicate double density diskettes under version 2 of OS/A+. Unpredictable and disastrous results may occur! DO use DUPDBL (see previous section) for this purpose.

Section 4.10
---

command: HELP

purpose: This program provides a MENU of system commands to help the beginning user.

usage: HELP

arguments: none

options: none

Description

Although we firmly believe that the command system of the OS/A+ Console Processor (CP) is superior to a menu approach, we can readily understand how the wealth of flexibility offered may overwhelm the new user. Therefore, we have provided this HELP command which provides menu access to the most frequently used system commands.

To use the menu, simply type HELP (followed by a RETURN, please!) any time the CP system prompt appears (usually D1:, followed by the cursor).

The available options are numbered from 1 to 9. To choose an option, simply type a digit from 1 to 9, followed by a RETURN. (Any invalid choice will exit the menu, back to OS/A+.)

Note that each of these options (except number 9) are exactly equivalent to an OS/A+ CP command. In the list which follows, the menu option is followed by its OS/A+ equivalent and a short description of its effect.

1. CAR — Runs any cartridge plugged into the left cartridge slot. Always does a "cold start" of the cartridge.

2. COPY — Allows the user to specify two filenames. Will copy a file from the "FROM" file to the "TO" file. [Allows use of ambiguous (wild card) names. Use "*" and "?" in filenames with caution.]

3.	DIR	Allows the user to specify a filename (including an ambiguous filename with "*" or "?") and then lists all files which match the given name. If just RETURN is given instead of a filename, will list all files on the "current" disk drive. [See section 2.2 for info on how to change the "current" disk.]
4.	COPY	"Duplicate a File". Special access into the COPY utility to copy a single file using a single disk drive. Not as fast as option 2, but for use on systems with only one disk drive.
5.	ERA	Allows the user to specify a filename. Erases the named file if it is on the current disk and if it is not protected.
6.	PRO	Allows the user to specify a filename. Sets the system status for the named file to "Protected" if the file exists. [Protected files cannot be ERAsed or written to.]
7.	REN	Allows the user to specify a FROM name and a TO filename. RENames the FROM file (if it exists and is not protected) to the TO filename (if it does not already exist).
8.	UNP	Allows the user to specify a filename. Resets the system "protected" status for the named file if it exists.
9.		Exit to OS/A+. An unnecessary option, actually, since a simple RETURN will exit also.

Section 4.11
---

command: INIT

purpose: This program initializes floppy disks so that they may be read from or written to

usage: INIT

arguments: none

options: none

Description

The INIT utility allows the user to format a floppy disk so that it may be read or written by programs. Under OS/A+ version 2, the user will be prompted for information on exactly how to initialize the disk (i.e., with or without a system file, etc.). Under version 4, a system file, DOS.SYS, is not written by the init program. In either case, the program will prompt for a drive to init. When the initialization process is complete, the floppy disk may now be used to store data. Under OS/A+ version 4, use COPY or DUPDSK to transfer a system file to the new disk, if desired.

Section 4.12

command: MAC65

purpose: Loads and executes the MAC/65 macro assembler

users: MAC/65 owners only

usage: MAC65 [file1 [file2 [file3]] [-A][-D]]

arguments: an optional set of one to three filename, construed to be the source, listing, and object files (respectively) of a MAC/65 assembly.

options:

-A	source file is Ascii
-D	assembly must be Disk-to-Disk

Description:

This command loads and executes the file MAC65.COM, the OSS Macro Assembler/Editor. If no filenames are given, MAC/65 will be invoked in its interactive (Editor) mode. Programs or text may then be edited and/or assembled. See the MAC/65 manual for further details.

If one or more files are specified, MAC/65 will be invoked in its "batch" mode. That is, it will perform a single assembly and then return to OS/A+. Generally, this command line will perform the assembly in a manner equivalent to giving the "ASM" command from the MAC/65 Editor. That is, if only one filename is given, it is assumed to be the source file, implying that the listing will go to the screen and the object code will be placed in memory (but only if requested by the .OPT OBJ directive). If a second filename is given, it is assumed to be the name of the listing file. Only if all three filenames are given will the object code be directed to the file specified.

Note: if an assembly needs no listing but does need an object file, the user may specify E: as the listing file, thus sending the listing to the screen.

Options

The -A option is used to specify that the source file is not a standard MAC/65 SAVEd file but is instead an Ascii (or Atascii) file. This is equivalent to using the interactive Editor mode of MAC/65 to use the sequence of commands "ENTER#D..." and "ASM ,...".

The -D option is used to specify that the assembly MUST proceed from disk to disk. If this option is not given, the source file is LOADed (or ENTERed) before the assembly, and then the assembly proceeds with the source in memory (generally producing improved speed of assembly). If, however, the source file is too large to be assembled in memory, the user may use this option to allow assembly of even very large programs. (And remember, even if the source fits, the macro and symbol tables must reside in memory during assembly also.)

NOTE: the -D option can NOT be used in conjunction with the -A option. The source file assembled under the -D option MUST be a properly SAVEd (tokenized) file.

*** FOR MORE INFORMATION, SEE YOUR MAC/65 MANUAL ***

Section 4.13
---

command: RS232

purpose: installs the serial device handlers ("Rn:") for use with the Atari 850 Interface Module.

users: Atari users with 850 Modules

usage: RS232

arguments: none

options: none

Description:

Using the command RS232 from OS/A+ is functionally equivalent to using Atari's AUTORUN.SYS file (which boots the R: handlers at power on time under Atari DOS). The driver for the various RS232 functions is loaded at LOMEM, LOMEM is moved, and the R: device is hooked into the handler table.

CAUTION: due to a bug in the software in the 850 Interface Module, hitting RESET will destroy the proper LOMEM pointer, effectively ignoring the space occupied by the RS232 handlers.

CAUTION: the 850 Interface Module is sometimes too intelligent for its own good. In particular, one cannot generally reload the software from the module without turning the module off and back on again.

After giving the RS232 command, if the Dn: prompt appears again below the line containing the "RS232" command, the Interface Module has loaded its software properly. If, however, the screen clears and the Dn: prompt appears at the top of the screen, something went wrong during the loading process. Unfortunately, the software in the Interface Module does not return a usable error code, preferring instead to do a system warmstart (hence the cleared screen).

---this page intentionally left blank--

+

+

---this page intentionally left blank--

+

+

Chapter 5: Interfacing to OS/A+

As mentioned in Chapter 1, OS/A+ is designed as a layered operating system. Application programs (including languages such as BASIC A+) are expected to call the operating system "properly", through the system call vector (labled "CIO"in SYSEQU.ASM). In turn, CIO will determine which device is to receive what I/O request and handles most of the work tranparent to the calling program.

If a program restricts itself to proper calls to CIO using labels provided in SYSEQ.ASM, the program should transfer virtually without change form one version of OS/A+ to another. (Probably the only other areas of change would involve memory map usage and the length of file names--12 bytes under version 2 and 30 bytes under version 4.)

In any case, here with is a description of the proper assembly language calling sequnces and parameters under OS/A+.

5.1 The Structure of the IOCB's

When a program calls the OS through location "CIO", OS expects to be given the address of a properly formatted IOCB (Input Output Control Block). (This does not apply to global commands under Apple version 4; see section 5.6 for more details on global OS commands) For simplicity, we have predefined 8 IOCB's, each 16 bytes long, and the program specifies which one to use by passing the IOCB number times 16 in the 6502's X-register. Thus, to access IOCB number four, the X-register should contain $40 on entry to OS. Notice that the IOCB number corresponds directly to the file number in BASIC (as in PRINT #6, etc.). Actually, the IOCB's are located from $0340 to $03BF on the Atari and from $BD00 to $BD7F on the Apple II (but you really should use the equates from the disk file "SYSEQU.ASM" rather than relying on hard-coded addresses.)

When OS gets control, it uses the X-register to inspect the appropriate IOCB and determine just what it was that the user wanted done. Figure 5.1 gives the OS/A+ standard names for each field in the IOCB along with a short description of the purpose of the field. Study the figure before proceeding.

The user program should NEVER touch fields ICHID,ICDNO, ICSTA and ICPTL/ICPTH. In addition, unless the particular device and I/O request requires it, the program should not change ICAX1 through ICAX6. The most important field is the one-byte command code, ICCOM, which tells the operating system what function is desired.

FIGURE 5-1

IOCB STRUCTURE

FIELD NAME	OFFSET WITHIN IOCB (bytes)	SIZE OF FIELD (bytes)	PURPOSE OF FIELD
ICHID	0	1	SET BY OS. Index into device name table for currently OPEN file, set to $FF if no file open on this IOCB.
ICDNO	1	1	SET BY OS. Device number (e.g., for "D1:xxx" or 2 for "D2:yyy")
ICCOM	2	1	The COMMAND request from user program. Defines how rest of IOCB is formatted.
ICSTA	3	1	SET BY OS. Last status returned by device. Not necessarily the status returned via STATUS command request.
ICBADR	4	2	BUFFER ADDRESS. A two byte address in normal 6502 low/high order. Specifies address of buffer for data transfer or address of filename for OPEN, STATUS, etc.
ICPTL	6	2	SET BY OS. Address minus one of device's put-one-byte routine. Possibly useful when high speed single byte transfers are needed.
ICBLEN	8	2	BUFFER LENGTH. Specifies maximum number of bytes to transfer for PUT/GET operations. NOTE: this length is decremented by one for each byte transfered.

ICAUX1	10	1	Auxiliary byte number one. Used in OPEN to specify kind of file access needed. Some drivers can make additional use of this byte.
ICAUX2	11	1	Auxilliary byte number two. Some serial port functions may use this byte. This and all following AUX bytes are for special use by each device driver.
ICAUX3 ICAUX4	12	2	For disk files only: where the disk sector number is passed by NOTE and POINT. (These bytes could be used separately by other drivers.
ICAUX5	14	1	For disk files only: the byte-within-sector number passed by NOTE and POINT.
ICAUX6	15	1	A spare auxilliary byte.

FIGURE 5-1

IOCB STRUCTURE

IOCB field name	0	1	2	3	4	5	6	7
					BUFFER ADDRESS		PUT-A-BYTE ADDRESS	
Type of command	IOCHID	ICDNO	ICCOM	ICSTA	ICBADR		ICPUT	
OPeN	*	*	3	*	filename		*	
CLOSE	*		12	*				
dynamic STATus		*	13	*	filename			
Get TeXT Record			5	*	buffer			
Put TeXT Record			9	*	buffer			
Get BINary Record			7	*	buffer			
Put BINary Record			11	*	buffer			
EXTENDED COMMANDS: DISK FILE MANGER ONLY								
REName		*	32	*	filename			
ERAse		*	33	*	filename			
PROtect		*	35	*	filename			
UNProtect		*	36	*	filename			
NOTE			38	*				
POINT			37	*				

LEGEND:

'*'	Set by OS when this command is used
'buffer'	Address of a data buffer
'filename'	Address of a filename

8 / BUFFER LENGTH / ICBLEN	9	10 / ICAUX1	11 / ICAUX2	12 / ICAUX3	13 / ICAUX4	14 / ICAUX5	15 / ICAUX6	IOCB field name (as given in SYSEQU.ASM) / COMMAND NAMES
		mode						COPN
								CCLOSE
								CSTAT
length								CGTXTR
length								CPTXTR
length								CGBINR
length								CPBINR
(See section 5.4)								
								CREN
								CERA
								CPRO
								CUNP
				sec num	byte			CNOTE
				sec num	byte			CPOINT

'length' Length of a data buffer
'mode' Mode of OPEN (i.e., read, write, etc.)
'sec num' Sector number, see section 5.4.3
'byte' Byte in sector, see section 5.4.3

5.2 The Standard OS/A+ Commands

The OS itself only understands a few fundamental commands, but OS/A+ also provides for extended commands necessary to some devices (XIO in BASIC). In any case, each of these fundamental commands deserves a short description.

5.2.1 OPEN

Open a device (synonyms: file, IOCB, channel) for read and/or write access. OS expects ICAX1 to contain a byte that specifies the mode of access: ICAX1 = 4 for read access, 8 for write access, and 12 for both read and write access. (Note: the disk file manager and the screen device handler allow other modes, and they will be discussed in a later section.) The name of the device (and, for the disk, the file) must be given to OS; this is accomplished by placing the ADDRESS of a string containing the name in ICBAL/ICBAH.

5.2.2 CLOSE

Terminate access to a device/file. Only the command must be given.

5.2.3 STATUS

Request the status of a device/file. The device can interpret this request as it wishes, and pass back a (hopefully) meaningful status. As with OPEN, the ADDRESS of a filename must be placed in ICBAL/ICBAH.

5.2.4 GET TEXT

A powerful command, this causes the OS to retrieve ("GET") bytes one at a time from a device/file already OPENed until either the buffer space provided by the user is exhausted or a RETURN character (Atari $9B, Apple $0D) is encountered. The user specifies the buffer to use by placing its ADDRESS in ICBAL/ICBAH and its maximum size (length) in ICBLL/ICBLH.

5.2.5 PUT TEXT

The analogue of GET TEXT, OS outputs characters one at a time until a RETURN is encountered or the buffer is empty. Requires ICBAL/ICBAH and ICBLL/ICBLH to be specified.

5.2.6 GET DATA

Extremely flexible command, this causes OS to retrieve, from the device/file previously OPENed, the number of bytes specified by ICBLL/ICBLH into the buffer specified by ICBAL/ICBAH. NO CHECKS WHATSOEVER ARE PERFORMED ON THE CONTENTS OF THE TRANSFERRED DATA.

5.2.7 PUT DATA

Similar to GET DATA, except that OS will output ICBLL/ICBLH bytes from the buffer specified by ICBAL/ICBAH. Again, no data checks are performed. Figure 5.2 provides a table of OS commands and their usage of the various fields of the IOCB's. For convenience, the disk file manager extended commands are also shown, and they will be discussed in section 5.4.

5.3 Device Names

Device names under OS/A+ are very simplistic; they consist of a single letter (optionally followed by a single numeral). Traditionally (and, in the case of all Apple II files and both Atari and Apple disk files, of necessity) the device name is followed by a colon. The following devices are implemented under standard OS/A+:

E: The keyboard/screen editor device. The normal console output.

K: The keyboard alone. Use this device to bypass editing of user input.

S: The screen alone. Can be either characters (ala E:) or graphics.

P: On the Atari, the printer. The standard device driver allows only one printer. On the Apple, any standard "Port" card may be accessed via this driver. The device number (e.g.,"P3:") specifies the slot number of the card.

C: The cassette recorder. (CAUTION: not imple- mented on the Apple II)

D: The disk file manager, which also usually requires a file name.

Other device names are possible (e.g., for RS-232 interfaces), and in fact the ease with which other devices may be added is another mark for the claim that OS/A+ is a TRUE operating system. The structure of device drivers is material for a later chapter, but we should like to point out that, on the Atari, the OS ROM includes drivers for all the above except the disk. In fact, the drivers account for over 5K bytes of the ROM code. The screen handler, with all its associated editing and GRAPHICS modes, occupies about 3K bytes of that. Of course, on the Apple II, all drivers are loaded in at system boot time from the disk.

5.4 Commands Unique to the Disk File Manager System

Figure 2 showed several OS/A+ system commands not discussed up until now. Generally, these "extended" commands are accessed via the extended request routine in a device driver's handler table (see chapter 8 for details on device drivers). However, some of these extended commands as implemented for the disk device in the File Manager System are important enough to deserve their own discussions. In this section, we examine each of the extended disk operations in a little detail:

5.4.1 Erase, Protect, and Unprotect

Also known as Delete, Lock, and Unlock, these three commands simply provide OS with a channel number (i.e., the X-register contains IOCB number times 16), a command number (ICCOM), and a filename (via ICBAL/ICBAH). When OS passes control to the FMS, an attempt is made to satisfy the request. Note that the filename may include "wild cards", as in "D:*.??S" (which will affect all files on disk drive one which have an 'S' as the last letter of their filename extension).

5.4.2 Rename

Very similar to ERASE, et al, in usage. The only difference is in the form of the filename. Proper form is: "Dn:oldname.ext,newname.ext" Note that the disk device specifier is not and CAN NOT be given twice.

5.4.3 Note and Point

Other than OPEN, these are the only commands encountered in standard OS/A+ which use any of the AUXilliary bytes of the IOCB. For these commands, one specifies the channel number and command number and then receives or passes file pointer information via three of the AUX bytes. ICAX3/ICAX4 are used as a conventional 6502 LSB/MSB 16-bit integer: they specify

the current (NOTE) or the to-be-made-current (POINT) sector within an already OPENed disk file. ICAX5 is similarly the current (NOTE) or to-be-made-current (POINT) byte within that sector. In the case of OS/A+ version 4, the word sector might be more properly replaced with the word "page", since the NOTE/POINT addressing always uses 256 byte pages, regardless of the physical sector size.

5.4.4 FMS Extensions of the OPEN Command

Open is not truly an extended operation, but for disk I/O we need to know that the FMS allows two additional "modes" beyond the fundamental OS modes (which are 4, 8, and 12 for read, write, and update). If ICAX1 contains a 6 when DFM is called for OPEN, then the disk DIRECTORY is opened (instead of a file) for read- only access. The filename now specifies the file (or files, if wild cards are used) to be listed as part of a directory listing. Note that FMS expects this type of OPEN to be followed by a succession of GETREC (get text line) OS calls (and we present an example of this below). If ICAX1 contains a 9, the specified file is opened as a write-only file, but the file pointer is set to the current end-of-file. Caution: version 2 FMS only appends on sector boundaries (normally this is transparent to the user, but caveat artificer). Finally, under version 4 FMS, mode 13 is also legal, specifying that the file be opened in "Append/Update" mode. See section 11.1 for more details on the meanings of other bits in ICAX1 and ICAX2 under version 4 of OS/A+.

5.5 ERROR HANDLING

This may not be the best place to introduce this topic, but the information is needed for examples which follow. There are four fundamental kinds of errors that can occur with OS/A+:

5.5.1 Hardware Errors

Such as attempting to read a bad disk, write a read-only disk, etc.

5.5.2 Data Transfer Errors

Errors which occur when data is transferred between the computer and a peripheral device. Examples include Device Timeout, Device NAK, Framing Error, etc.

5.5.3 Device Driver Errors

Found by the driver for the given device, as in (for the DFM) File Not Found, File Locked, Invalid Drive Number, etc.

5.5.4 OS Errors

Usually fundamental usage problems, such as Bad Channel Number, Bad Command, etc.

5.5.5 Error Codes Returned

On return from any OS call, the Y-register contains the completion code of the requested operation. A code of one (1) indicates "normal status, everything is okay". (I know, why not zero, which is easier to check for. Remember, we based this on Atari's OS ROMs, which are good, not perfect.) By convention, codes from $02 to $7F (2 through 127 decimal) are presumed to be "warnings". Those from $80 to $FF (128 through 255

decimal) are "hard" errors. These choices facilitate the following assembly language sequence:

```
     JSR CIOV  ; call the OS
     TYA               ; check completion code
     BMI OOPS  ; if $80-$FF, it must be an error
```

In theory, OS/A+ always returns to the user with condition codes set such that the TYA is unnecessary. In practice, that's probably true; but a little paranoia is often conducive to longer life of both humans and their programs.

5.6 CIO GLOBAL CALLS UNDER APPLE VERSION 4

OS/A+ Apple version 4 adds the capability of performing global system calls which do not reference an IOCB. These commands allow the user to change certain system parameters and default values. Just as CIO expects the number of the IOCB times 16 in the X register for normal commands, the number of the global command must be in the X register when CIO is called. Global commands are numbered from $C0 (192 decimal) to $FF (255 decimal), thereby avoiding a conflict with valid IOCB numbers (the values from $80 to $BF are reserved for future expansion of IOCB's and global commands). The use of the A and Y registers will vary depending on the particular global command desired.

Global system calls are accomplished in the following manner:

1) set up A, X, and Y to conform to the requirements of the particular command desired
2) call CIO through the CIO entry point (the same location which is used for normal system calls)
3) A, X, and Y now contain any values returned by the particular system call which was performed

The following global system calls are implemented:

5.6.1 Set Default Device

command value	(X reg) $FF 255 decimal
A register	device name-- ascii value (i.e., 'D'=$44)
Y register	sub-device number-- '1' to '9'
returns	no return values

When a file is opened under Apple version 4 and a device is not specified (i.e., a file name of "GEORGE" instead of "D1:GEORGE"), CIO will automatically use the current default device. This command allows the user to set the value of that default.

5.6.2 Return Default Device

command value	(X reg) $FE 254 decimal
A register	nothing
Y register	nothing
returns	device name in A sub-device number in Y-- '1' to '9'

This command allows the user to inspect the current value of the device system parameter.

5.6.3 Set System Date

command value (X reg)	$FD 253 decimal
A register	upper 4 bits: year lower 4 bits: month
Y register	day
returns	no return values

Apple version 4 supports a system date, although it does not have an internal clock. This command allows the user to set that date. The command expects the high 4 bits of the A register to contain the number of years past 1980 (i.e., a value of 4 means 1984) and the lower 4 bits to contain the current month number (1-12). The Y register will the current day of the month (1-31).

5.6.4 Return System Date

command value	$FC	252 decimal
A register	nothing	
Y register	nothing	
returns	system date in A and Y as in set system date command	

This command allows the user to inspect the current value of the system date paramater.

CHAPTER 6: Batch Processing Detailed

Chapter 1 provided an introduction to the capabilities of "Batch" execution. This chapter provides further details for the user who would like to create a custom "EXeCute" file.

6.1 Execute File Format

An execute file is simply a text file. Each line of the text file will become a CP command when executed. The three basic rules of text file lines are that:

1) they must contain valid OS/A+ console processor commands,
2) they must be less than 60 characters in length
3) they must end in a carriage return (ATASCII $9B on the Atari; ASCII $0D on the Apple).

OS/A+ allows the commands in an execute file to be preceeded by numbers and blanks. This feature allows the command lines to be numbered for readability and to document their purposes. The execute file line:

```
LOAD      OBJ.TEST <return>
```

and the line:

```
100 LOAD   OBJ.TEST <return>
```

are the same to OS/A+ . OS/A+ scans the line for the first non- numeric, non-blank character before starting to scan the command word. Virtually any text editor, including the editors of MAC/65 or EASMD, can be used to create and modify execute files.

Note for Atari users: One may also create an execute file (or, for that matter, any text file) by using "COPY E: <diskfile>". (COPY will clear thescreen, at which time you simply type in your text, line by line. You terminate the copy by pressing CTRL-3, the end of file signal for the E: device.)

6.2 Execute Intrinsic Commands

OS/A+ has four special intrinsic commands designed for use exclusively with execute files. These commands are:

REMARK	Remark or comment (does nothing)
NOSCREEN	Turn of Echo of execute file command lines to the screen. (Default Mode)
END	Stop executing the execute file and return OS/A+ to keyboard entry mode (the CP).

See the section on intrinsic commands for a more detailed explanation of these commands.

6.3 EXECUTE FILE STOPS

While an execute file is being processed, various conditions may occur which will warrant a halt in the batch execution. These conditions may occur because of system-detected errors or because of a user program detecting a condition it considers hazardous to the system's health.

6.3.1 STOPS ISSUED BY OS/A+

Humans are not quite perfect in the eyes of computers and sometimes make mistakes. OS/A+ commands specified in error will generate error messages. If OS/A+ discovers an error while executing an EXECUTE file, it will print the error message as usual and STOP executing the EXECUTE file. Note that this error stop only occurs if the error is found by OS/A+, not just because a program generates an error.

Execution of an execute file will also stop after the CARTRIDGE commmand is executed.

6.3.2 STOPS ISSUED BY USER PROGRAMS

It is sometimes desirable for a program in a chain of executing programs to stop the execute process. The usual reason for this is that the program has detected an error severe enough to invalidate the processes performed by the following program(s). The continued execution of the execute files is provided for by a single byte flag within OS/A+. If a program sets this byte to zero, then upon returning to OS/A+ (DOS or CP BASIC statements) the execute file execution will immediately stop. The execute flag is located 12 bytes from the start of OS/A+, which is pointed to by memory location 10 ($0A). The following Basic program segment will turn off the execute file and return to OS/A+.

```
1000      CPADR = PEEK(11)*256 + PEEK (10)
1010      EXCFLG = CPADR + 12
1020      POKE  EXCFLG,0
1030      DOS
```

6.4 STARTUP.EXC

The execute filename STARTUP.EXC has special meanings in the OS/A+ system. When the system is first booted (power up), OS/A+ will search the directory of the booted disk volume for a file named STARTUP.EXC. If STARTUP.EXC is on the booted volume, OS/A+ will execute that file before requesting keyboard commands.

Chapter 7: User Written Extrinsic Commands

The writer of assembly language code will most likly need to interface with the Atari Operating System (OS). If the assembly code is to become an extrinsic command, there may be a need to interface to OS/A+ . See chapter 5 for further information about the OS interface. Sections 7.1 through 7.4, below, give general information about the operations of OS/A+, especially when running in a "batch" environment. Sections 7.5 and 7.6, then, give more detailed information about command line parsing and the like.

7.1 SYSEQU.ASM

Every OS/A+ master disk contains an assembler source file, SYSEQU.ASM, that has various commonly used Atari OS and OS/A+ system equates. This file may be included in an assembly language program via the OSS MAC/65 include function (.INCLUDE #D1:SYSEQU.ASM)

7.2 OS/A+ MEMORY LOCATION

OS/A+ is designed to placed just after the Atari File Manager. Since the actual location of OS/A+ may vary with different versions of a file manager, a fixed location has been assigned to point to OS/A+. The location CPALOC($0A on the Atari, see SYSEQU.ASM for Apple value) is the same one Atari uses to point to DUP. Most Atari programs the use a DOS exit vector through CPALOC.

7.3 EXECUTE PARAMETERS

The OS/A+ execute flag is located CPEXFL ($0B) from the start of OS/A+. The CPALOC may be used as an indirect pointer to access the execute flag.

```
       LDY     #CPEXFL          ;GET DISPL TO FLAG
       LDA     (CPALOC),Y       ;LOAO FLAG
```

The Execute Flag has four bits that control the execute process.

EXCYES	$80	If one, an execute is in progress
EXCSCR	$4&	If one, do not echo execute input to screen
EXCNEW	$10	If one, a new execute is starting. Tells OS/A+ to start with first line of the file
EXCSUP	$20	If one, a cold start execute is starting.Used to avoid file-not-found error if STARTUP.EXC is not on boot disk.

OS/A+ performs the execute function by opening the file, POINTing to the next line, reading the next line, NOTE the new next line and closing the file. To perform these functions, OS/A+ must save the execute file name and the three byte NOTE values. The filename is saved at CPEXFN ($0C) into OS/A+. The three NOTE values are saved at CPEXNP($1C) into OS/A+. (CPEXNP+0=ICAUX5; CPEXNP+1=ICAUX4; CPEXNP+2=ICAUX3). By changing the various execute control para- meters, a programmer can cause recursion and/or changing of ex- ecute files.

7.4 DEFAULT DRIVE

Under Atari version 2, the OS/A+ default drive file spec is located at CPDFDV ($07) into OS/A+. The Default Drive here is ATASCII Dn: where "n" is the ATASCII default drive number.

7.5 EXTRINSIC PARAMETERS

The extrinsic commands may be called with parameters typed on the command line. The OSS command

```
D1:COPY          FROMFILE          D2:TO FILE
```

is an example of this. The entire parameters line is saved in the OS/A+ input buffer located at CPCMDB ($40) bytes into OS/A+ and is available to the user. Since most command parameters are file names OS/A+ provides a means of extracting these parameters as filenames. The routine that performs this service begins at CPGNFN ($03) bytes into OS/A+ . The routine will get the next parameter and move it to the filename buffer at CPFNAM ($21) bytes in OS/A+. If the parameter does not contain a device prefix, then OS/A+ will prefix the parameter with tlhe default drive prefix. The first time COPY calls CPGNFN the file spec "D1:FROMFILE" is placed at CPFNAM. The second time COPY calls CPGNFN the file spec "D2:TO FILE" is placed in CPFNAM. If CPGNFN were to be called more times, then the default file spec would be set into CPFNAM at each call. To detect the end of parameter condition, the user may check the CPBUFP ($0A into OS/A+) cell. If CPBUFP does not change often a CPGNFN call then there are no more parameters. The filename buffer is always padded to 16 bytes with ATASCII EOL ($9B) characters. The following example sets up a vector for calling the get file name routine.

```
        CLC
        LDA     CPALOC          ;ADD CPGNFN
        ADC     #CPGNFN         ;TO CPALOC VALUE
        STA     GETFN+1         ;AND PLACE IN
        LDA     CPALOC+1        ;ADDRESS FIELD
        ADC     #0              ;OF JUMP
        STA     GETFN+2         ;INSTRUCTION
        ::::
GETFN   JMP     0
```

The following routine gets the next file name to CPFNAM.

```
        LDY     #CPBUFP         ;SAVE CPBUFP
        LDA     (CPALOC),Y      ;VALUE
        PHA
        JSR     GETFN           ;GET NEXT FILE PARM
        LDY     #CPBUFP
        PLA                     ;TEST FOR NO NEXT
        CMP     (CPALOC),Y      ;PARM
        BEQ     NONEXT          ;BR IF NO NEXTPARM

        LDY     #CPFNAM         ;ELSE GET FILE
        LDA     (CPALOC),Y      ;NAME FROM BUFFER
```

7.6 RUNLOC

The OS/A+ RUNLOC ($3D into OS/A+) is used as the OS/A+ vector to routines with the RUN,LOAD and extrinsic commands. An application that allows exits to OS/A+ can change RUNLOC to provide a warmstart re-entry to the application (if the user enters RUN with no parameters). If the application is not reusable and wishes to forbid re-entry, the high order byte RUNLOC ($3E into OS/A+) should be set to zero, thusly:

```
        LDY     #RUNLOC+1       ;FORBID RE-ENTRY
        LDA     #0              ;TO ME
        STA     (CPALOC),Y
```

Chapter 8: DEVICE HANDLERS

As we have noted before, CIO is actually a very small program (approximately 700 bytes). Even so, it is able to handle the wide variety of I/O requests detailed in the first two parts of this chapter with a surprisingly simple and consistent assembly language interface. Perhaps even more amazing is the purity and simplicity of the OS interface to its device handlers.

Admittedly, because of this very simplicity, OS/A+ is sometimes slower that one would wish (probably only noticeably so with PUT BINARY RECORD and GET BINARY RECORD) and the handlers must be relatively sophisticated. But not too much so, as we will show.

8.1 The Device Handler Table

At location "HATABS" in RAM, OS/A+ has (loaded from ROM on the Atari, loaded from disk on the Apple II) a list of the standard devices (P:, D:,E:,S:, and K:) and the addresses thereof. To add a device, simply tack it on to the end of the list: you need only specify the device's name (one character) and the address of its handler table (more on that in a moment).

In theory, all named device handlers under OS/A+ may handle more than one physical device. Just as the disk handler understands "D1:" and "D2:", so could a keyboard handler understand "K1:" and "K2:". (In fact, the Apple version implements multiple port drivers via "Pn:".) OS/A+ supplies a default sub-device number of "1" if no number is given (thus "D:" becomes "D1:").

Following is the layout of the HAndler TABleS on the Atari. The Apple II version is very similar.

```
*=      $031A
        HATABS
.BYTE   'P'     ; the Printer device
.WORD   PDEVICE ; and the address of its driver
.BYTE   'C'     ; the Cassette device
.WORD   CDEVICE
.BYTE   'E'     ; the screen Editor device
.WORD   EDEVICE
.BYTE   'S'     ; the graphics Screen device
.WORD   SDEVICE
.BYTE   'K'     ; the Keyboard device
.WORD   KDEVICE
.BYTE   0       ; zero marks the end of the
                  table
.WORD   0       ; ...but there's room for
                several
.BYTE   0       ; ...more devices
                et cetera
```

8.2 Rules for Writing Device Handlers

Each device which has its handler address placed into the handler address table (above) is expected to conform to certain rules. In particular, the driver is expected to provide six (6) action subroutines and an initialization routine. (In practice, the current OS/A+ only calls the initialization routines for its own pre-defined devices. Since this may change in future OS's and since one can force the call to one's own initialization routine, we must recommend that each driver include one, even if it does nothing.) The address placed in the handler address table must point to, again, another table, the form of which is shown below (Figure 8.1).

```
HANDLER
   .WORD  <address of OPEN routine>-1
   .WORD  <address of CLOSE routine>-1
   .WORD  <address of GETBYTE routine>-1
   .WORD  <address of PUTBYTE routine>-1
   .WORD  <address of STATUS routine>-1
   .WORD  <address of XIO routine>-1
   JMP    <address of initialization routine>
```

FIGURE 8.1

Notice the six addresses which must be specified; and note that in the table one must subtract one from each address (the "-1" simply makes CIO's job easier...honest). A brief word about each routine is given in the following pages.

8.2.1 Device OPEN

The OPEN routine must perform any initialization needed by the device. For many devices, such as a printer, this may consist of simply checking the device status to insure that it is actually present. Since the X-register, on entry to each of these routines, contains the IOCB number being used for this call, the driver may examine ICAX1 (via LDA ICAX1,X) and/or ICAX2 to determine the kind of OPEN being requested. (Caution: OS/A+ preempts bits 2 and 3, $04 and $08, of ICAX1 for read/write access control. These bits may be examined but should normally not be changed.)

8.2.2 Device CLOSE

The CLOSE routine is often even simpler. It should "turn off" the device if necessary and possible.

8.2.3 Device PUT and GET BYTE Routines

The PUTBYTE and GETBYTE routines are just what are implied by their names: the device handler must supply a routine to output one byte to the device and a routine to input one byte from the device. HOWEVER, for many devices one or the other of these routines doesn't make sense (ever tried to input from a printer?). In this case the routine may simply RTS and OS/A+ will supply an error code.

8.2.4 Device STATUS Routine

The STATUS routine is intended to implement a dynamic status check. Generally, if dynamic checking is not desirable or feasible, the routine may simply return the status value it finds in the user's IOCB. However, it is NOT an error under OS/A+ to call the status routine for an unOPENed device, so be careful.

8.2.5 Device Extended I/O Routine(s)

The XIO routine does just what its name implies: it allows the user to call any and all special and wonderful routines that a given device handler may choose to implement. OS does nothing to process an XIO call except pass it to the appropriate driver.

8.2.6 General Comments on Device I/O Routines

In general, the AUXilliary bytes of each IOCB are available to each driver. In practice, it is best to avoid ICAX1 and ICAX2, as several BASIC and OS commands will alter them to their will. Note that ICAX3 thru ICAX5 may be used to pass and receive information to and from BASIC via the NOTE and POINT commands (which are actually special XIO commands). Finally, drivers should not touch any other bytes in the IOCBs, especially the first two bytes.

Notice that handlers need not be concerned with PUT BINARY RECORD, GET TEXT RECORD, etc.: OS performs all the needed housekeeping for these user-level commands.

8.3 Rules for Adding Things to OS

1. Inspect the system MEMLO pointer (see SYSEQU.ASM for the actual location).
2. Load your routine (including needed buffers) at the current value of MEMLO.
3. Add the size of your routine to MEMLO.
4. Store the resultant value back in MEMLO.
5. Connect your driver to OS by adding its name and address into the handler address table.
6. For Atari handlers only:
   Fool OS so that if SYSTEM RESET is hit steps 3 thru 5 will be reexecuted (because SYSTEM RESET indeed resets the handler address table and the value of MEMLO).

In point of fact, step 2 is the hardest of these to accomplish. In order to load your routine at wherever MEMLO may be pointing, you need a relocatable (or self-relocatable) routine. Since there is currently no assembler for OS/A+ which produces relocatable code, this is not an easy task. But it may not be necessary if you are writing code for your own private system, as opposed to for general consumption.

Step 6 is accomplished by making Atari OS think that your driver is the Disk driver for initialization purposes (by "stealing" the DOSINI vector) and then calling the Disk's initializer yourself before steps 3 thru 5 are performed again.

8.4 AN EXAMPLE PROGRAM

This driver, included in source form on your disk as "MEM.LIS", builds a new driver and adds it to the operating system. The "device" being driven is simply excess system memory within your computer. Thus, you may (for example) use this are as a pseudo-disk file for passing data between sequentially called programs.

Some words of caution are in order. This driver does NOT perform step 6 as noted in the last section (but it may be reinitialized via a BASIC USR call). It does NOT perform self-relocation: instead it simply locates itself above all normal low memory usage (except the serial port drivers, which would have to be loaded AFTER this driver). If you assemble it yourself, you could do so at the MEMLO you find in your normal system configuration (or you could improve it to be self-modifying, of course).

Other caveats pertain to the handler's usage: it uses RAM from the contents of MEMTOP downward. It does NOT check to see if it has bumped into BASIC's MEMTOP ($90) and hence could conceivably wipe out programs and/or data. To be safe, don't write more data to the RAM than a FRE(0) shows (and preferrably even less).

In operation, the M: driver reinitializes upon an OPEN for write access (mode 8). A CLOSE followed by a subsequent READ access will allow the data to be read in the order it was written. MORE CAUTIONS: don't change graphics modes between writing and reading if the change would use more memory (to be safe, simply don't change at all). The M: will perform almost exactly as if it were a cassette file, so the user program should be data sensitive if necessary: the M: driver will NOT itself give an error based on data contents. Note that the data may be re-READ if desired (via CLOSE and re-OPEN).

A suggested set of BASIC programs is presented on the next page.

Ending of PROGRAM 1:

```
9900 OPEN #2,8,0,"M:"
9910 PRINT #2; LEN(A$)
9920 PRINT #2; A$
9930 CLOSE #2
9940 RUN "D:PROGRAM2"
```

Beginning of PROGRAM 2:

```
100 OPEN #4,4,0,"M:"
110 INPUT #4,SIZE
120 DIM STRING$(SIZE)
130 INPUT #4, STRING$
140 CLOSE #4
```

BASIC A+ users might find RPUT/RGET and BPUT/BGET to be useful tools here instead of PRINT and INPUT. And, of course, users of any other language(s) might find this a handy inter-program communications device.

CHAPTER 9: VERSION 2 FILE STRUCTURE

Structure of the OS/A+ version 2 File Management System

OS/A+ version 2 was produced to provide the maximum compatibility possible with Atari's DOS 2.0s. In fact, the FMS used is identical to that used by Atari (for a simple reason: we wrote Atari's DOS). For reasons known best to Atari, we were instructed to create Atari's FMS around a linked-sector disk space management scheme. In essence, this means that the last three bytes of each sector in a disk file contain a link to the next sector in that same file. The positive result of this is that one produces a relatively small, memory-resident, disk manager which is nevertheless capable of dynamically allocating diskette space (unlike, for example, a contiguous file disk manager). The biggest disadvantage of the scheme seems to be that one may not do direct (random) access to the bytes of such files, as one CAN do with either a contiguous or mapped file allocation technique. Also, a disk error in the middle of a linked file means a loss of access to the rest of the file.

The purpose of FMS is to organize the 720 data sectors avilable on an 810 (or its double density equivalent) 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 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 dat sector in the file. Figure 9-1 illustrates the relation between the Directory and the Data files.

NOTE: since double density diskette sectors contain 256 bytes whereas single density (810 drive) sectors contain only 128, certain absolute byte number references may vary depending upon the diskette in use. Throughout this chapter, in such cases, the single density number is given followed by the double density number in square brackets [thus].

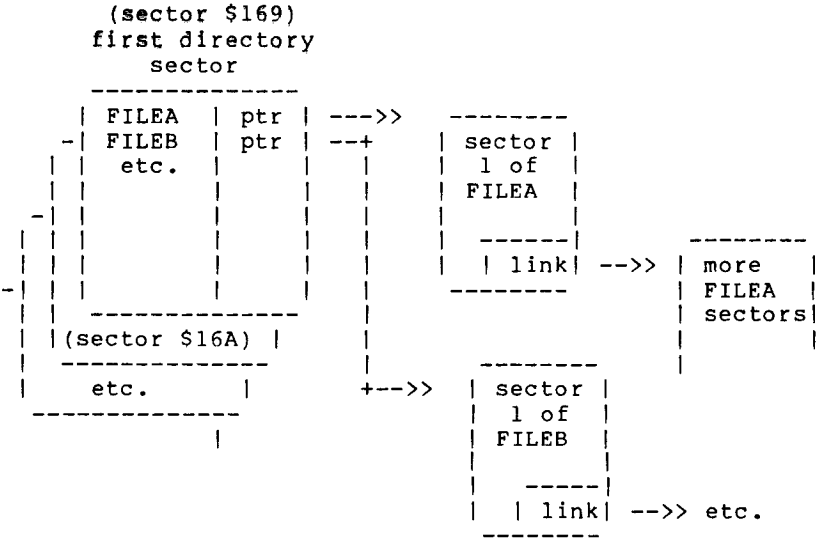

Figure 9-1

Version 2 Directory Structure

Note that only eight file directory entries are stored per sector, even on double density diskettes.

9.1 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 9-2. 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.

Starting Byte # of Field	Length of Field (bytes)	Purpose of Field
0	1	Flag byte. Meanings of bits: $00 Entry never used $80 Entry was deleted $40 Entry in use $20 Entry protected $02 a version 2 file $01 Now writing file
1	2	Count (LSB,MSB) of sectors in file
3	2	Start sector (LSB,MSB) of link chain
5	8	File name, primary
13	3	File name, extension

Figure 9-2

Directory Entry Structure

9.2 DATA SECTORS

A Data Sector is used to contain the file's data bytes. Each 128 [256] byte data sector is organized to hold 125 [253] bytes of data and three bytes of control information as shown in Firgure 9-3. The data bytes start with the first byte (byte 0) in the sector and run contiguously up to, and including, byte 124 [252]. The control information starts at byte 125 [253].

The sector byte count is contained in byte 127 [255]. This value is the actual number of data bytes in this particular sector. The value may range from zero (no data) to 125 [253] (a full sector). Any data sector in a file may be a short sector (contain less than 125 [253] data bytes).

The left six bits of byte 125 [253] contain the file number of the file. This number correspoinds to the location of the file's entry in the Directory. Directory entry zero in Directory sector $169 has a file number of zero. Entry one in Directory sector $169 has a 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 [253] (and all eight bits of byte 126 [254]) 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.

9.3 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 9-3 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 respresents 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.

Starting Byte # of Field	Length of Field (bytes)	Purpose of Field
0	1	Reserved (for type code)
1	2	Total number of sectors
3	2	Number of unused sectors
5	5	Reserved
10	90	Sector usage bit map Each bit represents a particular sector: a 1 bit indicates an available sector, a 0 bit indicates a sector in use.
100	28	Reserved (could be used for version 2 type DOS with more than 720 sectors per disk)

Figure 9-3

Structure of the VTOC Sector

CHAPTER 10: OS/A+ -- Version 4

OS/A+ version 4 is an operating system which provides all the power and flexibility of the Atari CIO scheme. But OS/A+ also uses an advanced File Management System (FMS) to provide fast and effective random access files for virtually any disk drive. Although OS/A+ comes with a console processor (which is functionally equivalent to that used by Digital Research's CP/M), the Console Processor (CP) is NOT necessarily an integral part of this system. In any case, the CP provides many features and conveniences not available via the standard Atari menu DOS.

10.1 AN OVERVIEW OF THE STRUCTURE OF VERSION 4

On the Apple II, the FMS of OS/A+ is fully upward compatible with the file scheme of Apple DOS 3.3, even though it allows disk drive capacites from 128K bytes to over 15 Megabytes. (In all cases, said capacities also reflect the maximum size of file accessible under OS/A+, except that a disk size might be 30 MB while a file cannot exceed 16MB.)

OS/A+ version 4, as it appears on the Atari, is virtually identical with OS/A+ as implemented on the Apple II, excepting only that it will not be compatible with ordinary Apple II diskettes (which are not readable by standard controller chips in any case). A diagrammetic overview of the file structure of OS/A+ version 4 follows:

```
-------------------------------------------------------
| filename1 / pointer 1 | filename2 / pointer2 | ...
-------------------------------------------------------
        [ disk directory structure ]
```

where the file pointers in turn point to:

```
-------------------------------------------------------
| file creation data | map of blocks ..... (link)|
-------------------------------------------------------
```

The link, which occurs only if a file's map exceeds its block size, points to another block containing a continuance of the map of blocks.

The map of blocks (actually a map of sets of sectors, to be discussed below) simply consists of a series of 2-byte disk addresses, each such address pointing to a block of sectors.

```
-----------------------------------------------
| sectors... | ...any... | ...data... | ...valid. |
-----------------------------------------------
```

There are no links to succeeding sectors or blocks, so the entire space of each sector (and block) is available for any data in any form.

There are several non-obvious advantages to this scheme, so bear with us as we try to explain some of them.

A. We are able to handle disks with 128, 256, or 512 bytes per sector. (To be truthful, with 128 byte per sector drives we would use pairs of sectors to emulate 256 byte sectors, since a 128 byte file map is not really adequate.)

B. We allow each disk drive to be assigned its own "drive blocking factor". That means that a quadruple density floppy might have blocks consisting of two 256-byte sectors while a 10 MB disk might use blocks of four 512-byte sectors. Note that this concept of blocking factors is not new or unique: CP/M 2.2 allows blocking factors of 1KB, 2KB, and 4KB, depending on disk size. We are simply a little more flexible.

C. We allow each file to be assigned its own "file blocking factor". Thus, even on a floppy with 512 byte blocks, a given FILE may use 8 KByte blocks, thus guaranteeing at most one disk read to access any given sector of the file. (On the Apple II version of this product, where the drive blocking factor is perforce 1 for standard Apple drives, a file blocking factor of 8 -- 2 KByte blocks -- essentially doubles random access speed.)

D. Although not yet implemented nor planned for first release, the directory structure is set up in such a way that, if desired, we could implement multiple and/or hierarchical directories (ala UNIX, for example). Even CIO (on the Apple II) has been altered to support the concept of a default device and/or directory.

E. Random access files are easy and practical. Unix-like "LSEEKs" are accomplished (via the "POINT" XIO call) to any byte of any file.

F. Except for those rare programs that somehow depend on having 125 bytes of data per sector, current Atari application programs (including Atari BASIC and programs written thereunder) will notice NO CHANGE in their interface with the operating system. Of course, some of the currently unused options will be available to take advantage of such features as file blocking factors, but they will not be necessary to the proper functioning of the system.

10.2 DISK FILE STRUCTURE

OS/A+ version 4 utilizes a mapped file structure which allows true random access to data files. In such a scheme, special segments of a file act as pointers to the blocks of data comprising a file. By allowing the user to specify the size of the data blocks pointed to, OS/A+ is able to handle large and small files and disks with unsurpasssed speed and utility.

The OS/A+ random access file managment system treats each disk under its control as a collection of continuos physical sectors of either 256 or 512 bytes in length, which are numbered starting with sector zero. These sectors are logicaly grouped into blocks of n sectors in length, where n is a power of two between 1 and 128. All files on a disk are allocated space in segments of at least one block in length.

10.2.1 THE VTOC

In order to keep track of what blocks on a partiular disk are available for use,the file manager maintains a special section on each disk known as the table of contents, or VTOC. The VTOC on a disk consists of from 1 to 255 sectors which contain a sequence of bytes, known as the bitmap. Each bit of the bitmap may be turned off or on to allocate or free a specific block on the disk. Note that the VTOC is the only data area on the disk allocated by sectors instead of blocks. The format of the VTOC is as follows:

byte(s)	value
0	unused (must be 2 on Apple Disk II)
1-2	block no. of first directory sector
3	unused (must be 4 on Apple Disk II)
4-5	unused
6	unused (must be $FE on Apple Disk II)
7-26	unused
27	max. number of pointers in file map sectors ($7A for disks with 256-byte sectors; $F4 for 512-byte sector disks)
28-2F	unused
30-33	unused (must be $FF,$FF,0,0 on Apple)
34	unused (must be $23 on Apple)
35	unused (must be $0on Apple)
36-37	unused (must be 0,1 on Apple)
38-	disk block bit map

10.2.2 THE DIRECTORY

The disk directory holds information describing existing files on a particular disk. The directory is allocated on the disk by disk blocks, each sector of which holds information on a certain number of files. The blocks themselves are singly linked, each one holding the disk address (block number) of its successor with the last block having a null link. Each sector of the directory contains a number of entries (7 for 256-byte sectors, 14 for 512-byte sectors) each describing a particular file. The entry for a single file contains the file's name, its length in sectors (see exceptions for dos 3.3 diskettes), its type and blocking factor (see sction immediately following), and a pointer to the start of the file map for that file.

The file name consists of a string of up to 30 characters excluding spaces, commas, carriage returns, or nulls. Characters within a directory entry will have their upper bit inverted. the file name will be padded at the right with inverted blanks (hex $A0).

The file type byte is used as follows:

bits 0-3: file use type -- values 0,1,2, and 4 are currently used on the Apple system while only 0 is used on the Atari.

bits 4-6: file blocking factor -- a value from 0 to 7 (see section 10.2.3 on file map)

bit 7: file protection -- if this bit is set then the file is protected from accidental write access, erasure, or renaming. The pointer to the start of the file map is a two byte value which is the disk block number of the first file map map block.

Format of directory sectors:

byte(s)	value
0	unused
1-2	block number of next directory block (0 if this is the last block)
3-A	unused
B-	directory entries (35 bytes each)

Format of directory entries:

byte(s)	value
0-1	block number of first block in file map
2	file type
3-20	file name
21-22	length of file in sectors (on Apple Disk II files, the count of sectors actually used by the file)

10.2.3 THE FILE MAP

As previously mentioned OS/A+ version 4 utilizes a mapped file structure where special portions of a file point to the locations on the disk holding the actual data. These special sections comprise the file map, which is a singly linked list of disk blocks which contain pointers to the data blocks of a file. Each sector within a file map block has the following format:

byte(s)	value
0	unused
1-2	block number of next file map block (zero if this block is the last)
3-4	unused
5-6	unused (reserved on Apple)
7-B	unused
C-D	block number of first data block in file
E-	block numbers of data blocks

A pointer to a file block is merely the disk block number of the start of a file block. For most purposes, a file block is equivalent in size to a disk block, However, the file manager allows the user to specify a file blocking factor which alters the size of data blocks for a single file. A file blocking factor of zero implies that file blocks are equivalent to disk blocks. A file blocking factor of 1, though, makes file blocks equivalent to 8 disk blocks in length. Similarly, a factor of 2 creates file blocks which are 16 disk blocks in length. While the use of a file blocking factor has little or no consequence for sequentially accessed files, it offers 2 distinct advantages for random access files. First, the size of the file map will be reduced, thereby decreasing the average number of disk accesses requrired to access data. Second, the file's data sectors are likely to be less fragmented on the disk, thereby decresing the average head movement required to access data sectors. The only disadvantage of using a file blocking factor is that disk space is allocated much more rapidly than otherwise, making this technique undesirable for small files.

10.3 BUFFER ALLOCATION

The file manager requires a continuous block of memory from which to allocate buffer space. The address of the start of this space is contained in location BUFSTART (see system memory map), and its length in pages (256 bytes) is in BUFLEN. The values in these locations may be changed by the user whenever all files are closed; however, the system must then be re-initialized either by jumping to the system reset location or by calling the file manager initialization subroutine (see memory map).

Space is allocated at system initialization time for the buffers which will contain the VTOC's of each drive in the system. When a file is opened, buffers are allocated for both the file map and data sectors. All buffers allocated are the same length as the sectors on the particular disk involved.

Example: Suppose a system has 2 disks with 256-byte sectors. There will be 2 VTOC buffers (one for each drive), a total of 512 bytes. Each open file will have file map and data buffers, a total of 512 bytes per open file. Therefore, allowing for 3 open files at one time, there should be 3 512-byte buffers for the files, plus the 512 bytes of buffer for the VTOCs, or 2K bytes of total buffer space available (i.e., BUFLEN = 8 pages).

10.4 ADDING DRIVES

In order to integrate a new disk drive into the system, the disk must first be initialized with the OS/A+ verion 4 file structure. Please refer to the section describing the INIT utility for instructions on using INIT. Source code of INIT is available for users contemplating adding their own disk devices.
In order to access a new drive, it must be installed into the disk drive table within the file manager. This table consists of 8 entries of 16 bytes each starting at location DRVTAB (see memory map). Each entry in this table contains the folowing information:

byte(s)	value	
0	drive type--	1=Apple Disk II: 0=other disk
1	reserved	
2	size of disk sectors--	1=256 byte 2=512 byte
3	disk blocking factor; sectors/block	
4-5	sector no. of disks VTOC	
6	no. of sectors in VTOC	
7-8	address of read/write sector routine minus one **	
9-10	(set by file manager)	
11	file map sector size-- $7A for 256 byte sectors; $F4 for 512 byte sectors	
12-15	(set by file manager)	

As can be seen, the drive table entry contains the address of the routine to read and write sectors on the disk. This routine may be located anywhere in memory. (See the next section for a description of the parameters to this routine).

Once the drive has been added to the drive table and the read write sector routine has been loaded, the system must be re-initialized by jumping to the system reset location. The new disk may then be accessed as Dn: where the disk is the nth entry in the drive table (n starts from 1).

** Note that more than one disk drive may be serviced by a single read/write sector routine.

10.5 THE READ/WRITE SECTOR ROUTINES

Each read/write sector routine receives its parameters through the Device Control Block, or DCB (see memory map). The format of the DC is as follows:

Byte 0:	machine dependent (unused on Apple)
Byte 1:	DCBDRV disk drive number (1-8).
Byte 2:	DCBCMD command (0=null, 1=read, 2=write).
Byte 3:	DCBSTA status byte (set by read/write sector).
Bytes 4-5:	DCBBUF buffer address in low, high order.
Bytes 6-7:	machine dependent (unused on Apple)
Bytes 8-9:	machine dependent (unused on Apple)
Bytes 9-A:	DCBSEC sector number--to be accessed in low, high order

See the table in Chapter 13 for valid status values.

CHAPTER 11: VERSION DIFFERENCES

As much as we would like to, we can't produce all three versions of OS/A+ in such a manner that they will be 100% compatible. And, of course, there are also minor differences between OS/A+ and Atari DOS or Apple DOS, as appropriate.

Some of the differences are obvious: with the Apple version of OS/A+, you can have file sizes of up to 16 megabytes versus the Apple DOS limit of 130K bytes. Other differences are subtle: file names under OS/A+ cannot contain space characters, as can Apple DOS names.

In this chapter we will try to document as many known major differences as we can. We will probably miss one or two, so please don't hesitate to call or write us if you intend to write software which will run on more than one version of OS/A+. We will try to keep a list of any other differences we (or you) find.

Of course, if you are working entirely within one of our OS/A+ configurations, most of this is unneeded information. Perhaps you need to be aware only of the differences between OS/A+ and your computer's "normal" operating system.

11.1 FEATURES SPECIFIC TO VERSION 4 OF THE FILE MANAGER

This section applies to both the Apple II and Atari editions of Version 4 OS/A+.

11.1.1 RANDOM ACCESS:

As previously mentioned, OS/A+ version 4 allows true random access to data files. This capability is provided through the usage of the NOTE and POINT operating system calls. Unlike its predecessor, version 4 expects the data in ICAX3-ICAX5 to be a 24 bit RELATIVE position within a file (in mid, high,low byte order). This allows the user to use the POINT command to easily move to any position within a file of 16 megabytes or less in size. Similarly, the data returned by the NOTE command is a 3 byte RELATIVE position value which is placed in ICAX3-ICAX5 by the file manager. Notice also that a random access file may contain "holes" created by writing non-contiguous portions of a file without writing the intervening data.

11.1.2 FILE TYPES:

Version 4 of OS/A+ also supports a file type byte which is also used to indicate the file's file map blocking factor (see section 10.4) and its protection. When a file is created, this byte is set into the file's directory entry; its value is zero by default. In order to place a value other than zero into the file type byte when a file is created, bit 6 of ICAX1 must be set (i.e. add $40, or 64, to the mode during an OPEN command). The desired file type must be placed in ICAX2 (the second value in a BASIC open statement). Whenever a file is opened, the current value of its type byte is returned in ICAX2.

11.1.3 SUPPRESSING AUTOMATIC CREATION:

Normally, when a file is opened in mode 8, it is created, if the file did not previously exist, or truncated, if it did exist. It is occasionally desirable to suppress creation or truncation when opening a file in mode 8, in which case an error would

occur when attempting to open a non-existent file. If a file is opened in mode 8 and bit 5 of ICAX1 is set (i.e., add $20, or 32, to the mode value), then the creation or truncation of the file will be prevented.

11.2 FEATURES UNIQUE TO APPLE VERSION OF OS/A+

11.2.1 DEFAULT DEVICES

In all versions of OS/A+, filenames given at the CP level need not specify the disk device if it is the same as the current default device (that is, the same as the current CP prompt characters). Under version 4 OS/A+ for the Apple II, this default device concept is carried throughout the system, even in BASIC A+ and MAC/65.

CAUTION: this implies that all devices must then be specified with a following colon. (On the Atari, specifying "P" as a file is equivalent to specifying "P1:". On the Apple, "P" is the same as "Dn:P". On both systems, "P:" will be interpreted correctly.)

See also sections 5.6.1 and 5.6.2 for other details.

11.2.2 FILE SIZE INFORMATION:

Whenever a DIR command is issued at the CP level, or a file is opened for directory access (mode 6), the information returned contains information about the size of the file in sectors. However, on any disk used on an Apple Disk II unit, this information does not always reflect the length of the file, as it does on other disks. Rather, the file size shown reflects the actual number of physical sectors currently occupied by the file. Therefore, if a file contains "holes" (see section 11.1.1 on random access), the file size may not match the usual interpretation of "length".

11.3 DIFFERENCES: ATARI DOS AND OS/A+

There are very few points of difference between Atari DOS and version 2 of OS/A+, other than the fact that Atari DOS uses DUP.SYS while OS/A+ uses its Command Processor. And, actually, there are few differences between version 2 and version 4 of OS/A+ AS SEEN BY AN APPLICATIONS PROGRAM (including BASIC A+, etc.). Since the differences are generic, rather than specific to a particular version, they will be discussed by category.

11.3.1 MEMORY USAGE

The only real problem that can exist here is in the location of LOMEM, the beginning of user application memory. If a program written for Atari DOS (or OS/A+ version 2, for that matter) has assumed a particular LOMEM, it may not run under a version with a higher LOMEM. To illustrate, the following table lists the LOMEM value that will result in each of several cases if we assume a system configuration of 2 disk drives allowing 3 disk files open at the same time.

version	LOMEM contents
Atari DOS 2.0s, single density disks	$1C00
Atari DOS 2.0s, double density disks	$1E80
OS/A+ version 2, single density disks	$1F00
OS/A+ version 2, double density disks	$2180
OS/A+ version 4	$2C00

Of course, by changing the contents of SASA and SABYTE (see chapters 12 and 13), the user may change the location and number of buffers, so a certain measure of LOMEM independence may be obtained by, for example, placing the buffers somewhere within an application program's memory space. However, even this is not foolproof: examine the memory map of chapter 13 for more details.

11.3.2 END OF FILE

Atari DOS and OS/A+ version 2 both are capable of knowing exactly where a file ends, since each sector "knows" (via its 3 byte link information) how many bytes it contains. OS/A+ version 4, however, DOES NOT KNOW exactly where the END OF FILE is. In fact, version 4 will not report end of file until it reaches the end of the last BLOCK in the file (see chapter 10 for a discussion of file blocks versus sectors). Normally, this has little if any effect on a user program.

In particular, if reading text lines, the system will read the last line the same on either version. Then, when an attempt is made to read the first line past the end of file, version 2 reports an immediate error. Version 4, however, will begin passing the trailing zero bytes to CIO. However, CIO will not end the transfer until it receives an ATASCII RETURN code ($9B) (it expects to return an error 137, truncated record, if the user buffer is too short for all those null bytes). But the file manager finally returns the end of file signal, and CIO ignores any previous errors to return the EOF code.

With binary files, the problem is more subtle, since a binary record of all nuls is perfectly legal. However, most user programs on the Atari will have been built in such a way as to be aware of how many records are in a given file. And, if they have not been so built, there is usually at least one field in the record which cannot contain a null result. That field may be checked for nuls as an end of file test. N.B.: virtually any Atari DOS program which used indexed files (via NOTE and POINT) will function properly under version 4. Of course, programs which "take over" the entire disk may fail, but that is because there are 256 bytes per sector and their direct SIO calls are now improper, which has nothing to do with DOS.

11.3.3 RANDOM ACCESS

This subject has been treated more thoroughly in preceding sections and pages, but let us at least mention here that programs using NOTE and POINT properly under DOS 2.0 or OS/A+ version 2 will need no changes to move to version 4.

Of course, programs using the direct random access capabilities of version 4 (i.e., the ability to POINT relative to the beginning of a file, without the need to have previously NOTEd) will not transfer back to version 2. Sorry, but that's the price one must pay for using advanced features.

11.3.4 BUFFER ALLOCATION

All three Atari systems use the concept of a "Begin Buffer Allocation Here" address and a "Allocate This Many Buffers". The label SASA in the memory map (or, better, SYSEQU.ASM) defines a word containing the starting address of the buffers. The label SABYTE defines the number of buffers to be allocated. All very similar.

CAUTION: under Atari DOS and version 2 of OS/A+, SABYTE refers to the number of 128 byte buffers to allocate. Under version 4 of OS/A+, SABYTE contains the number of 256 byte PAGE buffers to allocate.

SECONDARY CAUTION: Remember the allocation requirements differ between single and double density disks under version 2 and Atari DOS.

11.3.5 SECTORS 1, 2, and 3

To insure compatibility with the Atari Computer boot process, OS/A+ (both versions) always reads and writes Sectors 1, 2, and 3 in single density (128 byte) mode. This single density "force" occurs at the BSIO level, so programs using BSIO may do so in a compatible fashion.

CHAPTER 12: MODIFYING OS/A+

In addition to discussing various system parameters which may be changed (in order to "customize" OS/A+ for a particular usage), this section will prove useful should OSS, Inc., send out corrections ("patches") to OS/A+. The user would make the patches in accordance with the directions provided and then save the corrected version following the directions of this section.

12.1 BUFFER ALLOCATION

Both versions of OS/A+ allow the user to specify the starting address of the system file buffers and the number of buffers to be used. The location of the words which specify these parameters varies between version 2 and version 4 and, in any case, is not guaranteed to remain fixed in future releases. Therefore, it is strongly suggested that the user desiring to change one or both of these values check the file "SYSEQU.ASM", supplied on the OS/A+ disk, to be sure of the latest system value. As of the printing of this manual, the following locations were in use:

version	label	location	use
2	SASA	$070C	start of buffers
4	SASA	$1613	start of buffers
2	SABYTE	$0709	# of buffers
4	SABYTE	$1616	# of buffers

Presuming the user wishes to change SABYTE, the first question that needs answered is "How many buffers do I need?" The rules follow:

For OS/A+ version 2: For single density diskettes, use 1 buffer per active drive AND 1 buffer per simultaneously open file. For double density diskettes, use 2 buffers per active drive and 2 buffers per simultaneously open file. EACH BUFFER IS 128 BYTES LONG.

For OS/A+ version 4: For disks with 256 byte sectors (e.g., floppy disks under double density), use 1 buffer per active drive AND 2 buffers per simultaneously open file. For disks with 512 byte sectors, double both figures. EACH BUFFER IS 256 BYTES LONG.

CAUTION: Note the difference in the size of the buffers specified by the SABYTE contents.

12.2 SPECIFYING ACTIVE DRIVES

Under version 4, the only way to specify an active drive is to add its parameters to the drive table (see Chapter 10). Also, the CONFIGure extrinsic command will configure this drive table for you.

Under version 2, the byte location DRVBYT (at $70A, but consult SYSEQU.ASM to confirm current location) controls which drives are active. Each bit of DRVBYT represents a given drive. The least significant bit of DRVBYT represents drive 1, the next bit represents drive 2, etc., up to the most significant bit which represents drive 8.

If a bit is DRVBYT is on (set to one), the drive is active. If a bit is off, the drive is inactive. Thus a value of $05 would imply that "D1:" and "D3:" are active.

CAUTION: simply changing the bits in DRVBYT is NOT sufficient to change the system configuration. After changing the bits, you must call the DOS initialization routine, via DOSINI.

12.3 SAVING YOUR MODIFIED VERSION

Saving a modified version of OS/A+ is extremely simple. With version 2, simply use the INIT command and, when the menu appears, specify "Write DOS.SYS file only" (or go ahead and initialize the disk if it is a new disk...just be careful not to reinitialize a disk with valuable goodies on it).

With version 4, the process is both more complicated and simpler. Since the version 4 boot process simply searches for the filename "DOS.SYS", any file may be renamed DOS.SYS and the system will attempt to boot it. Saving a modified OS/A+, then, is as simple as SAVing the proper segment of memory. The EXeCute file "WRITESYS.EXC" performs this function for you if you simply type "@WRITESYS" from the CP prompt level.

12.4 MOVING VERSION 2 TO DOUBLE DENSITY

Version 2 of OS/A+, as delivered, is ready to use on double density disks (with 720 sectors of 256 bytes each per diskette). The only requirement is that the system be "moved" to a double density disk. The steps to do so are very simple, presuming you have two disk drives:

A) Configure drive 1 (D1:) to be single density.
B) Configure drive 2 (D2:) to be double density.
C) Boot up OS/A+ version 2 on drive 1 (single density).
D) Run the INIT utility to initialize drive 2 and write DOS.SYS to drive 2.
E) COPY the desired files from drive 1 to drive 2.
F) You may now reconfigure D1: to be double density and boot from your new double density OS/A+.

If you only have a single drive, the problem is only slightly more difficult, depending upon the type of disk drive you have. Some manufacturers (e.g., PERCOM) supply a "SDCOPY" (Single to Double COPY) program. Simply copy DOS.SYS (and other files) from the single density disk to the double density. If your drive has a "hard switch" for single versus double, follow the manufacturer's directions (or borrow a friends drive, if possible).

CHAPTER 13: SYSTEM MEMORY MAPS

13.1 APPLE ZERO PAGE MAP:

location		usage
0-1F		user zero page
$20-$50		Apple monitor zero page
51-67		user zero page
68-6B		OSS utilities zero page
6C		warmstart flag
6D-6E	CPALOC	pointer to CP warmstart
6F-72		read/write sector zero page
73-7D		CIO zero page
79-7F		device handler zero page
80-FF		language or user zero page (OSS languages such as BASIC, MAC/65, and C/65 use some or most of this area; consult respective manuals for detailed information)

13.2 APPLE SYSTEM MEMORY MAP - 64K version:

location	usage
100-1FF	6502 stack area
200-2FF	Apple monitor GETLN input buffer (not all of this page is used; consult Apple documentation)
300-3FF	Apple monitor vector locations
400-7FF	Primary text and lores graphics page
800-BFF	Secondary text and lores graphics page
C00-1FFF	user and language ram
2000-3FFF	primary hires graphics page
4000-5FFF	secondary hires page
6000-AFFF	user and language ram (with default buffers)
B000-B7FF	disk buffers (default)
B800-BCFF	OSS CIO
BD00-BD7F	system IOCB's - 7 at 16 bytes each
BD80-BD8B	system DCB - disk i/o control block
BD8C-BDA4	device handler table
BE00-BFF5	file manager ram storage
BFF6-BFF7	low memory pointer (default is 800)
BFF8-BFF9	high memory pointer (default is AFFF)
BFFA-BFFC	system reset vector
BFFD-BFFF	OS/A+ return vector
C000-CFFF	Apple II peripheral and i/o locations
D000-D6FF	OSS CP
D700-D7FF	E: and Pn: device handlers
D800-DDFF	user memory {!!! graphics somewhere !!!}
DE00-F1FF	fms - D: device handler
F200-F7FF	read/write sector routine for Apple II disks
F800-FFFF	autostart rom

13.3 APPLE SYSTEM MEMORY MAP - 48K version:

---- to be defined ----

F800-FFFF monitor or autostart rom

13.4 ATARI ZERO PAGE MAP:

location		usage
0-9		system zero page
A-B	CPALOC	known to Atari DOS as DOSVEC
C-D	DOSINI	vector to FMS initialization
E-42		system zero page
43-49		fms zero page
4A-7F		system zero page
80-FF		user and language zero page
80-BF		BASIC A+ zero page
D2-FF		floating point zero page

13.5 ATARI SYSTEM MEMORY MAP-- version 2:

location		usage
100-1FF		6502 stack area
200-319		system ram
31A-33F		device handler table
340-3BF		IOCB's - 8 at 16 bytes each
3C0-57F		system ram
580-5FF		E: text buffer
600-6FF		user ram
700-1C7F		OS/A+ -- file manager and CP
709	SABYTE	number of 128 byte file buffers
70A	DRVBYT	bit map: accessible drives
70C	SASA	address of start of buffers
1C80-1EFF		file manager buffers-- default size
1F00-BFFF		user, language, and graphics memory
C000-FFFF		I/O locations and system rom

13.6 ATARI SYSTEM MEMORY MAP-- version 4:

location		usage
100-1FF		6502 stack area
200-319		system ram
31A-33F		device handler table
340-3BF		IOCB's - 8 at 16 bytes each
3C0-57F		system ram
580-5FF		E: text buffer
600-6FF		user ram
700-1C7F		OS/A+ file manager
1613	SASA	address of start of buffers
1616	SABYTE	number of 256 byte buffers
1C80-1CFF		read/write sector routine
1D00-22FF		OS/A+ CP -- console processor
2300-2AFF		file manager buffers-- default size
2B00-BFFF		user, language, and graphics memory
C000-FFFF		I/O locations and system rom

CHAPTER 13: Errors

All OS/A+ operations return a status value in the IOSTAT field. OS/A+ convention is that status values of $80 or greater indicate some sort of error.

ERROR CODE HEX	DECIMAL	MEANING
$01	1	No error or warning.
$02	2	Truncated ASCII line. The OS did not find a CR within BUFLEN for ASCII line I/O.
$03	3	End of file look ahead. The last byte transfered from the device driver was its end-of-file byte. The device driver must set this status, so it is best to verify that the device being used is capable of returning this status before depending on it.
$80	128	Operation aborted. Set by Device Handler. (Also BREAK abort on Atari.)
$81	129	File already open. Program is trying to open a channel (IOCB) that has already been OPENed.
$82	130	Device does not exist. The device was not found in the OS device table. Often caused by forgetting the disk drive name when using a disk file (Atari only). On the Apple II, since a default disk drive is assumed, this error is rarer.
$83	131	File is write only. Program tried to read from a file which can only be used for writing (i.e., file was OPENed with AUX1 set to 8 or 9).

$84	132	Invalid Command. CIO has rejected your requested command. (Example: program tried to do XIO to a device which has no extended operations defined.)
$85	133	Device/File not open. The IOCB has not been OPENed for the operation. Most I/O requests require that the channel be OPENed before a request can be made.
$86	134	The IOCB specified is invalid. (In both Atari and Apple II versions, only IOCB numbers $00,$10,$20,$30,$40,$50,$60, and $70 are valid. From some languages, these will be seen as channels 0 to 7.
$87	135	File is read only. Program tried to write to a file which can only be used for reading (i.e., file was OPENed with AUX1 specified as 4 or 6 [or 5 on Apple]).
$88	136	End of file. No more data in file.
$89	137	Truncated record error. Usually occurs when the line you are reading is longer then the maximum record size specified in the Call to CIO (line oriented I/O). Can't occur with binary I/O on version 2 OS/A+.
$8A	138	Device timeout error. Atari: usually set by the serial bus I/O handler ("SIO") because a device did not respond within the alloted time as set by the OS. Apple II: set by a device handler.
$8B	139	Device NAK error. Atari: serial I/O error. Apple II: set by a device driver (rare).
$8C	140	Serial framing error. Atari: serial I/O error. Apple II: undefined.
$8D	141	Cursor out of range for specific graphics mode you are in. (Could be used for similar meaning by a non-graphics device.)

$8E	142	Serial bus overflow. Atari: computer could not respond fast enough to serial bus input (SIO error).
$8F	143	Checksum error. Communications over the serial bus are garbled (Atari SIO error).
$90	144	1) Device done error. A valid command on Atari: disk rotational speed needs adjustment. 2) Write protect error. The diskette has a write protect tab in place.
$91	145	Illegal screen mode error. Bad graphics mode number. Other devices: AUX1 and/or AUX2 bytes in IOCB are illegal.
$92	146	This error means the function you tried to do has not been implemented in the device handler. (Example: attempt to POINT the graphics device.)
$93	147	Not enough RAM for the graphics mode you requested. (Could be used by custom drivers for a similar message.)

NOTE: Errors $A0 through $AF are file manager errors.

$A0	160	Either a drive # NOT between 1-8 or drive was not powered on.
$A1	161	Too many OPEN files. No free sector buffers to use for another file.
$A2	162	Disk FULL. No free space left on disk.
$A3	163	Fatal system error. Either DOS has bug or bad diskette.
$A4	164	File mismatch. Bad file structure or POINT values wrong.
$A5	165	Bad file name. Check for illegal characters in file name. Version 4 is more liberal in this regard than version 2.

$A6	166	The byte count in your POINT Call was greater then 125 (for single density version 2) or 253 (for double density version 2).
$A7	167	The file specified is locked (PROtected). Protected files cannot be erased or written to.
$A8	168	The software interface for the specific device recieved an invalid command (example: tried to access a non-existent track or sector).
$A9	169	All space allocated for the directory has been used up (too many filenames in use).
$AA	170	The file you requested does not appear on this diskette.
$AB	171	You have tried to POINT to a byte in a file that is not OPENed for update (version 2 only).
$AC	172	Tried to OPEN a DOS 1 file with DOS II (version 2 only).
$AD	173	The disk drive has found bad sectors while trying to format the disk.

OOPS

No software product is bug free; and, unfortunately, OS/A+ is no exception. This page reports the known bugs, omissions, etc. in the current releases of OS/A+. PLEASE read this and make notes in your manual accordingly.

CAR [section 3.1, page 14]

The intrinsic command CARtridge currently makes no check to see if a cartridge actually is present in the system. Using the command when no cartridge is present will cause the system to "hang".

MOVING VERSION 2 TO DOUBLE DENSITY [section 12.4, page 103]

VERSION 2 ONLY: You can NOT use COPY (or any variant of COPY) to transfer DOS.SYS to a newly created double density disk unless you COPY it under a different name.

EXAMPLES THAT WORK:

1) COPY D1:DOS.SYS D2:GORP (This example assumes that drive 1 is configured to single density and drive two is configured double density.)
   REN D*:GORP D2:DOS.SYS

2) REN D1:DOS.SYS D1:JUNK
   SDCOPY D1:JUNK
   REN D1:JUNK D1:DOS.SYS (ON SINGLE DENSITY DISK)
   CONFIG 1D
   REN D1:JUNK D1:DOS.SYS (ON DOUBLE DENSITY DISK)

Of course you can create DOS.SYS on your double density disk using INIT or INITDBL, see addendum sheet for details.

HERE ARE STEP BY STEP INSTRUCTIONS FOR COPY24 USAGE:

SINGLE DISK DRIVE SYSTEM

1)Boot master Ver 4 disk
2)Type in ADOS command
3)Type COPY24 A1:FN D1:FN
4)Read prompts on screen to finish the copy.

TWO DISK DRIVE SYSTEM

1)Boot master Ver 4 disk
2)Type in CONFIG 2S (configure drive 2 to single density)
3)Type in ADOS command
4)Type in COPY24 A2:FN D1:FN

NOTE: "FN" means the filename you are going to copy

NEW UTILITIES, VERSION 2 ONLY

CLRDSK - This utility is used to initialize disks just like the Atari 810 disk drive does. Hopefully any program that does not work with Percom drives because of the way they format disks will work with a disk that has been formatted using CLRDSK. (NOTE: CLRDSK formats the disk first, then writes zeroes to all sectors except the DIRECTORY, BOOT and VTOC sectors.)

INITDBL - This utility is used ON A ONE DRIVE SYSTEM to initialize a double density diskette and write DOS.SYS to it. The procedure that INITDBL uses to initialize a double density disk is as follows: (1) it configures the drive to double density; (2) it initializes the disk; (3) it reconfigures the drive back to single density so that you can still read your single density master.

To get a directory of your new double density disk you must first configure the drive to double density (see section 4.4 in the OS/A+ manual) or boot the new double density master. Or use SDCOPY (below) to copy other files, etc., to the double density disk.

If using a two drive system refer to section 12.4 for specific details on initializing a double density diskette.

SDCOPY FN1 [FN2] [-FQWVR]

This utility is used to COPY a Single density file to a Double density file (hence SDCOPY). The command works the same way as the COPY command except:

-R Reverse orientation of copy: COPY instead from double density to single density.

NOTE: "from" and "to" drives MUST be the same when using SDCOPY. If you want to copy from single density to double density between two different drives, just use the CONFIG command to set the drives up properly and use the normal COPY command.

VERSION 2 and VERSION 4, CAUTION !!!

YOU SHOULD NOT USE A WILD CARD SPECIFICATION IN THE DESTINATION FILE NAME OF ANY VARIETY OF THE COPY COMMAND (COPY, SDCOPY, COPY24, etc.).

ANNOUNCEMENT of a new BEGINNER'S GUIDE to OS/A+

By the time you read this, we will have completed a brand new Beginner's Guide which is particularly oriented toward the Atari BASIC user. All of BASIC's disk specific capabilities are explained, and step by step explanations of most used functions are given (including making double density masters, double sided masters, etc.). This guide will become part of a future OS/A+ manual. For now, it is available to registered owners for $3.00, check/cash only, including postage, etc.

ADDENDUM

This sheet is meant to be used as an addendum to the OS/A+ manual supplied with your disk drive. Since the manual was printed, OS/A+ (Ver 2 and Ver 4) have been updated (to Ver 2.1 and Ver 4.1). This sheet reflects the changes and additions that have been made.

NOTE: Anything between the two braces {} are meant to be prompts by the computer. Characters following the braces are to be typed by the user.

1) DIR [FN1 [FN2]] If FN2 is specified the directory will go to that file or device.

{D1:} DIR D2: P: This example will put the directory on the disk in drive 2 to the printer.

2) TYPE FN1 [FN2] This command TYPEs ATASCII text files from FN1 to E: (the screen). Or, if FN2 is specified, then FN1 gets copied to FN2.

{D1:} TYPE D1:SYSEQU.ASM	Copies the file SYSEQU.ASM to screen.
{D1:} TYPE D1:SYSEQU.ASM P:	Copies the file SYSEQU.ASM to the printer
{D1:} TYPE D1:MYFILE D2:YOURFILE	Copies the file MYFILE on drive 1 to the file YOURFILE on drive 2.

NOTE: The TYPE command may not be used to copy binary files!

REVISED VERSION 4 UTILITY ONLY

3) COPY24 [-FQDVN] FN1 FN2
The flags used are the same as the COPY command except

-V Verbose: Filenames are echoed after they are copied. This flag is also available to the COPY command.

-D The version 2 disk involved is double density.

Filenames may be specified with either "D" or "A" as the device name where "D" refers to the Version 4 disk and "A" refers to the Version 2 disk. The -W flag (wait) is assumed when using only one disk drive.

{D1:} COPY24 A2:MENU.LIS D1:MENU.V4
This example will copy the Version 2 file MENU.LIS on drive 2 to the version 4 file MENU.V4 on drive 1.

{D1:} COPY24 D2:MENU.V4 A1:MENU.LIS
This example copies the version 4 file MENU.V4 on drive 2 to the version 2 file MENU.LIS on drive 1.

{D1:} COPY24 A2:*.* D1: -Q
This example will copy ALL version 2 files on drive 2 to version 4 files on drive 1. NOTE: since the -Q flag was used, you will be prompted before each file is copied.

{D1:} COPY24 A1:MYFILE YOURFILE
This example is assumed to be a single drive copy. The version 2 file MYFILE will be copied to the version 4 file YOURFILE.