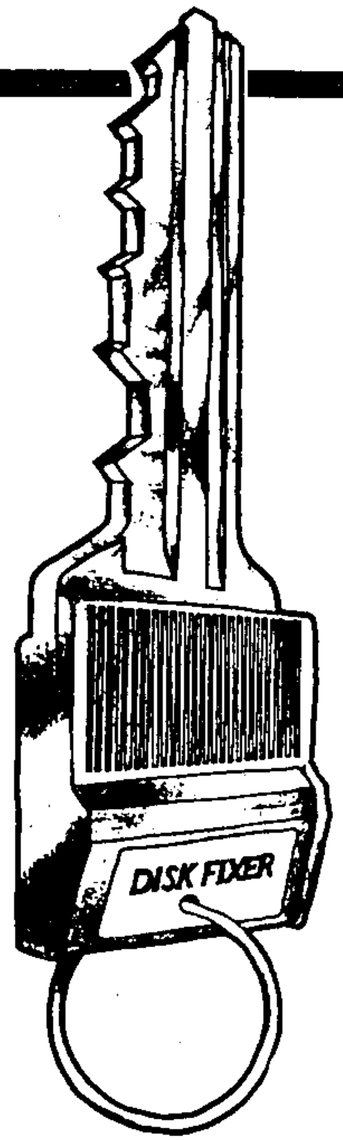
Navarone Industries, Inc.
510 Lawrence Expressway, #800
Sunnyvale, CA 94086

DISK FIXER

Unlocks the secrets of the disk and lets you access hidden or "lost" information.



Includes
HIDDEN POWERS
by Bill Gronos

OPERATING INSTRUCTIONS

The DISK FIXER is one of the most powerful utilities available for the TI-99/4A home computer. With DISK FIXER, you can access Floppy disks by sector rather than by FILE NAME. You can now display or print the actual "binary" contents of any sector of a floppy with a single command.

Other features of the DISK FIXER allow you to change any byte on any sector, or move data from one sector to another.

This program is ideal for fixing "blown directories," improperly closed files, and recovering data from diskettes otherwise inaccessable.

*** * * WARNING * * ***

THIS PROGRAM HAS THE CAPABILITY OF DESTROYING DATA ON YOUR DISKETTE IF IMPROPERLY USED.

OPERATING INSTRUCTIONS

1. Insert cartridge into "Game Port" on the console then select "DISK FIXER" from main menu screen.

2. The Disk Fixer will load and begin execution and display the banner:

*** * DISK FIXER VER 2.0 * ***
(C) Copyright 1983 By
NAVARONE INDUSTRIES

SELECT OPTION
R - READ
P - PRINT
W - WRITE
F - FIND STRING
A - ALTER DATA
D - DISPLAY BUFFER
C - CURRENT SECTOR
M - INSPECT RAM
H - HELP
Q - QUIT

3. Select the option by entering the appropriate command, and press "enter." If you enter the command incorrectly, the message, INVALID COMMAND will be displayed and you may then re-enter the command.

READ SECTOR

To **"read"** a sector of data, enter the following command

R sss,d

where: sss = The actual sector address you wish to read. If you do not enter this parameter, the program will use the last sector address entered plus one. This can be used to read sequentially through the diskette.

d = The disk drive device code. If you omit this parameter, the program will use the previously entered device code.

WRITE SECTOR

To **"WRITE"** a sector of data, enter the following command. This command should be used with caution as it will write the buffer contents into the sector defined. If no sector address is entered, the data will be written into the **"CURRENT"** sector location.

CAUTION: Do not use the "W" option immediately following a "P" rint function without specifying the entire syntax. The current sector pointer is always incremented after a print function and the use of a "W" without a sector address will cause the current buffer to be written on the next sector. So don't do it!

W sss,d

Where: sss = The actual sector address you wish to read. If you do not enter this parameter, the program will use the last sector address entered.

d = The disk drive device code. If you omit this parameter, the program will use the previously entered device code.

ALTER DATA

This command is used to alter or "change" the data in the buffer. The Disk fixer reads data from the diskette into an internal memory buffer. This buffer can be written back out to the same sector it was read from by just entering a **W** command. Use the **ALTER** command to change the contents of the buffer before writing it back.

A oooo

Where: oooo = The address of the data to alter. This is a relative address, and as displayed by the **D** command.

After changing the contents of the buffer, you can use the **D** command to display the buffer on the screen. The **PRINT** command cannot be used because it performs a read from the diskette which will overlay any change you have just made.

DISPLAY BUFFER

This command will display the current memory buffer on the screen. No reading or writing will occur with the use of the command.

D

INSPECT/CHANGE

This command is identical to the **M** command in the TI-**DEBUGGER.** This option allows you to inspect or change any **RAM** location in the system.

M oooo

where: oooo = The address of the data to inspect. This must be an **ABSOLUTE** address. You can display the contents of **VDP** memory by using a **"V"** after the address.

PRINT SECTOR

To **"PRINT"** a sector of data, enter the following command. You may optionally print a consecutive series of sectors with this command.

P sss,d,n

where: sss = The actual sector address you wish to read. If you do not enter this parameter, the program will use the last sector address entered plus one. This can be used to read sequentially through the diskette.

d = The disk drive device code. If you omit this parameter, the program will use the previously entered device code.

n = The number of consecutive sectors to print.

A message will request you to enter the **LIST DEVICE:**
Enter the device to list the data. The default is
RS232.BA = 9600

HELP

Use this command to display the options on the screen.

H

QUIT

This command causes **DISK FIXER** to return to the editor assembler.

FIND CHARACTER STRING

Use this command to locate the sector address of a character string. Enter the command as follows:

F SSS, EEE, d

Where SSS is the Starting Sector for the Search,

Where EEE is ending sector of the Search,

and d is the disk drive number.

The Prompt:

ENTER CHARACTER STRING

Enter up to 30 characters. The Disk Fixer will search between the sector limits specified on the Disk (d) and display the Sector address of the First occurance of the character string entered.

If the Disk Fixer cannot find the character string it will display the message:

STRING NOT FOUND - SECTOR ONNN (Sector number indicates Last Sector Searched).

The Disk Fixer will allow you to change any byte on any sector, move data from one sector to another, and, one of the most powerful options, will search for a character string and DISPLAY the sector address of where it found the data.

THE HIDDEN POWERS OF DISK FIXER

by Bill Gronos

* INDEX *

FORWARD

I'm sure few of you Disk Fixer owners really know the power you possess when you plug that module into your console.

And that is precisely the purpose of this booklet, to share the secrets of that power with you. To spare you many hours of labor spent unlocking those secrets so that you may spend the time putting the power to work.

I have written this booklet for the novice user who knows little or nothing about the inner workings of either the disk controller or the 99/4 console. I will give you step by step procedures that will allow you to do some pretty amazing things. The Disk Fixer is a magician's wand. I will show you how this wand can do many tricks. Some of these tricks have important programming applications, such as making some lines of a BASIC program invisible or preventing a program from being altered. Other tricks are designed to amuse you and to mystify your friends.

Also, I will share some techniques to aid you in performing the delicate surgical procedure that gives this unique module its name—a mini medical school for the treatment of sick disks.

Throughout this booklet I have made references to hexadecimal number notation (base 16). This was absolutely necessary because we will be dealing with the internal structure of disks and programs and hexadecimal is the language spoken in these realms. Your Disk Fixer program speaks the same language, but your Disk Manager converses in standard decimal (base 10) numbers. This is a little confusing and a giant pain in the posterior. For example, when you use the Disk Manager to do a disk test, the bad sectors will be given in decimal notation. You have to convert these values to hexadecimal when you use them with Disk Fixer. Hexadecimal numbers will be preceded by a "greater than" symbol. Decimal 10 is >A, etc.

All the examples of Basic programs in this booklet are written for TI Basic (the language supplied with the console) and most of them will work with Extended Basic. However, be aware that some format changes do exist and may cause results to vary.

I have tried not to bore you by being overly technical, though a treatise of this nature by necessity requires a certain amount of detail. Let's begin slowly so that you may gain confidence in using this module; let's start by having a little fun with Basic programs.

1. USING DISK FIXER TO ENHANCE BASIC PROGRAMS

One of the most valuable uses for Disk Fixer is to radically change Basic programs as they are stored on disks. The modified programs have properties that are both interesting, useful and impossible to duplicate through normal use of the Basic Language Editor.

When the modified programs are loaded into the console via the OLD command, the changes are not detected by the Basic Editor. Incorrect Basic lines will produce errors when the programs are run, but many changes will execute correctly and produce beneficial results. Once these special programs are in console memory, they can be resaved to disk or tape. Thus users without disk drives can be mystified and benefited by your newly acquired powers. We'll start with a simple example to illustrate this technique.

1-1 CREATING 'ILLEGAL' LINE NUMBERS

Did you ever try entering a Basic Statement with line number 0 or 60000? If you did, you got an error. This error checking routine is active only when you are creating or editing a program in the normal way. Line numbers are not checked when programs are Run or Saved. Therefore, illegal line numbers created with Disk Fixer will be accepted by the Language Interpreter. Let's make a simple change to illustrate this point:

1. Power-up your computer in TI Basic and type in this line:
   1 PRINT "0 IS AN ILLEGAL LINE NUMBER"

DEMO1

2. Using a blank, initialized disk, save the program with the command:
   SAVE DSK1.DEMO1

3. Exit Basic by typing BYE, and insert your Disk Fixer module. Select DISK FIXER from the main menu.

4. With the disk containing DEMO1 in drive 1, read the sector containing the test program by entering the command:
   R 22,1 [ENTER]

5. Next, press enter to return to the menu and bring up the line number to be altered with the command :
   A 8 [ENTER]

6.Change the line number to zero by typing in:
   0 [ENTER]

7. Save the altered sector back to disk with the Write command:
W [ENTER]

8.We're done, so exit Disk Fixer by typing :
Q [ENTER]

FIGURES 1 AND 2

```
NAVARONE IND. *** DISK FIXER V2.0 ** SECTOR DUMP     SECTOR ADDRESS  0022
ADDR =  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  INTERPRETED

0000 = 0003 37B7 37B4 37D7 0001 37B9 1F9C C71B  **77747W**79**G*
0010 = 3020 4953 2041 4E20 494C 4C45 4741 4C20  0 IS AN ILLEGAL
0020 = 4C49 4E45 204E 554D 4245 5200 AA3F FF11  LINE NUMBER**?**
0030 = 0300 0000 0200 03C4 454D 4F31 2020 2020  *******DEMO1
0040 = 2000 0000 0000 0100 0000 0000 0000 0000   ****************
0050 = 0000 0022 0000 0000 0000 0000 0000 0000  ***"************
0060 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0070 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0080 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0090 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00A0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00B0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00C0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00D0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00E0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00F0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
```

```
NAVARONE IND. *** DISK FIXER V2.0 ** SECTOR DUMP     SECTOR ADDRESS  0022
ADDR =  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  INTERPRETED

0000 = 0003 37B7 37B4 37D7 0000 37B9 1F9C C71B  **77747W**79**G*
0010 = 3020 4953 2041 4E20 494C 4C45 4741 4C20  0 IS AN ILLEGAL
0020 = 4C49 4E45 204E 554D 4245 5200 AA3F FF11  LINE NUMBER**?**
0030 = 0300 0000 0200 03C4 454D 4F31 2020 2020  *******DEMO1
0040 = 2000 0000 0000 0100 0000 0000 0000 0000   ****************
0050 = 0000 0022 0000 0000 0000 0000 0000 0000  ***"************
0060 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0070 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0080 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0090 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00A0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00B0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00C0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00D0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00E0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00F0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
```


I'm sorry if this detailed, step-by-step process offends any of you who are old hands at using Disk Fixer, but I wanted to be sure that users who have just inserted their modules into the console for the very first time haven't been left by the wayside. It would have been far simpler to say:

READ SECTOR >22, ALTER THE VALUE AT WORD >0008 to >0000, WRITE THE SECTOR BACK TO DISK AND EXIT.

It would have been even more concise to simply say:

CHANGE WORD >0008 OF SECTOR >22 to >0000

I won't go into as much detail on disk changes as I did here. When I say to alter a byte or a word at a certain disk location, you will need to read the sector, alter the appropriate values and save the changed sector back to the disk. If you are ever confused, refer back to the detailed instructions in this section.

Do you all know why I said to use an empty disk in step two? The first program saved to a "clean" disk will always begin at sector >22. Since we know what sector the program would be saved to, we don't have to hunt through the File Directories looking for the file name DEMO1. Now let's take a look at the results of our alteration.

Return to Basic and load the altered program with the command OLD DSK1.DEMO1. and list it. Instead of 1 PRINT "TEST" you have 0 PRINT "TEST". Try erasing or editing the line and Basic will refuse to do it. Giving you a BAD LINE NUMBER error message. However, the program will still run correctly!

Does this mean we have found a way to keep others from deleting lines from our programs? No. That line 0 can be deleted if we first RESequence the program. Type in RES and hit enter, list the program and you will see that line 0 is now 100 and can be edited or deleted. Therefore,changing line numbers into illegal values isn't enough to protect them from tampering. We must find a way to keep programs from being resequenced--Disk Fixer can do it! But before we see how this can be done, let me explain the memory format of a Basic program.

1-2 HOW BASIC PROGRAMS ARE STORED IN CONSOLE MEMORY

Figure 3 shows the contents of the sector that holds the Basic program, 1 PRINT "TEST".

FIGURE 3

```
NAVARONE IND. *** DISK FIXER V2.0 ** SECTOR DUMP     SECTOR ADDRESS  0023
ADDR =  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  INTERPRETED

0000 = 0005 37CE 37CB 37D7 0001 37D0 089C C704  **7N7K7W**7P**G*
0010 = 5445 5354 00AA 3FFF 1103 0000 0003 0003  TEST**?*********
0020 = C445 4D4F 3220 2020 2020 0000 0000 0001  DEMO2       ******
0030 = 0000 0000 0000 0000 0000 0000 2300 0000  ************#***
0040 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0050 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0060 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0070 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0080 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0090 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00A0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00B0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00C0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00D0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00E0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00F0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
```

By analyzing several short programs, I was able to figure out what all that garbage means. This process is called "hacking" and, since Texas Instruments treated all knowledge of the inner workings of the 99/4 as top secret nuclear bomb plans and apparently viewed us hobbiests as Soviet spies, it is the only way one can learn the full capabilities of their computer.

To begin to understand how a program is stored on disk, we must know how it is stored in the computer. Programs are stored in two sections: the Line Number Table and the Program Statement Table. When a program is listed, the line numbers precede each instructions, but this is simply for your convenience, not the computer's.

LINE NUMBER TABLE:

The line number table contains four bytes of information for each program instructions. The first two bytes are the line number you used for that instruction (in hexadecimal notation, of course) and the last two bytes are the memory address where the computer actually puts the instruction. The line numbers are stored in reverse order from the highest to the lowest.

When a program is run, the line numbers aren't used unless the normal sequential flow is altered by a GOTO, GOSUB, etc. When a transfer instruction is encountered, the computer finds the new line number in the table and begins executing instructions at the indicated memory point.

PROGRAM INSTRUCTION STATEMENTS:

The instruction statements come after the line number table and are stored in "tokenized" format. What does this mean?--you won't find the word "PRINT" anywhere in Figure 3. If TI Basic weren't tokenized, "PRINT" would require five bytes of memory. All the commands would require a byte for every character; "RANDOMIZE" would take nine bytes if commands weren't "crunched".

Instead of suffering under such a colossal waste of memory, every command is symbolized by a one byte value. This results in a subsantial reduction in memory "overhead".

The token for the command "PRINT" is hexadecimal >9C, which can be found at byte >D of Fig. 3. The decimal value would be 156. Let me give you an interesting demonstration of this:

1.POWER-UP YOUR COMPUTER IN TI BASIC AND TYPE IN THE LINE:

    1 REM [CONTROL] [;] [ENTER]

2. LIST THE PROGRAM AND, MYSTERIOUSLY, THE WORD "PRINT" HAS APPEARED.

DEMO

Actually, since we know how programs are tokenized, it isn't all that mysterious. If you check the appendix in the TI Basic "User's Reference Guide", page III-2, you'll find that [CONTROL] [;] has a character value of 156. Seeing command tokens within REMs being printed out as command names is just a quirk of TI Basic.

Now that you understand how programs are kept in the console memory, you'll see that they are stored on disk in almost an identical way.

1-3 HOW BASIC PROGRAMS ARE STORED ON DISK

On disk, programs are still stored in two sections: the line number table and the instruction statements. However, the first eight bytes of the first program sector are pointer values that are used by the Basic program loader. These pointers are used to load the program into the proper area of console memory and also to indicate where the Line Number Table ends and the program instructions begin. We will consider these pointer bytes as a third section of the stored program. Thus, programs are stored on disk in three segments:

PROGRAM LOAD INFORMATION.

LINE NUMBER TABLE.

INSTRUCTION STATEMENTS.

The Program Load Information always takes up the first eight bytes, but the lengths of the other two sections will vary depending on the number of lines and the lengths of the program statements.
Let's examine the disk storage format of our one line program. Look at Fig. 3 again while I give you a byte-by-byte explanation of what that "garbage" means.

PROGRAM LOAD INFORMATION.

Bytes 0-1(005): This value is used by the Basic Program Loader (when you use the 'OLD" command) to check that what you are trying to load is a valid program. It's value is obtained from the next four bytes. Knowing how to find this value is of minor importance unless you want to build Basic programs from scratch. For the sake of completeness, I'll show you how to find this value.

OR → You "exclusive or" words >2 and >4. To do this, expand the values into their binary equivalents and line up the columns. If the columns are different (0/1 or 1/0), write a one below that column. Change the resulting binary value back into hex. This is the value that is placed in bytes >0-1

EXAMPLE OF "EXCLUSIVE OR"

```
  >2=37CE=  0011 0111 1100 1110
  >4=37CB=  0011 0111 1100 1011
            ---------------------
            0000 0000 0000 0101
              0    0    0    5
```

THE VALUE >0005 IS THE RESULT

Bytes 2-3 (37CE): The console address of the end of the line number table:

Bytes 4-5 (37CB): The starting address of the line number table.

Bytes 6-7(37D7): The address of the highest memory location used by the program.

These program location pointers are the values that were being used by the Basic Interpreter when the program was stored. I changed these on the disk in an attempt to get the program to load in a different area of memory, but it didn't work. It seems that the Basic loader is going to put the program in the first available area of memory. It only uses the stored addresses to get the relative locations of the Line Number Table and the Program Statements Table. Example: If you save a one line program and then use Disk Fixer to subtract >3700 from all the address values, the program still loads in the same area as the original values. Oh well, being able to change the memory area where the program would load is of very limited use anyway.

LINE NUMBER TABLE

Bytes 8-9(0001): The last line number of the program (and, in this case, the only line number).

Bytes >A-B(37D0): the beginning address of the program line.

PROGRAM STATEMENT TABLE

Byte >C(08): length of instruction.

Byte >D(9C): command token (PRINT).

Bytes >E-14: data for the print statement.

If this were a much longer program, say about 8000 bytes, the Line Number Table would require several disk sectors. A 100 line program requires 400 bytes of disk space to store the Line Number Table, which is about a sector and a half.

"Wait a minute", you say, "Haven't you forgot something? What's all that stuff after the instruction statement?" This remaining information is the data contained in the buffer space used by the disk drive. It won't affect the storage or loading of Basic programs. It just happens to get saved if the program doesn't exactly fill up the last sector completely.

1-4 MAKING BASIC PROGRAMS INVISIBLE

Now that you've learned the mechanics of manipulating Basic program statements, let me show you how to put this knowledge to good use.

Have you ever wanted to keep people from listing or altering your Basic programs? Perhaps you have an educational program and you wanted to protect it against those who are cunning enough to list it out and extract the answers. If you have, then Disk Fixer can do the job!

We've already learned how to change line numbers into illegal values that can't be edited or deleted, but we found that resequencing the program easily changes the out of range numbers back to normal. Can Disk Fixer keep a program from being resequenced?--you bet! Let me demonstrate this process on the following two line program:

```
100REM COPYRIGHT 1984 NAVAR
ONE INDUSTRIES
200 PRINT "THIS PROGRAM IS P
ROTECTED FROM ALTERATION AND
 LISTING BY DISK FIXER"
```

Figure 4 shows how the program is saved on disk. Figure 5 shows the changes you make using Disk Fixer to protect the program.

FIGURE 4

```
NAVARONE IND. *** DISK FIXER V2.0 ** SECTOR DUMP     SECTOR ADDRESS  0022
ADDR =  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  INTERPRETED

0000 = 0009 3768 3761 37D7 00C8 376A 0064 37B2  **7h7a7W*H7j*d72
0010 = 479C C743 5448 4953 2050 524F 4752 414D  G*GCTHIS PROGRAM
0020 = 2049 5320 5052 4F54 4543 5445 4420 4652   IS PROTECTED FR
0030 = 4F4D 2041 4C54 4552 4154 494F 4E20 414E  OM ALTERATION AN
0040 = 4420 4C49 5354 494E 4720 4259 2044 4953  D LISTING BY DIS
0050 = 4B20 4649 5845 5200 269A 2043 4F50 5952  K FIXER*&* COPYR
0060 = 4947 4854 2031 3938 3420 4E41 5641 524F  IGHT 1984 NAVARO
0070 = 4E45 2049 4E44 5553 5452 4945 5320 00AA  NE INDUSTRIES **
0080 = 3FFF 1103 0000 0002 0003 C445 4D4F 3320  ?*********DEMO3
0090 = 2020 2020 0000 0000 0001 0000 0000 0000      ************
00A0 = 0000 0000 0000 2200 0000 0000 0000 0000  ******"*********
00B0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00C0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00D0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00E0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00F0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
```


FIGURE 5

```
NAVARONE IND. *** DISK FIXER V2.0 ** SECTOR DUMP     SECTOR ADDRESS  0022
ADDR =  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  INTERPRETED

0000 = 0009 3768 3761 37D7 0000 376A FFFF 37B2  **7h7a7k**7j**72
0010 = 479C C743 5448 4953 2050 524F 4752 414D  G*GCTHIS PROGRAM
0020 = 2049 5320 5052 4F54 4543 5445 4420 4652   IS PROTECTED FR
0030 = 4F4D 2041 4C54 4552 4154 494F 4E20 414E  OM ALTERATION AN
0040 = 4420 4C49 5354 494E 4720 4259 2044 4953  D LISTING BY DIS
0050 = 4B20 4649 5845 5200 279A 2043 4F50 5952  K FIXER*'* COPYR
0060 = 4947 4854 2031 3938 3420 4E41 5641 524F  IGHT 1984 NAVARO
0070 = 4E45 2049 4E44 5553 5452 4945 5300 00AA  NE INDUSTRIES***
0080 = 3FFF 1103 0000 0002 0003 C445 4D4F 3320  ?*********DEMO3
0090 = 2020 2020 0000 0000 0001 0000 0000 0000      ************
00A0 = 0000 0000 0000 2200 0000 0000 0000 0000  ******"*********
00B0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00C0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00D0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00E0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00F0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
```

It would be pretty mean of me to make you scan both entire sector print-outs for the four changes, so I'll spell them out. Type in the program in Basic EXACTLY as it is shown above. If you put in one extra space, the addresses won't match up with the instructions below. There isn't a space at the end of the REM statement. Save the program to a blank initialized disk.

1. USE DISK FIXER TO READ SECTOR >22 INTO MEMORY
   R 22,1 [ENTER]
2. RETURN TO THE DISK FIXER COMMAND MENU BY PRESSING 'ENTER'.

3. CHANGE LINE NUMBER 200 TO 0 BY ALTERING WORD >8 (00C8) TO BECOME 0:

   A 8 [ENTER] 0 [ENTER]

4. CHANGE LINE NUMBER 100 TO 65535 BY ALTERING WORD >C (0064) TO BECOME >FFFF:

   A C [ENTER] FFFF [ENTER]

5. ADD ONE TO THE LENGTH INDICATOR FOR LINE 100 (THE REM STATEMENT), WHICH IS LOCATED AT WORD >58. THE CURRENT VALUE IS >269A. The FIRST BYTE(>26) IS THE LENGTH AND THE SECOND BYTE(>9A) IS THE TOKEN FOR THE REM COMMAND. WE ONLY WANT TO CHANGE THE FIRST BYTE TO >27, SO BE SURE TO

RE-ENTER THE SECOND BYTE. THIS IS DONE WITH THIS KEY SEQUENCE:

A 58 [ENTER] 279A [ENTER]

6. LIKE THE PERIOD AT THE END OF A SENTENCE, BASIC LINES USE >00 TO MARK THE END OF THE STATEMENT. WE'LL CONFUSE THE INTERPRETER BY PUTTING TWO CONSECUTIVE "PERIODS" AT THE END OF THE REM STATEMENT. TO DO THIS CHANGE THE LAST SPACE CHARACTER (>20) AT BYTE >7D TO >00:

A 7C [ENTER] 5300 [ENTER]

7. ALL THAT'S LEFT TO DO IS TO WRITE THE ALTERED SECTOR BACK TO DISK WITH THE "W" COMMAND AND TEST THE RESULTS BY LOADING THE PROGRAM INTO BASIC.

Once the program is loaded into Basic, LIST it and see that only the REM statement appears and it is now line number 65535. Run the program and the PRINT line works just fine. Now, for the acid test, try to change the line numbers with the RES command. You'll notice a short pause followed by the loss of the screen's syncronization followed by a flurry of color bars. What has happened?

This is only speculation, but it seems the execution of the RES instruction causes the interpreter to search for line numbers buried within statements such as 100 GOTO 130. We have altered the beginning and end of a line, so perhaps it is attempting to find the end of the program. The screen loss occurs when the addresses memory-mapped for use by the Video Processor are inadvertently accessed, causing the sync loss. However it works, it's effective!

Now that you've seen how it works, let me give you the general procedure so you can protect your own programs from prying eyes.

GENERAL PROCEDURE FOR ALTERING A PROGRAM SO IT CAN'T BE LISTED

To prevent a program from being listed or altered, make your first and last program statements REMs, then four changes must be made: change the highest line number to >0, change the lowest line number to >FFFF, add one to the byte that contains the statement length of the first REM, and change the final "space" character (>20) to >00. Can't figure out how to do this?--let me spell it out:

1. Make the first line of your program a REM statement. Be sure no

transfers (e.g. GOTO, GOSUB) are made to this line. If you use a combination of letters that aren't likely to be duplicated elsewhere in your program (such as 100 REM ZCFIRST), you can use the Disk Fixer's FIND STRING (F) command instead of having to hunt fo the line manually.

NOTE : DISKETTE VERSIONS OF DISK FIXER DO NOT HAVE THE "FIND STRING" COMMAND.

2. To make it easy to find the last entry in the stored program's line number table, make your last line another distinctive statement, such as 30000 REM ZCLAST

3. After saving your program to disk, use Disk Fixer to locate the start of the Line Number Table. Since this will be stored on the first sector of the program, you can find the sector number by reading the file's Directory sector.

Example, If the name of the file where the Basic program is stored is "MYPROG", use the following Disk Fixer command:

F 1,21,1 [ENTER] (you will be prompted to enter the string to search for)

MYPROG [ENTER]

The drive will start clicking off tracks until the string is found. When it is found, press D [ENTER] to display the directory. Byte >1C of the directory gives you the first sector number of the file.

Remember, you can eliminate searching for the first sector if the program is saved to an empty disk, for then you know that sector >22 will be the beginning sector. You can copy the program to whatever disk you want after all the changes are made.

4. When you have found the Line Number Table, alter the word at location >8 to >0. This had been the highest line number used and now you have changed it to zero.

5. Finding the highest line number in step 4 was easy because it will always be in the same place, not so with the lowest line, its location will depend on the length of the program. If the program has more than 62 lines, the Line Number Table will require more than one sector.

This is where making our highest numbered REM a distinctive character string pays off. If the statement-- 30000 REM ZCLAST--is the last statement we entered, it will be the first statement following the Line Number Table. The Line Number Table always re-sorts its contents when a new line is added, this isn't true for the Statement Table. Pro-

gram statements are added to the top of previously entered statements regardless of their line numbers. Let me go off on a tangent and give you a simplified example of how this works. I'll symbolize the line number and statement tables so it will be easier to follow.

Suppose a program is typed in as follows:

1 REM 1
2 REM 2
3 REM 3

We would have a Line Number Table ordered like this:

0003
0002
0001

And the statement table would line up as:

REM 3
REM 2
REM 1

However, if the statements had been entered in this order:

1 REM 1
3 REM 3
2 REM 2

The Line Number Table would look the same, but the statements would be stored in a different order:

REM 2
REM 3
REM 1

Therefore, if you want to make it easy on yourself, you'll make sure 1 REM ZCFIRST (or something similar) is the first line you enter and 30000 REM ZCLAST is the last line entered. If you want to add the listing and alteration protection to a program you've already created, and if the first line isn't a REM you've got a problem. Not an unsolvable problem, mind you, just some extra work that I will explain latter. First, let's get back to the easier example where the REMs have been entered correctly.

You can find the sector that contains the lowest line number entry by having Disk Fixer search for that ZCLAST sequence. Look at this sector and find that table entry for line number >0001. Alter the word that contains "0001" to become "FFFF".

6. With these first five steps you have changed the Line Number Table so that only the first REM will show if the program is listed. Now let's

finish the process by preventing the RESEQUENCE command from undoing all our hard work. We do this by altering the stored format for our 1 REM ZCFIRST statement. At least it's easy to find, it will be on the last sector of the file.

Find the program's last sector and display its contents. It will be stored as the byte sequence 0B9A205A4346495253542000. Of course, Disk Fixer displays sector contents with spaces between bytes, so the sequence will appear as:

0B 9A 20 5A 43 46 49 52 53 54 20 00

To prevent resequencing, change that first "0B" to "0C" and that last "20" to "00". To verify that you did this correctly, display the sector after you have made the changes and you should see the following changed format:

0C 9A 20 5A 43 46 49 52 53 54 00 00

If you see something else, you screwed up.

WHAT IF I WANT TO USE THIS PROCEDURE ON AN EXISTING PROGRAM?

As I said earlier, this involves a little extra work. I discovered three ways to do this. The first way is very tedious if the program is quite long. You add the REMs 1 REM ZCFIRST and 30000 ZCLAST and then you edit every line, retyping the first character of each statement. This puts the program in the correct order and you can see the general method. A semi-trained ape could use this method, but he's going to need a lot of bananas if it's a 500 line program.

The second method took very little time, but required a lot of expertise to understand the complex change that had to be made to the Statement Table. I'm not even going to bother to describe it because the third solution is a piece of cake and equally effective.

The easy way is to change that last program statement into a REM so we can use the general method. We need to replace that missing line, of course, or it's highly doubtful the program will run as it was intended. Let's step through this process :

1. Make a copy of the program or at least have a printed listing of it.

2. Load the copy into Basic and RESEQUENCE it. Save the copy back to disk. This step isn't essential as long as none of the line numbers are consecutive, e.g. 200, 201.

3. Fire up Disk Fixer and find the last sector of the program. Display this

sector and find the last program statement. You can recognize the last statement by scanning for the byte sequence AA 3F FF that will come after the last statement.

4. Right before the "AA" byte will be the end of statement marker "00". Follow this backwards until you come to another "00", which will be the end marker for the next to last statement. The first byte after this "00" will be the length indicator for the statement we want to change into a REM. Add one to the length and alter the byte with the new value. After the length indicator will be the token for the statement's command. Don't worry about what it stands for, change it to "9A" (the REM token). Change the remaining bytes to "41" (the code for the letter "A") except for the one just before the "00". Make it into a second "00".

Do you see what we did? We have changed the format of whatever the last statement was into a non-resequenceable REM statement. Let me give you an actual example in case my instructions are confusing.

If the last bytes of the program are:

00 08 9C C7 04 54 45 53 54 00 AA 3F FF

We would alter them to become:

00 09 9A 41 41 41 41 41 00 00 AA 3F FF

5. Exit Disk Fixer and return to basic. Load the program copy and list it. Scan the statements until you find the new REM statement made of "A"s. Note the line number and view the original program to see what was lost. Go back to the copy and type in the lost line using a line number one greater than the REM statement. This will restore the program to original form. That added REM statement won't hurt a thing even if other program lines reference it.

6. Type in the **1 REM ZCFIRST** and **30000 REM ZCLAST** statements. Save the program back to disk and return to Disk Fixer.

Now all you have to do is alter those line numbers using the general method given above. No need to make the last statement Resequence proof, as step 4 already took care of that.

After you've done a couple of these, you shouldn't have any problems.

1-5 CREATING PROGRAM ODDITIES

After going through all the details of the listing/alteration protection, you deserve a rest. Let's create a couple of bizarrely numbered programs. These are sure to raise a few eyebrows when they are listed out by the unsuspecting. These examples have little practical use, but they are a lot of fun.

1-5-1 A PROGRAM THAT "DOESN'T DO WHAT IT IS SUPPOSED TO DO"

Did you ever see those computer quizzes that test your knowledge of programming by having you analyze the flow of a program and predict what will occur when the program is run? Of course, those tests assume the program wasn't created with Disk Fixer!! Take the following example:

WHAT WILL BE PRINTED WHEN THIS PROGRAM IS RUN?

```
100 REM PROGRAM QUIZ #1
110 PRINT "ALL IS NOT WHAT IT APPEARS TO BE"
```

"Come on, don't insult my intelligence", your friends will say. We wouldn't do that, but we will pull their leg a little. Run the program for mister-know-it-all and look what appears:

THE DARK SIDE OF DISK FIXER STRIKES AGAIN!!

Can you figure out how this is done? Answer: two tricks are involved.

(1) Line 110 isn't a real PRINT statement. It is actually part of the REM statement that has wrapped around the screen. "110" only looks like a line number!

(2) The real line after 100 is invisible because its line number is greater than that of the last line in the program, which is an altered REM. The hidden lines are:

```
110 PRINT "THE DARK SIDE OF DISK FIXER STRIKES AGAIN!!"
105 REM
```

HOW TO MAKE THIS PROGRAM

1. Type in the code as follows:

```
100 REM PROGRAM QUIZ #1bbbb1
10 PRINT "ALL IS NOT WHAT IT APPEARS TO BE"
```

This is how line 100 will appear on the screen when it is typed. The fake line number will line up correctly when the program is listed. The four "b"s are only there to show how many spaces are needed between the two "1"s

```
110 PRINT THE DARK SIDE OF THE DISK
FIXER STRIKES AGAIN!!"
120 REM
```

2. Save it to disk and engage Disk Fixer. The line number to change is at word 8. Remember, it's in hexadecimal notation, so look for 0078, not 120. Alter it to become 0069 (hex for 105).

3. Write the sector back to the disk and give the program a test drive in Basic.

1-5-2 A PROGRAM WITH LINE NUMBERS THAT RUN BACKWARDS??

Remember, Basic arranges line numbers for your convenience, it really doesn't care what order they go in. Why?--because the real line number is the actual memory address where the line is stored in console memory. When the Basic Interpreter encounters a line number in a statement such as GOTO 100, it has to look up the memory address before it can continue.

Let's make an inverse numbered program. Start with the "legal" program:

```
1 PRINT "RUN";N
2 FOR X=1 to 10
3 PRINT X
4 NEXT X
5 PRINT ::
6 N=N+1
7 GOTO 1
```

Run the program to see what it is supposed to do. Stop the program and change line seven to 7 GOTO 6. We do this because we are going to change the line number of the first line from 1 to 6.

Save it and use Disk Fixer to reverse the line number order to become 6,5,4,3,2,1,7. Why didn't we change 7? If we did, the rest of the lines wouldn't list and we wouldn't see our results.

If you are becoming proficient with Disk Fixer, you should be able to make the changes in less than a minute. The last line number can always be found at word 8 of the program's first sector. We want to start our changes at the next to last line number, so we add 4 bytes to 8, which tells us word >C is where the first change needs to be made. Less experienced users would probably scan the sector for the value 0006. I use small line numbers in most examples so you won't have to make hexadecimal to decimal conversions when manually searching for line number values.

Once you have displayed the first sector, you can zip through the needed changes with this key sequence:

A C [ENTER] 1 [SPACE] [SPACE] 2

[SPACE] [SPACE] 3 [SPACE] [SPACE] 4

[SPACE] [SPACE] 5 [SPACE] [SPACE] 6 [ENTER] W [ENTER] Q [ENTER]

After the reversed program is saved back to disk, all that's left to do is load it back into Basic. List the program to see the change. Run the program to see if it still works normally.

Test your proficiency at changing line numbers. Create the following program on your own:

```
0 FOR X=1 TO 10
0 PRINT X
0 NEXT X
50000 PRINT ::
```

Duplicating line numbers doesn't cause any problems, unless you transfer program control with a GOTO, GOSUB, IF...THEN, etc. If any of these use a line number that has been duplicated, the program will transfer control to the line number nearest the end of the Line Number Table:

```
0 GOTO 1
1 PRINT 1
1 PRINT 2
1 PRINT 3
```

When this program is run, only "3" will print.

2. UNPROTECTING PROGRAMS AND DISKS

WARNING: UNAUTHORIZED DUPLICATION OF COPYRIGHTED MATERIAL IS A VIOLATION OF U.S. LAW.

I do not advocate program piracy, as I co-authored a game program that lost substantial sales due to pirating. If you copy a program in order to avoid having to pay for it, you are a thief. It's no different from shoplifting.

However, legal owners of software are allowed to make additional copies strictly for their own use as protection against damage. This section is included in the hope that you will never have to use my instructions for fixing disks. It isn't cost effective to save the $2 cost of a back-up disk and risk throwing away hundreds of hours of hard work

you spent creating the contents of a disk. I learned my lesson the hard way. The University of Hard Knocks has an extremely high tuition, but you're not likely to forget the lessons. If you spend $100 on a program disk, $2 is cheap insurance.

"Wait a minute", you say, "I would gladly make back-up copies, but when I try all I get is a warning that the disk is 'proprietary' ". Yes you used to have a problem, but not any more--you own Disk Fixer! You can remove the company protection in seconds and then make the number of extra copies your particular level of paranoia warrants.

2-1 REMOVING THE COMPANY DISK PROTECTION

The TI protection is pathetically weak. I assume the company was relying on the read/write sector subroutine that is the heart of Disk Fixer remaining a deep, dark secret. Without Disk Fixer, knowing how the protection works is of little value.

A disk is protected by giving byte >10 on sector 0 a value of >50 (which is the ASCII code for "P"). To unprotect a disk, you simply change the >50 to >20 (the "space" character). It's as simple as that.

2-2 COPYING DISK WITH "HOME BREW" PROTECTION

Since the company protection is so easily broken, many independent programmers develop more effective methods, they built a better pirate trap. These protection schemes fall into two general groups: those that use standard tract and sector formatting and those that don't. Disk Fixer can duplicate the first type of disks, but usually it is ineffective against the second. We'll look at both kinds.

2-2-1 PROTECTION BASED ON STANDARD TRACT AND SECTOR FORMAT.

When you initialize a new disk, the Disk Controller marks it with a pattern of sector addresses that are similar to house addresses. These addresses are used to find data that has been stored on the disk. If all of these addresses are numbered as they are supposed to be, the disk is standard format and can be copied by Disk Fixer.

There are many ways to protect disks. Some assembly language programmers will purposely destroy File Directories and then use a special subroutine to load the data sector by sector. Another method is to erase part of the header on sector 0. You can still load programs from the disk, but if you try to copy them with the Disk Manager you will get a "DISK NOT INITIALIZED" message instead of your copy. None of these methods will prevent you from duplicating the disk.

It's a tedious process. You use Disk Fixer to read all 360 sectors on one

disk and write them to another. If you don't get any bad sector messages, the deed is done.

2-2-2 NON STANDARD PROTECTION METHODS

The release of Disk Fixer panicked many software companies. Programs that once were safe from most pirates became vulnerable to large scale theft. The first version of Disk Fixer was sold on disks and people didn't hesitate to use it to unprotect itself. Now it is only sold on module.

PSEUDO NON STANDARD PROTECTION METHODS

Creative programmers discovered ways to make Disk Fixer users think they were dealing with non-standard disk. When Disk Fixer tries to read these damaged areas, it will indicate a bad sector. Of course, all you have to do is ignore these messages and continue copying all of the good sectors and you will get a working disk.

A fairly good way to throw most pirates off track is to use assembly language to initialize an odd number of tracks, perhaps 30 instead of the usual 35 or 40. The program on the disk would check the last sector on the 29th track and the first sector of the 30th track. If these registered good/bad respectively, the program would execute normally. If the sectors came out good/good or bad/bad, the program knows it isn't on the original disk and will refuse to run. If you are REALLY knowledgeable in assembly language, you simply initialize a disk with the same number of tracks and then copy the disk on it. Not many people know how to do this, but there is a way around it.

HOW TO UNPROTECT A DISK THAT ISN'T FULLY INITIALIZED.

To do this, we make use of another fine Navarone product, the WIDGIT. The Widgit has a reset button that will halt the disk initialization at just the right point with no ill effects. You might be able to use the console on/off switch, but it's chancy.

First, you have to test to see if the disk is not fully initialized. Use the Disk Manager to run a non destructive test on the disk in question. It is not fully initialized if the first bad sector is evenly divisible by nine and all the remaining sectors are also bad.

Example: if the first bad sector is 90 (90 is evenly divisible by nine) and sectors 91-359 are bad, the disk is only partially initialized. Here's what you do to make an identical disk:

* Install the Disk Manager into your Widgit and plug it into your console. You have to use a disk that has never been initialized so there will

be no tracks already formed. I use a one pound magnet to bulk erase a formatted disk and this removes any previous tracks.

* Divide the number of the first bad sector by nine (90/9=10)--this tells you when to halt the initialization process.

* With your finger ready on the reset button, start the Initialization. The number you got from the last step tells you how many "clicks" to listen for from your drive. In our example, we would listen for 10 clicks and just as soon as we heard number 10 we would press the reset button. You have to be quick, but it's easy to hit it just right.

* Use Disk Fixer to test the disk. If the next to last sector (in this case 89 or >59) is good and the next sector (90 or >5A) is bad you did it right. If not try again.

* Now that you have the disk, all you need to do is have Disk Fixer copy all of the good tracks from the original to the copy.

DISKS THAT HAVE TRULY NON STANDARD FORMATS

You're out of luck--Disk Fixer is powerless against these. My new protection system uses this method and without giving away any house secrets, I'll tell you a little about how I do it.

I analyzed the assembly language instructions contained on the read only memory (ROM) inside the Disk Controller. With a lot of experimenting, I learned how to alter the addresses of just a few sectors. The protected program stored on these special disks contains a subroutine that can detect these address changes. It knows when it is residing on the original disk. It works so well that I named it CIPHRDISK and it is available for licensing .Serious inquiries can be sent to:

BILL GRONOS
9505 ½ SE 15th SUITE B
MIDWEST CITY, OK 73130

Now we leave disk protection and examine program protection.

2-3 UNPROTECTING EXTENDED BASIC PROGRAMS

This is an easy task, if you understand how hexadecimal "twos complement" arithmetic works.

To illustrate this protection, we'll create two BASIC programs. If you start with a blank initialized disk, you won't have to search for where the two programs have been stored. Enter Extended Basic and Type in the following program:


```
1 REM
```

Now save it unprotected with the following command:

```
SAVE DSK1.TEST1
```

Next, save it to a new file in protected format:

```
SAVE DSK1.TEST2,PROTECTED
```

We can now plug in our Disk fixer modules and inspect these two programs. Since the disk is empty, the unprotected program will be stored at sector >22. Bytes 0 and 1 of this sector contain the value >0005. The protected program is stored at sector >23 and its first two bytes contain the value >FFFB. Hexadecimal value >FFFB is the twos complement of >0005. Another way of looking at twos complement numbers is as positive and negative values. The value >FFFB can be considered as ->0005. This is how assembly language handles negative values.

So, if you want to unprotect an Extended Basic program, all you have to do is change the value in the first two bytes of the program's first sector into its twos complement value.

If you can do hexadecimal arithmetic, this can be done by subtracting the value from >FFFF and then adding one. E.G.,>FFFF-FFFB =>0004. Add one and you get >0005. Use the Disk Fixer Alter(A) command to replace >FFFB with >0005, save the changed sector to the disk with the Write(W) command and the protection is nullified!

3. USING DISK FIXER TO REPAIR BAD DISKS

Disk Fixer is like a bottle of glue: it'll put an arm back on a broken figurine, but it won't be much help in putting a pile of plaster dust back into the shape of a statue. If your prize disk accidentally got picked up by the magnetic crane that lifts cars in junk yards, I recommend you try using Buddhist chants or prayers to Saint Jude because Disk Fixer isn't going to do you a bit of good.

Disk Fixer has saved three disks of mine that had blown bit maps (located on sector) and can handle a variety of other disk diseases. Here is a brief run down of maladies for which Disk Fixer is a specific remedy, plus the difficulty of the required operation:

* BLOWN BIT MAP/DISK HEADER (SECTOR 0)--VERY EASY.

* BLOWN DIRECTORY LINK MAP (SECTOR 1)--EASY.

* FILES DELETED BY ACCIDENT--EASY IF NO NEW FILES WERE WRITTEN TO THE DISK AFTER THE DELETION OCCURED.

* DAMAGED SECTORS OF BASIC PROGRAMS OR TEXT FILES--PARTIAL DATA RECOVERY IS FAIRLY EASY.

* BLOWN FILE DIRECTORY (USUALLY SECTORS 2 TO >21)--DIFFICULT TO VIRTUALLY IMPOSSIBLE).

The above text is the course listing for the Floppy Disk School of Medicine. Completion of this program results in the award of a degree in Surgical Discology.

First, let me differentiate between blown and damaged sectors. Blown sectors are those that have had the data changed, but still have the proper sector and track addresses. Blown sectors can be reused because they can still be re-written.

Damaged sectors, however, can not be reused unless you re-initialize the disk. These sectors have had their identification markings altered or destroyed. You can not read these markings with Disk Fixer, nor can you repair them. If your disk has encountered a stray magnetic field, you will have some damaged sectors.

It's easy to tell if a sector has been blown or damaged; Disk Fixer will read or write to blown sectors, but it will indicate an error if you try to access a damaged sector. Let's start your lessons with the easiest case.

3-1 HOW TO FIX A BLOWN BIT MAP (SECTOR 0)

Ruined Bit Maps may go undetected. Even if this sector is completely lost, you will be able to load Basic programs, read TI-Writer files, etc. You may not know the Bit Map sector has been lost until you try to catalog or copy the disk with the Disk Manager and you get the "DISK NOT INITIALIZED" message.

The danger of an altered Bit Map looms into view when you save more data to the disk. For instance, if the Bit Map sector was changed to all zeroes, new data would be saved on top of data already on the disk--the Disk Controller thinks all those sectors are still available.

Fixing bad Bit Maps is a piece of cake. Even if sector 0 was completely erased, you can easily recover every bit of data stored on the disk.

A quick fix is to copy a good sector 0 from another disk and write it to the sick disk. This restores all the initialization data but not the bit map. You can now read or write to the disk, but since you no longer have a correct bit map to indicate which sectors are used and which are available, writing to the disk will likely overprint files on top of good data. No problem! Simply make a back-up of the fixed disk with the Disk Manager and reinit. the fixed disk for re-use.

Be sure that the disk you copy sector 0 off of is the same type as the disk you are repairing; e.g., if it is a double-sided disk, copy sector 0 from another double-sided disk.

WHAT DO I DO IF SECTOR 0 IS DAMAGED RATHER THAN BLOWN!

Trying to write to a damaged sector will produce an error, so copying a good sector 0 is out. The fix is still easy, but it is tedious.

You start with an empty good disk and copy all of the sectors from the sick disk to it. Then you simply make a back-up disk to restore the bit map.

Copying 359 sectors from one disk to another is going to take a little time, but it sure beats having to rewrite all of the files that would otherwise be lost.

3-2 FIXING BLOWN DIRECTORY LINK MAPS (SECTOR 1)

This is almost as easy as the first case. Just as with sector 0, if sector 1 is bad the entire disk will be unreadable by normal means.

You fix sector 1 by manually reconstructing it. You locate all the File Directories and note the file names and which sectors the File Directories are located on. Next, you place the file names in alphabetical order. Once this is done, creating a new directory link map is a snap. Let's do one together.

File Directories are usually located on sectors >2->21 (2-33 in decimal notation). First, we use Disk Fixer to scan these areas, locate all the file directories and write down the sector number and the contents of the first 10 bytes. These bytes are the hexadecimal form of the file names. File Directory format is described in appendix A3. They're easy to spot by the large number of 00 bytes towards the end of the sector.

In our example, four File Directories were found and we wrote down the following work sheet:

SEC#	FILE NAME									
2	4D	59	50	52	4F	47	20	20	20	20
3	54	45	53	54	31	20	20	20	20	20
4	42	41	54	54	4C	45	5A	4F	4E	45
5	4A	55	41	4E	49	54	41	20	20	20

No need to translate the hex codes into letters. The alphabetical order will be the same as the numeric order of the file names. Just treat the names as if they were 20 digit numbers and rank them from lowest to highest. Write down the new order of the directory sectors:

4,5,2,3

Next, make them into four digit work values:

0004 0005 0002 0003

The last step is to use Disk Fixer to write these values to sector 1 starting at the first byte (byte 0). You must start with sector 1 being all zeroes. So, if a lot of garbage has been dumped there, change every byte to 00 and then put in the Directory Link Map.

There is one snag that slightly complicates this otherwise simple operation. If you delete files, the directory link map is instantly changed, but the old File Directories will remain until they are reused. Thus, you may find some of these during your search. They will look like valid directories, but you don't want to use them in re-creating the directory map. What can be done?

You can avoid this problem by checking your list of file directories against the Bit Map on sector 0. The directories for current files will be allocated in the Bit Map. If the file has been deleted, the Bit Map will show that the sector where the old directory sits is available for reuse Appendix A2 will show you how to read the bit map.

The above instructions will work if sector 1 is blown. If it is damaged, Disk Fixer will give you an error when you try to recreate the Directory Link Map. Just as in the first corrective procedure, you are going to have to do the tedious job of recreating the directory link on to a good disk and then transferring all the other good secors on to the new disk.

3-3 WHAT TO DO IF YOU ACCIDENTALLY DELETE A FILE

If you studied the previous operation on fixing blown Directory Link

Maps, you may have already figured out how to "un-delete" a file.

When a file is deleted, the location of its File Directory sector is removed from the Directory Link Map at sector 1. The Bit Map at sector 0 is changed to show that the sectors used by the deleted file are now up for grabs when new files are added to the disk. The actual file is not changed in any way.

All you do is add the location of the file's directory back to the Directory Link Map as explained in section 3-2. Of course, you need to change the sector 0 Bit map to show that the file's sectors are back in use, but there is a simpler method of doing this.

To have the Bit Map corrected automatically, all you do is this:

1. Add the location of the deleted file's directory sector back to its proper place on sector 1.

2. Copy the file to another disk.

3. Copy the file from the new disk back to the original disk using the same name.

When the file is recopied to the original disk, its sectors will be shown as in use and the data will be protected from overwriting.

3-4 PARTIAL RECOVERY OF DAMAGED FILES

If an important sector, such as the Directory Link Map, is bad, the disk is unusable. In other cases, sectors within a file that actually contain the data may be damaged and only that one file is unreadable. Disk Fixer lets you recover the undamaged file portions which otherwise may have been inaccessable.

3-4-1 BASIC PROGRAM RECOVERY

If the partially damaged sector is a Basic program, loading will halt when the damaed sector is encountered and you'll get the "WARNING, CHECK PROGRAM IN MEMORY" advisory. Most of the program could still be good, but the Basic Loader refuses to let you see any of it.

What if you had a 500 line program with a single bad sector? If you had a listing, you could type the program back in, but that would be a big job--especially if you didn't know how to type. If the bad sector was in the Program Statement Section, you wouldn't have to reenter the entire program, just the lost lines. The Basic Editor won't let you do this, but Disk Fixer will!

Even if you don't have a program listing, partial recovery will aid you in

rewriting the program if it is one of your own creations. You may be able to salvage valuable subroutines or perhaps all you will need to do is reprogram a few lines that have been lost. Before I get your hopes up too much, let me explain the limitations of this process.

The partial loss of a Basic program could have occured in three places: the File Directory (which has nothing to do with the Basic program and will be covered separately in section 3-5), the Line Number Table, or the Program Statement Table. The difficulty of the recovery will depend on where the damage occured, so let's look at what we can hope to achieve from each section.

BAD LINE NUMBER TABLE

Earlier in this booklet, I explained how Basic program lines will be saved in reverse consecutive order if they are entered in their proper sequence and none of the lines were edited. If this is the case, we can compute all statement starting addresses and put in new line numbers. There is a problem. If any of the statements transfer program control to another line number, such as GOTO 100, we have to know which of the lines was the old number 100. We may be able to figure this out by analyzing the program, but then again we may not.

If the Line Number Table was not saved in consecutive order our job becomes more difficult. We can reconstruct all the lines, but we can't be sure what order they go in. This could be very tough if the program was written by someone else.

BAD PROGRAM STATEMENT TABLE

In this case we can amputate the bad sectors and replace them with filler material. Then we can get a listing of what's left and try to work it out from there. The missing statements will show up as line numbers only. The recovered listing will be something like:

```
100 PRINT "SAMPLE PROGRAM"
110 PRINT "AMPUTATED LINES"
120 FOR X=1 TO 100
130 PRINT X
140
150
160
170 REM INTEREST SUBROUTINE
180 INT=R*P
190 RETURN
```

If you have a printed listing of the program, all you need to do is fill in the gaps. If you don't have a listing you'll have to recreate those missing

lines by analysis. If the program has been heavily damaged, you may have to deduce the entire animal from its jaw bone.

Since the bad Program Statement Table is the easier repair to describe, we'll start with it. We'll use the simplest example: a program with one damaged sector.

PATCHING A BAD PROGRAM STATEMENT TABLE

Our patient is a middle-aged male basic program. When we try to load the program into basic we get an error message. We take an "x-ray"of the patient by using the disk manager to do a non-destructive test. The test results indicate sector 36 (hex >24) has a bad address code. The cause could have been a scratch, a piece of dust or a memo magnet your wife used to pin your priceless disk to the 'fridge so you wouldn't lose it. Your trained eye scans the sector contents displayed by Disk Fixer and you see that the line number is complete. The bad sector took a hunk out of the middle of the Program Statement Table. It's a text book case and the school solution is a sector by-pass operation.

Your pre-operative preparation is to make a program filler out of an unused disk sector. To do this, make a sector which has 00 in every byte location. The quickest way to do this is to copy sector 1 from a blank initialized disk. Note the sector address of the filler.

The by-pass must be performed on the program's File Directory. To find this, use the Find string (F) command in Disk Fixer to search for the file's name. Display the indicated sector. Figure 6 is a print out of our example program.

Figure 6

```
NAVARONE IND. *** DISK FIXER V2.0 ** SECTOR DUMP    SECTOR ADDRESS  0002
ADDR =  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  INTERPRETED
-------------------------------------------------------------------
0000 = 4A55 414E 4954 4120 2020 0000 0100 0007  JUANITA   ******
0010 = 2100 0000 0000 0000 0000 0000 2260 0000  !**********">**
0020 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0030 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0040 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0050 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0060 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0070 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0080 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
0090 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00A0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00B0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00C0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00D0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00E0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
00F0 = 0000 0000 0000 0000 0000 0000 0000 0000  ****************
```

The part of the sector we need to change begins at byte >1C. This is the Block Link and it is explained in detail in appendix A3. If you are not familiar with how these work, now is a good time to review that appendix. Briefly, Block Link is used to join together the fragments of a file that have not been stored in a consecutive block of sectors. If there is only one link, the file has not been fragmented and all the sectors are continuous. Each link is three bytes long.

We perform the by-pass by adding additional Block Links to the File Directory sector. These extra Links will point to the location of the program filler sector. We alter the other link(s) so that the bad sector will not be part of the file.

Our example program (figure 6) has one Block Link, so all the file sectors are located in one continuous group. In the middle of this group is a bad sector that we need to by-pass. The three bytes that make up the Block Link have the values 22 60 00. This tells us that the file consists of sectors:

22, 23, 24, 25, 26, 27, 28

Our program filler is at sector >5C and the bad section we need to by-pass is at sector >24. Therefore, we need to alter the Block Links so the file structure will be:

22, 23 5C 25, 26, 27, 28

Instead of one continous area of sectors for our program we now have three. Therefore three Block Links are required. Each Link must contain the location of the first sector in the block and also the highest numbered sector within the block. There are seven sectors in the block and these are numbered 0-6. Sector > 22 is number 0, sector > 23 is number 1, sector >5C is number 2 and sector >28 is number 6. So, our Block Links will be:

22 10 00
5C 20 00
25 60 00

If you don't understand the peculiar format Block Links have, you need to take a look at appendix A3.

We are now ready to perform the actual by-pass operation by writing these nine bytes to the File Directory sector starting at byte >1C. The by-pass is finished. If you have done everything right, the program will now load in Basic and you can list it and view the gap where the filler has been inserted.

For practice why don't you create this file, pretend sector >24 is bad and perform the by-pass. Start with an empty initialized disk. Use Disk Fixer to copy sector 1 to sector >5C to create the needed filler. To create a Basic program that will not have any of its Line Number Table in sector >24, type in the following line 15 times. Use different line number for each line:

100 REM 123456789012345678901234567890123456789012345678901234567890

Save the program, perform the by-pass as outlined above and view the results. Two lines have only line numbers followed by blanks and one line has been partially lost. The important thing is that 12 good lines have been recovered. This recovered portion could have been several hundred lines of coding if this had been a long program.

For additional practice, pretend sector >28 is damaged and change the block Links to by-pass it. Since the bad sector is at the end of the file, only two Links are required:

22 50 00
5C 60 00

Be sure the remainder of the sector following the Block Links is all 00's.

As you can see, it is fairly simple to patch programs that have one bad sector in the Program Statement Table. It only takes a bit more work to patch a program with multiple bad sectors. As an example, assume the sample program is bad at sectors >24 and >25.

We are going to need two sectors of filler material, so copy sector >5C into >5D. Three Block Links are required for the by-pass:

22 10 00
5C 30 00
26 60 00

What if we had two bad sectors that were not consecutive, such as >24 and >26? Simple? We do a double by-pass and the Block Links would be:

```
22 10 00
5C 20 00
25 30 00
5C 40 00
27 60 00
```

Did I lose you on this one? Let's take a closer look. Here are diagrams of the original file and the patched file:

ORIGINAL FILE

```
SEC ORDER   00 01 02 03 04 05 06
SEC NUMBR   22 23 24 25 26 27 28
```

PATCHED FILE

```
SEC ORDER   00 01 02 03 04 05 06
SEC NUMBR   22 23 5C 25 5C 27 28
```

If you use tables such as these, the Block Links are child's play to make. This should be enough examples of statement patching. We'll move into the trickier realm of mending Line Number Tables.

PATCHING LINE NUMBER TABLES

This is pretty much like fixing Program Statement Tables, but there are two differences: the filler material is different and there are several difficult complications.

The worst complication occurs when the program's first sector is bad. Sure, it can be patched around, but that isn't going to help much. The first sector contains the program's critical loading information. It tells Basic where the Line Number Table ends and where the program statements begin. It also has error checking values that will prevent the program from loading if they are not correct.

It is possible to recover some data if this first sector is bad, but it is a complicated, painstaking process that does not have general procedures that can be readily followed. As I said earlier, if the Line Number Table is completely wiped out you can still recover every statement, but you have no certainty as to what order they go in. Have you ever written a long program where you didn't have to edit a single statement or squeeze in an extra line? Not likely. If any editing was done to the program, the statements will not be stored in consecutive order. I will limit the discussion on patching Line Number Tables to cases where the first sector is still intact.

If the bad sector is somewhere in the middle of the Line Number Table, we can patch it just as we did the Program Statement Table. First, we need to create a special filler sector that will replace all missing line numbers with 0 and leave the rest of the line blank. To make this filler, find an unused sector and repeat the byte pattern 00 00 37 D7 00 00 37 D7 until the entire sector is filled. Write this data to an unused disk sector and prepare to operate.

Now that the filler is created, the rest of the process is just like patching Program Statement Tables. You create the new Block Links and place them in the File Directory sector in exactly the same way.

When you list the patched programs, you will see continuous sections of 64 blank lines that are numbered 0. These sections will occur wherever the lost lines would have normally listed.

Thats all I have to say about attempting to fix Basic programs. While the patching process is easy, recoding the missing section can be next to impossible. Text files are just as easy to patch and far more easier to reconstruct. We'll look at them next.

3-4-2 TEXT FILE RECOVERY

Unlike Basic programs files which have a format that is very foreign to us, text files lend themselves to partial recovery. If we lost one sector of a Line Number Table, the reconstruction is going to be tough. But if a single sector of a text file was lost, we would only need to rewrite a short paragraph. This paragraph is about one sector's worth of data.

Plus, text files are very forgiving. If we don't reword the missing section in exactly the same language, no real harm will probably be done. We may even improve on the content since it is our second crack at it.

Most text files the average user will produce will likely be TI-Writer documents. I had a magazine article stored on a disk with files totaling over 100 sectors. The disk developed three bad sectors in one of the files and I felt like sticking my head in the oven. The bad sectors were near the beginning of a 49 sector file. When I tried to load the file with TI-Writer all I got was the first five sectors and then the disk error message. It looked like 44 sectors worth of hard work had been lost. Luckily, I had Disk Fixer. It was this file loss that got me started on the techniques in this booklet.

I learned I could patch around the bad sectors and recover 46 sectors. The lost sectors werre easily rewritten in about half an hour. If it hadn't

been for Disk Fixer, it would have been no less than five hours of work. It only took 15 minutes to patch the file.

The actual patching operation is just as described for fixing Basic program files, just the filler material is different. The filler material isn't as critical as with Basic programs. Anything that will readily show us the lost areas will do. Personally, I use a filler sector of asterisks(*) because they really stand out.

To make the asterisk filler, power up TI-Writer and type in eight lines, each containing 32 asterisks. Save the file to disk and all you need to do is locate the first sector of the file. Jot down the sector number and you're ready to patch. When you load the patched file into TI-Writer, here's what it may like:

THE QUICK BROWN FOX JUMPS OVER THE LAZY
****
****
****
****
****
****
****
TO COME TO THE AID OF THEIR COUNTRY

3-5 HOW TO FIX A BLOWN FILE DIRECTORY

I rated this repair the most difficult because you have nothing to show you where to begin. The sectors that make up the file could be anywhere on the disk and in any order. If the file is long and highly fragmented you've got a tough job ahead.

If you knew the location of every sector in the file and what order they are supposed to be in, the job would be easy. Our example will be just such a case.

3-5-1 CONSTRUCTING TEXT FILE DIRECTORIES

Our patient is a TI-Writer file that has a blown File Directory. The file is 32 sectors long (not counting the sector used by the File Directory) and occupies consecutive sectors numbered >30 to >50. With this information we can easily construct a replacement file directory by following these steps:

1. Find an empty sector to use for the file dictionary and note its sector number. See appendix A2 on how to read the bit map on sector O if you don't know how to find unused sectors. Fill all the bytes in this sector

with OO's.

2. We know everything required to construct the File Directory except for one item--the contents of byte >10. At first glance, it was difficult to see what this byte had to do with the file contents, so I had to do some experimenting.

It seems that this byte shows how many bytes of the file's last sector are actually used. If the TI-Writer document that was saved to this file did not exactly fill all of the 32 sectors, this byte will tell the computer how much of the last sector contains data. How do we figure this out?

To find this value we need to find where the last record written to the file ends. TI-Writer ends a file by writing a tab and margin record to the disk. This is what we need to look for in the file's last sector. A typical tab and margin record will look similar to this when displayed by the Disk Fixer:

16 80 86 80 AD 86 8B 90 95 9F A9 D5 D5 D5 D5 D5
D5 D5 D5 D5 D5 80 86 FF

Every line of a text file has the same format and the tab/margin record is no exception. The first character will be the length of the line in bytes and the last character of the line will be the end of line marker "FF".

The tab/margin record is normally the last record on the last sector and is usually followed by 00 values, but not always. I've examined a bunch of text files and sometimes what seems to be garbage will follow the T/M record.

The easiest way to find the T/M record is to start from the end of the last sector and look backwards for the FF marker. Another sign post is that string of D5 values. These are unused tab stops. If you find what seems to be the T/M record, but you still have some doubt, count backwards 23 bytes from the FF. If this is a T/M record, the 23rd byte backwards will be the value 16.

Don't let this scare you, the tab/margin records are much easier to find than this would indicate. Practice finding these by examining some of your text with Disk Fixer and soon they will jump right out at you.

IF THIS EXPLANATION HAS YOU TOTALLY LOST, YOU CAN PROBABLY USE A VALUE OF FF WITH NO ILL EFFECTS. I did a little experimenting with this and found that the files will load correctly. You may get a little garbage displayed when you load the file back into TI-Writer, but it's easy to clean it up. To be on the safe side, I recommend deleting everything that shows up after the last bit of recognizable text.

Some of the garbage could be invisible control characters.

Getting back to our example, the value we need is the byte location of the T/M record's FF marker which is shown by Disk Fixer's index scales located above and to the left of the displayed sector. In our sample case we found this value to be >8C. Now we have all the information we need. The rest is all down hill.

3. The next step is to construct the File Directory contents. It's best to do this on paper and then use Disk Fixer to transfer the proper values to the File Directory sector. Here are all the values:

A. BYTES 0-9, FILE NAME: 7A 20 20 20 20 20 20 20 20 20

This sequence causes the file to be named lower case z. This should, alphabetically, be the last file on the disk and thus will make adding the File Directory's sector location to the File Index (located on disk sector) very simple.

B. BYTES >A-E, FILE SPECIFICATIONS : 00 00 80 03 00

These values are always the same for TI-Writer text files.

C. BYTE >F, NUMBER OF SECTORS IN FILE : 20

This value will vary according to the file size. In our example, we have 32 sectors (>20). Remember, we don't count the one sector that is used by the File Directory.

D. BYTE >10, BYTES USED IN LAST SECTOR : 8C

This is the value we found in step two. If you can't figure it out, try using FF.

E. BYTE >11, MAXIMUM RECORD SIZE : 50

This too, will always be the same.

F. BYTE >12, NUMBER OF RECORDS IN FILE : 20

This value varies with the file size. For variable record length files (such as TI-Writer output) it will be the same as the number of sectors given previously in step 3B.

G. BYTES >13-1B, FILL WITH 00's: 00 00 00 00 00 00 00 00 00

H. BYTES >1C-1E, BLOCK LINK: 30 FO 01

This is formed just as described above in the section 3-4-1 on partial file recovery. Block Links are explained in detail in appendix A3.

In this example, the file begins at sector >30 and is >20 sectors long. Since it is one consecutive block of sectors, only one Block Link is needed. Within the file itself, sectors are numbered >00-1F. The highest sector number contained in this block will be the last sector, >1F.

I. BYTES >1F-21, END OF BLOCK LINKS: 00 00 00

The end of the Block Link section is indicated by these 00 bytes. If garbage follows these values, the file will still function. However, it's best to zero out the rest of the sector or you may have problems if you increase the size of the file.

4. The File Directory is now complete and ready to save to disk. It's a good idea to use the Disk Fixer Dump(D) command to check over your work--it's awfully easy to make mistakes in altering bytes.

We'll assume the unused sector you've chosen is number >A. Write the sector to this location.

5. Last step. We need to add the location of this File Directory to the index at sector 1. This is where naming the file "z" comes in handy. Instead of having to reorder the entire index, all we do is add 00 0A to the end of the list:

BEFORE: 00 05 00 02 00 01 00 03 00 00 00 00

AFTER: 00 05 00 02 00 01 00 03 00 0A 00 00

6. Power up TI-Writer and load the file--the name will be DSKI.z (remember to use a small "z"). If you did everything right, you should have your file back.

3-5-2 CONSTRUCTING BASIC PROGRAM FILE DIRECTORIES

Basic program file directories are handled in nearly the same way that text files were handled in section 3-5-1. However, Basic uses "memory image" files rather than "record" files, so the bytes that indicate the file type will be different. We only need to change steps 2 and 3; the others are the same as in section 3-5-1.

1. See section 3-5-1

2. Byte >10 of a Basic program File Directory has the same purpose as in text files; it indicates how much of the last file sector needs to be loaded. Figuring out the correct value is easier than for text files because all Basic program files end identically.

LAST BYTE OF FILE

To find the program's final byte, display its last sector and scan the contents for the sequence 00 AA 3F FF. The byte containing 00 that precedes the AA value is the last byte of the file. Jot down the location of this byte within the sector and save it for step three.

If for some reason you just can't locate this last byte, you can get away with using a value of FF. This will cause extra data to be loaded along with the actual program, but Basic always knows where the program contents start and ends by using the load data at the start of the program.

3. Just as in section 3-5-1, we now are ready to construct the File Directory contents.

A. BYTES 0-9, FILE NAME: 7A 20 20 20 20 20 20 20 20 20

We name the file "z" for the same reason we did in 3-5-1.

B. BYTES >A-E, FILE SPECIFICATIONS: 00 00 01 00 00

These values are always the same for program files.

C. BYTE >F, NUMBER OF SECTORS IN FILE: 20

Two points to remember: we don't count the sector used by the File Directory and the count is expressed as a hexadecimal number, not decimal. Appendix A1 gives you the conversion values.

D. BYTE >10, BYTES USED IN LAST SECTOR: 8C

This is the value we found in step two. Plug in FF if you can't figure out the exact value.

E. BYTE >11, MAXIMUM RECORD SIZE: 00

Program files do not use records; this will always be zero.

F. BYTE >12, NUMBER OF RECORDS IN FILE: 00

Same reasoning as with byte >11.

G. BYTES >13-1B, FILL WITH 00's: 00 00 00 00 00 00 00 00 00

H. BYTES >1C-1E, BLOCK LINK: 30 F0 01

This is formed just as in section 3-4-1. It varies with where the file is stored on the disk and how many fragments it is made up of.

I. BYTES >1F-21, END OF BLOCK LINKS: 00 00 00

The byte location for this will change with the number of links. It will follow the last link and, just to be safe, you should zero out the rest of the sector that follows the Links.

Steps 4 and 5 are identical to section 3-5-1. Write the File Directory to disk and add its location to the index at sector 1.

6. Power up TI Basic and see if it loads with the command OLD DSK1.z. If it doesn't, check your work again.

With both text and program files, the hardest part of reconstructing File Directories is locating all of the pieces if the file has been fragmented. This requires expert sleuthing in extreme cases. If you know where all the fragments are and what order they go in the rest is not too difficult.

APPENDIX A1

HEX TO DECIMAL CONVERSION

Rather than explain how to convert between these two number systems, this program will do the conversion for you. I've kept it "no frills" so that it will be short and easy to type.

When first run, it will ask you to enter a hex number and then will convert it into decimal and request another number. Entering a "0" for a requested value will put you in the decimal to hex conversion mode. Entering another "0" will place you back in hex to decimal conversions.

```
100 REM HEXADECIMAL/DECIMAL CONVERSION
110 REM ENDTERING A 0 WILL FLIP-FLOP CONVERSION
120 HCR$="0123456789ABCDEF"
130 CALL CLEAR
140 INPUT "ENTER HEX NUMBER":H$
150 IF H$="0" THEN 310
160 N=0
170 LENH=LEN (H$)
180 FOR L=1 TO LENH
190 DIG=ASC [SEG$(H$,L,1)]
200 IF (DIG>47)*(DIG<58) THEN 240
210 IF (DIG>64)*(DIG<71) THEN 260
220 PRINT H$;" IS NOT A VALID HEX NUMBER"
230 GOTO 140
240 DIG=DIG-48
```

```
250 GOTO 270
260 DIG=DIG-55
270 N=N+DIG*16'(LENH-L)
280 NEXT L
290 PRINT H$;"=" ;;N
300 GOTO 140
310 INPUT "ENTER DECIMAL NUMBER" :D
315 D=ABS(D)
320 IF D=0 THEN 140
330 DIV=0
340 IF (D/16'DIV) <16 THEN 370
350 DIV=DIV+1
360 GOTO 340
370 D1=D/16'(DIV+1)
380 HNUM$=" "
390 FOR L=1 TO DIV+1
400 D1=D1*16
410 D2=INT (D1)
420 HNUM$=HNUM$&SEG$(HCR$,D2+1,1)
430 D1=D1-D2
440 NEXT L
450 PRINT D; "=";HNUM$
460 GOTO 310
```

APPENDIX 2

HOW TO READ THE BIT MAP (SECTOR 0)

The bit map is an area of sector 0 that is used to keep track of what sectors are in use and which are still available. When a new file is added to the disk, the bit map is searched for the most efficient arrangement of sectors that will store the file. Then it is altered to show that those sectors are no longer available. Likewise, when a file is deleted the bit map is changed to reflect the extra sectors that are now available for use.

Each sector is represented by a single bit. If the bit is a "1", then that sector has been used. If it is a "0', the sector is available.

The bit map begins at byte >38 of sector 0. Since a byte contains eight bits, eight sectors are controlled by each byte. Here is a diagram of the byte >38 to show how this works. Bits are numbered 0 to 7 from the left to the right:

BIT MAP BYTE >38

BIT	0 1 2 3 4 5 6 7
CONTROLS SECTOR	7 6 5 4 3 2 1 0

A table could be constructed for every byte to show exactly what sectors it controls, but this would be a waste of time and paper since all bytes function the same. Each byte in the bit map has a group of eight sectors it controls. To find the first sector controlled by a byte we can use the formula:

FIRST SECTOR = (BYTE NUMBER ->38) *8

The highest sector controlled by a particular byte is found by adding seven to the number of the first sector.
Example : We need to find an unused sector on a disk. We know that a value of 00 in the map indicates a block of eight consecutive unused sectors. To find out what those sectors are, we use the above formula. In this example the byte with the 00 value is number >60. Plugging this into the formula we get:

```
FIRST SECTOR   = (>60->38) *8
               =>28*8
               =>140
SECTORS CONTROLLED BY BYTE >60:>140 to >147
```

If hexadecimal arithmetic makes you queasy, you can convert all the numbers to decimal, perform the calculation and then turn the decimal numbers for the sectors back to hex. Again using byte >60 as an example:

```
>60 = 96    >38 = 56
FIRST SECTOR   =(96 - 56) *8
               =40 * 8
               = 320
SECTORS CONTROLLED BY BYTE >60: 320-327
```

A bit map value of 00 means all the sectors in that group are unused. A value of >FF means all eight sectors are in use. Values falling between 00 and >FF tells us that some sectors are used and some are still available. Here's how to tell which are which:
Example: Bit map byte >3A contains the value >0D. First we find the first sector controlled by that byte using the formula given above. The first sector works out to be >10, therefore some sectors between > 10 and >17 are used and some are unused.
Next, we change the value >0D into binary and get 00001101. The three "1's" show us three sectors are used. Likewise, each of the five "0's" represents an empty sector. Going from right to left, we can chart this section of the bit map:

USED SECTORS : >10,>12,>13
UNUSED SECTOR : >11,>14,>15,>16,>17

If you wanted to use sector >11 and wished to prevent it from being destroyed when more files were added to the disk, you would alter the 0 that represents that sector to a 1, change the binary number back to a hex number and then use Disk Fixer to alter byte >3A to the new value. Let's do it:

WITH SECTOR >11 UNUSED : 00001101=>0D
WITH SECTOR >11 USED : 00001111=>0F

Change byte >3A to >0F and sector >11 will be protected.

What if you want to figure out what part of the bit map controls a specific sector, say sector number >22? We divide the sector number by 8 to get a whole number and a remainder. The whole number plus >38 tells us the number of the bit map byte that controls that sector. The remainder, which will be between 0 and seven, tells us the exact bit:

>22/8=4 with a remainder of 2
4+>38=>3C

REMAINDER	7 6 5 4 3 2 1 0
BYTE >3C	0 0 0 0 0 * 0 0

The asterisk (*) shows the bit in byte >3C that controls sector >23. If we take a blank disk and save a one line Basic program to it, byte >3C of sector 0 will change from 00 to 04 to show that sector >22 is now in use.

APPENDIX 3

The information contained in this appendix may not be factual. The actual formats of data recorded on diskettes is not published by **Texas Instruments** and therefore not available for public use. The formats described here are simply an interpretation of the actual format and may not be accurate. It is however, information which you may find helpful when using **Disk Fixer** to locate and recover lost data from diskettes.

Several conventions apply to the directory information.

1. All addresses and other number values are in hexadecimal.
2. A byte is an eight bit field and nibble is a four bit field.
3. The term 'low order' refers to the rightmost (least significant) bit, nibble, or byte in a field. The leftmost (most significant) bit, nibble, or byte is the 'high order'.
4. Within a byte or nibble the bits are numbered from right to left with the rightmost bit designated as bit 0 and leftmost bit designated as bit 7 (for a byte) or bit 3 (for a nibble).

Sector 0 contains the diskette name in bytes 0-9. Bytes A and B taken as a full word, contain the value for the total number of sectors on the diskette. Byte C contains the value for the number of sectors per track. Bytes D-F contain the ASCII value "DSK". Bytes 10-13 contain the hexadecimal value X'20280101' on every diskette I have examined so far. It may be some kind of identifier. The sector bit map, which begins at address X'38' in sector 0, is a byte oriented table where a one bit indicates sector in use and a zero bit a free sector. Each byte in the map represents eight sectors with the low order bit representing the first sector in the group and the high order bit representing the last sector in the group. For example, in byte 38 bit 0 represents sector 0 and bit 7 represents sector 7. In byte 39 bit 0 represents sector 8 and bit 7 represents sector F. Therefore, when we look at the first word at the bit map (bytes 38, 39) the bits represent, from left to right, sectors 7, 6, 5, 4, 3, 2, 1, 0, F, E, D, C, B, A, 9, 8,. Each following word takes the same format for its group of sectors.

Each directory entry has the following format:

Bytes 0-9 File Name

Bytes A,B Unknown - I have seen them in use.

Byte C Bit O - on = program file, off = data file
Bit 1 - on = internal format, off = display format
Bit 2 - unused
Bit 3 - on = write protected, off = not write protected
Bit 4 - unused
Bit 5 - unused
Bit 6 - unused
Bit 7 - on = variable length records, off = fixed length records

Byte D Number of records per sector. Used only for data files. For fixed length records it is the actual number of complete records in each sector. For variable length records it is always 1.

Bytes E, F Number of sectors in the file. This word value is always one less than the file size indicated by the Disk Manager because the Disk Manager includes the sector containing the directory entry in the size.

Byte 10 Number of sectors used in the last sector of the file. This byte is present only for program files and data files having variable length records. An end sentinel is found at the position in the last sector pointed to by this value. For program files the end sentinel is X' AA'; for data files it is X' FF'.

Byte 11 Maximum number of bytes allowed in a record. This is used by all data files.

Byte 12 For variable data files - number of sectors used for the file (low order byte, equal to byte F).

For fixed data files - number of records in the file (low order byte).

Not used for program files.

Byte 13 For variable data files - number of sector used for the file (high order byte, equal to byte E.)

For fixed data files - number of records in the file (high order byte).

Not used for program files.

Bytes 1C-1F First file segment entry. Since the sectors of a file may occupy one or more separate areas on the diskette, each group of contiguous sectors is represented by the three byte file segment entry. Each file segment entry has the following format:

First byte -Low order byte of sector number of first in the segment.

Second byte -High order nibble (bits 4-7) = low order nibble of number of sectors (-1) in this segment added to this number of segments in all previous segments.

Low order nibble (bits 0-3) = high order portion (for sectors above X' FF') of sector number of first sector in this segment.

Third byte -High order nibble (bits 4-7) = high order nibble of number of sectors (-1) in this segment added to the number of segments in all previous segments. This nibble is used only if the number of sectors exceeds X' FF'.

Low order nibble (bits 0-3) = middle nibble of number of sectors (-1) in this segment added to the number of segments in all previous segments.

The following examples should make clear how this information applies to the directory entries.

1. 4155 544F 4C4f 4144 2020 0000 0100 0006 1700
0000 0000 0000 0000 0000 4851 0000 - This is a program file named AUTO-LOAD.
It occupies 6 sectors, the end sentinel is X'17' bytes into the last sector and the file begins at sector 148.

2. 4120 2020 2020 2020 2020 0000 0803 0002 0050
0600 0000 0000 0000 2210 0000 - This is a write protected data file named A. It is organized as a display format file with fixed length records of 80 bytes. It occupies 2 sectors with 3 records per sector; there are a total of 6 records in the file. The file starts at sector X'22'.

3. 5649 5441 4D49 4E20 2020 0000 0202 0024 0080
4700 0000 0000 0000 0000 5030 0200 - This is a data file named VITAMIN. It is organized as an internal format file with fixed length records of 128 bytes. It occupies 36