INCLUDES THE ULTIMATE

DUPLICATOR

DISK DOCTOR will let you read, modify, or copy any sector of a disk, check and adjust RPM, copy all or part of any standard ATARI® formatted disk at machine language speed — including those with bad sectors — and let you create bad sectors on your own disks; fix damaged files, modify directories, recover "lost" data or deleted files; repair damaged VTOC's and sector counts automatically, and format damaged disks.

BUT ONLY DISK DOCTOR will let you:

READ "UNLISTABLE" BASIC PROGRAMS right off of the DISK — even make working, modifiable copies of them automatically!

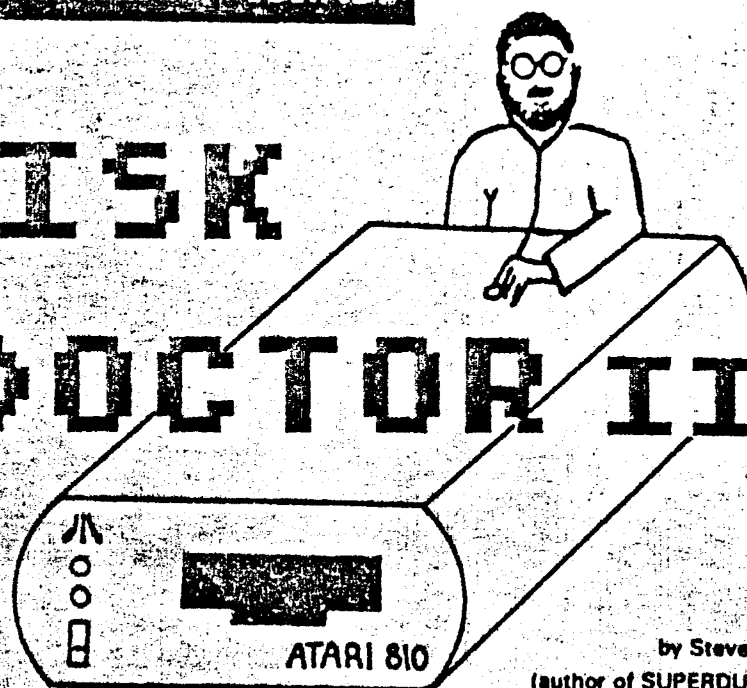
DISASSEMBLE sequential or DOS files WITH ADDRESSES AND SYSTEM LABELS or labels you create! You have to see it to believe it!

SEARCH the disk for any sequence of up to SIX BYTES WITH WILDCARDS!

RECOVER automatically OSA+ version 4 FILES

BJ SmartWare presents:

DISK DOCTOR II



by Steve Kaufm
(author of SUPERDUPER DU

THE ONLY DISK UTILITY YOU'LL EVER NEED

NOW FOR ALL DISKS: SINGLE OR DOUBLE DENSITY, and DOUBLE SIDED

Requires 32K; 810 or equivalent drive.
Printer Optional

COMPLETE DISK TUTORIAL INCLUDED
plus an intro to 6502 Assembly language.

DISK DOCTOR II

by Steve Kaufman

1 INTRODUCTION

DISK DOCTOR II is a collection of utility programs designed to let you directly examine and, if desired, modify data stored on a diskette. You can edit individual sectors, fix damaged or mistakenly deleted files, read "locked" BASIC programs, disassemble machine language programs with labels, search for a series of bytes (up to six, with wildcards) on the disk, duplicate any standard format diskette at machine language speed, including those with bad sectors, modify the disk directory and correct its table of contents, check and adjust the speed of your 810 disk drive, and even format damaged disks.

DISK DOCTOR II has a combination of features and conveniences unavailable on any other disk utility and should enable any user to master the mysteries of DOS (Disk Operating System) and the Atari disk system. An appendix to this manual offers a tutorial on the format of Atari disks, DOS and the disk boot process, as well as a brief introduction to assembly language for the beginning programmer. Note: an understanding of assembly language is not required in order to use DISK DOCTOR effectively.

Printer options are supported for all relevant functions.

The disk includes a file of system labels to be used with the disassembler as well as a program that allows the user to create and edit his/her own label files.

DISK DOCTOR II is totally compatible with all current single density, double density, and double sided drives, and includes a program for recovering OSA+ version 4 double density files.

Minimum RAM and accessories

REQUIRED:
- 32K RAM
- Atari BASIC Cartridge
- 810 disk drive or equivalent

OPTIONAL:
- Atari printer or equivalent
- 850 interface or equivalent

Warning

Attempting to modify data on a disk can be hazardous to the health of your disk if you don't know what you are doing. Be sure you have mastered the material in the appendix before

attempting to use DISK DOCTOR, and, whenever possible, always "operate" on copies, not originals.

At the end of this manual is a form to be returned to BJ Smartware. Please register your purchase of this software so that we can keep you informed of revisions and new products that may be of interest to you. Please note: Approximately 30% of all software brought to market after Oct., 1982 (and as much as 50% of that issued after Feb. 83) is protected by advanced formatting schemes that no software product alone can break. Fortunately and wisely, the vendors of such software generally offer reasonably priced backup disks. By all means avail yourself of those offers. DISK DOCTOR II, of course, will copy itself. Registered purchasers are authorized to use DISK DOCTOR II to make one backup copy of the original disk. Do not try to copy the individual files of DISK DOCTOR II onto another disk. They will behave very strangely!

STARTING UP

To use DISK DOCTOR, insert the BASIC cartridge into your computer, turn on your disk drive and wait for the motor to stop; then, as usual, insert the disk into the drive before turning on the computer. If you are planning to use a printer to print some of the data generated by DISK DOCTOR, turn it and your interface on now to avoid noisy error messages later.

Turn on your computer. The drive will whir for a few moments, the screen will flash, and the DISK DOCTOR logo will emerge

The following "menu" will then appear:

1 Disk Doctor, main module
2 THE ULTIMATE DUPLICATOR
3 DOS file recovery/directory-edit
   and format bad disks
4 OSA+ 4 recovery/directory-edit
5 RPM check/adjust -- make bad
   sectors
6 Read a BASIC program from disk
7 List Object file blocks/locations
8 Create/edit a Label file

WHICH?

That's it, type the number of the module of your choice, hit the RETURN key, and off we go.

1 DISK DOCTOR, MAIN MODULE

This is the program that you will be using most. It is the longest of the programs on the disk, so it takes a bit longer to load than the others. (Be patient, really fast drives for the Atari are right around the corner!) The screen will darken to a comfortable hue (a good compromise, I think, for both TV and monitor users), and after the drive stops, you will see this:

DISK DOCTOR II
by Steve Kaufman

A SECTOR EDITOR
B DISASSEMBLER
C SEARCH FOR A CHAIN OF BYTES
D DISK DIRECTORY
E UTILITIES
F FIND AN ADDRESS ON DISK
M SYSTEM MENU

All it takes to activate any of the above commands is a keystroke.

At this point, the disk that you wish to examine or modify must be inserted in the default drive. If you (like most of us) have only one drive, remove the system disk and insert the target disk. If you have a second drive (set up as drive #2) and it was on line when the program started running it will be the default drive for all disk analysis and modification functions. You may leave the system disk in drive #1 (but please don't take the write-protect tab off, just in case!)

In order to change the default drive, or to configure your drive for double density operation, use the "Drive Configuration option ("B") from the "Utilities" submenu. See below. All of the options of this program will run in single density, double density or double density/double sided drive configurations.

A SECTOR EDITOR : This module is the workhorse of the program, performing the sector by sector analysis and modification of the disk. Let's try it out:

Hit the "A" key and the following mini-menu will appear at the bottom of your screen:

R,E,P,W, or M?

These letters stand for Read (a sector of the disk), Edit a sector, Print whatever you seen on the screen now to the printer, Write the sector on the screen out to the disk, and go back to the main Menu of this module.

Hit "R" now and you will be asked which sector of the disk you wish to read. Let us read sector number 1 off the disk. Type the number 1 followed by a RETURN and the following will appear on your screen if you have a DOS formatted disk in your default drive:

```
SECTOR 1 ($ØØ1) FP. 515  FØ Ø

ØØ  ØØ Ø3 ØØ Ø7 4Ø 15 4C 14   ....@.L.
Ø8  Ø7 Ø3 Ø3 ØØ 8Ø 1C Ø1 Ø4   ........
1Ø  ØØ 7D CB Ø7 AC ØE Ø7 FØ   ..K.,...
18  36 AD 12 Ø7 85 43 8D Ø4   6-...C..
2Ø  Ø3 AD 13 Ø7 85 44 8D Ø5   .-...D..
28  Ø3 AD 1Ø Ø7 AC ØF Ø7 18   .-..,...
3Ø  AE ØE Ø7 2Ø 6C Ø7 3Ø 17   ... l.Ø.
38  AC 11 Ø7 B1 43 29 Ø3 48   ,..1C).H
4Ø  C8 11 43 FØ ØE 81 43 A8   H.C.1C(
48  2Ø 57 Ø7 68 4C 2F Ø7 A9    W.hL/.)
5Ø  CØ DØ Ø1 68 ØA A8 6Ø 18   @P.h.('.
58  A5 43 6D 11 Ø7 8D Ø4 Ø3   %Cm.....
6Ø  85 43 A5 44 69 ØØ 8D Ø5   .C%Di...
68  Ø3 85 44 6Ø 8D ØB Ø3 8C   ..D'....
7Ø  ØA Ø3 A9 52 AØ 4Ø 9Ø Ø4   ..)R @..
78  A9 5Ø AØ 8Ø 8D Ø2 Ø3 8C   )P .....
```

This is a "picture" of the first of the 72Ø sectors (144Ø if double sided) on your disk, in this case, from the boot code (see appendix I). The information on the top row tells us that this is sector 1 ($ØØ1 in hexadecimal) and that the last three bytes of the sector (were they to be interpreted as DOS file information) show that this is (or was) a part of File number Ø (FØ) and that the next sector of this file is to be found at sector 515 (FP=file pointer; both numbers are given in decimal). The left hand column of numbers gives the relative hexadecimal address within the sector of the starting byte of each line, while the subsequent columns show the bytes themselves in both hexadecimal notation and (at the far right) in character representation. Note that if a byte is not equivalent to a normal alphanumeric character a period is printed in its place (this is to prevent your printer from having heart failure when encountering certain control characters, as well as making the alphanumeric characters stand out).

The cursor is now back at the end of the mini-menu asking you for another command. To print what you now see on the screen to your printer simply hit P. (The screen print routine is in machine language, so if you have a printer buffer the program will be ready for another command almost instantly.) To write the sector that you now see back to the same or another disk, hit W and you will be asked "Write to which sector?". If you respond to this query with a carriage return the program assumes you want to write to the same sector you took it from (see EDIT below). If you respond with a Ø (zero) it assumes you have changed your mind and takes you back to the mini-menu for another try. Otherwise give it a sector number from 1 to 72Ø (or $Ø1 to $2DØ in hex) and the program will ask you to verify that you really want to write to such and such a sector. Respond to that prompt with Y and the sector will be written and the word DONE appear on the screen. Any other response returns you to the mini-menu. You may want to try this now by copying sector 1 to sector 72Ø, since sector 72Ø is not used by DOS and should have no data on it except zeros. OK, now try to read sector 72Ø. If you have done everything correctly, it should now be an exact duplicate of sector 1.

Any time a sector is showing on the screen, you may read the subsequent sector merely be hitting the > key. Similarly, the previous sector may be viewed by hitting <. If you are viewing a double density sector, you can only see half of it at one time (128 bytes). The left and right cursor control arrows will scroll you through the sector, 8 bytes at a time (no need to hold down CONTROL), or you can flip from the low 128 bytes to the high 128 bytes instantly by hitting the SELECT button.

By the way, throughout DISK DOCTOR, unless indicated otherwise, you may answer any request for sector number or RAM address information with either a decimal number (the default mode) or with the hexadecimal equivalent (always preceded by a dollar-sign, e.g. $A3BØ). DISK DOCTOR itself adheres to standard conventions and prefers to give sector numbers in decimal, but addresses(both in RAM and within a sector) in hex.

OK. Now you have copied a sector from one place on the disk to another, and perhaps printed out a copy. Now let's modify the sector before we write it back to disk. To do that simply hit E in response to the mini-menu. If you want to edit the same sector that you already see (or just saw) on the screen, simply hit RETURN in response to the prompt:"edit which sector?". Otherwise give a valid sector number (or Ø to return to the mini-menu).

Now you see a display very similar to the former one, except that now there is a cursor at the upper left hand corner of the sector image and the message "HIT ESCAPE TO EXIT". Simply move the cursor with the usual cursor arrow keys to the byte or bytes you want to modify, type in the proper HEX value, and when you are finished with your modifications hit the ESC key. DISK DOCTOR will not let you type in the wrong places, nor enter illegal values while editing. When you exit the edit mode you are automatically put into the Write mode so that you can write out your edited sector to the disk. Follow the directions given above. After you have written the modified sector to somewhere on the disk, you may find it reassuring (I do!) to Read that same new sector to make sure that your corrections have "taken". By the way, when modifying valuable disks, do yourself a favor and always keep a careful record of what the modified bytes looked like before you modified them! That way if you manage to make things even worse than they were before you can always go back to where you started.

If you are editing a double density sector, the SELECT button will flip the screen as usual, but to use it you must first position the cursor in one of the blank columns. Once used, SELECT can not be used again (nor can you see the cursor!) until the cursor is moved with one of the arrow keys.

When you are done with all of your sector reading, editing and copying, hit M in response to the mini-menu prompt in order to return to the main menu.

B DISASSEMBLER:

If you hit B in response to the menu, you will enter the disassembly mode. A disassembler turns machine language code back into easier to remember and interpret mnemonic "opcodes" and "operands" (see Appendix),i.e assembly language. Even if you cannot yet program in assembler, you should be able to use these disassemblies to some advantage if you learn the material in the appendix on 65Ø2 assembly language.

The first question this module asks is if you want your printout to go to the Screen or Printer. Answer appropriately. If the output goes to the printer it will not appear on the screen. Next you must determine if you want to disassemble Sequential files or DOS files. "Sequential" is just another way of saying sector after sector, all 128 bytes of information. If you are examining a "boot" disk this will probably be the option that you will choose. If you choose the DOS option, only 125 bytes (253 for double density sectors) will be displayed for each sector, and disassembly will proceed through the file using DOS file pointers. For now let us choose Sequential.

If everything has gone smoothly up to this point, you will now be asked what sector to start with. Any time this prompt appears on the screen, answering it with 0 will return you to the menu. For now let us type 2 (RETURN) to disassemble starting with sector 2 (almost always a part of the boot code of the disk). Then you must specify what byte of that sector you wish to start with. Let us enter 0 for now. Finally, you must indicate what starting memory address should be assigned to that byte. Since most disks have the DOS boot code on them, let us respond to this one with $780, the proper address for the code we are going to look at. If you have answered all the prompts correctly and are examining either the DISK DOCTOR disk or another disk with DOS on it, you should see the following to start with.

```
SECTOR 2    ADDRESS=$0780
00  0780    03              ???
01  0781    03              ???
02  0782    A9 31           LDA #$31
04  0784    A0 0F           LDY #$0F
06  0786    8D 00 03        STA $0300
09  0789    8C 06 03        STY $0306
0C  078C    A9 03           LDA #$03
0E  078E    8D FF 12        STA $12FF
11  0791    A9 00           LDA #$00
13  0793    A0 80           LDY #$80
15  0795    CA              DEX
16  0796    F0 04           BEQ $079C
18  0798    A9 01           LDA #$01
1A  079A    A0 00           LDY #$00
1C  079C    8D 09 03        STA $0309
1F  079F    8C 08 03        STY $0308
22  07A2    20 59 E4        JSR $E459
25  07A5    10 1D           BPL $07C4
```

Of course as you now know, the disassembly process will go on forever if you let it. TO STOP THE DISASSEMBLY AT ANY TIME SIMPLY HIT ANY KEY. There. Let's try it again. Simply answer the "which sector?" prompt by 2, repeat the other same answers as above and this time hit a key when you see about 3/4 of a page of stuff. As with any ATARI program, to temporarily stop a scrolling listing on the screen, you may also hit CONTROL and "1" at the same time. Repeat it to start it up again. (Note that once you have started the disassembly module you cannot change the major parameters--sequential/DOS, printer/screen. To do so simply answer the "which sector?" prompt with 0 to return to the main menu and hit B to return to disassembly from the beginning.) Here's what all that you see means:

At the beginning of each sector to be disassembled the sector number is displayed along with the memory address in hex that you assigned to it (either directly or, by implication, when you specified a starting address several sectors earlier). The display itself is divided into four sections: The leftmost column displays the hexadecimal number of the first byte of the assembly language command relative to the start of its sector. The second area gives the equivalent memory address that you have assigned to that byte. The third area consists of from one to three columns, depending on the Assembly language instruction encountered, the first byte being the machine language instruction itself, the second and third, if present, the data on which that instruction operates. The rightmost column, of

course, is the disassembled version of the third column.

At this point you might well want to experiment to see that the bytes depicted by the disassembly process are indeed the same bytes discovered in that sector by the SECTOR EDITOR (option A from the menu). Go ahead.

Some notes: If the disassembler can't figure out what the machine language instruction is it will indicate that fact by returning "???" instead of an assembler opcode in the righthand column. This will usually happen in only one of two cases: You have asked it to start disassembly in the middle of an instruction, or you are trying to disassemble data instead of a machine language program. If the former is the case, simply back up a few bytes and try again. In the latter instance, read the data with the SECTOR EDITOR rather than with the disassembler. Most good programs have lots and lots of data tables of various kinds. It is possible, however, for data bytes to have values deciperable into opcodes by the disassembler. This situation is usually obvious, however, for it typically produces strings of inordinately rare opcodes interspersed with a multitude of question marks.

If the disassembler comes upon the end of a sector in the middle of instruction, the operand of that instruction will be indicated by question marks (e.g. JSR $????), and the actual bytes to be assigned to that instruction will be found at the beginning of the next sector marked with <<< (e.g. <<< $xxxx 00 80, all of which simply means JSR $8000). This should produce no problems in most cases. Just remember that the LSB comes first (see appendix!). In the rare instance of a branch instruction occurring at the very last byte of a sector, refer to the relative branch table included in the appendix to determine the branch location. (Remember, the relative offset is from the byte following the operand of the branch instruction.)

There will be many times, of course when you cannot determine in advance what address to assign to the starting byte of the disassembly. Just give it the starting address of 0. (It would be wise to note on your printout that the addresses of this disassembly are not correct in such cases.

NOTE: If you use a starting address of 0, you may encounter a negative branch instruction that points to a minus address. If so, the disassembler will display the illegal address as "****".

MORE ON ADDRESSES: If you see a disassembled program with a lot of good code with a few question marks, it probably indicates that you are trying to disassemble across a block boundary (see the appendix and the instructions to module 7 of DISK DOCTOR). The proper procedure here is to run "7 Object file blocks / locations" from the main DISK DOCTOR menu fist and disassemble the blocks of memory separately. Obviously, the indicated address of any byte that is across a block boundary vis-a-vis the beginning of the disassembly will be either 4 or six bytes too high per block boundary depending on the number of header bytes of each block. The same caution applies when using option F FIND AN ADDRESS ON DISK. Searching for an address across a block boundary will provide erroneous results.

EVEN MORE ON ADDRESSES: On rare occasions you may encounter a DOS file created by an append process and never recopied. In such cases, the last sector of the original (pre-append) file may have less than 125 bytes of data even though it is not the last sector of the new file. This can be fixed by recopying the entire file into a new one. If you try to disassemble or find an address in such a file, however, you will end up analyzing a lot of extraneous information since the algorithms used by DISK DOCTOR do not examine the last byte of each sector to see how many bytes are in use. (See Appendix I.) Again, the proper method in such cases is to use Option 7 from the main DISK DOCTOR menu first.

Now for the good stuff. I have designed the disassembler to allow you to disassemble with labels, either system labels (recognizable, by and large to experienced Atari programmers) or labels that you wish to assign. Any two-byte address (not zero-page) can be assigned a label, up to a total of 96 different labels. To create or edit a label file to be used by the disassembler you will have to use the special program that must be called from the main DISK DOCTOR menu, but for now let us just use the file of system labels that comes on the disk. To do so, answer the "which sector?" request with 0 to return to the menu; then choose option E, UTILITIES, and choose option D from the Utilities sub-menu.

TO LOAD THE SYSTEM LABEL FILE just hit the carriage return when you are asked for a file name. To load another label file just give the file name proper without extender (all label files are automatically assigned the extender .LDT). The file will take a few moments to load. (If the program can't find your label file on the disk it will provide a directory of all .LDT files for you.) Now you can go back and try disassembly just as before, only this time whenever the disassembler encounters an address for which a label has been assigned, it will substitute the label for the address: Here's an example of disassembly with labels (from the same sector we viewed before):

```
18  0798  A9 01     LDA #$01
1A  079A  A0 00     LDY #$00
1C  079C  8D 09 03  STA $0309
1F  079F  8C 08 03  STY $0308
22  07A2  20 59 E4  JSR $E459
25  07A5  10 1D     BPL $07C4
27  07A7  CE FF 12  DEC $12FF
2A  07AA  30 18     BMI $07C4
2C  07AC  A2 40     LDX #$40
2E  07AE  A9 52     LDA #$52
30  07B0  CD 02 03  CMP DCOMND
33  07B3  F0 09     BEQ $07BE
35  07B5  A9 21     LDA #$21
37  07B7  CD 02 03  CMP DCOMND
3A  07BA  F0 02     BEQ $07BE
3C  07BC  A2 80     LDX #$80
3E  07BE  8E 03 03  STX DSTATS
41  07C1  4C A2 07  JMP $07A2
44  07C4  AE 01 13  LDX $1301
47  07C7  AD 03 03  LDA DSTATS
4A  07CA  60        RTS
```

Now you don't have to constantly refer to your memory map literature to determine what on earth a program is doing. Just remember a few mnemonics, or assign your own meaningful labels.

NOTE: Once a label file has been loaded it remains in memory and will be used for all subsequent disassemblies until either another label file has been loaded in its place or the "MAIN MODULE" has been abandoned to return to the main DISK DOCTOR menu.

A listing of the system label file included on DISK DOCTOR can be found on page 12.

C SEARCH FOR A CHAIN OF BYTES

Sometimes you will want to find a specific sequence of bytes or several similar sequences of bytes that you know or suspect are on a disk, but you don't know precisely where. This section of the main module lets you search the disk, either sector by sector ("sequentially") or to search through a single DOS file, for any sequence of up to six bytes. If you want any given byte or bytes in the sequence to be ignored (a "wildcard"), just enter an asterisk ("*") in that position. Other bytes may be entered in either decimal or hexadecimal notation (you needn't even be consistent), although the program will transform all of them into hex when it displays the string of bytes that it is searching for. If you wish to specify less than six bytes, simply enter a carriage RETURN alone in response to the prompt. For example:

SEQUENTIAL SECTORS OR DOS FILES? Here let us enter S in order to search the entire disk.

```
ENTER SEARCH BYTE 1 ?$A9
ENTER SEARCH BYTE 2 ?0
ENTER SEARCH BYTE 3 ?$8D
ENTER SEARCH BYTE 4 ?*
ENTER SEARCH BYTE 5 ?3
ENTER SEARCH BYTE 6 ?<RETURN>
```

In this case We have told the program that we want to search for every occurrence of the following five bytes of machine language code: LDA #0; STA $3xx = load the accumulator with zero; store the accumulator anywhere on page three of memory. Of course you needn't search for machine code; it could be ATASCII characters, a display list (look for 112,112,112!), etc.

Next the program will ask where you want to start the search; then it will display the string you are looking for and begin to read the disk and display (sector/byte) locations wherein it finds your searched-for sequence. YOU CAN ABORT THE SEARCH AT ANY TIME BY HITTING ANY KEY. If you are searching sequential sectors, the program will only abort at the end of the disk. If reading a DOS file, it will stop its search at the end of the file or if it encounters a bad file pointer. When the search is over you will be in the mini-menu of option A so that you can print out the results of the search or read a relevant sector to the screen. If you want to search for more, simply hit M for the Menu and C for the search option once more.

D DISK DIRECTORY

This option allows you to examine the directory sectors of a DOS disk (sectors 361-368) in directory format rather than in sector format. This is primarily designed to allow you to locate the beginning of files quickly for tracing or search purposes. Of course you can modify directory entries by using the Sector Editor of this main module. Unless you are very sure of yourself, however, you'd be better off to use the separate Directory Fixer module called from the main DISK DOCTOR menu. (For the meaning of the heading line in the directory table, see the instructions to that module)

After each sector of the directory is displayed, you have the option of seeing the next sector (RETURN), returning to the menu (M), or tracing a file (T).

Tracing a file is particulary useful for quickly locating the place where a file link has gone bad, for example when you try to load a program from the disk and you get an ERROR 164. The file trace will display each sector of the file, starting with the sector you indicate, and will continue until it reaches the end of the file, finds a file number mismatch, or can't read a disk sector; then you will be in the mini-menu, where you can print the trace to your printer or read the offending sector to determine exactly what is wrong.

If you are examing a disk that runs in BASIC but seems to have no directory on it, don't worry. It is there. It is just been moved to a non-standard location. Examine the sectors from about 340 to 380 or so with the sector editor until you find it. Then recopy those sectors onto sectors 361-368 of that or another disk, and you will be able to read and edit the directory as usual.

E UTILITIES

The following sub-menu will appear when you select this option:

A. Numeric Conversion
B. Drive Configuration
C. Establish XOR mask
D. Load Label File
E. Erase Disk
F. Fill Sectors
G. Calculate DOS pointer
H. Hi to Lo
M. Main Menu

A. This option simply converts from decimal to hexadecimal numbers and vice versa. Simply enter the number (in the range 0-$FFFF) and the equivalent will appear (minus values are not supported).

B. Use this option either to change the configuraton of a drive or simply to change the default drive. If you have a single density drive, the program will be unable to read its configuration and will tell you so, but that drive will now be the default drive and single density operation will be assumed. (If you have two or more drives, and a drive other than number one is double density by default or has previously been configured as double density before running this program, you will still have to go through the configuration routine, just so the program will know what its density/sidedness is). Moreover, if you have a PERCOM, be careful. They will accept configuration commands for functions they are incapable of and the program will behave accordingly. A drive with only one head will accept a command to become double sided. But there is no way that you are going to be able to read sector 721 on that drive, even though the program will let you try.

C. This important option lets you read or write a sector using an "exclusive or" byte. (If you don't know what that means, you probably couldn't use it to advantage anyways). Both the A and B options of the main menu will use the XOR byte here established. If an XOR value is in effect, that fact will appear on the upper right of the sector display. Morevoer, when you write a sector, the program will first ask you if you want to recode using the XOR value. It is possible to read a sector, change the XOR value using this option, then return to the sector (using the EDIT option of the sector editor, followed by a carriage return). The sector will appear as before, only changing when you answer Y to the appropriate prompt after you leave the EDIT mode. To eliminate the current XOR mask value quickly, simply ask the main menu for the directory of the disk.

D. This has been covered above, under the Disassembler.

E. and F. These options allow you to fill a series of sectors with a chosen byte (F) or the entire disk with zeros (E) without reformatting the disk. Note: If you just want to wipe the old files off a disk but don't care what data remains in the sectors themselves, use the relevant option of the DOS file recovery program (main DDII menu option 3).

G. Enter the sector number to be pointed to, and the file number and this option will display the proper bytes to enter into the second and third last bytes of the preceding sector in the file chain. See Appendix I.

IMPORTANT: Here, and in all functions involving DOS file structure, the reference is to files created and linked using standard Atari DOS, either in single density or double density mode. (See Appendix I, and note that OSA+ versions 2 and 3 use the same file structure). Standard Atari DOS cannot handle disks with more than $3FF (1023) sectors however (Indeed, the rumor is that that's how many DOS 3.0 will use). Thus any call to the program for a DOS file structure function involving the second side of double sided disks (sectors 721-1440) asumes that the modified Atari DOS available from BJ Smartware has been (or will be) used to create those files. This is a fair assumption, given that BJ DOS is the only known Atari-DOS-compatible DOS available for full double sided operation. Any file that runs in normal DOS, will run in BJ DOS. Of course, BJ DOS may also be configured to run on single sided double density drives.

H. Choose this option and the data currently in the sector buffer in bytes 129-256 will be moved to positions 1-128. I really can't imagine why you might want to do that, however!

DOUBLE DENSITY USERS PLEASE NOTE: Due to a quirk in the Atari operating system, the first 3 sectors on any disk must be single density. Don't try to find the second half of those sectors' It's not there. Due to a bug in early Percom double sided drives, sectors 1436-1439 are also single density and sector 1440 cannot be accessed. If you have a recently manufactured dual head Percom, try answering the "Is it a Percom" question you will be asked upon configuration with "N". Then try reading those sectors. If you are successful, the bug has been fixed!

The System Label File
(LABELS.LDT)

770	($Ø3Ø2)	DCOMND
771	($Ø3Ø3)	DSTATS
772	($Ø3Ø4)	DBUFLO
773	($Ø3Ø5)	DBUFHI
778	($Ø3ØA)	DSECLO
779	($Ø3ØB)	DSECHI
794	($Ø31A)	HATABS
832	($Ø34Ø)	IOCB
834	($Ø342)	ICCOM
835	($Ø343)	ICSTAT
836	($Ø344)	ICBAL
837	($Ø345)	ICBAH
840	($Ø348)	ICBLL
841	($Ø349)	ICBLH
559	($Ø22F)	SDMCTL
560	($Ø23Ø)	SDLSTL
561	($Ø231)	SDLSTH
623	($Ø26F)	GPRIOR
53760	($D2ØØ)	AUDF1
54273	($D4Ø1)	CHACTL
54003	($D2F3)	CHART
54281	($D4Ø9)	CHBASE
756	($Ø2F4)	CHBAS
712	($Ø2C8)	COLOR4
708	($Ø2C4)	COLORØ
709	($Ø2C5)	COLOR1
710	($Ø2C6)	COLOR2
711	($Ø2C7)	COLOR3
704	($Ø2CØ)	PCOLRØ

774	($Ø3Ø6)	DTIMLO
769	($Ø3Ø1)	DUNIT
768	($Ø3ØØ)	DDEVIC
743	($Ø2E7)	MEMLO
763	($Ø2FB)	ATACHR
54018	($D3Ø2)	PACTL
54279	($D4Ø7)	PMBASE
632	($Ø278)	STICKØ
633	($Ø279)	STICK1
624	($Ø27Ø)	PADDLØ
625	($Ø271)	PADDL1
53260	($DØØC)	SIZEM
53256	($DØØ8)	SIZEPØ
53775	($D2ØF)	SKCTL
54283	($D4ØB)	VCOUNT
54277	($D4Ø5)	VSCROL
512	($Ø2ØØ)	VDSLST
522	($Ø2ØA)	VSERIN
546	($Ø222)	VVBLKI
548	($Ø224)	VVBLKD
702	($Ø2BE)	SHFLOK
53761	($D2Ø1)	AUDC1
53768	($D2Ø8)	AUDCTL
53762	($D2Ø2)	AUDF2
53763	($D2Ø3)	AUDC2
53764	($D2Ø4)	AUDF3
53765	($D2Ø5)	AUDC3
53766	($D2Ø6)	AUDF4
53767	($D2Ø7)	AUDC4

53279	($DØ1F)	CONSOL
53277	($DØ1D)	GRACTL
53248	($DØØØ)	HPOSPØ
53774	($D2ØE)	IRQEN
54276	($D4Ø4)	HSCROL
53769	($D2Ø9)	KBCODE
53770	($D2ØA)	RANDOM
54286	($D4ØE)	NMIEN
53275	($DØ1B)	PRIOR
53264	($DØ1Ø)	TRIGØ
644	($Ø284)	STRIGØ
54282	($D4ØA)	WSYNC
53265	($DØ11)	TRIG1
645	($Ø285)	STRIG1
53274	($DØ1A)	COLBK
53270	($DØ16)	COLPFØ
53271	($DØ17)	COLPF1
53272	($DØ18)	COLPF2
53273	($DØ19)	COLPF3
53278	($DØ1E)	HITCLR
53249	($DØØ1)	HPOSP1
53250	($DØØ2)	HPOSP2
53251	($DØØ3)	HPOSP3
53252	($DØØ4)	HPOSMØ
53253	($DØØ5)	HPOSM1
53254	($DØØ6)	HPOSM2
53255	($DØØ7)	HPOSM3
564	($Ø234)	LPENH
565	($Ø235)	LPENV
54016	($D3ØØ)	PORTA
54017	($D3Ø1)	PORTB
53257	($DØØ9)	SIZEP1
53258	($DØØA)	SIZEP2
53259	($DØØB)	SIZEP3

F FIND AN ADDRESS ON DISK

This option enables you to specify the equivalent memory address of a byte on the disk and to let the computer find for you any subsequent memory address relative to that starting byte. (Once you have defined a starting address, it remains the default starting address until you define a new one or leave this module for the DISK DOCTOR system menu. Simply hit RETURN in response to the prompt in order to use the same address.) As usual, you may choose to have the computer assume sequential sectors (in which case the calculation is immediate) or use the DOS file pointers (in which case the computer must read every sector of the file in order to follow the pointers). When the proper byte is located, the sector will be displayed with a pointer to the correct byte and a confirmatory message at the bottom of the screen.

NOTE: Please refer to the caution expressed with regard to relative addresses in the DISASSEMBLER instructions. If you are examining a DOS file, use the Object file analyzer from the main DISK DOCTOR menu first.

This concludes our introduction to the main module, to return to the DISK DOCTOR system menu, simply choose option M.

2 THE ULTIMATE DUPLICATOR

This machine language program allows you to duplicate a complete single density disk; indeed to make up to 255 copies of a single disk on one read of the original. If you have two drives, you may run it from the main menu, but if you only have one, you may save yourself some disk swapping by removing the BASIC cartridge and rebooting DISK DOCTOR II. When so booted, THE ULTIMATE DUPLICATOR comes up immediately.

All destination disks must be formatted before using the DUPLICATOR. Preferably, use the format option from the DOS file recovery module in order to verify the integrity of the disk.

When the DUPLICATOR first comes up you will see the following display:

```
      DISK DOCTOR II
   ULTIMATE DUPLICATOR
           ØØØ
      SOURCE       1<
      DESTINATION  1
      VERIFY       -
      START      ØØ1
      FINISH     72Ø
      COPIES     ØØ1

   <>,START,SELECT(map)
```

These values represent the default parameters for making 1 copy of all the sectors (1-72Ø) from a source disk on drive 1 to a destination disk on drive 1 and writing the destination without verify. To change any of these values, simply move the arrow cursor with the > and < keys and hit the return key when the cursor is positioned next to the parameter you wish to change. As you hit the return key, the drive numbers will toggle through the 4 possible values and the verify parameter will toggle from plus (with verify) to minus (without verify). Writing without verify is more than twice as fast as writing with verify, but is slightly less reliable.

When you hit return when the cursor is resting on one of the lower three values, the bottom most line (the "prompt" line) will change to "enter value". Enter a decimal number from 1 to 72Ø for the sector numbers, or from 1 to 255 for the number of copies you want the program to make. Hit return, and that number will be transfered to the proper parameter.

Once the parameters are properly established (make sure the starting sector is lower than the finish!) and your disks are inserted in the proper drives, hit the START button to begin. (The START button is the proper response to all general prompts on the prompt line, but any time the computer is waiting for you to push START, you may abort the entire procedure by hitting the ESC key.)

You can abort the copy while the program is reading your source by holding the OPTION button, whereupon the computer will write out whatever has been read up to that point to the destination disk(s).

The DUPLICATOR uses a compaction scheme that allows it to copy almost any disk in only two passes on a 48K machine. Many will copy in only one pass.

As each sector is read or written, its number appears on the top of the display. If the computer encounters any bad sectors, that sector number will be displayed to the right of the current sector number. (Such sectors will not be written to the destination disks.) When the entire process is completed, you may request a listing of all the bad sectors on the source disk by hitting the SELECT button (sector "map") To copy another disk, simply hit START to return to the original display.

If the duplicate you make with the ULTIMATE DUPLICATOR does not run properly, and the drive "squawks" when the original loads, you will have to use the bad sector create module of option 5 from the DISK DOCTOR II main menu.

3 DOS file recovery/directory edit and Format bad disks

This module allows you to edit directories with ease, to recover mistakenly deleted or lost files, to repair the allocation map (Volume Table of Contents) of the disk and to format disks that you cannot or don't want to format using DOS. Double density Atari DOS and double density/double sided BJ DOS are fully supported.

As in the main module, use the configuration option ("C") both to change default drives and to reconfigure or establish double density default parameters.

To edit the directory, enter option A. You will see a display like the following:

F#	NAME	EXT	START	LEN	FLAG
Ø	DOS	SYS	4	39	L
1	TEST	DAT	43	1Ø8	D

The first column is the file number assigned by DOS to this file. The file name proper and extension (note no period between them) follow. In column 4 you will see the starting sector of the file. Column 5 has the length of the file (the same number you see when you do a directory from DOS.) The last column will usually be of most interest. L means the file is locked (as far as DOS is concerned). If there is nothing in that column it means the file is unlocked but in use. If there is a D there it means the file is deleted (as far as DOS is concerned). That means that next time DOS tries to write a new file to this disk it will use that file # for the new file since the user doesn't want the old file anymore. But until DOS actually writes a new file into the sectors where the old file was, the old file is still on the disk. If you have accidentally deleted a file and want to recover it, and you have not written any other new information to the disk since then, your file should be totally recoverable.

TO RECOVER AN ACCIDENTALLY DELETED FILE or to edit any other information in the directory (you may want to put some special characters in the file name. Warning: if you do, you won't be able to read that file from DOS), simply type in the appropriate file number in response to the prompt line

ENTER F# TO EDIT,W TO WRITE SECTOR OR <CR> TO SEE MORE

Follow the prompts from that point on. Each time you make a change, that sector of the directory will be re-displayed on the screen as changed. If you want to make more changes, enter another file #. When you are satisfied with your editorial work, enter W in order to write the modified sector to the disk. If you don't write it to the disk, it won't be there!

(If you want to add a file, or any other information, to the end of the currently active directory area, you will have to write out some dummy file names to the disk first, be it from DOS, BASIC or the DISK DOCTOR main module, then go back to this module.)

After you have changed the flag of a Deleted file back to locked or in use, you still are not home free; for when DOS deleted the file it also marked all of the sectors in the file as free to be used again for other purposes. Were you to try to

save a new file to the disk now, you would almost certainly lose much of the old file as the sectors were reused. You must fix it so those sectors are marked "in use" again. To do so, simply choose B FIX THE ALLOCATION MAP.This part of the program reads through every file that is currently marked as active on the directory, rebuilds the allocation map accordingly, and corrects the free sector count.

If you should make a mistake and run the allocation map repair module before the directory was properly prepared, simply make your directory changes and rerun the module.

An alternative to fixing the allocation map after mistakenly deleting a file is simply to mark it in use, copy it to another disk (using DUP.SYS) and then write it back to the original.

E. Erase all files: Use this option to create a clean DOS disk without reformatting. The reformatting process is a rather drastic one to subject your disks to every time you want to clean off old data or programs, and frequently you may find that an old disk when reformatted will result in bad sectors. Avoid this problem by using this option.

F. Recover lost files: This option should be used if a directory sector (361-368) has gone bad. (Since these sectors of the disk are modified frequently, they tend to become bad more easily than others. You will know you have this problem if you get a 144 error when you ask for a directory.) The proper procedure is to copy the defective disk to a new disk with THE ULTIMATE DUPLICATOR. Note which directory sectors are bad on the original. Then run this option on the new disk.

This procedure takes a while, so be prepared to wait. As each file that belongs to the missing directory sector is discovered, its sectors will be printed to the screen. When the process is over, the recovered files will now be listed in the directory as FILEA, FILEB, etc. (Note: the recovered files may well include some you have earlier intentionally deleted but not yet overwritten.) You will now have to go in with the main DDII module (or DOS, or BASIC, etc.) to try to determine just what each of those anonymous files is. Once you have made that determination, give the files meaningful names with option "A" and delete those files that have to be deleted. Just for safety's sake, you should now probably run the allocation map repair module, too.

FORMAT

The last option from this sub-menu allows you to format defective disks. If you have some disks that DOS refuses to format for you, run them through this program. Of course you can also use this option to format healthy disks as well.

If there are indeed damaged areas on the disk, the drive will try to format the disk twice. Then the program will write out the boot code, the VTOC and the directory sectors on the disk. The first entry in the directory will not be a file, but will be a statement of how many bad sectors are on that disk. Those sectors will have been marked as in use in the VTOC, so from now on DOS will avoid them entirely.

PERCOM OWNERS: Unlike Atari drives, Percom drives do not send back an error message to the disk operating system telling it that it could not format successfully. Therefore, if you format normally you may well be using disks that are actually faulty. Use DISK DOCTOR instead. There is one drawback. Some Percom drives (especially slave drives) seem to have great difficulty reading the inside sectors of a disk. Thus, although successfully formatted, the drive may erroneously inform the computer that some of the higher number sectors are bad. The program will mark them as bad sectors. In order to determine if your drive is in this category, run SuperDuper Duplicator on disks known to be sound. If your drive struggles to read the innermost tracks, I would suggest you see if you can get the drive replaced.

4 OSA+ 4 recovery / directory edit

<u>NOTE: DO NOT TRY TO RUN THIS MODULE UNLESS YOU HAVE A DOUBLE DENSITY DRIVE!</u>

Since unmodified Atari DOS 2.0 cannot access more than 1023 sectors, a need was felt to provide an alternative DOS to handle double sided drives and other non-Atari manufactured drives. Optimized Systems Software, Inc., the creators of Atari DOS filled this need with the introduction of their OSA+ version 4 Disk Operating System. As of this writing, OSA+ 4 is the only DOS being provided with PERCOM drives. (This is lamentable. Originally PERCOM provided a very usable modification of DOS 2.0 for their single sided double density drives that, unlike OSA+ 4, was compatible with all Atari software. Such a DOS is still available with MicroMainframe drives and from BJ Smartware. Single density drive owners may ignore all that follows. As delivered, OSA+ 4 will not run single density.)

By the way, if you have OSA+ version 4.0 only, insist on getting an update to version 4.1. The former is bug-ridden and practically unusable.

The file structure of OSA+ 4 is totally different from that of Atari DOS, therefore totally different procedures are required to recover lost files or to trace the sectors of a file. This module is intended to provide those procedures.

As the program comes up, you will first be asked what drive to configure for double density operation.

A. Edit the Directory: This option functions similarly to the equivalent option in the DOS directory edit program. The information contained in an OSA+ 4 directory sector is somewhat different, however. In addition to the file name and length, a file is marked as locked or unlocked and a sector number, the number of the first sector "map" of the file, is given. That "map" sector is simply a list of all the data sectors in the file. (If the file is a long one, there may be more than one map sector; the pointer to each map sector being found in the previous map sector.) When you delete a file, as with Atari DOS, the file name remains in the directory until overwritten with another file, but the starting map sector number is simply changed to 0. So, once deleted, there is no simple way to find out out where that mistakenly deleted file used to begin. Thus the need for option B, RECOVER LOST FILES. Run this option and the program will search the disk for all possible file maps not currently found in the directory. (Wait, this takes time!)

Once you have a list of possible file maps, you should run option D, DISPLAY FILE MAP to determine the actual sectors included in the lost file(s) and check them out with the main module if there is some doubt as to which file is which. Note that if a file has more than one map, the second map will automatically be displayed when the first is finished. Once you are sure which maps go with what file names, put that information in the directory with option A. Lastly, recopy that recovered file onto a new OSA+ 4 disk (else next time you write a file to the old disk the recovered data will be overwritten since the relevant sectors were marked as free in the VTOC when you originally deleted that file.)

This same procedure should be followed in order to recover files should a directory sector go bad; though of course in this case you will have many more file maps to deal with. Unfortunately, there is little alternative. OSA+ 4 offers several distinct advantages over DOS 2.0 when everything works right, but when it doesn't, it can be a real mess.

5 RPM test and create bad sectors

This option of the main menu allows you to test the speed of your 810 drive and to adjust it. It also enables you to create sectors on the disk that the drive cannot read when operating at normal speed and to check the error status of any sector on the disk.

You will see the following menu:

A CHECK RPM
B WRITE BAD SECTORS
C READ SECTORS FOR ERROR CODE
D SYSTEM MENU

To check the RPM (revolutions per minute) of your drive, simply hit the A key in response to this menu. (It's wise to have a rather unimportant [but formatted!] disk in the drive during this process since the disk will be read many times during the check.) After the RPM begins to display, simply hold down the OPTION key in order to abort the speed check and return to the menu. The normal speed of the 810 drive is 288 RPM, but if you are within a range of 284-292 I wouldn't mess with it. If you have to adjust it (and if you've been having trouble with it, that may be the only maintenance it needs) refer to appendix III for the location of the adjustment controls. You may adjust the speed while the RPM tester is spinning the disk.

The proper speed of PERCOM and MicroMainframe drives is 295-300 RPM.

Bad Sectors:

There are several ways to create "bad" sectors on a disk, i.e. sectors that the drive is unable to read. The easiest is simply not to format them in the first place. To do this, follow your normal process for formatting disks, but as the drive starts to whir, count the clunks as the drive head moves from sector to sector. (The drive formats a disk by laying down sector map information and data (all zeros on an 810, all 1A's on a Percom), starting at sector one on the outside of the disk and proceeding to sector 720 on the inside. After it has written data to each sector it reads the sectors in reverse order to determine that the data has been properly written. Only the writing process is formatting proper. The verification read does not have to occur for the disk to be formatted.) Each clunk is one track (eighteen sectors). If you want to format, say, only the first 400 sectors or so, when you hear the twenty-third clunk (400/18=22.22), quickly pop open the drive door and remove the disk. This will not damage the disk, since the read/write head is lifted off of the disk as soon as the door is opened. (Do it too often ,though, and it could loosen things up in your drive!)

If you have a disk that is already formatted, however, or want to create bad sectors only at precise points on the disk, you will have to use another method. This is where our program module comes in. What we want to do is write data to the sector in question so that under normal circumstances it cannot be read. We can do this either by writing the data at a speed substantially different from the normal one, or by locating the data somewhere that it is not supposed to be.

The former method is the most reliable when writing large blocks of bad sectors. To use it, choose option A and adjust your drive speed to about 220 RPM. (The speed is not as important as how the drive sounds. It should be reading the sectors evenly --beep - beep - beep -- but clearly struggling to do so.) When you have set the speed properly, hit the OPTION key to go to the menu. Be sure the proper disk is inserted in the drive. Hit B for the bad sector option. The program will ask you to indicate the starting and ending sectors of the block of bad sectors you wish to create. (Enter them in decimal.) When you are finished creating bad sector blocks, enter 0 to the next prompt and you will be returned to the menu. (To adjust RPM, see Appendix III.)

Now you will probably want to adjust your drive speed back to normal and test the results of your labors with option C. When the drive is set to normal speed again, that option will let you see if the sectors you have worked on are indeed bad. If the error number returned is 1, then the sector is good. An error number greater than 128 indicates a bad sector.

Owners of Percom drives (whose speed apparently is not user adjustable) and those who want to create only a few bad sectors at a time, can use another method. Attach a looped piece of cellophane tape to the top edge of the disk, about a third of the way from the right hand side when looking at the labeled side, creating a "tab" that allows you to pull on the disk while it is inserted in the drive and the door is closed. Pull gently but firmly on this "tab" as the drive is writing to a sector, and, with some practice, you should soon be able to create bad sectors more often than not - without having to undergo the rather nerve-wracking experience of fiddling with the innards of your drive.

Yet another method for those with access to a computer that can format selected tracks of a diskette, is to format the disk on your Atari and then to reformat the selected track on the other computer.

6 Read a BASIC program from disk

Many times authors selling easily deciphered (i.e. well-written) BASIC programs wish to protect their efforts from such decipherment; but under normal circumstances, when you are running a BASIC program and hit the SYSTEM RESET button, you will be able to LIST the program that you have been running.

There are two ways to circumvent this possibility. One is to put garbage into the variable name table. This is the table of variable names that is stored by BASIC along with the program when it is saved to disk and read back into memory when the program is loaded. But BASIC does not use this table when running the program; within the program itself each variable is represented by a number, not by a name. The variable name table is used only when you ask BASIC to LIST the program, or when you add or edit program lines. If you try, then, to list a program whose name table has been destroyed, you will get something rather unhelpful.

A further element of protection is to destroy the last statement pointer of the program. After the line number of each statement in a BASIC program, BASIC puts a pointer consisting of the offset from the beginning of the current line to the next line. That way BASIC can find the beginning of any program line without reading through every byte of the program. It only has to read the pointers. Due to a peculiarity in BASIC, you can use this system to your advantage. BASIC automatically assigns the line number 32768 (i.e. 32K) to the immediate command line (e.g. when you type LIST). If the pointer to that line number is missing, BASIC is unable to find the command line, and will not respond to commands.

Many of the programs of DISK DOCTOR are protected precisely in these two ways. Try to load them in from BASIC and see what happens. Up until now the only way you could read such programs was to physically trace through the file on the disk with a sector editor, restore the line pointer at the end of the program, and put in some dummy variable names into the name table. This module lets you avoid all these steps. Moreover it lets you create a working, modifiable copy of such programs with very little effort.

What DISK DOCTOR does is to read the program directly off of the disk and to translate the BASIC "tokens" back into normal BASIC words. Instead of the original variable names used by the author (whether or not they are still on the disk), it will assign the new variable "names" V0 thru V127. (Hardly very meaningful, are they, but certainly much better than a string of control characters.)

If you need a directory of the disk to remind yourself of the name of the program you wish to read, the program will provide it. Then simply enter the name of the BASIC program you wish to examine. If you choose to have the output sent to the printer or to the screen, any control characters in the program will appear as an inverse C in the listing. If you send the listing to a disk file, the correct control characters will be incorporated into the file. When the listing is done, you may request to see the equivalent variable names in the variable name table itself, but don't be surprised to see garbage.

The listing will also include sector/byte information when it discovers a faulty line pointer or other erroneous information.This should enable you to go back and repair certain troubles in your own BASIC programs as well.

In order to create a working copy of the file, simply send the listing to a disk file. When you ENTER that file from BASIC (be sure to ENTER not LOAD it) you will undoubtedly see a few ERROR messages, due to the fact that it is not always readily determinable whether or not a variable is a string variable. If you have any programming experience at all, however, you ought to be able to fix those easily. Some examples:

```
A$(1,4)=B$ will appear as V1$(1,4)$=V2
B$=A$(1,4)  will appear correctly as V2$=V1$(1,4)
ADR(A$) will appear as ADR(V1)
```

In any case, there will be far fewer errors to correct if you use this method then if you try to go in and restore the variable name table. In that case nearly every string and array statement will need correction. You will also have to eliminate any extraneous statements such as "end of program detected at".

If you see that the variable name table is intact, however, the best way to get a useful listing of the program is to use this module to find the location of the broken line pointer and to restore it. But this is best left to advanced users. Novices may experiment a bit on their own programs to see how this works: Write short programs; save them to disk; then go in with the main DISK DOCTOR module and examine what the saved program actually looks like on disk.

7 List Object file blocks/locations

Machine language programs (binary load DOS files, loaded thru DOS option L) are frequently stored on disk in blocks of memory rather than as a single long piece of code to be loaded into one location. (This is due to several factors; the style of the programmer, the peculiarities of the assembler or linker used, and whether or not several files have been appended to another.) The first two bytes of any such file are always $FF $FF, followed by the two bytes of the starting address where the following code is to be loaded and the two bytes of the last address of the load block. If there are additional blocks, they may be introduced by a similar six byte header or only have the four bytes of address information.

This module lets you quickly analyze this information. Simply type the file name (use wildcards if you wish) as you would with DOS, and the memory blocks in the file, as well as their locations on the disk, will be listed to the screen or printer. The locations given refer to the start of the blocks of code themselves, not to the headers.

If you wish the information to be sent to your printer, answer the "hardcopy?" prompt with Y.

The program will also indicate the initialization addresses and run address of the target program, as well as its length.

8 Create or Edit a Label file

The last option from the main system menu allows you to create and edit files to be used with the disassembler of the DISK DOCTOR main module. The program is self prompting and you should have no trouble with it. It includes an option for printing out a label file of equivalents to the screen or a printer.

You can create a file of up to 96 addresses and labels. Addresses can be given in decimal or hex notation, but must be non zero-page addresses. (I.e. $000A is a legitimate address for the file, but will not be recognized by the disassembler.) Labels can be from one to six characters long.

Label file names are automatically given the extension .LDT

As usual, it is not advisable to use file names only one character long.

Now you know all there is to know about DISK DOCTOR. Be careful, enjoy, and happy DISK DOCTORing.

THE ATARI DISK SYSTEM

Atari compatible disk drives store data on the 5 1/4 inch floppy disk in a series of 40 concentric circles or "tracks". These tracks are numbered by convention from outside in, so that the highest numbered track is the smallest circle near the center of the disk.

When you format a disk, each track is divided up into 18 divisions or "sectors" of data. Each sector is able to store 128 bytes of information in a single density environment and 256 bytes in a double density environment. The 810 disk drive can only read and write 128 bytes to a sector. Even if you are using "double density" disks, you will only get 128 bytes to a sector on the 810. You must have a double density capable drive in order to pack those additional 128 bytes into a sector.

Now since there are 40 tracks, and 18 sectors per track, there are obviously a total of 18*40=720 sectors on one side of a disk. Since each sector can store 128 bytes of information, a single-sided, single density Atari drive can provide "on-line" access to 128*720=92,160 bytes of data on one disk.

The operating system of ATARI computers allows the experienced programmer immediate access to any one of those 720 sectors. Indeed many games and other machine language programs are stored directly on the disk, sector by sector.

THE BOOT PROCESS

When the computer is cold-started (that is to say, turned on) it checks to see if disk drive #1 is on. If so, it tries to read the data on the first sector of the disk. If successful in doing so, the computer looks at the first six bytes of the data in that sector in order to determine what to do next. The computer is primarily concerned to know how many sectors of data it should read off of the disk and where it should put that data. A typical (and well known) series of such bytes is:

00 03 00 07 40 15

This is a very meaningful statement to an Atari computer. It means: read in three sectors of data (including sector 1) and put them at sequential locations of RAM memory starting at address $0700; start executing the machine language code at location $0706; and when that is finished jump (=GOTO) to location $1540 to run the program. But three sectors of 128 bytes each beginning at $0700 will only reach to $087F. How does the program that is to be executed at location $1540 get into the computer? It gets there because the program starting at location $0706 reads it in to RAM sector by sector.

This is what as known as the boot ((bootstrap) process. The computer automatically loads some of the program, and that part of the program loads the rest. This is the way that the great majority of machine language programs are loaded from disk into the computer.

DOS

For most users, however, the normal way to access the disk drive in order to store or retrieve data is through DOS, the Disk Operating System, be it ATARI DOS or any of the various third-party DOS equivalents that use the same format. DOS is nothing more than a "booted" machine language program that allows the user to access the disk using filenames rather than worrying about where precisely on the disk each byte of information is going to be placed. DOS does all of those calculations automatically. DOS keeps track of what sectors are in use, how many sectors are available to be used, and which files are locked, in use, or deleted.

In order to do all of this, however, DOS has to sacrifice some of the room available for data. Only 125 bytes of each sector on a DOS disk are available for data storage; the last three bytes are used by DOS for housekeeping. Moreover, due to a bug in DOS, it cannot use sector 720 of the disk. Thus, since DOS reserves the three first sectors of the disk for its "boot" code, uses 8 sectors (361-8) to keep a directory of the files on the disk, and uses 1 sector (360 the VTOC or Volume Table of Contents) to keep a "map" of all the other sectors on the disk and their status, only 707 (720-1-3-8-1) sectors are available for data storage on a DOS disk, and only 707*125=88375 bytes of data may be stored.

When DOS starts putting files onto a newly formatted disk, it writes the files all out in sequential sectors, one after the other. But after a few files have been deleted, DOS soon finds that it can no longer store information in a single file in sequential sectors but must jump around the disk looking for free space to put the data. In order to do this, DOS uses a system of pointers. Each sector of a file has a pointer to the next sector of the file. At the end of the file, the pointer is 0. The location of the first sector of a file is kept as part of the directory.

The file pointer is a part of the last three bytes of each sector. These three bytes (bytes 125,126,127; remember the first byte of a sector is 0, but the first sector of a disk is 1!) mean the following:

byte 127 how many bytes of the sector are used. If the sector is full it will contain the value $7D=125

byte 126 the low eight bits of the file pointer

byte 125 the file number to which this sector belongs (bits 2-7) and the high two bits of the file pointer (bits 0-1)

Examine, for example, the last three bytes of sector 4 on most disks and you will probably see the following:

00 05 7D

This means that this is a part of file number 0, the next sector of the file is sector 5, and this sector has 125 bytes of data.

When a new file is written to the disk, before writing out each sector DOS determines where it is going to put the next sector and inserts the appropriate file pointer into the end of the sector. When DOS reads a file, whether it is providing data for a GET command in BASIC, loading or entering a BASIC program, or loading a machine language program, it retrieves sector after sector according to the pointers until the end-of-file marker is reached. But DOS also checks to make sure the file numbers in each sector are consistent, and if they are not, you will get an ERROR 164 message.

Why should the file number be wrong? Although ATARI drives are relatively reliable, a bad byte can always occur now and then. And if one should occur in the VTOC, which is rewritten to the disk every time a file is added or deleted, the next time DOS needs file space it could appropriate a sector in the middle of another file.

DOS also seems to have a particlarly hard time of it when it tries to write a new file on the disk and runs out of space in the middle of the write. This will frequently result in a "free sectors" statement that is incorrect.

DISK DOCTOR allows you to recover all or most of a file wherein something has gone amiss, be it a single byte, a file pointer, or a mistakenly deleted file. To fix a file where you receive an ERROR 164, for example, trace the file, examine the erroneous sector in the file chain (probably part of a totally different file), and fix the file pointer in the sector that points to the bad one to point to the next sector in the file. You ought to be able to recover the entire file except for the bad sector.

SOME DOS FILE POINTER EXAMPLES (bytes 125,126 of each sector):

125	126	125 in binary	file #	next sector
1C	12	000111/00	07(=000111)	012
32	A0	001100/10	0C(=001100)	2A0
51	F0	010100/01	14(=010100)	1F0

PROTECTING PROGRAMS

Since a program such as DISK DOCTOR can give you access to every sector and byte of data on the disk, how can programs be protected against indiscriminate copying?

The earliest disk software for ATARI was not really protected. Since most users only had access to the "duplicate disk" option in DOS, all that had to be done was to put data into sectors that were marked as unused in the VTOC, for the "duplicate disk" option does not really duplicate everything, only those sectors that are marked as in use.

Soon, however, ponderous (in speed, not in size) BASIC programs (that take about 20 minutes to write) that could copy all the sectors of a disk became widely available for ridiculously high prices, and software authors perfected another scheme, still widely used, called bad sectoring. The trick was to format a disk in such a way that not all the sectors on it were readable. In order for the program to run, it must try to read one or more of those "bad" sectors and find that they are bad. If they are not bad, the program will stop, reboot the disk, or some other such thing; for the program "knows" that it is not on its original disk, the one with the bad sectors.

This scheme, too, was soon easily defeated. First by skilled programmers who could examine the machine code on the program and alter the bad-sector checking code. Indeed a skilled programmer can turn a typical bootable disk program into a binary DOS file and store six games in the space that used to be taken up by one. But few (fortunately) are those able to do things like that. If you wish to try to achieve such skill, DISK DOCTOR can help you do it.

DISK DOCTOR enables you to use another method to back up disks protected by bad sectoring, that is to create bad sectors on the destination disk exactly where you found them on the source disk. I do not believe that making a backup disk of software that you own is a crime. It is certainly not immoral. But I am not a lawyer. Check with your own, if you are concerned. Unquestionably, however, distribution of unlawfully duplicated opyrighted material, especially if for personal gain, is a serious crime. Please don't do it.

The ATARI disk drive is "intelligent". It has its own on board computer that does most of the serious work. It is for this reason that protection techniques for ATARI software remained so primitive for so long. When the 810 is told to format a disk, it formats it the way it wants to, and the user has no control over the results. On other computers, the user can specify how the sectors are to be organized in any given track. For the sectors are not simply arranged sequentially within a track. Ideally one tries to arrange the sectors so that the read/write head is positioned over the next sector as soon as it is done processing the data from the previous one. Thus optimum placement of the sectors of a track depends on how long it takes to process that data and how fast the disk is spinning. To see how this is true, find someone with a Percom drive and listen to a disk that has been formatted on a Percom drive. On the Percom, that spins at 300 RPM, the disk loads substantially more quickly than does a normal 810 formatted disk. But try writing to that disk on an 810. It takes audibly longer than usual. That's because the sectors are arranged differently on the track by the Percom drive.

How, then, does the drive know where the sectors are if they are not always in the same order? It knows because there is actually more data in each sector of a disk than just the data visible to the normal ATARI user. When formatting the disk the drive lays down all sorts of additional information to enable to find the various sectors, marking each sector with a sector number. But when called by the computer, the chip in the drive will only respond with the data of the sector (and the checksum).

As more and more clever people became interested in ATARI, it was inevitable that special drives and modifications would be developed that would allow large scale software houses to format their disks in special ways. When these disks are read according to the code (usually itself encoded) in the boot program of the disk, the data is read correctly and the program runs. When copied by any straightforward software process, however, (such as SuperDuper Duplicator), crucial code-bearing sectors are misread.

An example: One sophisticated manufacturer regularly revises every last sector of a track (18,36,54,etc.) so that the drive thinks it is another second-last sector (17,35,53). The only way to read that sector is to read the previous sector twice very quickly in succession. The drive tries to read the same sector it just read before,but since it hasn't had a chance to make a complete revolution yet, it actually is tricked into reading the second sector with the same "name". This is what we have referred to as special formatting; a procedure that cannot be defeated by software alone. Unfortunately, as more and more software houses (and individuals) acquire the hardware to enable them to produce this kind of protection, the percentage of them that seem to be offering backup disks for a nominal charge seems to be decreasing. This is as lamentable as the achievement of relatively "unbreakable" protection schemes is laudable.

6502 ASSEMBLY LANGUAGE

A brief introduction.

In order to make use of the full potential of DISK DOCTOR, familiarity with the machine language of the 6502 family of computers is obviously necessary. But even a novice can follow the logic of parts of the most complex programs, if the basics of 6502 Assembly language are mastered.

The fact is that computers are extremely simple machines. They can only count to 1, and they can only follow simple instructions. It is only by combining these simple instructions and these simple numbers into complex combinations that computers can be made to do complex things.

The result of this fact is that essentially assembly language, too, is relatively simple to understand, although not necessarily simple to program in.

The following discussion assumes that the reader has some familiarity with the hexadecimal number system, and such basic concepts as "byte", "K" and "RAM". If not, please go read a basic introductory book and come back.

Now that you are back, let's begin.

A machine language (assembly language is the same as machine language, except that it uses mnemonics instead of numbers as and aid to people) program is simply a series of instructions, just like a BASIC program, that moves bytes of data from one place to another in the computer's memory, performs simple arithmetic calculations and logical comparisons on that data, and puts the results somewhere where the user can access them. In order to do this, the CPU (central processing unit) must be able to keep track of where in the program it is, and where it is supposed to GOTO when it is done doing what it is doing. It must also have some "registers" (byte size locations in the 6502) where it can manipulate that data, and the ability to fetch and store data instantly from any place in RAM.

The 6502 CPU in ATARI home computers has the following registers:

- A the Accumulator, where most calculations are performed.
- X an "index" register.
- Y another "index" register.
- S the "stack pointer".
- P the status register.
- PC the (two-byte) program counter.

It must be kept in mind that these registers are not RAM addresses. Indeed from a high level language like BASIC the user has no direct access to these registers at all; but they are the things that are doing all the work in the computer. (Even assembly language provides no direct access to the program counter.)

When a computation is performed, a byte is fetched from a memory address, manipulated, and then returned to memory or (what is for the CPU absolutely equivalent) sent to a memory location that actually functions as a "port" to the outside world.

Thus the major programming commands of assembly language must necessarily be very similar to those in high level languages like BASIC: store a certain value in a variable (LET X=3), fetch the value of a variable (LET Y=X), fetch a value - manipulate it - and put it in another (or the same) variable (LET Y=Y+3). Program control commands too, are functionally identical: GOTO, GOSUB, RETURN, IF xxx then GOTO yyy.

Fetching and storing can be done by either the A, X, or Y registers, as can comparisons. Addition and subtraction, however, can only be performed on a byte in the Accumulator.

The fetch command is a "load" command, thus:

LDA load the Accumulator.
LDX load the X register.
LDY load the Y register.

The store command is just that:

STA store the value that is in the accumulator.
STX store the value that is in the X register.
STY store the value in the Y register.

The actual machine language code that corresponds to each of these mnemonics differs, however, depending on the "address mode", that is to say, depending on just where it is that you want the register to fetch from or store to. The 65Ø2 supports several esoteric address modes, but the most common ones are "immediate", "absolute" and "indexed absolute". Moreover the 65Ø2 assigns a very special role to addresses on "page zero" (the first 256 bytes of RAM).

The immediate mode, indicated by the use of #, e.g. LDA #Ø, means load the register in question with the value of the byte immediately following the "opcode" (the mnemonic 3 letter instruction), in this case, load the accumulator with the value Ø.

The absolute mode, indicated by the presence of a full 2 byte address, e.g. LDX $AØØØ means load the register with the value currently found at that RAM address. As is the case with BASIC when you say LET T=N, N (in our case $AØØØ) keeps the value that it had before, but that value is also now found in T (in our case, the X register). Thus the BASIC statement POKE 4Ø96,8 can be "translated" into assembly language as:

LDA #8 (load the Accumulator with the value 8.
STA $1ØØØ (store the value in the Accumulator into RAM location $1ØØØ). Indeed this simple sequence of commands is the one the beginning Atari user will be most interested in. You will find that most instructions to the computer involving color and sound and player missiles, will consist primarily of such sequences.

The indexed absolute mode, indicated by, e.g. LDA $1ØØØ,X means load the Accumulator with the value in RAM location $1ØØØ+whatever is in the X register. If X is 3, then load A from $1ØØ3.

Zero page addressing is particularly important because it only takes one byte to define an address on page zero of memory, but it takes two bytes to define all other addresses. Thus the execution of commands involving zero page addresses is much quicker than equivalent commands using other RAM addresses. There are some special address modes involving zero page addresses, but the only one the beginner should worry about is the following:

LDA ($ØA),Y

This means: Load the Accumulator with the value found at the address contained in RAM locations $ØA and $ØB offset by the value in the Y register! Complex, but very useful in many applications. If $ØA has value of Ø and $ØB has the value of $8Ø and Y contains the number 6, this command means: Load the Accumulator with the value currently found in location $8ØØ6.

NOTE: <u>In 65Ø2 language the least significant byte of a two-byte address is stored in the lower memory location.</u>

Other major opcodes to know are:

JMP $xxxx jump to location $xxxx and continue the program there (i.e. = BASIC GOTO).
JMP ($xxxx) jump to the location contained in the two bytes starting at $xxxx. (Note that "indirection" of addressing is indicated here as above by parentheses.)
JSR $xxxx jump to the subroutine that begins at $xxxx.
RTS at the end of a subroutine, return to the calling program (=BASIC RETURN).
ADC add a value to the current value of the Accumulator.
SBC subtract a value from the current value of the accumulator.
DEC, DEX or DEY decrease the current value of a memory location or of the X or Y register by 1.
INC, INX or INY increase the current value of RAM , X or Y by 1.
TXA,TYA,TAY,TAX transfer the value in the first register to the second: e.g. TXA = transfer X to the Accumulator.
CMP, CPX, CPY compare the value in the accumulator (X or Y) to another value.

Every time one of these operations is performed, the values in P, the "status" register may be changed. If an operation results in a zero value or an absolute equivalence, the so-called "zero" flag of P will be set. If an addition results in a carry, the "carry" flag will be set. (These flags are nothing other than single bits within the P register that are either on or off.) Based on whether or not a flag is set or not, a conditional branch can be accomplished (= BASIC IF - - THEN). The conditional controls are:

BEQ branch if "equals" zero.
BNE branch if not = zero.
BMI branch on "minus"; if the last byte result was $8Ø or larger (in computerese, all values with the most significant bit "on" are considered "minus", even if they are used as postive values by the program.
BPL branch on plus; if the highest bit was not set.
BCC branch if the carry bit is not set (clear).
BCS branch if the carry bit is set.

Let us see now how all this goes together to create a short assembly language program. A frequent operation in such programs is to clear a block of memory. Let us clear all the memory on "page six":

```
        LDA #Ø  (load the Accumulator with Ø).
        TAX     (transfer the A to X, now X contains Ø too).
loop    STA $6ØØ,X (put whatever is in A into $6ØØ+X).
        INX     (increment X).
        BNE loop  (if X is not = zero, go back to the statement
called loop.
```

In the above program, each time that X is incremented, the program will keep looping back upon itself (256 times) until X is finally equal to zero again, at which time control will be transfered to whatever statement follows the BNE instruction. Each time that the program loops, the value in the Accumulator is put into successively higher memory locations on page six. Of course all of this happens in a small fraction of a second.

An important thing to remember about branch instructions is that they only take a one byte operand, that is to say you can only branch to a total of 256 different locations from any one starting point. These instructions use the idea of "minus" arithmetic mentioned before, so that the byte sequence F0 $10 means branch (if equal to zero; F0 is the machine language equivalent of BEQ) hex 10 (decimal 16) bytes ahead in the program. F0 $FF means branch one byte back, for $FF is one "less" than 0, $FE = -2 and so on down to $80= - 128. The branch is calculated starting at the byte <u>following</u> the operand of the branch instruction itself.

BACKWARD RELATIVE BRANCH TABLE

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	96	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	16	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Additional 6502 opcodes

AND logical <u>and</u> the Accumulator.

ASL arithmetic shift left (shift all bits one to the left, highest bit goes into the carry flag.

BIT logical <u>and</u> the accumulator in memory, but change neither; zero,negative and "overflow" flags are affected.

BRK break (stop program execution).

BVS, BVC rarely used branch instructions: branch if the "overflow" flag is set or clear. Overflow is set if an operation has resulted in a carry from bit six to bit seven of a byte.

CLD, SED clear and set decimal mode: Unlike most small CPU's, the 6502 can actually do decimal arithmetic directly. When this mode is in force, 09 + 01 = 10 not 0A as usual.
CLI, SEI clear and set the interrupt flag bit of the status registers. If this bit is set on, so-called "non-maskable" interrupts cannot occur. An interrupt is a summons to the CPU to put aside what it is doing for some more urgent task.

CLV clear the overflow flag. I have never seen this one in a program.

EOR logical exclusive-or the Accumulator.

LSR logical shift right. Shift all bits one to the right, the former lowest bit going into the carry bit of P. Equivalent to dividing by two.

NOP no operation. This is a place holder, like REM in BASIC.

PHA, PHP "push" A or P to the stack for safekeeping.

PLA, PLP "pull" (retrieve) the first (lowest) value on the stack and place it in A or P.

ROL, ROR "roll" left or right. Shifts all bits of a memory location or A one bit left or right, putting the value in the carry flag into the vacated location, and the lost bit into the carry flag.

RTI return from interrupt. After interrupting the CPU to do an interrupt, the offending routine must do an RTI to send the CPU back to what it was doing before.

SBC subtract with carry. To get the correct results, the carry flag must be set before all subtractions.

TXS transfer X to the stack pointer.

A TYPICAL BOOT CODE SEGMENT

Since many users of DISK DOCTOR will want to try to examine booted software, let us have a look at a typical piece of code that you might encounter in a boot code, the first sectors on a disk that actually do the loading of the rest of the program:

```
 1.    LDA #$52
       STA $302
 2.    LDA #0
       STA $304
       STA $30B
 3.    LDA #$75
       STA $90
 4.    LDA #$10
       STA $305
 5.    LDA #$4
       STA $30A
 6.    JSR $E453
 7.    INC $30A
       BNE 02
       INC $30B
 8.    CLC
       LDA $304
       ADC #$80
       STA $304
       LDA $305
       ADC #0
       STA $305
 9.    DEC $90
       BNE E0  (= decimal -32, go back to step 6.
10.    CLC
       RTS
```

This routine or something equivalent to it must be performed by any program that wants to access the disk drive directly. What it does is poke certain values into an area of memory called the <u>D</u>evice <u>C</u>ontrol <u>B</u>lock and then jump to a subroutine in the Operating System ($E453) that takes over from there and performs the disk access as specified by the values it finds in the DCB. In fact, the procedure is so easy that it can be done without difficulty from BASIC (have a look at the DISK DOCTOR progams) by POKEing the appropriate values into the proper areas of memory and then doing a USR call to a string consisting of nothing more than the 4 bytes $68 $4C $53 $E4 (i.e. PLA, JMP $E453 refer to the BASIC manual for the USR function and the PLA).

1. This command puts the value $52 (='R' for 'read') into the location $302 that specifies the disk command, in this case a read of the disk. (Other possible command bytes are $57=write with verify, $53=status check. and $21=format.)

2. Here we put zero into the high byte (MSB) of the sector number we want to read from the disk andthe low byte of the RAM address into which that sector will be read.

APPENDIX II

3. Now we must set up a counter to keep track of how many sectors we are going to read. Because of the speed factor mentioned earlier, most programs use zero page addresses like $90 for this kind of a variable.

4. $305 is the MSB of the buffer. Here we are saying that we want the information on the disk to be stored starting at RAM location $1000 (remember we put a 0 into the LSB in 2.

5. $30A is the LSB of the sector number to be read. We want to start reading from the disk at sector 4.

6. Let the Operating System take over from here and do the read.

7. Increment the sector number.

8. Add 128 (hex $80, the size of one sector) to the buffer address. CLC means "clear carry" and must be done before all additions. Note that one can add a single byte value to a multiple byte amount simply by adding 0 to all the higher bytes without doing a CLC before those additions.

9. Decrement our counter. If it hasn't reached zero yet, go back and read the next sector into the next buffer area.

10. When finished, return to the Operating System routine that calls up the boot code. CLC is done first to show the OS that the initialization performed by the boot code has been successful.

APPENDIX III

Adjusting drive RPM

NOTE: It is strongly suggested not to open up your disk drive while it is still under warranty. You are liable to invalidate it. Under no circumstances open up your drive in a dirty environment or when loaded with static electricity! Moreover, any damage you may do to your drive when trying to follow any of the following instructions is entirely your own responsibility!

The cover of the 810 drive is easily removed. Simply pry off the four little round plastic covers on the top of the drive, loosen the screw beneath with a phillips screwdriver and gently lift the top up and off.

The speed control is located on the printed circuit board lying flat at the back of your drive.

If your drive (the older ones) has only two boards, the speed control is a dime-sized wheel (probably blue or white) labeled R142 at the left rear corner. Small movements of this control produce large changes in RPM.

If your drive has three boards, the adjuster is a tiny screw protruding from the top of a tiny "box" (green or tan, usually) labled R104, just to the left of the only IC on that back board. You will find that you will need a very tiny screwdriver or knife to adjust this screw, and that it will take many revolutions of the screw to change RPM substantially.

While doing all of this, please try not to displace or sever any of the wires in the drive. My local repairman has a $40 dollar minimum charge to work on 810's!

If you have a new 810 (made later than Jan. 83, or some PERCOMS) you may be unable to adjust your drive speed substantially. There are several solutions if you are unsuccessful at writing bad sectors. If you have an 810, you can replace the resistor in question; the part is cheap, but installation is quite complex. A better solution (indeed one good for all 810 drives) is simply to replace the ROM chip in your drive with one that will enable you to duplicate practically anything. Check with BJ Smartware for information on both of these products.

If you have a troublesome PERCOM, one approach (so we are told