DISKEDIT 2.Ø

A Soft Unlimited Product

DISKEDIT 2.Ø is a major upgrade from 1.Ø. New features include SuperSearch w/wildcards, mass boolean operations, two way buffer transfer, memory editting, SOFTPROMPT Defmode user input, an RPM Tester, and more.

This documentation and the DISKEDIT program are protected by U.S. copyright laws, and may not be copied for any other use than personal backup protection. Copying this software for redistribution or trade is a criminal offense and punishable by fines and imprisonment. Likewise, use of this software for software piracy is also illegal. Soft Unlimited developed this software as a disk analysis and development tool and may not be held responsible for illicit use of the program.

This product is sold as-is. Soft Unlimited will, however, replace defective disks at no charge, if returned within 1Ø days of purchase. Disks returned after 1Ø days must be accompanied by $5.ØØ to cover handling and postage for replacement. So, don't forget to make a backup of the program as soon as you get your disk. A program called PCOPY is included on the DISKEDIT disk to allow you to backup DISKEDIT. When you boot up with DISKEDIT the first time, press the BREAK key and then type RUN "D:PCOPY". PCOPY will prompt you for the source disk, so just press return. When you see the FORMAT? prompt, place a formatted disk in the drive and press return again. You now have a duplicate of DISKEDIT. Please don't start passing copies of DISKEDIT around. I am selling my software for what I consider a reasonable price, because I am gambling that the lower price will lower the tendency to trade. No single SOFT UNLIMITED program will ever have a suggested retail price of more than $25.ØØ, unless piracy destroys our profit margin.

```
*******************************************
* DISKEDIT and ULTRACOPY may be ordered   *
* FROM:                                   *
*   SOFT UNLIMITED                        *
*   3546 PILGRIM LANE                     *
*   PLYMOUTH, MN 55441                    *
*   DISKEDIT-$24.95 Suggested Retail      *
*   ULTRACOPY-$2Ø.ØØ Suggested Retail     *
*******************************************
```

DISKEDIT - A Disk Analysis Tool

Written by Todd Burkey 2/1/82

INTRODUCTION:

Program DISKEDIT was written primarily as a learning experience in order for me to better understand the internal workings of the ATARI Disk I/O subsystem. Starting with a simple routine to read and write sectors from the disk to memory and back, I soon found the need to add editing capability, viewing routines, data transfer routines, multi-disk selection, a boolean calculator, a disassembler with relocatable printout locations, and finally a disk mapper to 'see' the contents of a disk at a glance and a disk copier to move the data around. After getting some suggestions from users of 1.Ø, I added single key entry, a mass boolean operation option, search tools, memory editting, two way buffer transfer, and a default mode prompting system. I also sped up the mapper to full assembly speed and gave the user the ability to map around blocks of bad sectors. The 2.Ø release is also easier to upgrade to double density copy, read, and write modes. All of the control program is written in BASIC with extensive use of relocatable assembly code routines wherever speed was required (disk I/O, Mapping, Boolean operations, viewing, and copying). The final product is a program that will work in a system as small as 24K and loads very quickly as one complete program unit (few data statements were used to create assembly routines upon execution; instead, strings allow the high speed loading.)

As an added bonus, I have decided to market this software as listable source. All of the assembly routines used in DISKEDIT are fully relocatable and may be inserted in other programs by 'LIST'ing the line out to a disk file and then 'ENTER'ing the line back into the source program. None of the routines are what I would consider copyrightable, so feel free to use them in new software development projects. Just don't forget to credit the original developer. The program, however, is copyrighted and as such may not be copied for resale or redistribution.

Any suggestions for improving this software will be welcomed from the users. In fact, at regular intervals (possibly every 6 months) a new production release of DISKEDIT will be available to registered users for a $5.ØØ update fee (trading in your old disk). Users who return the registration form will be notified whenever a release is available. Registration is automatic for direct purchases from Soft Unlimited.

Diskedit was developed for use by both the experienced ATARI hacker and the beginning user. Anyone can use DISKEDIT as an everday DOS utility (i.e. recovering lost files, fixing scrambled sectors, etc.), but to fully utilize the capacity of DISKEDIT, some familiarity with Assembly language is important. Beginning ATARI users should obtain a 65Ø2 reference manual and the ATARI OS manuals for reference when using DISKEDIT to look at source code. Many users have informed me that DISKEDIT turned out to be very helpful in their learning of Assembly language (i.e. by looking at game code and figuring out what is being done, they learn Assembly as a side benefit.)

EXECUTION:

Program DISKEDIT will automatically load once the system is powered up. Make sure the BASIC cartridge is inserted prior to powerup. Loading will take less than one minute. Once loaded, the screen will blank for 1Ø seconds as the software deletes lines that are not necessary for the rest of the session and then a title page will appear, with a brief description of the command options. The RPM of the source disk will then be tested and everything will be all set to go.

COMMAND SUMMARY:

The following command descriptions cover all possible legal responses to the menu level prompt shown below.

S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?

This prompt will be returned to upon successful completion of any command and is also the trap return point for user errors. The menu can optionally be returned to at any point by pressing ESC in response to a command level question. Also, if you break out of the program and want to continue, type GOTO 23Ø to continue (or reboot the DISKEDIT disk). This version of Diskedit also uses a default mode data entry routine which which shows the user what the default response will be (i.e. in answer to a null input) for most of the prompts. This response is then echoed so that you know what was input.

COMMAND-A:

This command allows the user to toggle back and forth between the HEX/ASCII and the disk sector MAP displays (see figs. 1 and 2). DISKEDIT initializes in the HEX/ASCII mode, since this mode is used for editting and prompting. The MAP mode is most useful when the user wants to do a fast scan of which sectors have been mapped. This mode should not be used until the map option has been used at least once.

```
    SECTOR: 53 <$ØØ35>   NEXT S#=54
 #        HEXIDECIMAL          ASCII
ØØ: 55 52 4B 45 59 2C 2Ø 33   URKEY, 3
Ø8: 35 34 36 2Ø 5Ø 49 4C 47   546 PILG
1Ø: 52 49 4D 2Ø 4C 41 4E 45   RIM LANE
18: 2C 2Ø 5Ø 4C 59 4D 4F 55   , PLYMOU
2Ø: 54 48 2C 2Ø 4D 49 4E 4E   TH, MINN
28: 45 53 4F 54 41 2Ø 35 35   ESOTA 55
3Ø: 34 34 31 2E 2Ø 48 4F 4D   441. HOM
38: 45 3A 36 31 32 2D 35 34   E:612-54
4Ø: 32 2D 31 3Ø 32 37 2C 2Ø   2-1Ø27,
48: 57 4F 52 4B 3A 35 34 31   WORK:541
5Ø: 2D 32 3Ø 36 34 9B Ø4 ØØ   -2Ø64%..
58: ØD ØD 2B ØE ØØ ØØ ØØ ØØ   ..+.....
6Ø: ØØ ØØ 16 ØB ØØ 19 ØF 36   .......6
  : 8Ø 2D ØE 41 8Ø 1Ø ØØ ØØ   .-.A....
7Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ........
78: ØØ ØØ ØØ ØØ ØØ Ø4 36 7D   ........

S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?
```

Fig. 1: HEX/ASCII

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
eebex?eeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
    Press A to Return
```

Fig. 2: MAP (unformatted)

COMMAND-C:

This command allows selective copying of sectors from one diskette to another. Any number of sectors may be copied, but use of this software to copy more than 150/87/22 sectors (for 40K/32K/24K machines respectively) will require that the user reinsert the source disk one or more times (unless two drives are used). This copier has the unique feature of being able to relocate sectors through a user specified destination start sector. This allows you to place more than one game on a disk, if you use a menu to pick the game to load. A sample execution printout follows:

```
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?C
 READ 128 BYTES/SECTOR (Y/N)[Y]?Y   <---Reply N for 125 bytes
 START SECTOR[1]?3
 END SECTOR[720]?200
 DEST. START SECTOR[3]103
 INSERT SOURCE DISK, RETURN WHEN READY?
 INSERT DESTINATION DISK?
 INSERT SOURCE DISK, RETURN WHEN READY?
 INSERT DESTINATION DISK?
```

MAIL ORDER ONLY: Include $1.00 Shipping

A program called ULTRACOPY is available from Soft Unlimited for DISKEDIT owners for $12.00. This is a 100% assembly code copier that copies using advanced data compaction techniques. The data compaction will allow you to copy most game disks in one pass (a boon to us one disk drive owners.)

COMMAND-D:

This command allows the user to disassemble any part of memory within the range of byte 0 to byte 65532. This is a one pass disassembler which allows the user the option of setting up the printed addresses as different from the actual memory locations. This is a useful feature when disassembling binary load files or boot files when the start address is known. Another useful feature built into the disassembler is the SUPERSEARCH function. This allows you to search any place in memory (including the Transfer buffer) for a 1, 2, or 3 byte sequence of numbers. A special wildcard search is also allowed in that two bytes seperated by any byte can be searched for (i.e. search for all STA $03xx as shown below in the example. A sample execution printout is shown below in fig. 3.

```
PRINTER (Y/N/Q)[N]?N
SEARCH(0-NO,1-1,2-2,3-SPLIT,4-3)[0]?2
VALUE #1[$20]?$8D
VALUE #3[$E4]?$03
PRINTOUT RANGE [+/- 10 BYTES]?4
OUTPUT MODE (H-HEX,D-DEC,B-BOTH)[H]?B
# OF BYTES[ALL]?18000
OFFSET ADDRESS[$1000]?$714
START ADDRESS[T BUFFER]?=$14
SECTOR:22   BYTE: 34
  1817 0719 : AD 12 07  LDA $0712
  1820 071C : 85 43     STA $43
  1822 071E : 8D 04 03  STA $0304
  1825 0721 : AD 13 07  LDA $0713
USE PRINTER [N]?         <---Use the ESC key to exit
```

Fig. 3: Listing

The disassembly may be interrupted at any point by simply hitting any key on the keyboard. Pressing the space bar will stop the printout for viewing, and pressing the bar again will resume printout. Two final things are important to remember. When typing in the memory location to start viewing at, a C/R response will default to the start of the system buffer (from the Transfer commands' data) and an "=" followed by a number will start at the system buffer address + the number offset. Also, the current sector may be disassembled at any time by starting at address $68Ø (the page 6 buffer I used to store the sector.)

COMMAND-E:

This command allows editting of the last sector read or the memory viewed using the P command. The editting is performed by moving an arrow around on the screen using the keyboard arrows until the arrow points to the byte to be editted. The user then inputs the new value as either a decimal number, a HEXidecimal number (preceded by a dollar sign), or an ASCII character (preceded by a period). The screen will automatically be updated in both the HEX and ASCII displays, and the arrow will point to the next byte on the screen. When the user is finished editting, pressing ESC will return to the main menu. The information can then be written out to the disk drive with the W command.

```
      SECTOR: 53 <$ØØ35>   NEXT S#=54
 #         HEXIDECIMAL           ASCII
ØØ: 55 52 4B 45 59 2C 2Ø 33  URKEY, 3
Ø8: 35 34 36 2Ø 5Ø 49 4C 47  546 PILG
1Ø: 52 49 4D 2Ø 4C 41 4E 45  RIM LANE
18: 2C 2Ø 5Ø 4C 59 4D 4F 55  , PLYMOU
2Ø: 54 48 2C 2Ø 4D 49 4E 4E  TH, MINN
28: 45 53 4F 54 41 2Ø 35 35  ESOTA 55
3Ø: 34 34 31 2E 2Ø 48 4F 4D  441. HOM
38:>45 3A 36 31 32 2D 35 34  E:612-54
4Ø: 32 2D 31 3Ø 32 37 2C 2Ø  2-1Ø27,
48: 57 4F 52 4B 3A 35 34 31  WORK:541
5Ø: 2D 32 3Ø 36 34 9B Ø4 ØØ  -2Ø64%..
58: ØD ØD 2B ØE ØØ ØØ ØØ ØØ  ..+.....
6Ø: ØØ ØØ 16 ØB ØØ 19 ØF 36  .......6
68: 8Ø 2D ØE 41 8Ø 1Ø ØØ ØØ  .-.A....
7Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ........
78: ØØ ØØ ØØ ØØ ØØ Ø4 36 7D  ........
>>>NEW VALUE?
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?
```

Fig. 4: Editting Session

COMMAND-F:

This option is used to format a diskette (notice that the F prompt is not included in the menu-it is hidden to prevent accidental formatting of your disks by the curious.) The disk to be formatted will be the one last identified as the destination disk (disk drive 1 is default). You will be prompted as to whether you really want to format that particular disk after you enter the F option (just in case you typoed.)

COMMAND-M:

This option is used to create a map that will allow you to see where bad sectors on a disk reside and also where the sectors containing data are located. The map command will search every sector of a disk for data, determining the status of each sector (bad, empty, data filled, or repeating data). You will be able to watch the progress of the mapping on the screen as Diskedit creates its map buffer (described under the A option-see fig. 2). As each sector is analyzed, you will see a character appear on the screen. Four different characters are used in the map. An x means that the sector contains data. An e means that the sector is empty (all zeroes). An underscore (printed as a ? in the V option) means that the sector contains 128 identical non-zero bytes. A b means that the sector is bad. Since a complete map will take several minutes (yes, even at full assembly code speed like this is) the user has interrupt control by pressing any key during the scanning. Once interrupted, the program will return to the main menu, but mapping may be continued by again using the M option and specifying a new start sector. Popping in and out of the mapping process is common when mapping around blocks of bad sectors (unless you are wearing earplugs.)

```
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?M
START SECTOR[1]?1
PURGE MAP[N]?Y
```

COMMAND-P:

This command is used to relocate your view/edit buffer from the last sector read buffer ($680-$6ff) to any point in memory. Basically, you can look at any point in memory, and then edit that area using the E option. The most common use for this command is to view parts of the Transfer buffer (see T option) and then edit the data prior to transfering the data back to a disk. The T buffer may be accessed the same as it is in the D option (using the = prefix before the number of bytes offset into the buffer.) This command should be used with extreme care when looking at memory locations outside of the Transfer or last sector buffers. Editting the wrong bytes in the lower 4 pages of memory or inside the program itself can easily hang the system. As long as you have a write-protected disk in the drive, however, it is still fun to play around with page 2 locations (colors, sounds, etc.)

```
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?P
MEMORY START[T BUFFER]?$200
```

COMMAND-R:

This command drops the user into an arithmetic section which allows standard calculations (+,-,*,/) as well as Boolean operations between two operands. An extension of this command has been added which allows the same calculations to be performed at the BYTE level on a MASS of data. If the MASS function is selected, numbers entered as Dec or Hex will be interpreted as a constant value. Numbers preceded by a '.' will be interpreted as an actual memory location and numbers preceded by a = sign will be read as offsets into the transfer buffer. The user will be required to specify the number of bytes to

perform the arithmetic on as well as the area in memory he wishes to store the results. Use this option with care. A sample session is shown below.

```
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?R

 OP: (+,-,*,/,And,Or,Xor,Not)[-]? +
 MASS OPTION[N]?N
 FIRST OPERAND? $123
 SECOND OPERAND? 33
 HEX: $123 + $21  = $Ø144
 DEC: 291  + 33   = 324
 OP: (+,-,*,/,A-AND,O-OR,X-XOR,N-NOT)[-]?
```

COMMAND-S:

This command allows the user to select what drives will be used to read and write data from. The system will default to read and write from drive 1 unless told otherwise. A sample session is shown below.

```
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?S

 DRIVE # FOR READING DATA? 1
 DRIVE # FOR WRITING DATA? 2
```

COMMAND-T:

This command is much like the Copy command, except that now copying is performed from disk to memory or memory to disk. This command is used primarily to load an entire program or game into the Diskedit Transfer buffer for editting using the P command or Search and Disassembly using the D command. Data can be loaded or copied from any part of the buffer. This command does purge the map buffer, so be sure you have a good copy of your disk map before transferring or copying data. A sample execution printout follows.

```
S# (OR A,C,D,E,M,R,S,T,V,?,<,>)?T

 # BYTES INTO BUFFER[Ø]?Ø             ---# of bytes offset
 TRANSFER DISK TO MEM[Y]?Y
 READ 128 BYTES/SECTOR (Y/N)[Y]?Y
 START SECTOR[1]?3
 END SECTOR[15Ø]?14Ø
 INSERT SOURCE DISK, RETURN WHEN READY?
```

Note that the 128/125 byte readin option becomes very useful now, since binary load files contain only 125 bytes of object code. By specifying N in response to the prompt, the user can obtain a continuous disassembly listing (assuming the file is contiguous). See NOTES for more info.

COMMAND-V:

This command lets you get a formatted printout of the disk map. You can choose to print to either the screen or the printer. The character codes used are described under the M option. A sample printout is shown below.

```
ØØØØØØØØØ111111111122222222223
123456789Ø123456789Ø123456789Ø
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx<3Ø
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx<6Ø
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx<9Ø
xxxxxxxxx?????xx?????xxxxxxxxx<12Ø
xxxxxxxxxxxxxxxxxxxxxxxbxxbxxx<15Ø
xxxxxxxxxxxxxxxxxxxxxxeeexxxxx<18Ø
xxxxxeeeeeeeeeeeeeeeeeeeeeeeee<21Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<24Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<27Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<3ØØ
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<33Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeex<36Ø
xeeeeeeeeeeeeeeeeeeeeeeeeeeeee<39Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<42Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<45Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<48Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<51Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<54Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<57Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<6ØØ
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<63Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<66Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<69Ø
eeeeeeeeeeeeeeeeeeeeeeeeeeeeee<72Ø
```

Fig. 5: Disk Map

COMMAND-W:

This command will write out the current sector in memory to any user specified sector.

COMMAND-C/R:

A carriage return at the menu level will be interpretted as a request to read the next DOS sector. This command is only of use when you are tracing your way through a DOS created program or data file. Diskedit recognizes the next sector pointer (calculated from the last 3 bytes in a sector), but the info is useless on boot (non-DOS) sectors (see NOTES).

COMMAND-'>':

Use this command to read the next sector (i.e. if you are currently viewing sector 1Ø1, 1Ø2 will now be displayed.)

COMMAND-'<':

A less than sign will read the current sector # less one into memory (i.e. if you are currently viewing sector 101, then 100 will now be displayed.)

COMMAND-n:

Any number entered at the menu level will be interpretted as a request to read that sector. The number may be either DECimal or HEXidecimal. Hexidecimal numbers must be prefixed with a dollar sign. The allowable range of sectors is 1 to 720.

COMMAND-?:

This command reprints the list of options shown at boot time and performs an RPM test of the source disk for you. A good speed to keep your drive at is 288 +/- 2 rpm.

SUMMARY

This version of DISKEDIT has been tested for a variety of user responses and should never abort with an error message. If you ever see a BASIC error pop up and the program stop, feel free to write me a nasty letter. It is hard to take all possible situations into account, but hopefully I have made DISKEDIT foolproof. DO NOT write me if you make changes to the program and then have errors occur. For your reference, I suggest listing the program when you get it and then disassembling the assembly code routines (just hit break after the program boots up and find the starting addresses of the STRINGS; then GOTO 230 and use the D option). The program is modular (i.e. the main loop is in lines 0-1000, the copy routine is in lines 4000-5000, etc.)and fairly easy to understand.

NOTES:

There are only a few fundamental concepts to remember while working with the ATARI 810 disk system. First, a sector contains 128 bytes of information. Certain sectors, however, contain less actual data. For example, the very first sector read on a disk when the system is brought up, sector 1, contains 6 header bytes of information that tell the system:

1) Whether to silence the loading sounds (byte 0=$FF for silent load, else 00).
2) How many sectors to load in initially (byte 1).
3) Where in memory to start the load process at (bytes 3 and 2).
4) Where in memory to set the DOS init address (bytes 5 and 4).

After the initial number of sectors are loaded, control will pass to the instructions starting at byte 6 of the first sector (i.e. the load address+6). Also important to remember is that binary load files are stored as 125 bytes of information with a 3 byte pointer attached at the end. The pointer points to the next sector in the file and also contains a checksum for the file # and # of bytes in the sector. This is the reason I included the optional 125 byte readin capability. If you have a sequential, contiguous binary load file and move several sectors of 128 bytes each into memory for disassembly, junk would appear in the disassembly every 125 bytes as the disassembler would try to disassemble the pointer. The 125 byte readin will chop off the pointer as it is read into memory, giving you a continuous disassembly. Incidentally, the 125 byte readin is also useful if you wish to convert a binary load file to a boot file using the Copy command.

For those of you looking at DOS disks, the only other bit of information I can add is to look at sectors 360, 361, and 362 (362-368 if you have a lot of files.) Sector 360 is the disk map sector that contains all of the information about which sectors of the disk are free and which have already been used for file storage. This information is coded (a zero bit indicates a used sector and a 1 bit indicates an available sector), so I would recommend studying the ATARI operating system manual before trying to modify this sector. Sectors 361 to 367 are the sectors I usually work with. These sectors contain all of the file names that are stored on the disk, their size in sectors, and their starting sector number. Each file takes up 16 bytes (2 lines) of these sectors. The first byte is a status byte. If it is equal to an ASCII b then the file is active, otherwise the file is a deleted or 'accidentally lost' file. The next two bytes contain the size of the file and the following two bytes contain the start sector number. Finally, the last 11 bytes contain the name of the file. These sectors become very important when you lose a file by opening it for output and then turn off the machine, reboot, or hang the system. By simply editting the first of the sixteen bytes of the appropriate line, you can recover your data file and copy it to another diskette.

For the beginning user, I would suggest looking at these sectors on a DOS disk and experiment with tracing your way through some files.

BACKING UP YOUR BOOT DISKS

By popular demand of many DISKEDIT users who wish to back up copy-protected disks for personal backup protection, I have added this section for those unfamiliar with disk protection methods. Please check all applicable copyright laws before you copy any software (Soft Unlimited can not be held responsible for illicit use of the software.) There are many schemes on the market to copy-protect disks, but there is no unbreakable copy protection mechanism available yet for ATARI disks. This is because the computer always has to be able to read sector 1 on a disk and this will tell the computer (and you) what to do next. With sufficient patience any disk can be backed up.

The most common scheme used by manufacturers is to format a disk initially with bad sectors and then have the boot program check to insure that these sectors are indeed bad (remember those disks that go clunk?). These checks are easily found by disassembling the program and then simulating a bad check by editting the code. In the example below, assume that we have already determined that sector 258 is bad by doing a disk map.

```
Original Source            Modified Source
  LDA $Ø2                     LDA $Ø2
  STA $Ø3ØA                   STA $Ø3ØA
  LDA $Ø1                     LDA $86
  STA $Ø3ØB                   STA $Ø3Ø3
  JSR $E453                   NOP,NOP,TAY
```

Putting a negative number ($86) into location $3Ø3, the accumulator, and the Y register and NOPing out the JSR to the DISKIO routines will allow most programs to be backed up, and without those noisy sector checks. You can create bad sectors by slowing your disk speed down prior to writing a sector, but I don't recommend this procedure. If you are patient and the problem is just a bad sector check, you will find it.

Other things to keep in mind include: 1) the sequence of sectors in the original disk may allow faster read-in; hence you may need to null out timing checks, 2) multiple sectors with the same number may exist (i.e. the second time you read sector 17, you may get different data than the first time); hence you will have to be careful copying the two sectors surrounding bad (missing) sectors, and 3) don't forget that the program may check itself to see if you modified it.

In summary, the sequence of steps you must take to backup a disk include: 1) map the disk to find data and bad sector locations, 2) copy all sectors with data to a blank disk, 3) look at sector 1 and determine how many sectors are loaded and where in memory they are stored, 4) find and fix any bad sector checks by viewing sectors, disassembling, editting, and writing, and 5) cross your fingers and try booting the disk. Don't be afraid to go in and modify sections of code in DISKEDIT for special backup requirements you may have. By putting a loop in the copy routine code (4ØØØ-5ØØØ), you could copy all odd numbered sectors from on disk to another. By placing calls to the BOOLean routine in a loop, you could mix entire sections of code together (a common data scrambling technique). You should never find yourself doing something repetitive with DISKEDIT.

Most Asked Questions
****

In this section I will attempt to answer any questions people ask the most about DISKEDIT, disk backup, etc.

Q) Can I write bad sectors w/o modifying my 810?

A) No. The Atari 810 handles all of the formatting and track I/O on diskettes, so the only way to write bad sectors, duplicate sectors, or checksum errors is by installing a new ROM on your 810 (i.e. you provide new commands for your 810 processor.) Slowing your disk drive down (prior to writing a sector) will create a bad sector, but this will not back up most of the games on the market today. For your interest, a new 4K bank select EPROM will be available through Soft Unlimited in March, 1983. This unit has already been tested successfully on the best protected games on the market-no problems. The EPROM and software will go for around $75.00 (excellent when compared with other products of lesser capability on the market today.)

Q) What are bad sectors?

First I should state that bad sectors are a generic term used for any sector that can't be read on the 810 disk drive. To best answer the question, I must address bad sectors, duplicate sectors, and what I tend to call Sort-Of-Bad (SOB) sectors. I will have to explain this in laymans terms, since that is the way I think anyway. More detailed information can be gleaned from the Atari Technical User Notes (which have been signigicantly improved in the last release by the way-there is actually an index with page numbers!)

The best way to think of a sector is that it is a small area on the disk that contains 128 bytes of user information and several bytes of disk informattion. The disk info is used by the disk drive to tell what sector number it is looking at and to checksum the data. A Checksum is a simple means of quickly checking to see if data has been accidentally modified. By adding together all of the bytes of data in a sector and then comparing this value against the checksum value (calculated when the data was stored), the disk drive can tell whether data has gotten scrammbled. This reading and writing of the disk information is completely controlled by the 810's microprocessor and ROM instructions.

A bad sector (found on disks that go clunk when they boot up) is simply a sector that doesnt exist on the disk. It was either not formatted in the first place or was formatted, but the sector number info was changed so that the 810 thinks it is a different sector number. In any case, the 810 will not find the sector on the disk and after scanning back and forth a bit trying to find the sector (that is what the clunk is) it will return an error code to the computer. A game program will be set up to expect this code to be returned and if it doesn't see it, the program will usually crash, go out and format the copied disk, or something else equally as nasty.

Duplicate sectors are sectors created by renaming 2 or more consecutive sectors on a disk to the same number. If a disk has double sectors on it, the game will expect to be able to read two sector 18s for example, and obtain different data each time it reads the sector. Games that use this protection scheme usually load in quiet (no clunk), but can be easily identified when a copy of the game starts loading in fine, but gets to one point and then just seems to read the same sector over and over. SOB sectors are usually the trickiest to work with, because they have the characteristic of a quiet read-in, but the computer is told the sector is bad. Also, data is actually read in to the computers memory. This type of sector is originally created with a bad checksum value; hence a checksum error is returned even though the data is good. The game program will be expecting both the data as well as the checksum error.

I hope that this information has been of some use. Please keep in mind that there is no way to create these types of bad or duplicate sectors through software. A hardware modification of the 810 is required (replacing the microprocessors' instruction set) to do so. Some companies claim that their software will write bad sectors, but they will require you to pull out a screwdriver and fiddle with your disk drive speed to write a sector slower or faster than normal. This sometimes works but only on some peoples' drives and only for games that expect missing (bad) sectors. Two recently marketed programs, that purportedly clone or replicate a disk, operate using this feature (so the English tell me,) so be warned.

Q) How can I back up disks protected in such ways?

A) First, you need a program like DISKEDIT to allow you to get in and look at the code on a disk. Also, a program like ULTRACOPY will prove invaluable in performing the large backups of entire disks and also the specialized copying around bad sectors that can get tiring if you have one disk drive. DISKEDIT does have a sector copier built in, but you can only copy 150 sectors at a time (between swaps of the source and destination disks) unless you have two disk drives. I should also caution against trying to sell copied software or selling the originals and keeping the copies. This is clearly against the law. The only reason to copy a game is for backup protection in case you destroy the original accidentally.

Before going into the ways to get around copy protection mechanisms, I will give a brief description of how the ATARI reads and writes data to and from a disk. After looking through my notes on my presentation to the SPACE group on this subject I think I had best start at the beginning. An ATARI 810 diskette (when formatted) is composed of 720 sectors of 128 bytes each. The sectors are numbered from 1 to 720 and are arranged on the diskette in groups of 18. Each group of 18 is called a track and is best thought of as a ring of sectors on the diskette that can be accessed by the head with minimal movement. Movement from track to track can be heard when reading or formatting diskettes (unless you have greased your rails recently.)

The ATARI performs Disk I/O in much the same manner that it performs I/O to other peripherals. Basically, several memory locations are reserved in memory as pointers. To perform a disk operation, a program must do a jump subroutine into the OS at a particular point (JSR $E453 or JSR $E459; JSR $E453 sets up some pointers and then jumps into $E459) and the OS will inspect these memory locations to decide what has to be done. The memory locations tell the OS what sector to read/write, where in memory to read/write the data from/to, what operation to perform (i.e. read/write/format), what disk to do the operation to, and how many bytes to transfer (128 for the ATARI 810, 256 for a double density drive.) The memory locations are as follows: $301-disk drive number (0-3), $302-disk operation ($52=read,$21=format disk, $57=write), $304&$305-memory location for data I/O, $30A&$30B-Sector number, and if JSR $E459 is used then $308&$309 must contain the number of bytes to transfer and $303 must be set up with $80 for a write operation and $40 for a read operation ($303, $308, and $309 are set up by JSR $E453). The OS will return a value in memory location $303 and the Y register to indicate the successfulness of the last disk operation. A successful operation will return a 1, otherwise a negative number >$7F is returned. Knowing just this much, it is possible for most people to back up game disks that are protected with simple bad sector checks. Using a disk edittor, you can find out which sectors are bad and then disassemble the program to find out where it stores a bad sector pointer into memory locations $30A and $30B. The following JSR $E453 or JSR $E459 can then be no-oped out or converted to a LDA $80, TAY to simulate a bad sector read. Be warned that some games do a checksum of the sector containing the bad sector check (to see if you changed the code), but this can usually be sidestepped by switching the values that are stored into $30A and $30B instead of no-oping the JSR out (i.e. if the original bad sector was $30A=$0 and $30B=$13, setting $30A to $13 and $30B to $0 will tell the OS to try to read sector $1300 and return an error code since that sector doesn't exist.) Note that the ATARI reads two byte addresses second byte first. As I indicated above, there are a variety of ways to change the arrangement of sectors on a disk, and a game program will be written in such a way that it expects to see just such an arrangement (i.e. a specific bad sector, double sector, or a checksum sector.)

Disks that are protected with double sectoring (i.e. two sectors with the same number) are a little more difficult to back up. Usually, these are the disks that read in silently. The only way to find a double sector on a disk is to locate the bad(missing) sectors and then search the sectors that are adjacent to the missing ones (i.e. a missing one is just one that has been renumbered to become a double one). To identify a double sector, you must read in the suspect sector, note the first couple of bytes in the sector, and then read the sector again before the disk drive powers down (you have to do this so that the heads will be positioned correctly). Compare the bytes that you saw before with the new ones and you can usually see a double sector fairly easily. Once you find a double sector, do a sector copy of the disk onto a blank disk and then read the double sector until you get the 'second' sector and write this on the destination disk in the sector that was bad on the original disk. I

must stress here never to write anything out to your original disk. Once you have copied all of the data including double sectors over to the backup disk, you must go into the code and determine where the game is reading in the double sector. When you find this spot, you can usually just increment the sector pointer ($30A,$30B) by editting the code (i.e. change where it tries to read the same sector twice) and you will be all set. Some games scramble their code, but most disk editters have boolean unscrambling capability, so this is no problem. If you don't let the thought of assembly code overwhelm you, you will soon find that backing up software is relatively easy and quite interesting if you take time out to look at the actual game code. You will also discover that many of the more recent games are protected in quite complex, but surprisingly simple to break methods.

After talking with the author of a program similar to DISKEDIT (DISKASM), I have to say that I agree with his comment that the cost being added to software for protection schemes is getting ridiculous. Maybe if everyone becomes conversant in backing up their own software, the developers will just stop protecting the software and drop costs down to a reasonable price range. These aren't the days when there were just 10,000 ATARI computers out there and a high cost was justified based on percentage sales. Any comments from anyone on this subject?

Q) How come you sell DISKEDIT and ULTRACOPY so cheap?

A) Two main reasons. First, I think the potential market for each is big enough that I don't need to price it as if the product were only going to sell for a couple of months. I wrote DISKEDIT and ULTRACOPY in such a manner that they will continue to evolve and remain a useful tool for my own use and hopefully for other users. Second, I don't have any overhead from trying to copy protect the software and plastering the magazines with flashy ads. Occassionally I will advertize, but most of my sales will remain word of mouth. If a product is good, it will sell itself.

Q) What good is this program if I don't know assembly?

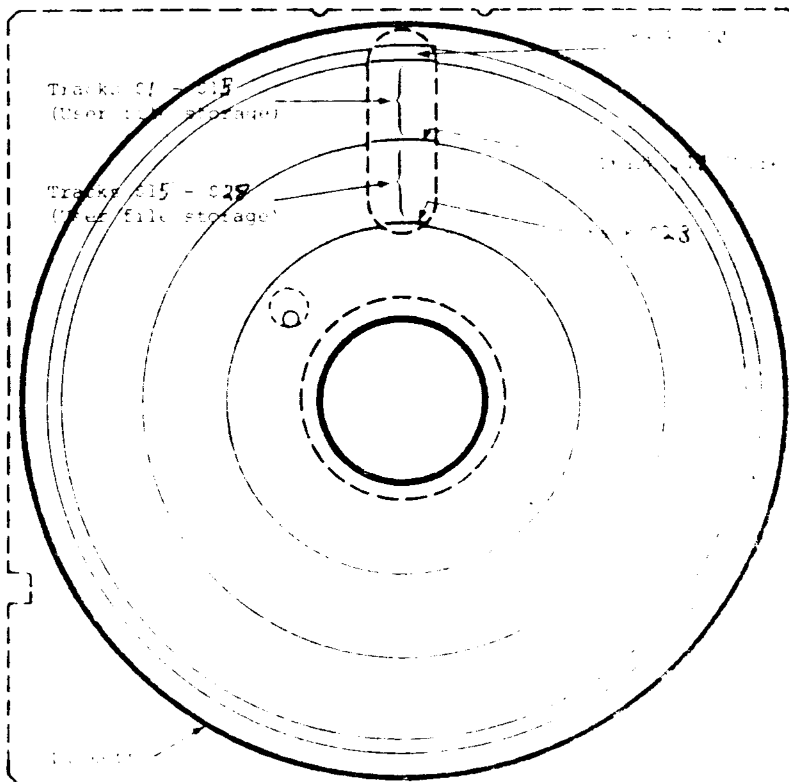
A) The only reason it helps to know assembly when using diskedit is if you are intent on backing up the more complicated games. I originally developed the software to help me learn more about the disk and to recover lost disk files and games. The assembly experience will come with time if you just use the disassembler on some games and study the printouts. Don't be afraid to ask your friends why things are getting stored to certain addresses. It also will help if you get an ATARI OS Users manual to use as reference (don't try to read it, just look up what you need as you need it.) When you reach the point where you start taking long printouts to bed with you for light reading, you can call yourself an experienced hacker and take a loooong break from your ATARI.

ADDENDUM A-DISK STORAGE

This is the first of hopefully many addendums that will address the fundamentals of the DISK system as well as some more advanced concepts. I have decided to devote the first section to disk storage, since this subject is glossed over in most documentation I have seen available to date. If you have further questions about the Disk system after reading this material, I strongly recommend that you obtain the ATARI technical user notes for reference.

DISK STRUCTURE:

A diskette that has been formatted on an ATARI 81Ø disk drive contains 4Ø tracks. Each track circles the disk in the form of a ring and is composed of 18 sectors. Each sector is composed of 128 bytes of information. A disk can contain, therefor, 4Ø*18*128=9216Ø bytes of information (a byte can be thought of as one character representing a value between Ø and 255-ØØ to FF hex). The diagram to the right depicts this structure in detail.



Note that this will be the last I talk about tracks. The disk structure is best thought of as having 72Ø sectors (4Ø*18), since this is the way they are addressed by the ATARI computer. Because of a discrepancy in the ATARI OS, we will think of these sectors as being numbered 1 to 72Ø, even though ATARI DOS can only read sectors 1 to 719.

SECTOR TYPES:

I have discussed before the types of bad sectors that exist, so I will now describe the types of 'good' sectors that you will encounter when looking at a disk. Aside from empty sectors (sectors containing all zeroes), there are five main sector types-autoboot (BOOT), DOS file (FILE), non DOS file (NOTF), volume table of contents (VTOC), and directory (DIR) sectors.

A boot sector is a sector that is automatically loaded when the computer is turned on. All DOS or game disks have one or more boot sectors in a contiguous block of sectors beginning with sector 1 (see INFO on page 1Ø.) In DOSII disks, sectors 1-3 are the boot sectors (these in turn load in the DOS.SYS binary program and execute it.) In non DOS (i.e. game) disks, all that may autoload in is sector 1 (which in turn loads in the rest of the game), or all of the game may

load in as part of the boot.

A NOTF sector is any sector on a disk that is accessed directly by sector number via the disk handler (via a call to the OS through $E453 for example). In other words, this type of sector only exists on a game boot disk or on a non standard DOS disk, since DOS can not be used to read or even create it.

The final 3 sector types (DOSF, DIR, and VTOC) are found on DOS disks. The DOSF file is the file that contains your programs and databases. DOS stores files on a disk in such a manner that it can always look at any sector of a file and determine which file the sector belongs to and what sector is next in the file chain. This capability is not without some cost however, since not only data must be stored on a DOSF sector, but also some bookkeeping overhead is required. A DOSF sector thus will contain only 125 bytes of user information followed by 3 bytes that will indicate the file index #, the next sector # in the chain, and the # of usable bytes in the sector. DOS keeps track of which files are stored where by using a set of sectors called the directory. Each DIR sector contains 8 index locations for file information (16 bytes/file). Each location contains information on the name of the file (bytes 6 to 16), the starting sector number of the file (bytes 4 and 5), the size of the file (bytes 2 and 3), and the status of the file (the first byte). Eight sectors are reserved on a DOS disk for the directory info (sectors 361 to 368). Finally, the VTOC (volume table of contents) sector is the sector that contains information regarding which sectors on a disk are used and how many are free.

All three of the DOS file types are used for any DOS file I/O operation. For example, lets say that you want to store a BASIC program to disk. When you do a SAVE of the program, DOS assigns the file a location in the first available location in the directory sectors. DOS then scans the VTOC directory to find the first available sector to start storing the data to. This sector number is placed in the file directory sector (bytes 2 and 3) and then 125 bytes of your file is transferred to this sector along with a trailing 3 bytes. DOS then updates the VTOC and continues the scan, transfer, and update process until all of the file is transferred. When done, the file status byte is updated in the directory sector to indicate that the file is good and closed.

FINDING FILES:

Now that you know something about how the disk is structured, the only thing left is to go through a quick example of how to find a file on a disk. The assumptions made in this example are that you have just SAVEd a BASIC program to a newly formatted disk which has DOS and DUP written to it. If you look at sector 361, you will see that your file is the third entry in the directory table and that it starts at sector number 85 (byte 4 at the file location equals 85). If you look at sector 85 now, you will see the start of your program (in tokenized form of course.)

In the following sample cases, I hope to give you a better understanding of the differences between standard DOS disks and most autoboot game disks.

SAMPLE #1: A newly formatted DOS II disk, with DOS&DUP.

SECTOR #	TYPE	DESCRIPTION/FUNCTION
1	BOOT	Tells computer to autoread sectors 1-3 into mem @ $700
2	BOOT	Once this three sector boot program is put into memory,
3	BOOT	the computer begins executing it at $706
4	FILE	DOS.SYS. This is loaded and executed by the boot program.
.	FILE	DOS provides the user the ability
42	FILE	to access (SAVE,LOAD,OPEN,etc) disk files.
43	FILE	DUP.SYS. This gives the user file deletion, copying,
.	FILE	and other file managment functions.
84	FILE	
85	FILE	START of available file space for user programs.
.	FILE	The first file stored on a disk after writing DOS&DUP
359	FILE	will be stored at the lowest sector number available.
360	VTOC	Tell DOS what sectors are available for storage.
361	DIR	These 8 sectors tell DOS where each file
.	DIR	is stored. Each sector contains 8 file entries
368	DIR	of 16 bytes each (giving you 64 file locs total).
.	FILE	More file storage space.
720	FILE	This sector is accessible only via the DISK handler.

SAMPLE #2:A boot game-non DOS.

SECTOR #	TYPE	DESCRIPTION/FUNCTION
1	BOOT	Tells the computer to boot the next 6 sectors.
2	BOOT	This is the continuation of the boot routine.
3	BOOT	Once loaded, this routine will begin executing
4	BOOT	and load sectors 257-400 into memory. Execution
5	BOOT	will then pass to the game code. Note that
6	BOOT	these 6 sectors may contain a bad sector check.
7-255	----	Empty sectors.
257	NOTF	This is the game code.
.	NOTF	Note that these sectors may also
400	NOTF	contain a bad sector check.
401-450	----	More empty sectors.
450-720	----	Bad (unformatted) sectors.

Assume that sector 1 used in case 2 contained the following first six bytes: byte 1=0 (normal noisy load), byte 2=6 (load 6 sectors), byte 3=0 and byte 4=6 (start loading into memory at address $600), and bytes 5=6 & 6=6 (set the system reset vector to address $606). This is just for example purposes only. It is best to just look at a game disk and play around with it.