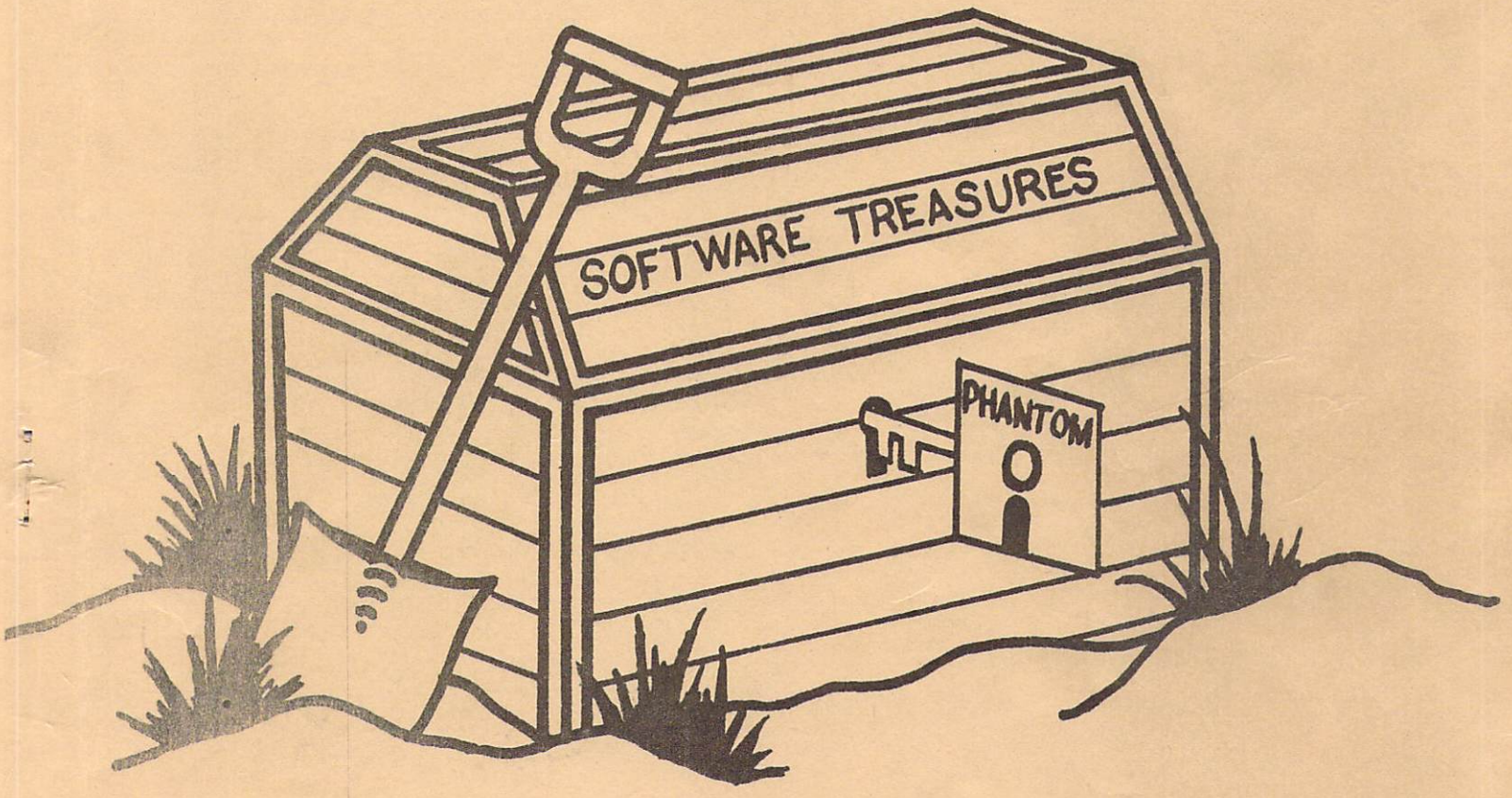
PROGRAM PROTECTION MANUAL
FOR THE C - 64



C S M SOFTWARE

P.O. BOX 563

CROWN POINT, IN

TABLE OF CONTENTS

COPYRIGHT NOTICE

This manual and the computer programs on the accompanying floppy disks, which are described by this manual, are copyrighted and contain proprietary information belonging to T. N. Simstad.

No one may give or sell copies of this manual or the accompanying disks or of the listings of the programs on the disks to any person or institution, except as provided for by the written agreement with T. N. Simstad.

No one may copy, photocopy, reproduce, translate this manual or reduce it to machine readable form, in whole or in part, without the prior written consent of T. N. Simstad.

WARRANTY AND LIABILITY

Neither C S M SOFTWARE, T. N. Simstad, nor any dealer distributing the product, makes any warranty, express or implied, with respect to this manual, the disks or any related item, their quality, performance, merchantability, or fitness for any purpose. It is the responsibility solely of the purchaser to determine the suitability of these products for any purpose.

In no case neither C S M SOFTWARE nor T. N. Simstad will be held liable for direct, indirect or incidential damages resulting from any defect or omission in the manual, the disk or other related items and processes, including, but not limited to, any interruption of service, loss of business, anticipated profit, or other consequential damages.

THIS STATEMENT OF LIMITED LIABILITY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Neither C S M SOFTWARE nor T. N. Simstad assumes any other warranty or liability. Nor do they authorize any other person to assume any other warranty or liability for it, in connection with the sale of their products.

UPDATES AND REVISIONS

T. N. Simstad and C S M SOFTWARE reserves the right to correct and/or improve this manual and the related disK at any time without notice and with out responsibility to provide these changes to prior purchasers of the program.

Although every attempt to verify the accuray of this document has been made, we cannot assume any liability for errors or omissions. No warranty or other guarantee can be given as to the accuracy or suitability of this book or the software for a particular purpose, nor can we be liable for any loss or damage arising from the use of the same.

DISK DIRECTORY

MOVE BASIC	PRG	1
DISK CHECKER	PRG	3
ID CHECKER	PRG	4
APPEND	PRG	1
BLOCK AL & FREE	PRG	3
DISK ADDR CHANGE	PRG	4
DISK DR	PRG	23
BACKUP 228	PRG	37
TEST #1	PRG	2
KEYBRD BUFFER	PRG	1
ERROR CHECK	PRG	2
USER FILES	PRG	3
USER.WOW	PRG	35
SUPER LINES	PRG	3
ML & BASIC	PRG	4
ML & BASIC #2	PRG	4
U1 & U2	PRG	8
MOD POINTERS	PRG	1
SPEED COPY	PRG	11
TEST ML	PRG	3
RESTORE	PRG	1
LLMON SYS8192	PRG	17
HIMON SYS49152	PRG	17
LOMON SYS32768	PRG	17

THE DIRECTORY OF THE DISK HAS BEEN MODIFIED TO PREVENT YOU FROM LISTING IT. IT WILL BE NECESSARY TO RESET THE TRACK AND SECTOR POINTERS TO MAKE THE DISK LISTABLE. THE POINTERS HAVE BEEN CHANGED TO PLACE THE DIRECTORY IN AN ENDLESS LOOP.

OTHER MODIFICATIONS HAVE BEEN MADE. SEE IF YOU CAN FIND THEM ALL. THE CHAPTER ON THE DISK DRIVE WILL DOCUMENT ALL THAT NEEDS TO BE CHANGED. GOOD LUCK!

THE VERSION OF DISK DR CONTAINED ON THIS DISK HAS BEEN SLIGHTLY MODIFIED TO MAKE IT WORK ON DISKS WILL ENDLESS LOOP DIRECTORIES.

ALL THE PROGRAMS CONTAINED ON THE DISK ARE PUBLIC DOMAIN. FEEL FREE TO DO WITH THEM AS YOU WISH.

SOFTWARE LAW

The purpose of this chapter is to inform the user of the C-64 computer what they may and may not do with the programs they have purchased. I am not a lawyer and I am not trying to give legal advice. What I am trying to do is make the average user more aware of some of the legal aspects of software law. If you have any specific questions go to your own lawyer or a lawyer who specializes in software law.

Programs may take many forms. They may be purchased on disks, cassette tapes, cartridges or stringy floppies for the C-64. The only difference between a blank disk and a word processing program is a small amount of magnetic information that has been placed on the disk. Usually the magnetic information can be placed on the disk in a matter of minutes. With todays high speed copy machines, programs may be duplicated within a matter of minutes. This will include the time necessary to verify the disk. This will make certain that the disk will perform as expected.

Many programs take thousands of hours to develop. A good program will need a great amount of time to develop and debug. Anyone who has written even a simple program in BASIC can verify this fact. Consider the time required necessary to write a good data base or a good word processor. Often times the program will be developed by a group of programmers, all working together to finish the program. Each programmer may be a specialist in a particular aspect of the program. How can a programmer make any money if it takes months to develop a program and only minutes to copy for a software pirate to copy?

Two methods currently exist to protect the program from unauthorized copying. Both of which offer the programmer some amount of protection for his software. First is the legal method, this is the law of the country where the program is used. Second is the copy protection method, this is the method that the programmer uses to actually prevent unauthorized duplication of the software. In this chapter I will cover a few of the more popular legal ways of protecting computer software.

The Congress of the United States has passed a number of laws to protect the author of a computer program. There are many ways that a programmer may legally protect his software from being copied.

1. Trade Secret:
Trade secrets will protect the program as long as the program is kept a secret. If you keep your program secret and the code that makes the program work a secret, you have the best protection of all. The difficulty comes in when you try to sell the program to the customer. If you don't require the customer and all the users to sign a non-disclosure agreement, your trade secret status may be lost. Trade secrets work well during the development phase of the program, but they are impractical if the program is to be mass marketed.

2. Patent Protection:
Copyrights only protects the expression of an idea, whereas a patent will protect the idea itself. If your program is granted a patent, you will have a seventeen year monopoly on your idea. This sounds like it might be the ideal way to protect your program. Right?

WRONG! Patents many times take two or more years to obtain, your program may be obsolete before it has patent protection. Also the patent office may be unwilling to provide your program with a patent.

3. Trademarks:
The trademarks can only protect the name of the program, not the program itself. If your program has a good name, you will want to use a trademark to prevent anyone else from using the same name on their products.

4. Copyright:
A copyright will protect the expression of an idea, not the idea itself. Although this last statement may sound confusing, it really is easy to understand.

Most lawyers agree the best legal protection for your software is through the use of the copyright protection laws. In recent years the copyright laws have been updated and protection has been specifically extended to computer programs. This coverage will apply if the program is on a disk, cassette tape, cartridge or part of the internal ROM memory of the computer.

I stated earlier that the copyright will protect the expression of an idea, not the idea itself. Let's look at this example. You, as a software author are working on word processing program. This is to be the best word processing program ever made. It will have all the functions of any other word processor plus a few new ideas of your own. While you are writing the program, you make every effort to insure that no one gets a copy of your code, thereby insuring your trade secret protection is maintained. Once the program is finished you copyright the program and begin to market the program. A few weeks later you find out that someone else has just marketed a new word processing program, this program has every feature that your program has. The two programs are very similar and perform all the same functions. Could this be a case of copyright infringement? Possibly, or it could be the case of two programmers simultaneously creating similar programs. Even though the programs appear to be similar, they have been created independently of each other. The other program may perform the same functions that yours does, but it does it in a different way. It is not what the program does, it is how it does it. Thus the statement: A copyright will protect the expression of an idea, not the idea itself.

Let's take a look at another example. You develop a word processing program. A software pirate buys a copy of your program. He changes the name and a few lines of code. The pirate then sells the program as his own. This is a clear cut case of copyright infringement. One can not just change a few simple lines of code and say that they are the author. If you take the pirate to court it would be an easy case to win. The program would have to be substantially different from your program in ordered to be considered unique.

The copyright is automatically born when the program is created and transferred from you to paper, disk of other media. You have up to five years to perfect your copyright with the Copyright Office. When you wish to perfect the copyright, you must follow a few simple steps. First, you need to place the proper copyright notice in a conspicuous place, you must file the proper form with the Copyright Office, send in a check for Ten dollars, the first twenty five pages and the last twenty five pages of your program. It would be advisable to contact your lawyer for further information on how to proceed.

You, as a software author, have copyrighted your program and have done it properly. What is to prevent some one from copying your program? The copyright law states that anyone who willingly copies your program is in violation of the law. They don't have to sell your program to violate the law, they only have to copy it to be in violation. The law does provide for the owner of the program to make one copy for archival purposes only.

If you find that someone has violated the law and is copying your program you can sue that person. You may recover any actual damages that you incurred, your attorneys fees, court costs and whatever other damages the court wishes to order. You may also request an injunction to prevent the pirate from any further copying of your program.

Your local library is a good source of reference on computer software law. Many books have been written on the subject in the past few years. Try to get the most recent one, because the law is changing almost daily.

5. Limiting liability:
This form of protection may very well be the most important for the software author. By limiting his liability the software author can protect himself from unhappy or dissatisfied customers. The personal computer is covered by consumer protection laws. Any time the consumer purchases a software program (or most any item) certain warranties go with.

Three types of warranties are: express warranty, implied warranty of fitness and implied warranty of merchantability. The express warranty is created by the wording of the program or a salesmans words (i.e. "This program will sort five thousand files in two seconds"). If the product won't do it, it shouldn't say it. The implied warranty of fitness only comes into play if a salesman states that the program will fulfill his needs and the customer buys the product based upon the salesmans recommendations. Again

if the product won't do it, don't say that it will. The implied warranty of merchantability states that the product is as good as any one elses. This warranty is created automatically when your program is sold.

Why then, is limiting liability the most important type of protection for the software author? Because if you, as the software author, don't properly disclaim each and every warranty, the author or seller may be open to a lawsuit if the product does not perform as the buyer expected it to. In many states the disclaimer must be placed in a conspicuous location, visible without opening the package, in order to be valid. If you put the disclaimer in the wrong location, it may be considered void. Contact a lawyer for specific information on limiting your liability if you are considering writing programs.

ARCHIVAL COPIES

The dictionary defines the word ARCHIVE as follows: The place where records or papers of historical interest are stored. The meaning in the computer industry has taken a slightly different turn. An archival copy of a program is a duplicate program that is stored in a safe place, to be used in the event that anything should happen to the original. Software laws today provide for the owner of a program to make an archival copy of the original program. It is your right to make a copy of any program that you purchase. You also have the right to modify the program that you purchase, providing that you don't make copies of the original or the modified programs for anyone else.

I think that we all have purchased a program, got it home and found that the program did not suit our needs. Sometimes the program only needed a small change to suit our particular needs. Other times the program was junk and we just wasted our money. If you wish to modify the program, you may do so. You may not give copies of the modified program to your friends. It is still protected by copyright laws. Changing a few lines of code or renaming the program will not let the purchaser usurp the copyright law.

Software stored on disk or tape is highly susceptible to damage. Should the original copy of a program become unuseable for any reason, the user only has to go to his archives and retrieve the archival copy and he is back in business.

How does one obtain an archival copy of a program? Some software companies provide a backup program for a nominal fee. Others do not. They leave it up to the individual to make his own copy. In the interest of preventing software piracy some companies make their software virtually uncopyable.

What is the owner of the program to do? The manufacturer will not supply a backup and the program has a great deal of protection built in to prevent illegal copying. This protection prevents the legitimate person from making an archival copy. It seems some software companies want people to 'break' their programs in order to obtain a backup copy. 'Breaking' a program refers to removing all the protection schemes from a program. 'Breaking' a program will allow the program to be copied by any convenient method. The broken program will perform exactly the same as the original. The only difference between the original and the broken version is the program protection. In some ways it seems that software companies are encouraging piracy, by forcing the end user to break a program in order to obtain an archival copy. Once the program is broken anyone can copy it and many times they do (illegally of course). Remember, once you have purchased the program it is yours, to do with as you wish. You may modify the program, you can change the program, you can even sell the original version of the program if you wish. You may NOT make copies of the program to give or to sell to other people, that is illegal.

I have just received a copy of a program that will copy almost any disk, errors and all. It takes less than five minutes to make a copy of a full disk and, in most cases, will make an exact duplicate of the original program, including any errors. The major problem with the copy program is that the copied program will perform just like the original.

You might ask why I think that this is a problem. If the original disk used 'bad blocks' the copy will use 'bad blocks'. 'Bad blocks' is a type of program protection that will literally beat your disk drive to death when the program loads in to memory. The programmer will intentionally write a bad block on the disk. This bad block does not contain any inform0tion, its only purpose is to generate an error when the disk drive tries to read the block. The disk drive will make a loud banging sound when it tries to read this bad block. This banging results from the cam (that moves the read/write head) bumping against its end stop. This bumping can be very hard on the disk drive. Many disk drives have been knocked out of alignment while trying to read a bad block.

It is the program author's right to protect his software from unauthorized duplication. It is your right to protect your disk drive from being beat to death. You have the right to protect your investment from being rendered useless. It is your right to 'fix' the program so that it will not beat your drive to death. You also have the right to make an archival copy of your programs. Don't let a protected programs keep you from having the copy you need.

COPY PROTECTION

Copy protection refers to the methods that a software author uses to protect his program from unauthorized duplication. These methods range from the simple to the bizzare. Most often copy protection is an afterthought. The software author will spend weeks or months writing a program. Then he usually spends a few hours protecting his work. I have seen programs that have taken literally thousands of hours to write, then the author spends thirty minutes on the protection scheme.

Programs on cassette may be protected by several methods. The program may be stored on the tape in several parts. Each part will load the next part. Information may be stored in such a way that it may be difficult to copy with only one cassette player. There is not a lot one can do to protect software saved on cassette. There are a few firms which make an interface which will allow the user to copy any cassette based program to another cassette. These interfaces will make exact copies of the original. When one considers the cost of such an interface, it will provide the most economical method of program duplication. Find a friend who has a cassette player and share the cost of the interface.

Disk based programs can not be copied as easily as cassette based programs. If they could there would not be any need for this book. Programs stored on disk have more options as to their copy protection. The BLOCK ALLOCATION MAP (BAM) may be modified. The DIRECTORY (DIR) can be hidden from the user or it may be modified to prevent the user from listing the directory. Special information may be stored on the disk in such a manner that it may not be easily retrieved by the average user. Many different types of errors may be intentionally placed on the disk. These errors will be checked by the program as it runs. If the error is of the proper type and at the proper location the program will execute. If some one makes a copy of the original disk and does not place the errors on the duplicate disk the program will not run. Disks may be formatted on a disk drive that is not totally compatible with the 1541. The program will load and run properly, but duplicates can not be made on the 1541 disk drive.

Information is stored on the disk in what is called a BLOCK. There are 683 blocks of information that may be used on the 1541 drive. Each block may contain up to 256 BYTES of information. In addition to the 683 blocks, the disk will also contain some special information, including SYNC MARKS, ID numbers, CHECKSUM, the TRACK and SECTOR numbers and another CHECKSUM. The disk drive uses this special information to process and identify the block. This special information is referred to as the HEADER. Some software manufacturers will modify the 'header' in such a fashion that this block of information is no longer readable by the disk drive. Once a block has been modified in this manner it is referred to as a BAD BLOCK. Generally a bad block does not contain any information, it is just there to create an error when the disk drive tries to read it.

I am sure that you have all tried to load a disk that has contained a bad block. While the program is loading the red light will flash and the disk drive will make a loud banging noise. This noise is generated by the disk drive when it tries to read the bad block. The disk drive can not read the information contained in the header. When this occurs the disk drive will mechanically re-position the read/write head. To do this it is necessary to pound the stepper motor cam against its end stop. The read/write head of the disk drive is attached to the stepper motor cam. When the bad block is encountered, an error will be generated and the read/write head will literally get beat to death. In other words, if the disk drive tries to read a bad block the read/write head will pound against the end stop in an attempt to retrieve the information from the disk.

I know that all of you have heard that there is a problem with the 1541 disk drives going out of alignment. Reading and writing bad blocks is a major contributor to this mis-alignment. Why would a software manufacturer put bad blocks on a disk when it may tear up the disk drive when their program tries to read the bad block??? Because he cares more about protecting his program from pirates than he does about your disk drive. If your disk drive gets beat to death trying to read his program, that's your problem (or so they think).

While I am on my soap box, I would like to tell a little story that happened to me. About six months ago I purchased a protected program (cost $95.00). After using this program for less than two months the program developed a flaw in it (due to its protection scheme). After contacting the manufacturer, I was told to send in the original program disk and they would send me a new copy (for $12.00). The trouble was that I needed the program and could not afford to wait two or more weeks, as they requested. It was necessary for me to modify the original disk so that it could be returned to working condition and it was also necessary to repair my drive.

In an effort to prevent any one from making a copy of the disk the company used a technique called bad blocks on the disk. As you all know, when the disk drive tries to read a bad block the drive makes a loud banging noise. This noise is a direct result of the drives stepper motor cam pounding against a stop. This pounding can be very harmful to the disk drives' mechanical parts. After the drive mechanism pounds against the stop enough times, the drives' stepper motor will become mis-aligned. The read/write head, which is attached to the stepper motor, will be beat out of alignment and the disk drive will no longer be able to read or write any information from the disk.

On this disk I purchased, the program would read a portion of the program into memory, modify it and re-write the information back to the original disk. While loading the program, the disk drive made an unusually loud and hard clicking noise (bad blocks were used). After which the disk drive had a hard time reading the information from the disk. After the program had run and all the information had been processed, the program attempted to write

the information back to the disk. After partially writing the information, the program stopped. The disk drive head had been knocked out of alignment when the program tried to read the bad block. My disk drive was damaged and the program was rendered useless, even when used on a good drive. The company's protection scheme prevented me from making a backup copy of the program and my drive was made useless.

I cannot begin to tell you of all the people who, after trying to load one of these protected programs, have had their disk drives damaged. If you have not had your disk drive beaten out of alignment, just wait. Your turn is coming!

For those of you who have had your disk drive beat out of alignment, you may be interested to know that I have just completed writing a disk drive alignment program. This disk drive alignment program requires no special equipment. No oscilliscope, no strobe light and most importantly you don't have to be a electronics wizard to use this program. The program is contained on one disk and a specially prepared calibration disk is supplied with the program. The calibration functions appear on your TV or monitor screen. Also I have included instructions for a 'fix' that has prevented many disk drives from ever going out of alignment again.

Cartridge programs may also be copy protected. The fact that the program resides on a cartridge, is copy protection enough for most people. Down loading the cartridge to disk (cassette) can usually be accomplished very easily. The information stored on the cartridge may then be loaded in the normal fashion and executed. More on this in the chapter on cartridges.

BASIC PROTECTION

Most BASIC programs are protected by a few simple POKEs. In order for the numbers to be poked into memory the program must be RUN. Following is a list of the most common locations and their original values, their function and their protection values.

Memory Location	Original Value	Function Affected	Protection Value
1	55	oper. system	00
43	01	bot of mem. low byte	20
44	08	bot of mem. high byte	09
192	00	tape interlock	
197		value of current key	
198		# of char. in buffer	
631-641		keyboard buffer	
649	10	keyboard buf. size	
775	167	eliminate list	200
808	237	eliminate stop	239
808	237	stop restore list clock	225
808	237	stop restore list	230
818	237	eliminate save	32
828-1019		cassette buffer	

Load the program called TEST #1 from the program disk. Run the program, then list it. This is a very simple BASIC program, it does not contain any protection. Then try changing the values at the locations listed above. First POKE 1,0 press 'RETURN'. The computer has just had its complete operating system turned off. The microprocessor thinks that there is no longer a BASIC interpreter or KERNAL. The value can not be restored to its original value from BASIC. The system has just crashed. To reset the computer turn the power off and then on again.

To see the effects of the other memory locations, try poking the other protection values into their memory location. First load the TEST #1 program and run it. Then list it to be sure every thing is all right. Then POKE 808,230 'RETURN'. Try to list the program. You will not be able to get a proper listing of the program any more, the STOP and RESTORE keys will be disabled. The program will function normally in the run mode. RUN the program and try to stop it by pressing the RUN/STOP and RESTORE keys. Many programs written in BASIC will alter the values at location 808 in an effort to prevent you from listing or stopping the program. After the program has ENDed it will be possible to restore the computer to normal operation. POKE 808, 237 'RETURN'. Once you have restored the protection value to its original value try another. Try all the memory locations that have a protection value listed. See what effects are caused by changing some values stored in memory. You can not damage your computer by POKING values into memory, at the very worst you may have to turn the computer off to regain control.

If you should see a program that pokes values into the memory locations of the cassette buffer, the program is probably storing a short machine language routine in the cassette buffer. These routines normally have nothing to do with program protection. They are use to speed the BASIC program by doing certain functions in MACHINE LANGUAGE (ML). Certain functions of your computer may be accomplished hundreds or even thousands of times faster in ML than in BASIC. To use this ML subroutine, BASIC has the SYS command. The SYS command will turn control of the computer over to the ML subroutine. When the ML subroutine is done performing its function, the control is returned back to BASIC.

When numbers are being poked into locations 631-641, many times the computer is getting ready to load another program into memory. 631-641 is the keyboard buffer, location 198 contains the number of characters currently contained in the buffer. The proper values must be stored in the keyboard buffer and the number of characters is stored in location 198. BASIC will then be given the END command and the characters in the keyboard buffer will be printed on the screen. Their function will be executed by printing the return command as the last character printed to the screen (CHR$(13)). Load and run the program called KEYBRD BUFFER, you will see how this works. To find out just what the program is doing you will need to convert the numbers that are being poked into the buffer to their ASCII equivalents (see the memory map section for ASCII equivalents).

Another way to protect BASIC programs is through the use of a character that BASIC does not understand. When the BASIC interpreter comes to this character it will cause a ?SYNTAX ERROR. We can not have this error occur while the program is running. It must only appear when the program is listed. Try entering the following lines:

```
10 PRINT "THIS IS A TEST OF LINE 10": REM   'RETURN'
20 PRINT "THIS IS A TEST OF LINE 20": REM   'RETURN'
30 PRINT "THIS IS A TEST OF LINE 30": REM   'RETURN'
```

RUN the program to be sure it will execute properly. Then after each REM type the following graphics character: shift L (hold the shift key while pressing the letter L) 'RETURN'. Try running the program, then list it. You will now get a ?SYNTAX ERROR every time you list the program. BASIC does not understand the function of the shift L character. When the program is run the REM statement will protect the character from generating a ?SYNTAX ERROR. When the program is LISTed BASIC will try to interpret this character. It can not, so a ?SYNTAX ERROR is generated. To remove the graphics character that causes the SYNTAX ERROR, try listing the program, move the cursor up to the line that contains the error, then press 'RETURN'. List the program again and the line will list properly. You will have to move the cursor up to EACH line that contains an error and press 'RETURN'. Do this at every line that contains the graphics character and the program will list properly.

Another way to keep people from listing your complete program is through the use of multiple delete characters inserted immediately after the statement. This line will list properly then be immediately deleted. This will happen so fast that you will probably not be able to notice it happening. If this technique is used in a large program, a few important lines may be deleted. This technique is not hard to do, but it can be a little tricky to perform the first time. Follow the instructions to the letter. Press 'RETURN' only when instructed to do so. Type in the following line:

```
10 POKE 53281,0:PRINT" "
```

DON'T press the 'RETURN' key just yet. Delete the last quotes sign (") with the 'DEL' key. Hold the 'SHIFT' key down, hold the 'DEL' key down for 3 seconds. The cursor should not move, but it should blink very fast (if the cursor moves, stop what you are doing and start over from scratch). Release both keys and press the 'DEL' key 30 times. The line should not be deleted, the reversed 'T' symbol should be printed. Now press 'RETURN'. Try to list the line. If you look very carefully you will just barely see the line flash on the screen and then be deleted. Run the program and you will see that the line does in fact still exist. This line will function properly, but it will be difficult to list. Try placing a 'hidden' line in the middle of a large listing and the line will become virtualy invisible. To remove a 'hidden' line from a program requires you to delete the entire line, then re-type the line without the deletes. If you run into a program that you suspect to have this type of line in it, try listing the program to your printer to find the 'hidden' line.

There are many different ways to make BASIC lines unlistable. Modification of the quotes mode can usually be found by listing the program to the printer. Then you can delete the line that contains the modified code. Re-type the line minus the special characters. That's all there is to it.

Moving the start of BASIC pointers is a method of 'hiding' a BASIC program. The start of BASIC pointers are contained at locations 43 and 44. The start of BASIC address may be calculated (in decimal) by the following line:

```
10 PRINT PEEK (43) + 256 * PEEK (44)
```

If the pointers have not been moved the value returned after running line 10 will be 2049. This means that the start of BASIC is at the memory location 2049. BASIC RAM memory actually starts at 2048, but the first byte of BASIC memory must contain a 00. Programs written in BASIC must have every line preceded by a 00, so the actual BASIC program will start at 2049. The start of BASIC memory can be moved higher with the following program:

```
5 X = 4096: REM NEW START OF BASIC
10 N = X - 1: REM NEW START OF BASIC - 1
20 H = INT (X / 256): REM HIGH BYTE
30 L = X - H: REM LOW BYTE
40 POKE N,0: POKE 43,L: POKE 44,H: CLR: NEW
```

This program is contained on the disk under the name of MOVE BASIC. Very few programs use this type of program protection. Occasionally a BASIC program will be relocated higher in memory. The memory below the BASIC program will contain a short ML routine. It will appear to the novice that the program has been written entirely in ML. A little later I will cover how to identify these programs. This will be covered in the chapter on ADVANCED PROTECTION SCHEMES.

Many programs will contain errors on the disk. These errors have been intentionally placed on the disk. When the program runs it will check the disk to see if the error is present. If the error is present the program will execute normally. If the error in not present the program will crash. This way if some one makes a copy of the original disk and does not have the capibilities to put the error on the copy disk, the program will not run. The following routine is typical of a error checking routine written in BASIC:

```
10 OPEN15,8,15,"IO:": REM  OPEN ERROR CHANNEL AND INITIALIZE
DRIVE

20 OPEN5,8,5,"#": REM  OPEN CHANNEL FOR DATA. IT IS NECESSARY TO
OPEN BOTH CHANNELS TO THE DRIVE.

30 PRINT#15,"B-R";5;0;1;3: REM  PERFORM BLOCK READ; USE CHANNEL
5; DRIVE 0; TRACK 1; SECTOR 3;: DISK DRIVE WILL TRY TO READ A
BLOCK OF INFORMATION FROM THE DISK

40 INPUT#15,A$,B$,C$,D$: REM  READ THE DISK DRIVE ERROR CHANNEL

50 IF VAL (A$) = 21 THEN 100: REM  CHECKS TO SEE IF ERROR TYPE
IS A #21. IF IT IS GOTO LINE 100 (START OF PROGRAM)

60 SYS 64738: REM  SYS 64738 WILL CAUSE THE PROGRAM TO CRASH AND
THE COMPUTER TO RESET ITSELF.

70 REM  THE ORIGINAL DISK HAD AN ERROR #21 AT TRACK 1, SECTOR 3.

80 REM  IF THE COPY DISK DOES NOT HAVE THIS ERROR THEN THE
PROGRAM WILL CRASH

90 REM  IF THE COPY DISK HAS THE SAME ERROR AS THE ORIGINAL THE
PROGRAM WILL RUN

100 CLOSE5: CLOSE15 : REM  CLOSE CHANNELS TO DISK DRIVE AND
BEGIN NORMAL PROGRAM
```

This is a typical listing of how a program will check to see if the disk has the proper error in the proper place. The error may be on any track or sector on the disk. Line 30 may be changed to the desired track and sector. The error type may be any error which is capable of being placed on the disk. The desired error type may be set in line 50. The most common error types are 20, 21, 22, 23, 27 and 29. Very few programs will have their error checking routine accessible to you. It may require a little work to find the routine. Some will try to hide the error checking routine with the methods used above. Others may use more sophisticated methods that I will cover in later chapters.

Most programs will use a variation of the error checking routine in one form or another. It does not matter if the program is written in BASIC or ML the same general format must be followed in order to read an error. This is a very important concept, be sure that you thoroughly understand it before proceeding. In the following chapters, I will refer back to the error checking routine many different times.

If the preceding example was part of an actual program, there are a few methods that can be used to defeat the error checking routine. The first thing to do is to make a copy of the disk with BACKUP 228. This will give you a copy of the original disk without any errors on the copy. Then load in the program that contains the error checking routine. Now let's look at the possible ways to defeat the error checking. Any one of the methods listed below will work equally well.

1). Change line 10 to: 10 GOTO 100: REM THIS WILL BYPASS THE ERROR CHECK ROUTINE COMPLETELY

2). Change line 50 to: 50 GOTO 100: REM THIS WILL CHECK FOR AN ERROR. IT DOES NOT MATTER IF THE ERROR IS PRESENT OR NOT, THE MAIN PROGRAM WILL BE EXECUTED

3). Change line 50 to: 50 IF A$ 21 THEN 100: REM IF THE ERROR IS NOT PRESENT THE PROGRAM WILL EXECUTE. THIS IS THE METHOD I PREFER TO USE

4). Delete line 60 completely: REM NOW THE PROGRAM WILL 'FALL THRU' TO LINE 100 AND WILL EXECUTE PROPERLY

Any of the above methods will work. It does not matter which line is changed. The important thing to remember is that the error checking routine must be disabled in one fashion or another. It will be up for you to decide on how to modify the program. If you can not decide on how to modify the program, just try any method that seems to make sense. Keep track of what you did, and what effects it had on the program. If the first item that you changed does not give the desired results, then change the code back to the original version. Try something else, don't be afraid to experiment.

DISK DRIVE OVERVIEW

The 1541 disk drive will format the new disk to be read and write compatible with may other Commodore (R) disk drives. The proper syntax to format a disk is:

```
10 OPEN 15,8,15,"IO:"                'RETURN'
20 PRINT#15,"NO:name of disk,ID"     'RETURN'
30 CLOSE 15                          'RETURN'
```

Whenever you open a channel to the disk drive be sure to initialize the drive ("IO:"). This will reset the disk drive to the same condition as if you just turned the power on. To achieve a properly formatted disk a loud clicking sound should be heard from the drive during the first few seconds of the formatting procedure.

In order to properly communicate we first need to understand the meaning of a few technical terms. Following is a list of terms that I will use in discussing the disk drive.

DEFINITIONS: Refer to pages 54 thru 58 of your Disk Users Manual:

BAM Block Allocation Map - how many blocks of information have been used and how many are available for use.

Block The area on a disk where information is stored. There are 683 Blocks, each capable of holding 256 bytes of information. The term block refers to a specific Track and Sector on the disk. A block is where the program data is stored

Byte A numeric method of storing information in the computer's memory or on the disk. One byte is required to store each letter or number in the computer's memory. All letters, numbers, graphics, symbols and punctuation marks are stored in the computers' memory as a number. The numerical equivalents are contained in a chart provided in the section on memory maps. A byte must be two digits (i.e. $00, $2A, $C0, $FF).

Directory A listing of each file (program, sequential, user relative) contained on the disk. The directory also contains the location of the track and sector on which the program starts and how long the programs is.

DOS Disk Operating System. This controls the internal workings of the disk drive. This will include the microprocessor and associated memory contained in the disk drive.

File A file is a group of blocks of information. Information may be stored on a disk in Program files, Sequential files, User files, Relative files, Random files or the Directory file. The disk files are similar to the files contained in a file cabinet, they contain any information that you wish to store in them.

Format Most small computers use the same floppy disks. The only difference between the disks is the way that the disk drive stores the information on the disk. The method that each disk drive uses to store its information is called the format. When a disk is formatted the disk is completely erased, a new I.D. number is placed on each sector and the disk is re-named.

Header The header is the part of a sector that contains the disk I.D., checksum, sync marks and other special information that the disk drive needs. The header and the block make-up one sector.

Hex Hexidecimal. This is a numbering system based upon the number 16. This system uses 16 different digits, whereas the decimal system uses 10. Hex is a convenient numbering system to work in when you are using the computer.

RAM Random Access Memory: This is the part of your computers memory that may be changed to suit a particular need (Games, Word Processing, etc.). Ram will contain the BASIC program or the ML instructions to perform specific tasks.

ROM Read Only Memory: This is the part of your computers memory that is a permanent part of the computer. ROM cannot be changed, modified or erased. The ROM in your computer allows you turn on the computer and begin typing, it also controls most of the internal functions of the computer. ROM may be thought of as the computers brains.

Sector A subdivision of a track. Each track is divided into many smaller parts, each part is referred to as a sector. The sector will contain the header and the block. It is where the disk drive will store the information. The sector will also contain the I.D. number of the disk, error checks (checksum), sync marks and its special identification numbers. The number of sectors per varies with the size of the track. Outer tracks, 1 thru 17, have 21 sectors, tracks 18 thru 24 have 19 sectors, tracks 25 thru 30 have 18 sectors and tracks 31 thru 35 have 17 sectors.

Track A concentric ring (circle) used for storing information on the disk. There are 35 tracks on the 1541 format. Track 1 is the outer most, track 18 contains the BAM and directory and track 35 is the innermost.

In the following pages we will take an in depth look at the BLOCK ALLOCATION MAP (BAM), the DIRECTORY and other associated information that is stored on TRACK 18. TRACK 18 is very special on the 1541 disk drive. One byte of information can be changed on this track and this will prevent any one from ever writing to the disk again. We can determine how many blocks of information have been used and which ones. The name and I.D. number of the disk can be found on this track. The names of all the files that have been stored on the disk can be found here. How long each file is and where the file starts may also be found on track 18. Track 18 is very special and time should be taken to understand the information contained on this track!

```
TB/EDITOR  DISK=8J TRK=18 BLK=0J MODE=H
POINTERS
   0 1 2 3  4 5 6 7  8 9 A B  C D E F
 TR SE      TRACK#1   #2        #3
J0 12014100 15FFFF1F 15FFFF1F 15FFFF1F
J1 15FFFF1F 15FFFF1F 15FFFF1F 15FFFF1F
J2 15FFFF1F 15FFFF1F 15FFFF1F 15FFFF1F
J3 15FFFF1F 15FFFF1F 15FFFF1F 15FFFF1F
J
J4 0FD5571F 00000000 11FCFF07 0AE0C307
J5 13FFFF07 13FFFF07 13FFFF07 13FFFF07
J6 13FFFF07 12FFFF03 12FFFF03 12FFFF03
J7 12FFFF03 12FFFF03 12FFFF03 11FFFF01
J   TRACK 32 TRACK 33 TRACK 34 TRACK 35
J8 11FFFF01 11FFFF01 11FFFF01 11FFFF01
J9 54455354 20444953 4B20A0A0 A0A0A0A0
JA A0A04944 A03241A0 A0A0A000 00000000
JB 00000000 00000000 00000000 00000000
J
JC 00000000 00000000 00000000 00000000
JD 00000000 00000000 00000000 00000000
JE 00000000 00000000 00000000 00000000
JF 00000000 00000000 00000000 00000000
```

BLOCK ALLOCATION MAP

4 BYTES FOR EACH TRACK

```
TB/EDITOR  DISK=8J TRK=18 BLK=0J MODE=A
   0 1 2 3  4 5 6 7  8 9 A B  C D E F
J0 . . A .  . R R .  . R R .  . R R .
J1 . R R .  . R R .  . R R .  . R R .
J2 . R R .  . R R .  . R R .  . R R .
J3 . R R .  . R R .  . R R .  . R R .
J
J4 . R W .  . . . .  . ± R .  . J ± .
J5 . R R .  . R R .  . R R .  . R R .
J6 . R R .  . R R .  . R R .  . R R .
J7 . R R .  . R R .  . R R .  . R R .
J
J8 . R R .  . R R .  . R R .  . R R .
J9 T E S T     D I S K    J J  J J J J
JA J J I D  J 2 A J  J J J  . . . .
JB . . . .  . . . .  . . . .  . . . .
J
JC . . . .  . . . .  . . . .  . . . .
JD . . . .  . . . .  . . . .  . . . .
JE . . . .  . . . .  . . . .  . . . .
JF . . . .  . . . .  . . . .  . . . .
```

THIS BYTE SHOULD BE AN 'A'

DISK NAME

FIG. 1

Refer to figure #1 for the following explanation:

This is a print out of the information contained on track 18 sector 0. This block contains the BAM. Below is a listing of the first line of the print out, all numbers are in hexidecimal. Starting in the upper left hand corner the first byte is number 0, not number 1. The last byte is number 255, not the number 256.

```
LOCATION 00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
VALUE    12 01 41 00  15 FF FF 1F  15 FF FF 1F  15 FF FF 1F
```

The bytes of information have been specifically arranged to provide easy identification. Starting at the left, location 00, we find the hexidecimal number 12, this equals 18 in decimal. The first number contains the pointer to the next track of the file, in this case the next block of information is located on track 18. The second byte (location 01), hexidecimal number 01 equals 1 in decimal, is the pointer to the next sector in the file (sector 1). From the first two bytes in this file we can determine where the next block of this file is located, track 18, sector 1. These first two bytes are pointers. The next byte (location 03) is very important. This is the hexidecimal number 41, decimal 65. The ASCII equivalent of this byte is the letter 'A'. This byte must be an 'A' or an error message will be generated: 73,DOS MISMATCH,18,00. This byte is checked every time a disk is inserted into the drive. If this byte is changed to the letter 'E' (hex 45, decimal 69) the disk will become permanently write protected. The disk drive will no longer be able to write any information to the disk. This 'E' fools the disk drive into believing that the disk was formatted on a disk drive that is not completely compatible with the 1541. Commodore makes many different disk drives. Some of these drives are completely read and write compatible with the 1541. Some drives are read compatible only, meaning that the information stored on the disk from one drive will be able to be read by the other drive. The two disk drives will not be able to write any information to the disk formatted on the other drive. The only way that the disk can be written to, is if the disk is re-formatted. Formatting will completely erase all the information from the disk, including the 'E'. The fourth byte of information is hex 00 (decimal 0), this byte is unused and will normally be set to 00.

The next 140 bytes contain the Block Allocation Map (BAM). Each group of four bytes contains the BAM for one track. Below is the BAM for Track 1 (location 04 - 07):

Location	04	05	06	07
Hex	15	FF	FF	1F

The hex number 15 equals 21 in decimal. This number contains the total number of blocks free in this track. The next three bytes contain a bit map for the track. Bits refer to the individual 1's and 0's that make up the hex number. Example:

Location	04	- 05 -		- 06 -		- 07 -	
Hex	15	F	F	F	F	1	F
Bit map		1111	1111	1111	1111	0001	1111

Track 1 has a total of 21 Sectors available. There is a total of 21 1's contained in the bytes FF FF 1F. If a block of information is used the DOS will change the proper 1 to a 0. This is called allocating a block. The disk drive will do this automatically for you when you save a program to the disk.

When a disk is first inserted into the drive the BAM will read into the disk drive controllers memory. The BAM will be updated every time a program is saved or scratched. Once a block has been allocated the disk drive will not normally write any other information to the block. The allocated block may be freed up by 'SCRATCHING' the file or by the use of the 'BLOCK-FREE' command. Track 35 contains a total of 17 sectors, below is the BAM for Track 35.

Hex	11	F	F	F	F	0	1
Bit map		1111	1111	1111	1111	0000	0001

The hex 11 equals 17 in decimal, 17 blocks are free. The 1's and 0's represent the bit map for the track.

Bytes 144 to 161 contain the disk name. The name of the disk must be 16 bytes long. If the name is not 16 bytes long the disk drive will automatically fill the unused bytes with shifted spaces (A0). Shifted spaces are used as fillers only.

Bytes 162 and 163 contain the disk I.D. The I.D. will be placed in the header of every block and in this location of the BAM.

Byte 164 is a shifted space (A0).

Bytes 165 and 166 contain the ASCII representation for 2A, which represents the DOS version and the format type.

Bytes 167 to 170 are shifted spaces (A0).

Bytes 171 to 255 are nulls (00). These bytes are not normally used, occasionally ASCII characters may be found in these locations.

The BAM may be modified in a number of ways. I will cover the type of modifications that may be performed to the BAM, starting at the location 00 of track 18, sector 0. If you would like to see the effects of the changes, you can modify your own disk. Make the changes to a disk that does not contain any valuable programs. If you should make a mistake, you don't want to lose anything valuable. The ideal disk to practice on is a copy of any disk. That way if you make a mistake nothing is lost, all you have to do is make another copy. If you should make a mistake, don't worry about it. Everyone who has ever worked with the computer has messed up at least once (sometimes more than once). If you have not modified the disk before, this will be good

experience. Make one change at a time, then try listing the directory. Keep a note book of just what you changed, what effects it had and how you returned the disk to normal. This can save a lot of time later, especially when you are looking at other programs that have been modified.

Bytes 0 and 1: These bytes may be changed to point to a different track and/or sector. If you change these bytes be sure to move the directory. I have not seen these bytes changed. I don't recommend that you modify them.

Byte 2: This byte may be changed from the ASCII 'A' to 'E' to prevent the disk from ever being written to again. Once this change has been made the 'E' may not be changed back again. By changing this byte to certain other values you can make the disk both read and write incompatible. One note, if you change this byte the disk must be removed from the disk drive to make the change permanent. The BAM is read into memory when the disk is first inserted into the drive. So, if you don't remove the disk after making the change, the drive will still have the 'A' in its memory. If this byte has been changed to the 'E' many copy programs will 'die' while trying to copy the disk. The copy programs will function normally during the first two passes (until the 'E' is written on the destination disk), then the motor of the drive will stop on the next pass. If this happens to you while making a copy it will be necessary use BACKUP-228 to copy the disk. Start your copy at track 19, block 0. Finish on track 35. Then copy track 1, block 0, finish on track 18. This way the 'E' will not defeat the copy program.

Byte 4: This byte contains the number of blocks free on track 1. If you change this byte to the hex value of FF, you can fool the drive into thinking that the track 1 has 255 blocks free. Try changing the first byte of the BAM on each of the tracks and see how many blocks free you can get. This will only give a false number of blocks free. Even though the drive can be fooled into thinking that there are more blocks free than actually exist, these blocks may not be used. Every fourth byte will starts a new track BAM.

Bytes 5,6&7: The bit map of available blocks. A bit value of 1 indicates that the block is available. A bit value of 0 indicates the block is allocated. By changing these bytes the disk drive will not know which blocks have actually been used. These bytes may be restored to their proper values by VALIDATING the disk.

Bytes 8 thru 143: same as 4 thru 7 in four byte groups. Each group represents the BAM for one track. Bytes 8-11 are for track 2, bytes 12-15 are for track 3, etc.

Bytes 144 thru 161: Contain the disk name and may be changed to any name desired. Illegal characters may be placed in the name by modifying these bytes (i.e. * or ?). This is the only place the name will appear on the disk. Special opcodes may also be placed after the first A0 (i.e. $03 $07). See the section on file names for more information. One of the easiest ways to modify the disk is by changing the first six bytes of the disk name to the hex

values: 14 14 14 00 00 00. The directory will no longer be able to be listed.

Bytes 162 & 163: Contain the disk I.D. and may be changed to any characters desired. Illegal characters may be placed in the I.D. by modifying these bytes. The I.D. will be placed on every sector on the disk when the disk is formatted. These I.D. numbers will not be changed by modifying Track 18, Sector 0.

Bytes 165 & 166: Contain the format designation. These bytes may be changed with no adverse affects on the operation of the disk. If they are changed to unprintable characters (00 03) it may cause problems with the directory listing.

Bytes 167-255: Any modification may be made to these bytes with no adverse affects. This is a good place to store messages to those who are trying to pirate software.

The directory begins on Track 18, Sector 1. The directory may contain up to 144 individual file entries. Each entry will represent one file. The directory will contain information on the type of file, where the file begins on the disk, the name of the file and its length. Refer to figure 2 for the following explanation.

Bytes 0 and 1: The first two bytes of this block contain the pointers to the track and sector of the next block in the file. If this is the only block in the directory file, the first byte will be 00, the second byte will be FF. If this is not the only block in the file, the first byte will point to the track, the second byte will point to the sector.

Bytes 2 thru 31: Contain the information for the file entry #1

Bytes 34 thru 63: Contain the information for the file entry #2

Bytes 66 thru 95: Contain the information for the file entry #3

Bytes 98 thru 127: Contain the information for the file entry #4

Bytes 130 thru 159: Contain the information for the file entry #5

Bytes 162 thru 191: Contain the information for the file entry #6

Bytes 194 thru 223: Contain the information for the file entry #7

Bytes 226 thru 255: Contain the information for the file entry #8

Any bytes not listed are unused. If there are additional blocks used in the directory file, they will be similar to the first one. The only change will be the listings for the file entry. The individual file entries will be similiar to the one below, the only change will be those that are specific to the individual file. The example is from figure 2. All numbers are in hex.

```
TB/EDITOR  DISK=8↵ TRK=18 BLK=1↵ MODE=H

   0 1 2 3  4 5 6 7  8 9 A B  C D E F

↵0 12048211 0050524F 504F5341 4CA0A0A0
↵1 A0A0A0A0 A0000000 00000000 00003200
↵2 00008211 01414343 4F554E54 494E472E
↵3 4332A0A0 A0000000 00000000 00004C00
↵
↵4 00008213 00414D4F 5254274E 20544142
↵5 4C452E43 32000000 00000000 00002700
↵6 00008215 01424F4E 44532E43 32A0A0A0
↵7 A0A0A0A0 A0000000 00000000 00002100
↵
↵8 0000820E 02425544 47455441 43434F55
↵9 4E542E43 32000000 00000000 00003400
↵A 00008216 0843414C 454E4441 522E4332
↵B A0A0A0A0 A0000000 00000000 00002600
↵
↵C 00008218 04435245 44495420 554E494F
↵D 4E2E4332 A0000000 00000000 00002700
↵E 0000820B 00444154 45532E43 32A0A0A0
↵F A0A0A0A0 A0000000 00000000 00002200
```

FILE 1

FILE 2

FILE 3

FILE 4

FILE 5

FILE 6

FILE 7

FILE 8

```
TB/EDITOR  DISK=8↵ TRK=18 BLK=1↵ MODE=A

   0 1 2 3  4 5 6 7  8 9 A B  C D E F

↵0 . . . .  . P R O  P O S A  L ↵ ↵ ↵
↵1 ↵ ↵ ↵ ↵  ↵ . . .  . . . .  . . 2 .
↵2 . . . .  . A C C  O U N T  I N G .
↵3 C 2 ↵ ↵  ↵ . . .  . . . .  . . L .
↵
↵4 . . . .  . A M O  R T ' N    T A B
↵5 L E . C  2 . . .  . . . .  . . ' .
↵6 . . . .  . B O N  D S . C  2 ↵ ↵ ↵
↵7 ↵ ↵ ↵ ↵  ↵ . . .  . . . .  . . ! .
↵
↵8 . . . .  . B U D  G E T A  C C O U
↵9 N T . C  2 . . .  . . . .  . . 4 .
↵A . . . .  . C A L  E N D A  R . C 2
↵B ↵ ↵ ↵ ↵  ↵ . . .  . . . .  . . & .
↵
↵C . . . .  . C R E  D I T    U N I O
↵D N . C 2  ↵ . . .  . . . .  . . ' .
↵E . . . .  . D A T  E S . C  2 ↵ ↵ ↵
↵F ↵ ↵ ↵ ↵  ↵ . . .  . . . .  . . " .
```

FIG. 2

```
Location 00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F

         12 04 82 11  00 50 52 4F  50 4F 53 41  4C A0 A0 A0
         A0 A0 A0 A0  A0 00 00 00  00 00 00 00  00 00 32 00
```

Bytes 0 & 1: The hex 12 equals 18 in decimal. The hex 04 equals 4 in decimal. The next block of the directory file may be found at track 18, sector 4. It is normal for the disk drive to separate blocks of a file by 3 blocks.

Byte 2: This contains the information that determines what type of file is stored on the disk. Remember byte 2 is the first byte of file entry #1. The first byte of all file entries will indicate the type of file. Following is a list of the bytes that would normally be found here and their meanings.

00 = no file or scratched file	82 = program file
80 = no file or scratched file	83 = user file
81 = sequential file	84 = relative file

Bytes 3 & 4: Contain the track and sector of the first block of information in the file. Hex 11 = 17 decimal. Hex 00 = 0 decimal. The first block of data will be found at track 17, sector 0.

Bytes 5 thru 20: Contain the name of the file. The name of the file must be 16 bytes long. If the file name is not 16 bytes long the disk drive will automatically fill the unused bytes with shifted spaces ($A0). The shifted spaces will be used as fillers only.

Bytes 21 thru 23: Will be used for relative files only. These byte are not normally used.

Bytes 24 thru 27: Unused. Will be nulls (00)

Bytes 28 & 29: Used for temporarily storing the track and sector of the replacement file when SAVE with replace is used (SAVE@0:).

Bytes 30 & 31: Contains a running count of the number of blocks in the file. The number is stored low byte (30), high byte (31). Location 30 contains the hex value 32, this is 50 in decimal. Location 31 contains 00. The file is 50 blocks long.

The following file entries will have the same format as the first. Just apply the same rules to the rest of the file entries to determine the necessary information.

The DIRectory may be modified in a number of ways. Most of the ways that the DIRectory can be modified are contained below. Following each modification there is a 'fix' listed. Try each of the following modifications and see what happens when something is changed.

Byte 0 & 1: These locations contain the pointers to the next track and sector of the file. If this is the last block of the directory file the bytes will contain the hex values 00 FF. If it is not the last block of the file, you will need to locate the

last block in the file. Change the first two bytes of the last block in the directory file to the hex values: 12 01 (decimal 18 01). This will point the directory file back to the first block in the directory file. You will have made a continuous loop, the directory will continue to search for the pointers that tell the disk drive that it has found the last block of the file. Instead it will find pointers to the first block of the file, track 18, sector 1. The disk drive will search forever for the end of file marker, but it will not find it. Programs will still load and run properly, but you will not be able to list the directory. To correct this condition you only have to find the last block in the file and reset the pointers to hex values 00 FF. The last block in the directory file should always have the pointers 00 FF.

Byte 2: This byte should not be modified. Some times programers will put bogus (fake) programs on the disk. These programs have no value, they are there only to confuse the software pirates. Do not change this byte unless you are trying to confuse someone or you are trying to delete a file from the disk.

Byte 3 & 4: These are the pointers for the starting block of the file. Do not modify these pointers. The only time they will be modified is when a bogus program has been placed on the disk. These bogus programs will be placed on the disk to prevent pirates from copying the files. The pointers may be changed to put the disk drive in an endless loop. Bogus programs may be deleted from a disk by changing byte 2 to 00. This will 'SCRATCH' the file from the disk.

Bytes 5 thru 20: This will contain the name of the file. The name can be any character(s). Many times the name will be set to a character that BASIC will not easily accept as a valid name. The name may also be set to a non-printable character. This can be CHR$(13), CHR$(05), CHR$(03), etc. BASIC will accept these names, but it is not something the beginner will know how to do. ML programs can easily use any name. This type of protection will discourage only the novice. Most of the commercial file copy programs will be able to copy any and all files from the disk.

A more advanced type of protection will be to modify the first few bytes that follow the name of the file. These are the bytes after the first shifted space ($A0). By inserting characters here, you can foil most any type of file copy program. Using the hex values of 03 05 90 93, immediately following the first shifted space ($A0), will give the best of file copy programs fits. Combine this with names that will not print out to the screen and you can keep most people from file copying your progams. Example:

```
Location 00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
ORIGINAL:
         12 04 82 11  00 0D 03 A0  A0 A0 A0 A0  A0 A0 A0 A0
         A0 A0 A0 A0  A0 00 00 00  00 00 00 00  00 00 32 00

MODIFIED:
         10 04 82 11  00 0D 03 A0  03 05 90 93  A0 A0 A0 A0
         A0 A0 A0 A0  A0 00 00 00  00 00 00 00  00 00 32 00
```

To correct the types of protection listed here requires that the modified bytes be returned to their normal values ($A0).

The names of these files may not be changed unless the boot programs, which call up these files, have the files names changed from within the boot programs. This will normally require some knowledge of ML and a ML monitor to accomplish. Remember that the name of the program will start at byte 4 and continue to the first shifted space, anything after the first shifted space will be bogus.

Some programmers will place delete characters immediately following the first shifted space. This will allow the name to be listed and then deleted. This can happen so fast that the average person will not be able to see the characters. Again, to correct this type of protection just change the modified bytes to shifted spaces.

The rest of the directory is just a repetition for each file. You will only have to adjust the locations for the appropriate file being examined. Directory modifications are not hard to perform, but they can give the novice programmer fits.

BAD BLOCKS

BAD BLOCKS, that must be read in order for the program to run, are the worst form of protection that a programmer can use!

A disk formatted on the 1541 disk drive will contain 683 blocks. During formatting the drive will completely erase the disk, place sync marks, I.D. numbers, track and sector numbers, checksums and other information needed for the proper operation of the disk drive. All of this information will be contained in the header of the sector. The disk drive will also allocate the space necessary to store the files (blocks). All the information needs to be present for the disk drive to be able to read the block of information from the disk. If all this information is present and the numbers contained in the header have the proper values the block is referred to as a good block.

Many programmers are able to modify or delete some of the information contained in the header. This will generate an error when the disk drive tries to read the modified block (bad block). You will see the red light flash on and off, the disk drive will try to find the information that has been modified or deleted from the sector. In an effort to extract the proper information from the block the disk drive will move the read/write head from the desired track to another track, and back again. After the drive has failed to read the information from the bad block, the read/write head will mechanically reset itself. This will cause a loud clicking noise, similar to the noise heard when formatting a disk. The noise is a result of the cam that the moves the read/write head pounding against its end stop. This can be very harmful to the disk drive, many disk drives have been pounded out of alignment while trying to read these bad blocks.

If these BAD BLOCKS can be so harmful, why do so many programs have them?? Until early 1984 there was no easy way for the user of the 1541 disk drive to duplicate these bad blocks. This seemed to be the ideal way to protect software from unauthorized duplication. The user of the disk might be able to copy the information from the original disk, but he had no sure way of duplicating the bad block on the copy. The software manufacturers had a fool proof protection scheme, or so they thought. Before the program would run, the bad block would have to be read by the disk drive. The program would then check the error channel of the disk drive and read the error. If the disk had an error of the proper type and at the proper location the program would run. If the disk did not contain the error the program would 'crash' or go into an endless loop. This seemed to be a very easy way to prevent pirates from copying the program. This also prevented the owner of the program from making an archival copy of the disk.

The bad blocks normally do not contain any useful information. They are usually placed on the disk as a means of protecting the disk from software pirates.
Below is a BASIC listing of how to read a block from the disk drive. This should be familiar to you.

```
0 REM LINES 10 THRU 90 CONTAIN THE ERROR CHECK ROUTINE

10 OPEN 15,8,15,"IO:" :REM   OPEN ERROR CHANNEL

20 OPEN 5,8,5,"#" :REM   OPEN DATA CHANNEL, USE ANY BUFFER (#)

30 TR = 1: SE = 0 :REM   USE TRACK 1  SECTOR 0

40 PRINT#15,"B-R";5;0;TR;SE :REM   PERFORM BLOCK READ, CHANNEL 5,
DRIVE 0, TRACK, SECTOR

50 INPUT#15,A$,B$,C$,D$ :REM   READ ERROR CHANNEL FOR ERROR, IF
NO ERROR THEN   00,OK,00,00  WILL BE INPUT

60 REM     IF   ERROR   DETECTED   THEN     ERROR   NUMBER,ERROR
TYPE,TRACK,SECTOR  WILL BE INPUT

70 IF VAL(A$)    00 THEN 100 :REM   IF  ERROR  EXISTS  THEN  GOTO
LINE 100 TO START THE PROGRAM

80 PRINT " THIS DISK IS A COPY" :REM   NOTICE TO PIRATES

90 SYS 64738 :REM   PERFORM WARM RESTART AND RESET COMPUTER.

100 REM   START OF MAIN PROGRAM
```

Most programs that have their error checking routine written in BASIC will have similar syntax to that used above. It is necessary to open the error channel and a data channel to the disk drive prior to performing the block read command. Some programmers will substitute "U1", "UA", "U2", "UB" or "B-W" commands for the "B-R" command. All of these commands will perform either a block read or a block write, see your disk drive manual for further information on these commands. It is not important which command is sent to the disk drive. What is important is that the block of information that they are checking for will return the proper error message.

Notice that the program in the chapter on BASIC PROTECTION SCHEMES contained an error checking routine that checked for a particular error (50 IF VAL(A$) = 21 THEN 100). Whereas the above program checks for any error (50 IF VAL(A$) 00 THEN 100). Some programs will check for a specific errors others will check for any error. Either method will provide similiar results.

The program disk contains a short program titled ERROR CHECK. Run this program to see how the disk drive will respond to a good block of information (track 2, sector 0) and how it will respond to a bad block (track 1, sector 0). The actual error message will be displayed on the screen after the disk drive has read the block from the disk.

Bad blocks may also be read from a ML program. The general syntax will be the same as the BASIC version, only the ML routine will rely on KERNAL subroutines located at $FF81 to $FFF5. These are the subroutines that will open channels, print the block read command to the disk drive and input characters from the error channel. BASIC instructions will perform the same function as the ML routine. A table has been prepared which will show the similarities between BASIC and ML.

BASIC COMMAND	ML AND KERNAL CALLS
OPEN	$FFBA $FFBD $FFC0
PRINT#	$FFD2 $FFA8
"U1:5;0;1;0"	U1: 5 0 01 00
INPUT#	$FFCF $FFA5 $FFE4
IF THEN	CMP BEQ
SYS 64738	JSR $FCE2
CLOSE	$FFC3 $FFCC $FFE7

If you were looking at a program written in BASIC you would hunt for the commands on the left to find the error checking routines. If you are looking for the ML program protection routines you should hunt for the subroutines on the right. If you don't understand the ML at this point, don't worry about it. This is just an introduction on what to look for in ML. More on program protection in ML in later chapters.

Keep in mind that any time the disk drive reads a bad block the read/write mechanism will be pounded against the end stop. This will result in a loud noise from the disk drive. The pounding can (will) cause the drive to become mis-aligned, preventing further use of the disk drive.

Bad blocks, that must be read in order for the program to run, are the worst form of protection that a programmer can use.

PRG, SEQ AND USER FILES

A file is just a method of storing information on a disk. Most copy protected programs will use either PROGRAM, SEQUENTIAL or USER files to store the information on the disk.

A PROGRAM file, as the name suggests, is the normal method of storing a program on the disk. When you save a BASIC program to the disk drive it is stored in a program file. When you save a ML progam to the disk drive it is stored in a program file. The program file will contain pointers to link the blocks of the file together. It will also contain information so that the file may be relocated to the same area of memory that it came from and it will contain the actual program. Figure 3 contains the first block of information from a program file. Below is the first line from the file:

```
Location 00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
Hex      11 04 01 08  15 08 00 00  44 4E B2 38  3A 8F 20 44
```

Bytes 0 & 1: Pointers to the next track and sector of the file (hex 11 = decimal 17, hex 04 = decimal 4, track 17 and sector 4). All program files will have pointers located at this location. If the byte 0 is 00, this will be the last block of the file and byte 1 will contain the number of bytes used in this block. Example: the first two bytes contain 00 79 , 00 means that this is the last block of the file and hex 79 = decimal 127, byte 127 is the last byte of this program file. The 00 is an end of file marker, this tell the disk drive that this block is the last in the file.

Bytes 2 & 3: These bytes will contain different values depending on whether this is the first block of the file or one of the other blocks in the file. If it is the first block of the file it will contain the location of where, in the computers memory, the program resided. If this is one of the subsequent blocks of information, these locations will contain the program data. It is important to remember that the first block of the program file contains the pointers so that the program may be relocated in memory.

If this is the first block of data in a program file the pointers will contain the memory location of where the program resided in memory when it was saved to the disk. The bytes are stored in the low byte, high byte fashion. This means that the program resided at the hex address of 0801, decimal 2049. Notice that these two bytes are stored in the reverse order of their actual location. The 6510 microprocessor chip in your computer stores all memory locations in this low byte, high byte order. Keep this in mind when examining hex numbers. The computer can load program files back into the same area of memory where the program resided when they where saved or the computer can load program files into memory at the start of BASIC. The start of BASIC RAM is at hex 0800 (decimal 2048), but every BASIC program line must be preceeded by the value 00 for BASIC to work properly. The BASIC

program files must be located immediately above this location. Hex 0801 is the actual start of where BASIC programs reside in memory. The example in figure 3 is a BASIC program and will locate itself at hex 0801 whether the normal load command (LOAD "NAME",8) or the relocate load is used (LOAD "NAME",8,1). If we load ML programs with the normal load command (,8) the program will load at the start of BASIC, hex 0801. If the relocate command is used (,8,1) the program will relocate it to the same spot where it resided before it was saved.

Bytes 4 thru 255: If this is the first block of a program file, byte #4 will contain the first byte of the actual program data. If this is a subsequent block of a program file it will be a continuation of the program data. Remember that the last block of a program file may contain less than 255 bytes of information in the block.

A SEQUENTIAL file is , as the name implies, information that has been stored on the disk in a sequential manner. These files are similar to those on cassette. The information stored in sequential files must be read from the beginning to the end. Sequential files may not be loaded directly into the computer. Data must be transfered byte by byte through a buffer. Sequential files are similar to program files in that the first two bytes of a block contain pointers that point to the next block in the file. All the rest of the information in the block will be the data. There will not any pointer that will tell the computer where to relocate the data in memory. The data contained in the sequential file must be stored in the computers memory by the program loading the file.

USER files contain information that has been stored on the disk by the user (programmer). These files will not contain any pointers. There will not be any pointers to link the blocks of a file togeather. There will not be any pointers to locate the files in memory.

These files can only be read into the disk drives buffer through the use of the "U1" or the "UA" commands. The information will then be read into the computer one byte at a time and stored at a location in memory specified by the program. This seems to be a complicated way to store information on the disk. It is! Many programs will be stored on the disk in user files one block at a time. The programmer can control what part of the program will reside on which block of the disk. The programmer can scatter the program clear across the disk if desired. The only way to load the data back in to the computer is to know how and where it was saved to the disk in the first place. This can present a challenge to software pirates. Either a complete copy of the disk must be made or the pirate must try to figure how the data was saved on the disk in the first place.

NEXT TRACK + SECTOR OF FILE

```
TB/EDITOR  DISK=8/ TRK=17 BLK=0/ MODE=H

    0 1 2 3  4 5 6 7  8 9 A B  C D E F
/0 11040108 15080000 444EB238 3A8F2044
/1 52495645 20233800 44080A00 99222020
/2 54484953 20495320 41205445 5354204F
/3 46205448 45204449 534B2044 52495645
/
/4 20202020 2022005E 08140099 22202044
/5 45564943 45204E55 4D424552 2E202022
/6 0081081E 00972038 30382C32 33303A97
/7 20353332 38312C30 303A9735 33323830
/
/8 2C303000 94082800 8154B230 A4203130
/9 30303A20 823A00BE 08320099 22931111
/A 20202050 52455353 20414E59 204B4559
/B 20544F20 434F4E54 494E5545 20202022
/
/C 00D7083C 004124B2 22223A8B 20412420
/D B2202222 20A72036 30000409 46009922
/E 93111120 2020494E 53455254 20424C41
/F 4E4B2055 4E464F52 4D415454 45442044
```

POINTER FOR MEMORY RELOCATION $0801

PROGRAM STARTS AT BYTE $04

```
TB/EDITOR  DISK=8/ TRK=17 BLK=0/ MODE=A

   0 1 2 3  4 5 6 7  8 9 A B  C D E F

/0 . . . .  . . . .  D N ♦ 8  : .   D
/1 R I V E    # 8 .  D . . .  . "
/2 T H I S    I S    A   T E  S T   0
/3 F   T H  E   D I  S K   D  R I V E
/
/4           "  . .  . . . .  "     D
/5 E V I C  E   N U  M B E R  .     "
/6 . . . .  . .   8  0 8 . 2  3 0 : .
/7   5 3 2  8 1 . 0  0 : . 5  3 2 8 0
/
/8 , 0 0 .  . . ( .  . T ♦ 0  ↑   1 0
/9 0 0 :    . : . ×  . 2 . .  "  . . .
/A       P  R E S S    A N Y    K E Y
/B   T O    C O N T  I N U E        "
/
/C . ö . <  . A $ ♦  " " : .    A $
/D ♦   " "    →   6  0 . . .  F . . "
/E . . .        I N  S E R T    B L A
/F N K   U  N F O R  M A T T  E D   D
```

FIG. 3

One can not simply load or save user files. User files may be loaded into memory thru the use of the following BASIC program. Notice that this program is similar to the program that checks the error channel after a block read.

```
0 REM   LOAD USER FILES ONE BLOCK AT A TIME

10 OPEN 15,8,15,"IO:" :REM   OPEN ERROR CHANNEL AND INITIALIZE
DRIVE

20 OPEN 5,8,5,"#" :REM   OPEN DATA CHANNEL, USE ANY BUFFER

30 INPUT "MEMORY LOCATION ";ME :REM   STORE USER DATA STARTING AT
THIS MEMORY LOCATION

40 INPUT "TRACK ";TR :REM   WHICH TRACK?

50 INPUT "SECTOR ";SE :REM   WHICH SECTOR?

60 PRINT#15,"U1";5;0;TR;SE :REM   PERFORM USER READ

70 FOR DA = 0 TO 255 :REM  256 BYTES OF DATA

80 GET#5,I$: I = ASC (I$+CHR$(0)): REM   GET BYTE FROM DRIVE  AND
CONVERT INTO DECIMAL

90 POKE ME,I :REM   STORE DATA AT MEMORY LOCATION

100 ME = ME + 1 :REM   INCREASE MEMORY LOCATION BY ONE

110 NEXT :REM   GET ANOTHER BYTE

120 INPUT " ANOTHER BLOCK  ";YN$ :REM   DO IT AGAIN?

130 IF YN$ = "Y" THEN 40 :REM   IF YES GOTO 40

150 CLOSE5:CLOSE15:END
```

This program will allow you to load user files into memory. If you wish to load user files at hex 0801 (2049 decimal) it will be necessary to relocate this program higher into memory. Use the program titled MOVE BASIC before loading USER FILES. Some programs rely only upon hiding the program on the disk in user files and loading it back in the same manner in which it was saved to disk. Others combine the user files with bad blocks in order to protect the disk from pirates. As you can see it can be quite a job to figure which blocks to load and in what order. I do not recommend that you use this routine to try to load a program that was saved to the disk in user files. This program is helpful when examining one or two blocks of information from the disk.

COMPILED PROGRAMS

Compiled programs can be the best friend of the programmer who is wishing to protect his program from pirates. A program, once compiled can be next to impossible to decipher.

BASIC is a high level, interpretive language. This means that BASIC is easy to use, has easily understandable commands and will be fairly slow in its operation. The computer will only understand ML. Each and every instruction in BASIC must be interpreted into ML. The ML will be executed and the computer will interpret another command. This is similar to the action of two people who speak different languages and must use an interpreter to converse. As you can see this could be quite time consuming and some of the instructions from one person may not mean the same thing to the other person. This is also true of BASIC programs and the BASIC interpreter that is used to convert BASIC instructions into ML. It is slow and sometimes not every instruction the computer is capable of performing may be implemented by BASIC.

A BASIC complier can help to remedy the speed problem of BASIC. The compiler will convert the BASIC program into a pseudo ML. This is not a true ML, nor is it ASSEMBLY LANGUAGE. It is a combination of ML and has its own brand of instructions (speed code). The BASIC program is first written and debugged. It is important to have a working version completely debugged and tested before compiling, because the compiled program will not allow editing. Many times BASIC must be re-written to accommodate the compiler being used in order to make the program bullet proof. Then the BASIC program is compiled and the new code is generated. This code will only slightly resemble the BASIC program from which it was derived, but it will run any where from 4 to 100 times faster than the BASIC version. Speed is not the only reason a programmer should consider the use of a compiled program or routine. Once the program has been compiled the code is much different from the BASIC that it was derived from. It can really give software pirates fits when they are trying to figure just how and what the routine is doing. Compiled programs can not do any thing more than a program written in BASIC can. Sometimes compiled programs do less than BASIC or may not duplicate every function of the BASIC program from which they are derived. This is one draw back that must be considered when using the compiler.

When should a programmer use a compiled program? whenever a BASIC program needs added speed or whenever he wishes to confuse a software pirate. The expert programmer, who will write his programs in ML, should not rule out the use of compiled subroutines to confuse the software pirates. I have seen teen age 'kids' that could understand programs written entirely in ML. So the use of compiled subroutines could add a new measure of security to the program protection scheme.

Trying to decipher the compiled program will be a task even for an experienced programmer. I have written programs in BASIC and was not able to follow the logic in the compiled version. There are three or four good compilers on the market today. Try to get a hold of one and see what I mean.

The only way to follow the logic of a program once compiled is to start a table of the code that the compiler uses and try to find the meaning for each and every instruction. Good luck! Each and every compiler will use its own set of commands. I have heard of a program that will decipher the compiled version of a program and return it to BASIC. Whether or not the program actually exists and the quality of its work is not known at this time.

Most 'professional' programmers do not use any compiled routines. I guess that they feel that a program written in BASIC and then compiled is not a very macho way to write programs. Very few programs are written in any form of a compiled language. The BASIC program from the last chapter is contained on the disk. Notice how much faster the program works. The program is called USER. WOW.

MACHINE LANGUAGE

Some knowledge of machine language (ML) is necessary to understand the program protection schemes of most programs.

Understanding the ML once written is far easier than writing programs in ML. One only has to follow the concept of the program to understand what the program does. Whereas, writing a program in ML requires attention to details and a thorough knowledge of ML.

I am not going to give you a complete course in ML. I will try to pass along the concepts of ML needed to understand program protection schemes. If you wish to learn more about ML I would suggest you borrow a few books from your library on 6502 ML programming. The 6502 and the 6510 microprocessor have the same commands and are almost identical in their operation. The only difference is in the internal registers of the 6510 and this will not affect the ML programming of the computer. The programmers reference guide for the C-64 has a section devoted to ML programming. If you are not familiar with ML, now is the time to acquaint yourself with it.

ML, just as the name implies, is the machines (computers) own language. ML can be difficult to understand, fortunately we have some tools to help us understand ML. The most important tool is the ML monitor. The ML monitor will convert (disassemble) ML into assembly language. Assembly language may be thought of as an abbreviated English and will allow us to work in the computers own language. For the rest of the book any time I refer to ML it will also mean assembly language. Below are some examples of a disassembly of a ML routine.

```
Memory   Machine    Assembly
Address  Language   Language    Comments

.,4000   A9 00      LDA #$00    LOAD THE ACCUMULATOR WITH 00
.,4002   8D 09 63   STA $6309   STORE THE ACCUMULATOR AT $6309
.,4004   88         DEY         DECREMENT THE Y  (Y=Y-1)
.,4005   C9 0D      CMP #$0D    COMPARE TO #$0D
.,4007   20 89 43   JSR $4389   JUMP TO SUBROUTINE AT $4389
.,400A   4C 56 32   JMP $3256   JUMP TO $3256
```

While ML is not hard to understand, the actions may not be that clear at this time. Why would any one want to load the accumulator? What is an accumulator? Who cares?

Most ML code is valid code written to perform a specific function. Only a small portion of the program is used for protection. It is this small portion of code that you will be interested in finding. Once you have found this code, you can begin to decipher its meaning.

The first thing to get familiar with are the commands that ML uses. The commands are all three letters which are abbreviations

of their function. JMP stands for jump, this is similar to GOTO in BASIC. BRK stands for break, this is similar to END in BASIC. JSR stands for jump to subroutine, this is similar to GOSUB in BASIC. RTS stands for return from subroutine, this is similar to RETURN in BASIC. Most of the rest of the commands will become familiar to you as you use them. Keep in mind that they are all abbreviations for the actual command. Let's look at a sample routine written in ML. This is the disassembled version of the routine. The first number will represent the location of the memory address that you are looking at. The next number will represent the value stored at that memory location. Any other numbers following will represent the values stored at the following memory locations. The three letters are the assembly language instructions that the values represent. The last number, if present, represents the value or memory location to be acted upon.

```
Memory  Machine    Assembly
Address Language   Language     COMMENTS

.,8000 A9 00       LDA #$00     LDA WITH THE HEX VALUE OF OO
.,8002 8D 30 80    STA $8030    STA AT THE LOCATION OF $8030
.,8005 20 CC FF    JSR $FFCC    JSR AT $FFCC - CLOSE CHANNELS
.,8008 4C 8A 40    JMP $408A    JUMP TO $408A
```

The comments are only added to help you understand what the ML commands mean, you will not find the comments when you use your ML monitor.

The machine language monitor will be your best tool when examining programs written in machine language. Three monitors have been included on the disk, all are identical except for where they reside in memory. The monitors are public domain programs and, as such, do not have every function that one could ask for, but they are more than adequate for every day use.

When you are looking at ML programs that someone else has written, the ML monitor is an indispensable tool. Be sure that you understand all the capibilities of the ML monitors before proceeding to the next chapters. It will be necessary to become familiar with the SYNTAX of the ML monitor. To load the monitor type: LOAD "HIMON SYS49152",8,1. After the program has loaded it will be necessary to type SYS49152 (press RETURN). You should see the following lines on the screen.

```
B*
   PC  SR AC XR YR SP
.,C03E 32 00 C3 00 F7
```

This means that you have successfully entered the ML monitor.
PC is the program counter
SR is the status register
AC is the accumulator
XR is the X register
YR is the Y register
SP is the stack pointer

Since I am not going to cover how to write machine language in this book, I will not go into lengthy discussions of the functions of the registers. The AC, XR and YR are somewhat similar to intersections in a small town. For now lets think of them as something that you must go thru prior to getting anywhere. It is necessary to go thru one of these registers before you store any values in memory. It is not important to understand why you need registers at this time, just remember that you do. If you wish to store a value in memory it will be necessary to go thru one of these registers before storing the value in memory. First load the accumulator (LDA), then you can store the accumulator (STA) at the desired memory location.

What's very important is your ability to use your ML monitor and be familiar with the commands available. Most ML monitors have similar commands, so if you have used other monitors the syntax will be similar.

Command	Function
A	Assemble (write) ML instructions A XXXX LDA #$10
C	Compare to two areas of memory C LLLL HHHH CCCC
D	Disassemble memory D LLLL (HHHH)
F	Fill a section of memory F LLLL HHHH XX
G	Go (run) G XXXX
H	Hunt for specific bytes H LLLL HHHH XX or H LLLL HHHH "XX"
I	Interpret memory (display ASCII symbols) I LLLL (HHHH)
L	Load program L "NAME",08
M	Memory display and modify M LLLL (HHHH)
R	Register display
S	Save program S "NAME",08,LLLL,HHHH
T	Transfer memory from one location to another T LLLL HHHH TTTT
X	Exit to basic

To use the monitor, simply type the letter corresponding to the command you wish to execute. Then type in the memory location that you wish to examine or modify. The LLLL refers to the low or starting address of memory that you wish to examine. The HHHH is the high or ending address of the block of memory that you wish to examine. The XX or XXXX will be any value or memory location you wish to use. The CCCC will be the low or the starting address of the memory location that you wish to compare the first block of memory with, no ending address is necessary. The TTTT is the low or starting address of memory that you wish to transfer the memory to, no ending address is necessary. The memory locations contained in parenthesis are optional and may be used if you wish to examine a block of memory rather than an individual location. Use the monitors along with a good book on machine language programming if you are just starting in ML. Practice using the commands until you have mastered all of them. It is more important that you know how to use the ML monitor than it is to understand everything there is to know about ML itself.

Try loading in a BASIC program from the ML monitor. Use the following commands to load the program.

```
L "TEST BASIC",08        'RETURN'
```

To examine the BASIC program with ML monitor use the 'I' command.

```
I 0800 0880    'RETURN'
```

Use the cursor up and cursor down keys to scroll thru memory, notice how BASIC programs are stored in memory. It is very important that you can recognize programs written in BASIC. Many good programs are written in BASIC and some have been modified with a ML monitor to prevent tampering by the user. There are many tricks that can be done to BASIC programs with a ML monitor. Most of these tricks are easy to do from ML, whereas they would require an extensive amount of time to perform from BASIC. These tricks will be covered in the chapter on ADVANCED PROTECTION SCHEMES.

Now fill the memory from hex $0800 to $9FFF with the value of 00.

```
F 0800 9FFF 00     'RETURN'
```

Use the 'I' command to examine the memory where the BASIC program used to reside. The area will be filled with 00s. If you want to find out where a ML program resides in memory it will be necessary to first fill the memory with 00s. Then load the program from the monitor.

```
L "TEST ML",08
```

The ML monitor will automatically relocate all programs into the same area of memory that the program was saved from. You can use the 'I' command to search thru memory until you find the ML program. Now examine the memory to find the starting and ending location of the program. Use the 'I' command to scroll thru memory while looking for the starting and ending address. The ML

code contained in "TEST ML" is not actually a program, it is just a subroutine from a program I wrote. Once you have found the begining and ending address of the program, you should write them down for future reference. If this were an actual program that you were going to modify it would be necessary to have the starting and ending addresses in order to save the program back to the disk.

Now that you have found the beginning and end of the program, it will be necessary to use some of the other functions of the ML monitor in order to further examine the ML code. Try using the 'H' command to hunt for specific information.

```
H 2000 2200 'U1              'RETURN'
H 2000 2200 55 31            'RETURN'
```

The computer will now hunt its memory from hex $2000 to hex $2200 for the hex representation of U1. When you wish to hunt for ASCII characters use the ' (single quote) before the character(s). If you wish to hunt for the hex value(s) use the value(s) separated by a space. Hex 55 31 is equivalent to 'U1. Try hunting for other value(s) or character(s).

The transfer and compare commands may be used in conjunction with one another. Transferring memory will cause a copy of the memory to be made, it won't actually transfer the memory. It will duplicate memory. First transfer the memory from one location to another, then compare the two blocks of memory.

```
T 2000 2200 3000             'RETURN'

C 2000 2200 3000             'RETURN'
```

If any of the bytes contain different values the location(s) will be displayed on the screen. Since you have just transferred the memory to hex $3000 there will not be any differences. Now use the 'M' command to display and modify memory.

```
M 2000 2040                  'RETURN'
```

Change a few of the displayed bytes to any other value. Then press 'RETURN' before moving the cursor from the line. This will change the value of the bytes in memory. Another comparison can be made to find any bytes which have been changed.

```
C 2000 2200 3000
```

The memory location(s) of the two blocks of memory that contain different values will be displayed. The use of the transfer and compare is very valuable when inspecting program protection schemes. Once you have modified the code it will be necessary to save the ML code back to the disk. When using the save command it will be necessary to specify the starting address and the ending address of the memory to be saved, plus one.

```
S "NAME",08,2000,2201        'RETURN'
```

This will save the code from hex $2000 to hex $2200 on the disk. The code will be saved in a program file so that it may be reloaded into the same area of memory from which it was saved.

Once you have loaded a ML program into memory it will sometimes be necessary to execute (run) the program. To do this the ML monitor has the 'G' command. Once you have located the proper entry point (where the program will start) the following command can be used.

G 2000 'RETURN'

The will cause the ML program located at hex $2000 to be executed. The program will continue to execute until the BRK command is encountered. The BRK command is similiar to END in BASIC. BRK may be inserted into ML at various points to help you understand the function of the code. The computer will execute the program, as it was written, until the BRK. At which time the program will stop and control of the computer will be turned over to the ML monitor.

If you wish to exit from the ML monitor to BASIC all you have to type is:

X 'RETURN'

ADVANCED BASIC PROTECTION

The first part of this chapter will cover programs which have been written in BASIC. There are many good programs written entirely in BASIC. Simple, but effective, protection schemes rei been used to protect these programs from the average user. If you remember from the chapter on BASIC protection schemes, the user could be prevented from using the RUN/STOP and RESTORE keys thru a simple poke (POKE 808,230). It will now become virtualy impossible to stop the program. When this occurs the easy way to defeat this is thru the use of a reset button. There are many commercial made reset buttons on the market for under $10.00. You can make your own for under a dollar. It will require a little bit of soldering, Keep in mind that any modifications that you make to your computer may void the warranty. If you are not sure of what you are doing take your computer to a qualified service person.

1) Locate the RS232 port (modem port) on the left rear of your computer. The board has 12 contacts on the top and bottom of the board. Starting from the center, top side of the board, the contacts are numbered from one to twelve. Locate contacts #1 and #3. Open the case of your computer and drill a small hole in the side of your computers case. Install a momentary contact, normally open switch in the hole. Solder a wire from one side of the switch to pin #1, solder another wire from pin #3 to the other side of the switch. Close the case of your computer and you're done.

2). For those of you who don't want to drill a hole in your computer or if you don't want take a chance on voiding your warranty you may purchase a 24 pin edge card connector from your local electronics supply house. Solder the momentary contact switch to the connector, pins #1 and #3. All you have to do is to is plug the connector into the modem port and your reset button is ready.

The reset switch is a very valuable tool. When the C-64 was made it could not have added $0.20 to the price of the machine to add the reset button. If your computer is not equipped with a reset button at this time, I would recommend that one be installed. You will need it later.

Contacts #1 and #3 contain the ground and the RESET lines of the computer. When these two lines are momentarily connected, the computer will perform a hardware reset. This is similiar to turning the computer off and back on again. The main difference is that the computer will not erase any memory stored in the BASIC RAM and above ($0800-$FFFF). When the computer is reset the memory below BASIC ($0000-$07FF) will be reset. BASIC programs stored in memory will not be erased, only the pointers that tell the computer where to find the BASIC program will be reset. The BASIC program still exists, you just can't see it. To restore the pointers back to where they were before reset requires the use of short ML program. The program is called RESTORE and can be found

on the program disk.

To demonstrate the use of the reset button and the program RESTORE, first load the "BASIC TEST" program into memory. Then POKE 808,230. RUN the program. It is now impossible to stop the program from running by using the RUN/STOP and RESTORE keys. To stop the program press the reset button. The screen will shrink and clear. Next you will see the same messages appear on the screen as if you had just turned the computer on. If you LIST the program it will not appear. To restore the program in memory all you have to do is:

```
LOAD "RESTORE",8,1          'RETURN'

SYS525:CLR                  'RETURN'
```

Now the program will LIST, RUN and can be SAVEd normally. Some programmers will go to extraordinary means to protect their BASIC programs. They will modify the BAM and DIRectory to prevent you from seeing what files are saved on the disk. They will save the BASIC program in user files scattered all across the disk. They will use ML boot programs to actually load the BASIC program into memory. Very complicated tricks are sometimes employed to prevent you from getting at the program. Then they don't protect the actual BASIC program other than using a few pokes. All these pokes do is prevent you from using the RUN/STOP and RESTORE keys. BASIC programs may be easily identified by LOADing and RUNning the program. Once it is in memory and running you will notice that the response time to your commands will be sluggish, not the same response that you would expect from your favorite ML game programs. These programs may be saved to disk with the use of the reset button and the "RESTORE" program. Don't be fooled by all that fancy protection. Most BASIC programs may be broken this easily!

SUPER LINE NUMBERS is a trick that I have not seen used by many programmers. It will provide better protection for programs written in BASIC than most. The only draw back is that the program must handle GOTO or GOSUB commands in a special manner. This is a very effective and easy way to write a boot program or the main program. Super line numbers will give most people fits when they first encounter them.

The version of BASIC that the C-64 uses will not allow line numbers larger than 63999. Any time that you try to enter a line number larger than 63999 a '?SYNTAX ERROR' will be generated and the line will be ignored. The trick is to enter the program in BASIC with normal line numbers. Then use the ML monitor to modify the line numbers of the BASIC program. With the ML monitor, it is possible to change the BASIC line numbers to any value up to 65535. It is even possible to change all the line numbers to the same number. When you combine the changed line numbers with a graphics character that BASIC will not accept (see BASIC protection schemes) it will be possible to make the program virtually unlistable, even before the program is run. To do this it will be necessary to use the ML monitor. Load the "BASIC TEST" program. Add the following command to the end of the first line

of the program. :REM shift L (hold the shift key while pressing the letter L). This will produce a graphics character similiar to the letter L. The graphics character will not list from BASIC, it will create a '?SYNTAX ERROR' when listed. Load and execute the HIMON. Use the 'M' command to examine the BASIC program.

M 0800 0820 'RETURN'

The following lines will be displayed.

```
:0800 00 13 08 0A 00 99 22 4C
:0808 49 4E 45 20 31 30 22 32
:0810 8F CC 00 22 08 14 00 99
:0818 22 4C 49 4E 45 20 32 30
:0820 22 00 00 00 00 00 00 00
```

The value contained at 0800 is 00. BASIC line numbers must be preceded by the value 00.

Locations $0801 and $0802 contain the pointers that tell BASIC where the next line begins. The pointers are stored in the familiar low byte, high byte format that is necessary for the 6510 microprocessor. Note that the pointers do not point to the 00 immediately preceding the BASIC line, they actually point to the pointers of the next line (location $0813 in the example). All BASIC lines will contain pointers to the location of the next line.

Location $0803 and $0804 contain the line number of the first line of the BASIC program. Again low byte, high byte (line 10, hex 000A). All BASIC lines must have a line number. The line number can be found immediately after the pointers of all BASIC lines. BASIC will store its program lines in ascending order. If we wanted to change the line number to a different value, these are the two bytes to change. If these two bytes are changed to FF FF, the first line number will now be 65535 when the program is LISTed. Change the line numbers to FF FF, then press 'RETURN' before leaving the line.

```
:0800 00 13 08 FF FF 99 22 4C
:0808 49 4E 45 20 31 30 22 32
:0810 8F CC 00 22 08 FF FF 99
:0818 22 4C 49 4E 45 20 32 30
:0820 22 00 00 00 00 00 00 00
```

Then use the 'X' command to exit to BASIC, if you wish to re-enter the ML monitor at a later time type SYS 49152 'RETURN'. The program will run properly, but the lines may no longer be deleted or edited. The program will not list properly because of the 'shift L' inserted after the first line. A '?SYNTAX ERROR' will be generated when the program is listed. The super line number is used to prevent the line from being deleted or edited from BASIC. The 'shift L' character did not have to be added to make the super line work. It was added only to make the program unlistable. Try using super line numbers on a program that does not contain the 'shift L' character. All of the line numbers may

be set to any number desired.

BASIC does not look at line numbers while the program is running, unless the GOTO or GOSUB commands are used. If a GOTO or GOSUB command is issued BASIC will search the line numbers for the proper value, starting with the first line. BASIC stores its line numbers in ascending order. When BASIC reads the number of 65535 it gets fooled into thinking that this must be the last line number and quits searching. Hence, no GOTO or GOSUB commands may be used when the first line number has been changed to these super line numbers.

If, instead, the first line of the program contains the line number 10 and the statement GOTO 63999, all of the subroutines can be stored with line numbers between 10 and 63998. The program will function just fine, all the GOTO and GOSUB commands will work normally. Keep the main body of the program at line 63999 and above, keep all GOTOs and GOSUBs at line 63998 and below. For an example of this, see the program called SUPER LINES. All of the lines above 64000 have the same line number, yet the computer knows where to RETURN to after the GOSUB command has been executed. BASIC keeps track of the actual address of where the GOSUB is located in memory, not the line number that contains the GOSUB. So when the 'RETURN' is found the program will return to the same address it came from, not paying any attention to the line number. BASIC only looks for the line number when searching for the line specified by the GOSUB or GOTO.

To correct a program that uses these super line numbers, you only have to use the ML monitor to assign line numbers corresponding to the proper ascending order. Line numbers of all lines may be changed by this technique and they may be changed to any value desired between 0 and 65535. Numbers may be repeated or a descending order may be used.

Locations $0805 thru $0811 in the preceding example contain the BASIC program line. All BASIC commands are tokenized or stored as single byte that the computer will interpret. This is done to conserve memory. All of the rest of the BASIC lines are stored in the same fashion. A 00 will precede the start of a BASIC line, then the pointers to the next line, the line number and the actual program line itself. The end of a BASIC program will contain three zero's (00 00 00) and the pointers of the last line will point to the middle 00. These three 00 bytes will be found at the end of all BASIC programs and they also mark the start of where BASIC stores its variables.

Line numbers are not the only item that may be changed to make the BASIC program appear to be different than it performs. The pointers that point to the next line may also be modified. For instance, the pointers that point to the second line may be changed to point to any other line. The pointers of any line may be changed to point to any other line or to the end of the program. If the pointers are changed to point to the end of the program, the only line that will list is the first line, but the program will execute normally. These pointers are only used when LISTing the program or searching for GOSUBs or GOTOs. Just as in

super line numbers it will be necessary to place any subroutines at lower line numbers than where the pointers have been modified. Load the program called MOD POINTERS. This program has already been modified. See if you can return it to normal using the ML monitor to reset the pointers. Remember that the first line should point to the second line, the second to the third, etc.

Programs written in BASIC can be protected in a number of ways. It is important that you can identify programs written in BASIC and how to fix these protection schemes. Rarely does any BASIC program use more than one of these protection methods and most don't get very sophisticated. Be sure that you can identify programs written in BASIC by using the 'I' command of the ML monitor. It is necessary to know when the pointers for the start of BASIC have been moved higher in memory and a short ML subroutine is inserted below the new start of BASIC. A sample of this is contained on the disk under the name of "ML AND BASIC". LOAD and RUN this program so that you know what the program does. Then try using the ML monitor and the 'I' command (or the 'M' command) to find the start of the BASIC program. Try also to detect and correct any other tricks used on this program. When you are done (or have given up) load the program called "ML AND BASIC #2". This will give you some hints as to what to look for in the first program.

When you are looking at BASIC programs keep in mind that some contain absolutely no protection schemes at all. Most contain only one or two small tricks. Very few will get in to the sophisticated protection schemes. Just be aware that they do exist and you will probably see more of them in the future. Program protection for most programmers is an afterthought. Many programmers spend hundreds or thousands of hours writing their programs, then spend only one or two hours protecting them. Another common occurrence is to find all the program protection at one location. Most programs are quite easy to break once you know where to look and what you are looking at.

ML PROTECTION

Hopefully you have mastered the use of the ML monitor and we can proceed to programs protected in ML. Now that we are going to be working in ML, I will no longer be giving any values in decimal. From here on out it will all be in hex.

ML is where the real meat of the program protection schemes reside. We can now look at how the more 'professional' programmers protect their software. I use the word 'professional' with some reservation. Most of these so called professionals use bad blocks, which must be read by the program before it will run. I am dead set against using this technique to protect software. I takes too high of a toll on the disk drive. When the drive tries to read a bad block the read/write head is literally beat to death. Unfortunately, too many programmers prefer to use this method of program protection.

The only good thing about this type of protection is that it is relatively easy to spot. Once you have spotted the protection routine, it will be necessary to obtain a disassembly of the ML code in the immediate area. As I stated earlier most programmers prefer to place all of their protection schemes in one small area of the program (usually at the begining or end). This makes things easier for them, and for us. It really is nice that almost all of the programs on the market today place all of their program protection in one little area. This sure makes it easy to fix the program so that the disk drive does not get beat to death every time the program loads.

First thing to do is get a copy of the program on a disk that does not contain any errors. The best way to do this is to use "BACKUP 228". This is a program that will enable you to make a copy of any disk, minus the bad blocks. BACKUP 228 is not fast, it takes almost 30 minutes to make a copy of the complete disk. What it does do is make an accurate copy of any data contained on the disk. I always use BACKUP 228 for the first copy of any disk. It works. The complete instructions are contained in the program, just follow the prompts. It is important that you do not change any code on the original disk. Keep the original program intact and do not perform any modifications to it. Make a copy of the disk and work with it. If you should make a mistake or anything else happens to the copy disk its no big deal. Remember you can legally make one copy of any program for archival purposes. I would strongly suggest that you make a copy of every program and use the copy, place the original in your archives.

Once you have a copy of the original disk, you can get to work on eliminating the protection schemes. Verify the fact that you can list the directory of the disk. If you can't, now is the time to fix the directory so that you can (see chapter on BAM and DIR). It will be necessary to load each and every program on the disk to determine which program contains the program protection scheme. Sometimes it may be necessary to rename the program so that it may be loaded into memory and examined. Be sure to change

the name back to its original name before trying the disk.

Some programs contain an auto-boot routine that will take over control of your computer as soon as it is loaded into memory. These programs are located from $0100 to $0400. This is below the screen memory. Try loading these programs from the ML monitor, if the program takes over control of the computer it will be necessary to reset the computer. One problem with the reset is that the computer will restore all the memory from $0000 to $0800 when the reset button is used. Any program that previously resided at this area will now be erased. The RAM memory from $0800 and up will not be affected by the use of the reset button. This will make it a little more difficult to examine the program that resides in the area from $0000 to $0800. In order to examine the auto-boot programs we must first relocate them in to a higher area of memory.

First load the ML monitor and execute it to be sure every thing is functioning properly. Exit to BASIC, then load the auto-boot program in from BASIC. Do not use the ',8,1' command to load the program, use the ',8' after the program name. This will load the program into the area of memory where BASIC normally resides. Occasionally the computer will lockup when you load a ML program into the area where BASIC normally resides. If this happens just reset the computer, the program and the ML monitor will not be erased from memory. To re-enter the monitor type SYS 49152 'RETURN'. You will now be able to disassemble the ML code starting at 0800. When you are looking at this code remember that this is not the normal location for this code to reside. It is only necessary to relocate this code when the program you are trying to examine is an auto-boot program and it takes over control of your computer when loaded from the ML monitor. Other programs may be examined at their normal location in memory.

Now that you have the program in memory what do you look for? If the program is a auto-boot routine it will normally just load another program and perform a JMP (jump) to the proper memory location to begin executing the next program. There is not quite enough memory available to perform the necessary error checks and have room left over to load another program. Use the 'I' command to look for the file name of the next program that will be loaded into memory. The file name may be just one byte or as many as sixteen. Some programs store the name in ASCII characters, this sure makes finding the name of the next program easy. Others use names that do not contain any ASCII characters, this only makes the name a little harder to find. Below you will see a sample ML routine. This routine is a boot. It will only load another program, then JMP (jump) to the proper location to execute the program. The comments that appear along side the disassembly are mine and have been placed there for convenience only.

SUBROUTINE # 1 ORIGINAL CODE

```
,2900 A9 00    LDA #$00
,2902 20 BD FF JSR $FFBD   SET FILE NAME (00 = NO NAME)
,2905 A9 0F    LDA #$0F
,2907 A2 08    LDX #$08
,2909 A8       TAY
,290A 20 BA FF JSR $FFBA   SET LOGICAL, 1ST AND 2ND ADDR (15,8,15)
,290D 20 C0 FF JSR $FFC0   OPEN FILE (15,8,15)
,2910 A9 01    LDA #$01    1 CHARACTER NAME
,2912 A2 55    LDX #$55    FROM LOCATION
,2914 A0 29    LDY #$29    $2955
,2916 20 BD FF JSR $FFBD   SET FILE NAME (#)
,2919 A9 05    LDA #$05
,291B A2 08    LDX #$08
,291D A8       TAY
,291E 20 BA FF JSR $FFBA   SET LOGICAL, 1ST AND 2ND ADDR (5,8,5)
,2921 20 C0 FF JSR $FFC0   OPEN FILE (5,8,5)
,2924 20 CC FF JSR $FFCC   CLOSE I/O CHANNELS
,2927 A2 0F    LDX #$0F
,2929 20 C9 FF JSR $FFC9   OPEN CHANNEL FOR OUTPUT (15)
,292C A0 00    LDY #$00
,292E B9 56 29 LDA $2956,Y
,2931 F0 06    BEQ $2939
,2933 20 D2 FF JSR $FFD2   PRINT CHARACTERS TO DISK DRIVE  (U1: 5 0 01 2)
,2936 C8       INY
,2937 D0 F5    BNE $292E
,2939 20 CC FF JSR $FFCC   CLOSE I/O CHANNELS
,293C A2 0F    LDX #$0F
,293E 20 C6 FF JSR $FFC6   OPEN CHANNEL FOR INPUT
,2941 20 CF FF JSR $FFCF   INPUT CHARACTER
,2944 C9 32    CMP #$32    CMP TO $32 (ASCII 2)
,2946 D0 0A    BNE $2952   BRANCH IF NOT EQUAL (TO $32)
,2948 20 CF FF JSR $FFCF   INPUT CHARACTER
,294B C9 39    CMP #$39    CMP TO $39 (ASCII 9)
,294D D0 03    BNE $2952   BRANCH IF NOT EQUAL (TO $39)
,294F 4C 64 90 JMP $9064   JUMP TO START OF MAIN PROGRAM
,2952 4C E2 FC JMP $FCE2   JUMP TO WARM RESET (CRASH PROGRAM)
.2955 23       ???         (ASCII #)
,2956 55 31    EOR $31,X   (ASCII U1)
,2958 3A       ???         (ASCII :)
,2959 20 35 20 JSR $2035   (ASCII  5 )
,295C 30 20    BMI $297E   (ASCII 0 )
,295E 30 31    BMI $2991   (ASCII 01)
,2960 20 32 0D JSR $0D32   (ASCII  2 )
```

THIS IS THE ORIGINAL VERSION OF THE CODE. THE SUBROUTINE WILL OPEN TWO CHANNELS TO THE DISK DRIVE (5 & 15). THEN IT WILL PRINT A USER #1 COMMAND (BLOCK READ). ALL CHANNELS WILL BE CLOSED TO THE DISK DRIVE. A CHANNEL WILL BE OPENED FOR INPUT AND A CHARACTER WILL BE INPUT. A COMPARISON WILL BE MADE TO SEE IF THE FIRST CAHRACTER INPUT IS $32 (ASCII 2). IF IT IS NOT $32 THE THE PROGRAM WILL BRANCH TO $2952 WHERE THE PROGRAM WILL CRASH. IF IT IS A $32 THE PROGRAM WILL 'FALL THRU' TO THE NEXT CHARACTER INPUT. A COMPARISON WILL BE MADE TO SEE IF THE SECOND CHARACTER IS $39 (ASCII 9). IF IT IS NOT, THE PROGRAM WILL BRANCH TO $2952 AND CRASH. IF IT IS EQUAL TO $39 THEN THE PROGRAM WILL 'FALL THRU' TO THE JMP $9064 WHERE THE MAIN PROGRAM RESIDES.

THE ERROR TYPE THAT THIS PROGRAM LOOKS FOR IS 29. IF THE DISK HAS THE PROPER ERROR, THE ERROR CHANNEL WILL RETURN: 29,DISK ID MISMATCH,01,02. AS YOU CAN SEE THE FIRST TWO CHARACTERS WILL BE 2 & 9. THE SUBROUTINE WILL CHECK EACH CHARACTER INDIVIDUALLY AND EXECUTE THE MAIN PROGRAM IF THE PROPER ERROR IS RETURNED.

SUBROUTINE # 2 ORIGINAL CODE

```
,3551 A9 03    LDA #$03    3 CHARACTER NAME
,3553 A2 C0    LDX #$C0    STORED AT
,3555 A0 35    LDY #$35    $35C0
,3557 20 BD FF JSR $FFBD   SET FILE NAME  (I/O)
,355A A9 0F    LDA #$0F
,355C A2 08    LDX #$08
,355E A0 0F    LDY #$0F
,3560 20 BA FF JSR $FFBA   SET LOGICAL, 1ST & 2ND ADDR (15,8,15)
,3563 20 C0 FF JSR $FFC0   OPEN FILE (15,8,15,"I/O")
,3566 A9 01    LDA #$01    1 CHARACTER NAME
,3568 A2 BF    LDX #$BF    STORED AT
,356A A0 35    LDY #$35    $35BF
,356C 20 BD FF JSR $FFBD   SET FILE NAME  (#)
,356F A9 03    LDA #$03
,3571 A2 08    LDX #$08
,3573 A0 03    LDY #$03
,3575 20 BA FF JSR $FFBA   SET LOGICAL, 1ST & 2ND ADDR (3,8,3)
,3578 20 C0 FF JSR $FFC0   OPEN FILE (3,8,3,"#")
,357B A2 0F    LDX #$0F
,357D 20 C9 FF JSR $FFC9   OPEN CHANNEL 15 FOR OUTPUT
,3580 A0 00    LDY #$00
,3582 B9 C3 35 LDA $35C3,Y
,3585 F0 06    BEQ $358D
,3587 20 D2 FF JSR $FFD2   PRINT CHRACTERS TO DISK DRIVE (U1:3 0 01 02)
,358A C8       INY
,358B D0 F5    BNE $3582
,358D 20 CC FF JSR $FFCC   CLOSE I/O CHANNELS
,3590 A2 0F    LDX #$0F
,3592 20 C6 FF JSR $FFC6   OPEN CHANNEL 15 FOR INPUT
,3595 20 CF FF JSR $FFCF   INPUT A CHARACTER
,3598 29 FF    AND #$FF
,359A 48       PHA         PUSH THE CHARACTER ON THE STACK (SAVE CHARACTER)
,359B 20 CF FF JSR $FFCF   INPUT ANOTHER CHARACTER
,359E C9 0D    CMP #$0D
,35A0 D0 F9    BNE $359B
,35A2 20 CC FF JSR $FFCC   CLOSE I/O CHANNELS
,35A5 68       PLA         PULL THE CHARACTER FROM STACK
,35A6 C9 32    CMP #$32    CMP TO $32 (ASCII 2)
,35A8 D0 03    BNE $35AD   BRANCH IF NOT EQUAL TO $32 (ASCII 2)
,35AA 4C 0A 26 JMP $260A   JUMP TO START OF MAIN PROGRAM
,35AD A9 AA    LDA #$AA
,35AF 85 FB    STA $FB
,35B1 A5 34    LDA $34     INDIRECT INDEXED ADDRESSING
,35B3 85 FC    STA $FC
,35B5 A4 00    LDY $00     USED TO ERASE ML CODE
,35B7 91 FB    STA ($FB),Y
,35B9 C8       INY         FROM $34AA TO $35AA
,35BA D0 FB    BNE $35B7
,35BC 4C E2 FC JMP $FCE2   THEN CRASH PROGRAM
,35BF 23       ???         ASCII #
,35C0 49 2F    EOR #$2F    ASCII I/
,35C2 4F       ???         ASCII O
,35C3 55 31    EOR $31,X   ASCII U1
,35C5 3A       ???         ASCII :
,35C6 33       ???         ASCII 3
,35C7 20 30 20 JSR $2030   ASCII  0
,35CA 30 31    BMI $35FD   ASCII 01
,35CC 20 30 32 JSR $3230   ASCII  02
,35CF 0D 00 00 ORA $0000   ASCII RETURN
```

THIS SUBROUTINE WILL CHECK FOR ANY ERROR ON THE DISK. IT WILL ONLY CHECK THE FIRST BYTE FROM THE ERROR CHANNEL (CMP #$32 OR ASCII 2). MOST ERRORS FROM THE DRIVE WILL BEGIN 2 (i.e. 20, 21, 22, 23, 24, 27 or 27).

```
,6A00 A9 05    LDA #$05
,6A02 A2 08    LDX #$08
,6A04 A0 05    LDY #$05
,6A06 20 BA FF JSR $FFBA     SET LOGICAL, 1ST & 2ND ADDR (5,8,5)
,6A09 A9 01    LDA #$01      1 CHARACTER NAME
,6A0B A2 6A    LDX #$6A      STORED AT
,6A0D A0 6A    LDY #$6A      $6A6A
,6A0F 20 BD FF JSR $FFBD     SET FILE NAME (#)
,6A12 20 C0 FF JSR $FFC0     OPEN FILE (5,8,5,"#")
,6A15 B0 2E    BCS $6A45
,6A17 A9 0F    LDA #$0F
,6A19 A2 08    LDX #$08
,6A1B A0 0F    LDY #$0F
,6A1D 20 BA FF JSR $FFBA     SET LOGICAL, 1ST & 2ND ADDR (15,8,15)
,6A20 A9 0C    LDA #$0C      12 CHARACTER NAME
,6A22 A2 6B    LDX #$6B      STORED AT
,6A24 A0 6A    LDY #$6A      $6A6B
,6A26 20 BD FF JSR $FFBD     SET FILE NAME (U1: 5 0 35 1)
,6A29 20 C0 FF JSR $FFC0     OPEN FILE (15,8,15)
,6A2C C6 FB    DEC $FB       DECREMENT $FB ($FB = $FB-1)
,6A2E A2 0F    LDX #$0F
,6A30 20 C6 FF JSR $FFC6     OPEN CHANNEL (15) FOR INPUT
,6A33 A9 0F    LDA #$0F
,6A35 20 CF FF JSR $FFCF     INPUT CHARACTER
,6A38 C9 32    CMP #$32      CMP TO $32 (ASCII 2)
,6A3A D0 09    BNE $6A45     BRANCH IF NOT EQUAL (TO $32)
,6A3C A9 0F    LDA #$0F
,6A3E 20 CF FF JSR $FFCF     INPUT CHARACTER
,6A41 C9 33    CMP #$33      CMP TO $33 (ASCII 3)
,6A43 F0 08    BEQ $6A4D     BRANCH IF EQUAL (TO $33)
,6A45 A9 00    LDA #$00
,6A47 85 01    STA $01       CRASH COMPUTER
,6A49 A9 05    LDA #$05
,6A4B D0 FC    BNE $6A49     ENDLESS LOOP TO $6A49
,6A4D 20 CC FF JSR $FFCC     CLOSE I/O CHANNELS
,6A40 A9 0F    LDA #$0F
,6A52 A2 08    LDX #$08
,6A54 A0 0F    LDY #$0F
,6A56 20 BA FF JSR $FFBA     SET LOGICAL, 1ST & 2ND ADDR (15,8,15)
,6A59 A9 01    LDA #$01      1 CHARACTER NAME
,6A5B A2 77    LDX #$77      STORED AT
,6A5D A0 6A    LDY #$6A      $6A77
,6A5F 20 BD FF JSR $FFBD     SET FILE NAME (I)
,6A62 20 C0 FF JSR $FFC0     OPEN FILE (15,8,15,"I")
,6A65 A9 00    LDA #$00
,6A67 4C 3B 5B JMP $5B3B     JUMP TO START OF MAIN PROGRAM
,6A6A 23       ???           ASCII #
,6A6B 55 31    EOR $31,X     ASCII U1
,6A6D 3A       ???           ASCII :
,6A6E 20 35 20 JSR $2035     ASCII  5
,6A71 30 20    BMI $2035     ASCII 0
,6A73 33       ???           ASCII 3
,6A74 35 20    AND $20,X     ASCII 5
,6A76 31 49    AND ($49),Y   ASCII 1I
```

THIS PROGRAM WILL CHECK FOR THE ERROR TYPE 23. TAKE SOME TIME TO EXAMINE THESE THREE EXAMPLES AND SEE IF YOU CAN DETERMINE HOW TO BYPASS THE ERROR CHECKING ROUTINE. COMPARE YOUR RESULTS WITH THE THREE BROKEN VERSIONS ON THE NEXT FEW PAGES. REMEMBER THAT THERE ARE MANY DIFFERENT WAYS TO DISABLE THE ERROR CHECKING ROUTINE. TRY TO LEAVE AS MUCH CODE AS POSSIBLE INTACT. MODIFY THE FEWEST BYTES POSSIBLE. LOCATION $CA2C CONTAINS THE DEC $FB COMMAND. THIS MAY BE IMPORTANT TO THE PROGRAM SO DON'T JUST CHANGE THE FIRST FEW BYTE OF THE SUBROUTINE TO JMP $5B3B.

The most important item to look for is the KERNAL subroutines. The logic of this program may be followed by examining the function of the KERNAL calls. The first thing the program will do is set the logical file #8 (AC), the device #8 (XR) and the secondary address #1 (YR). The program will then set the file name. The name is 4 bytes long (AC) and begins at the location of $082F (XR & YR). Then the program will load RAM (program file) from the disk drive. The reason we know that the program will load from the disk drive is because the device number 8 was set earlier. After the program has loaded into memory the computer will jump to location $6000 and begin executing the program that resides there. The AC, XR and YR have been loaded with specific values prior to performing the JSR to the KERNAL. By setting these registers to the proper values, the KERNAL will know which files to open or what program name to use. The programmers reference guide contains a listing of the KERNAL and how the associated registers must be set to perform the specific functions. Pages 272 to 306 contain the KERNAL functions, you may (will) find the layout confusing to use at first. The only time I use these pages is to find out what the function of the registers are when KERNAL calls are used. You will find that the memory map contained in the last few pages of this book will give you a more complete and easier to use listing of the KERNAL.

Notice that JSRs to KERNAL subroutines. Any time a ML program is going to access the disk drive it is necessary to use KERNAL subroutines. If the program is going to load another program from the disk, KERNAL subroutines will be used. If the program is going to read a block of information from a user file, KERNAL subroutines will be used. If the program is going to try to read a bad block on the disk, KERNAL subroutines will be used. If the program is going to check for an I.D. mismatch (error 29), KERNAL subroutines will be used. If it sounds like I am trying make a point, you're catching on. KERNAL subroutines are used extensively in program protection routines. For most of the programs on the market today, all one has to do is to find the KERNAL routines to find the program protection schemes. Most programs contain only one small area that contains the total program protection scheme.

Usually the program will contain a number of files on the disk. It may be necessary to check each file on the disk in order to find the program protection code. I usually start with the first file on the disk and examine it, then examine the next file until I find where the protection schemes resides. Lets take a look at a typical subroutine that will check a disk for a bad block or I.D. mismatch. It does not matter what type of error is on the disk, the error checking routines will be similiar to the example listed. All error checking routines will use similar syntax to accomplish their task. Examine the ML code in figure 4. Then review the same program written in BASIC, figure 2. Notice how similar the two programs are. Comments have been included to make the ML easier to understand.

SUBROUTINE # 1 MODIFIED CODE

```
,2900 A9 00    LDA #$00
,2902 20 BD FF JSR $FFBD
,2905 A9 0F    LDA #$0F
,2907 A2 08    LDX #$08
,2909 A8       TAY
,290A 20 BA FF JSR $FFBA
,290D 20 C0 FF JSR $FFC0
,2910 A9 01    LDA #$01
,2912 A2 55    LDX #$55
,2914 A0 29    LDY #$29
,2916 20 BD FF JSR $FFBD
,2919 A9 05    LDA #$05
,291B A2 08    LDX #$08
,291D A8       TAY
,291E 20 BA FF JSR $FFBA
,2921 20 C0 FF JSR $FFC0
,2924 20 CC FF JSR $FFCC
,2927 A2 0F    LDX #$0F
,2929 20 C9 FF JSR $FFC9
,292C A0 00    LDY #$00
,292E B9 56 29 LDA $2956,Y
,2931 F0 06    BEQ $2939
,2933 20 D2 FF JSR $FFD2
,2936 C8       INY
,2937 D0 F5    BNE $292E
,2939 20 CC FF JSR $FFCC
,293C A2 0F    LDX #$0F
,293E 20 C6 FF JSR $FFC6
,2941 20 CF FF JSR $FFCF
,2944 C9 30    CMP #$30      ORIGINAL VERSION: CMP #$32
,2946 D0 0A    BNE $2952
,2948 20 CF FF JSR $FFCF
,294B C9 30    CMP #$30      ORIGINAL VERSION: CMP #$32
,294D D0 03    BNE $2952
,294F 4C 64 90 JMP $9064
,2952 4C E2 FC JMP $FCE2
,2955 23       ???
,2956 55 31    EOR $31,X
,2958 3A       ???
,2959 20 35 20 JSR $2035
,295C 30 20    BMI $297E
,295E 30 31    BMI $2991
,2960 20 32 0D JSR $0D32
```

ONCE THIS PROGRAM HAS BEEN SAVED ON A DISK WITHOUT ANY ERRORS IT WILL BE NECESSARY TO MODIY THE CODE SO THAT THE PROGRAM DOES NOT WANT TO FIND ANY ERRORS ON THE DISK. THE ONLY CHANGE THAT IS NECESSARY TO MAKE IS: CHANGE THE TWO COMPARISONS SO THAT THEY WILL WANT TO FIND $30 & $30 (ASCII 0 & 0). WHEN THE PROGRAM GOES TO THE DISK AND READS THE BLOCK IT WILL BE GOOD. SO THE ERROR MESSAGE RETURNED WILL BE 00,OK,00,00. ASCII 0 = $30, THAT IS WHY THE COMPARISON WILL HAVE TO BE TO $30 NOT $00. THE PROGRAM THAT CONTAINS THIS SUBROUTINE CAN BE SAVED BACK TO DISK AND THE PROGRAM WILL FUNTION NORMALLY ON ANY DISK THAT DOES NOT CONTAIN AN ERROR.

OTHER CHANGES COULD HAVE BEEN MADE TO THE PROGRAM AND ACCOMPLISHED THE SAME TASK. THE CODE THAT REPRESENTS BNE $2952 COULD HAVE BEEN CHANGED TO NOP ($EA) AT BOTH LOCATIONS. THIS WAY THE PROGRAM WOULD HAVE 'FALLEN THRU' NO MATTER IF THERE WAS AN ERROR OR NOT. THE JMP $FCE2 COULD HAVE BEEN CHANGED TO JMP $9064. THIS WAY THE PROGRAM WOULD HAVE JUMPED TO THE MAIN PROGRAM WHETHER THERE WAS AN ERROR OR NOT. CAN YOU FIND ANY OTHER WAYS TO CHANGE THE CODE SO THAT THE PROGRAM WILL RUN ON ANY DISK?

SUBROUTINE # 2 MODIFIED CODE

```
,3551 A9 03    LDA #$03
,3553 A2 C0    LDX #$C0
,3555 A0 35    LDY #$35
,3557 20 BD FF JSR $FFBD
,355A A9 0F    LDA #$0F
,355C A2 08    LDX #$08
,355E A0 0F    LDY #$0F
,3560 20 BA FF JSR $FFBA
,3563 20 C0 FF JSR $FFC0
,3566 A9 01    LDA #$01
,3568 A2 BF    LDX #$BF
,356A A0 35    LDY #$35
,356C 20 BD FF JSR $FFBD
,356F A9 03    LDA #$03
,3571 A2 08    LDX #$08
,3573 A0 03    LDY #$03
,3575 20 BA FF JSR $FFBA
,3578 20 C0 FF JSR $FFC0
,357B A2 0F    LDX #$0F
,357D 20 C9 FF JSR $FFC9
,3580 A0 00    LDY #$00
,3582 B9 C3 35 LDA $35C3,Y
,3585 F0 06    BEQ $358D
,3587 20 D2 FF JSR $FFD2
,358A C8       INY
,358B D0 F5    BNE $3582
,358D 20 CC FF JSR $FFCC
,3590 A2 0F    LDX #$0F
,3592 20 C6 FF JSR $FFC6
,3595 20 CF FF JSR $FFCF
,3598 29 FF    AND #$FF
,359A 48       PHA
,359B 20 CF FF JSR $FFCF
,359E C9 0D    CMP #$0D
,35A0 D0 F9    BNE $359B
,35A2 20 CC FF JSR $FFCC
,35A5 68       PLA
,35A6 C9 30    CMP #$30    ORIGINAL VERSION: CMP #$32
,35A8 D0 03    BNE $35AD
,35AA 4C 0A 26 JMP $260A
,35AD A9 AA    LDA #$AA
,35AF 85 FB    STA $FB
,35B1 A5 34    LDA $34
,35B3 85 FC    STA $FC
,35B5 A4 00    LDY $00
,35B7 91 FB    STA ($FB),Y
,35B9 C8       INY
,35BA D0 FB    BNE $35B7
,35BC 4C E2 FC JMP $FCE2
,35BF 23       ???
,35C0 49 2F    EOR #$2F
,35C2 4F       ???
,35C3 55 31    EOR $31,X
,35C5 3A       ???
,35C6 33       ???
,35C7 20 30 20 JSR $2030
,35CA 30 31    BMI $35FD
,35CC 20 30 32 JSR $3230
,35CF 0D 00 00 ORA $0000
```

ONLY ONE BYTE OF THIS ROUTINE HAD TO BE CHANGED IN ORDER TO MAKE THIS PROGRAM WORK. THIS ROUTINE WILL NOW WANT TO SEE THE ERROR MESSAGE 00,OK,00,00. OTHER CODE MAY BE CHANGED TO ACCOMPLISH THE SAME TASK.

TRY TO COME UP WITH OTHER METHODS THAT WILL WORK.

SUBROUTINE # 3 MODIFIED CODE

```
,6A00 A9 05    LDA #$05
,6A02 A2 08    LDX #$08
,6A04 A0 05    LDY #$05
,6A06 20 BA FF JSR $FFBA
,6A09 A9 01    LDA #$01
,6A0B A2 6A    LDX #$6A
,6A0D A0 6A    LDY #$6A
,6A0F 20 BD FF JSR $FFBD
,6A12 20 C0 FF JSR $FFC0
,6A15 B0 2E    BCS $6A45
,6A17 A9 0F    LDA #$0F
,6A19 A2 08    LDX #$08
,6A1B A0 0F    LDY #$0F
,6A1D 20 BA FF JSR $FFBA
,6A20 A9 0C    LDA #$0C
,6A22 A2 6B    LDX #$6B
,6A24 A0 6A    LDY #$6A
,6A26 20 BD FF JSR $FFBD
,6A29 20 C0 FF JSR $FFC0
,6A2C C6 FB    DEC $FB
,6A2E A2 0F    LDX #$0F
,6A30 20 C6 FF JSR $FFC6
,6A33 A9 0F    LDA #$0F
,6A35 20 CF FF JSR $FFCF
,6A38 C9 32    CMP #$32
,6A3A F0 09    BEQ $6A45    ORIGINAL VERSION: BNE $64A5
,6A3C A9 0F    LDA #$0F
,6A3E 20 CF FF JSR $FFCF
,6A41 C9 33    CMP #$33
,6A43 D0 08    BNE $6A4D    ORIGINAL VERSION: BEQ $64AD
,6A45 A9 00    LDA #$00
,6A47 85 01    STA $01
,6A49 A9 05    LDA #$05
,6A4B D0 FC    BNE $6A49
,6A4D 20 CC FF JSR $FFCC
,6A40 A9 0F    LDA #$0F
,6A52 A2 08    LDX #$08
,6A54 A0 0F    LDY #$0F
,6A56 20 BA FF JSR $FFBA
,6A59 A9 01    LDA #$01
,6A5B A2 77    LDX #$77
,6A5D A0 6A    LDY #$6A
,6A5F 20 BD FF JSR $FFBD
,6A62 20 C0 FF JSR $FFC0
,6A65 A9 00    LDA #$00
,6A67 4C 3B 5B JMP $5B3B
,6A6A 23       ???
,6A6B 55 31    EOR $31,X
,6A6D 3A       ???
,6A6E 20 35 20 JSR $2035
,6A71 30 20    BMI $2035
,6A73 33       ???
,6A74 35 20    AND $20,X
,6A76 31 49    AND ($49),Y
```

THIS VERSION WILL WORK ON ANY DISK THAT DOES NOT CONTAIN ANY ERRORS. THIS TIME THE BRANCH INSTRUCTIONS WERE CHANGED. ON THE OTHER SUBROUTINES THE COMPARE INSTRUCTIONS WERE CHANGED. EITHER METHOD WILL WORK ON ALL THREE ROUTINES. JUST TRY ONE OR THE OTHER, NOT BOTH. WHEN YOU ARE TRYING TO FIND THE ERROR PROTECTION ROUTINES HUNT FOR THE KERNAL SUBROUTINES OR THE 'U1'. THEY WILL BE USED IN MOST PROTECTION SCHEMES.

The first thing that the protection scheme will do is open two files to the disk drive. One will be the error channel, the other will the data channel. It is necessary to have both channels open prior to attempting to read a block of information from the disk. Then the ML program will print the following ASCII characters to the disk drive: U1: 5 0 01 02. This the same command that BASIC will print to the disk drive in order to perform a user call or block read. The 'U1:' means to read a block of information from the disk drive. The '5' will be the data channel to use (previously opened). The '0' will be drive number 0. The '1' is the track to use. The '2' is the sector. Once the disk drive has tried to read this block of information the ML program will read the error channel. It is necessary to INPUT one byte at a time from the error channel. If the block was successfully read, the error channel will return the message of '00,OK,00,00'. If there was any type of error, the error channel will return the error number, type of error, track of error and sector of error (21,READ ERROR,01,02). For the error types 20, 21, 22, 23 and 27 the message will be similar to the preceding error message. The only difference in the error message will be the error type (21), track (01) and sector (02) of the actual error returned. If the error type 29 is encountered the following error message will be returned: 29,DISK ID MISMATCH,01,02. Once the first byte of the error message is returned the ML program will CMP #$32 (compare to hex $32, which represents 2 ASCII). If this byte does not equal ASCII 2 the program will BNE (branch if not equal) to where the program will crash. If the comparison is equal the program will 'fall thru' the branch and continue to the next INPUT, where another byte will be input from the disk drive. Another comparison will be made CMP #$31 (compare to hex $31, which equals ASCII 1). If this byte does not equal ASCII 1 the program will BNE (branch if not equal) to where the program will crash. If the comparison is equal the program will 'fall thru' the branch and continue to the next instruction, where the program will close the files and JMP (jump) to the start of the main program. The first comparison was for ASCII 2, the second comparison was for ASCII 1, this is the error type 21. If the program was looking for error type 29, the second comparison would be for hex $39 (ASCII 9). The program will check to see if the error type 29 exists at the proper location on the disk. If it does exist, the program will continue normally. If the error does not exist the program will crash. This is typical of most program protection schemes. Some programmers will try to make the protection scheme appear slightly different than what is listed here. Instead of INPUTing a character ($FFCF) some programs will INPUT a byte ($FFA5), others will GET a Character ($FFE4). Others programs will perform a 'B-R' instead of the 'U1'. Occasionally the programmers will try to confuse the novice by storing the ASCII characters in reverse order (20 10 0 5 :1U). ML can read the ASCII characters from front to back or vice versa.

The computer will store all numbers and letters in its memory as a number. The number that the computer uses will not be the same as the ASCII number. Example: hex $30 = 0 ASCII: hex $31 = 1 ASCII, hex $32 = 2 ASCII, etc. Use the chart at the end of the memory map section to find a complete listing the hex, decimal, ASCII, BASIC and screen codes.

How do you go about finding this protection scheme, and once you find it how can you modify the code so the program will run on a disk that has no errors?

First load every program into memory from the ML monitor. Second use the 'H' command to hunt for the KERNAL subroutines, the 'U1' or 'B-R'. The other way to find the protection schemes is to use the 'I' command to look for the 'U1' or the 'B-R'. Either way will usually turn up the proper location of the protection scheme. Keep in mind that some programs will have valid KERNAL calls that will have nothing to do with program protection. These are programs that will load or save information on the disk. Word processors, spread sheets, and data bases all access the disk for reasons other than program protection. Try to find the KERNAL calls that are located near the 'U1' or the 'B-R'. This will probably be the area of the program protection. Most programmers will put all their program protection in one small area. One interesting thing to do is to look at a number of different program protection schemes. This will allow you to find how a number of different programmers expand on the examples listed.

If you belong to a users group or if you have friends who have broken software, try to borrow the broken version of the program. Many times it will be quite clear as to where the protection schemes are and how they were defeated. Start with programs that have been broken by someone else first. These programs usually are the most straight forward and have easy to spot program protection schemes.

Remember: in order for the program to read a bad block or to check for an I.D. mismatch the following commands must be issued. This will apply for BASIC or ML.

1). Open the error channel (15) to the disk drive.
2). Open a data channel to the disk drive
3). Print 'U1:' or 'B-R' CHANNEL + DRIVE NUMBER + TRACK + SECTOR to the disk drive.
4). Read the error channel to check for the error.
5). Compare the error received to the desired error.
6). Execute the main program if the proper error is found.
7). Crash the program if no error or the wrong error is found.

This format is followed by the majority of the programs on the market today.

CARTRIDGE PROGRAMS

Many cartridges may be easily down loaded to disk. To do this it will generally be necessary to have $2.00 switch added to your computer or you will need a mother board. I prefer the $2.00 switch, it is easier to use and does not require any table space. One word of caution before we start. You can damage your computer or the cartridge by inserting the cartridge with the computer turned on. It is possible to save the cost of the switch and by inserting the cartridge after the power is turned on. This technique is used by many people, I do not recommend that you try it.

If you don't have a mother board it will be a good idea to install the switch that I mentioned before. This switch will make it easy to down load most cartridge programs. By installing the switch you may void the warranty of your computer. Installing the switch requires some knowledge of electronics and should only be attempted by qualified personel. Now is the time to decide to buy a mother board or install the switch. This switch will be located in the right rear of the computer, near the cartridge port. Below is a set of instructions to follow if you wish to install the switch.

1). Purchase a subminiature DPST toggle switch.
2). Remove the power, serial port cable and any other items from the computer before starting. Open the computers case and locate the cartridge port at the right rear of the computer.
3). Find an open spot just to the right of the cartridge port, carefully drill a hole to mount the switch. It is cramped in this area, use care.
4). The pins are numbered from the center of the computer. It will be necessary to locate pins 2, 3 & 8 of the top row. It is not necessary to modify any pins of the lower row on the connector.
5). Pins 2 & 3 contain the 5 volt power to the cartridge. Both pins are connected to the same place on the printed circuit board. You will need to cut both pins 2 & 3. Solder the lower half (printed circuit board side) of the pins 2 & 3 together, then solder the upper half (connector side) of pins 2 & 3 together. Remember they both contain power and it will be all right to connect these pins together.
6). Install wires to one side of the DPST switch from the upper half of pins 2 & 3 and the from the lower half of pins 2 & 3 to the DPST switch.
7). Cut pin 8 of the top row of the connector, run wires from the DPST switch to the top and bottom half of pin 8.
8). Verify that one side of the switch will turn on pins 2 & 3 and that the other side of the switch will turn on pin 8. Install switch and close the case of computer. Reinstall the cables and hook up your accessories to the computer.
9). Your done.

Pins 2 & 3 contain the 5 volt power to the cartridge. Pin 8 is the GAME line to the cartridge. These three pins must be switched off to allow for the cartridge to be inserted without taking over control of the computer at power up.

A). Turn your computer off and turn the cartridge switch off.
B). Insert the cartridge and then turn the computer on.
C). You should see 30719 BASIC BYTES FREE. The cartridge should not take over control of the computer, but the presence of the cartridge will be detected by the computer.
D). You can now load the HIMON and execute it.
E). Turn the cartridge switch on.
F). The ML monitor should still have control of your computer. If the screen goes berserk, or the characters on the screen appear to be scrambled when you flip the cartridge switch, you will not be able to down load this particular cartridge. Some cartridges will modify the character set when power is applied to them, I have not found an easy way to circumvent this problem. So for the time being, don't try to down load these cartridges that scramble your screen characters. Try another cartridge, most will not be any problem to down load.
F). Use the 'M' command to examine the memory starting at $8000.
G). If the cartridge is an auto start cartridge you should find the code similiar to below:

```
;8000 09 80 70 80 C3 C2 CD 38
;8008 30
```

The first two bytes contain the vectors to the cold start location of the cartridge ($8009 in this example). The next two bytes contain the warm start vectors for the cartridge ($8070 in this example). The next three bytes are the shifted letters CBM. The last two bytes must be the hex representation for 80. When power is first turned on, your computer will check to see if these nine bytes are located at $8000. If they are, the computer will suspend all of its normal activities and turn control over to the cartridge. The cold start vector is used if the computer was just powered up and the warm start vector is used if the 'RESTORE' key was pressed. These vectors will point to the appropriate location for your particular cartridge and may not be the same as listed in the example. What is important is the CBM80. The computer must see these five bytes, if the cartridge is to be auto start. It doesn't matter if a cartridge is present or not. If these five bytes are present when the computer is reset or when the 'RESTORE' key is pressed control will be turned over to the vectors. This is why cartridge programs that have been down loaded to disk will run after the computer is reset. The CBM80 will perform the same from RAM or ROM.

By turning the switch off before power up, we can keep the computer from recognizing the data contained in the cartridge. The computer will be able to tell that there is something located at the cartridge location, but it will not be able to recognize any of the data contained on the cartridge. This is why the computer will have only 30719 BASIC BYTES FREE.

Cartridges normally occupy the memory location from $8000 to $9FFF. This is 8K of memory. To save the information from the cartridge to disk is fairly simple to do.

H). Use the 'S' commmand of the ML monitor to save the memory to disk.

```
S "NAME",08,8000,A000          'RETURN'
```

$8000 is the starting address of the cartridge, $A000 is the ending location plus one, of the memory to be saved. Always add at least one extra byte of data, some monitors will not save the last byte of data.

The computer may now be turned off and the cartridge removed. The program can be loaded back in from disk. Use the ,8,1 command so that the information will be relocated back into memory in the same location. Try pressing the 'RESTORE' key. If the program does not start executing, use SYS 64738 or press your reset button to cause a warm start. This will normally bring the program to life.

Occasionally the cartridge program will not work after it has been down loaded to disk. Some cartridges put a small amount of program protection in the ML code. Cartridges programs are contained on ROM chip(s). This means that the information stored on the chip can only be read, no information can be written to the chip. ROM memory can be considered to be permanent memory and it cannot be modified. The data on the chip can not be modified or changed in any way. What some cartridges try to do is modify their own memory as the program runs. Because the program is contained on a ROM chip the memory can not be changed, so the program will execute normally. Once this program has been down loaded to disk and loaded back into memory, the program now resides in RAM memory. Programs which reside in RAM can be changed or modified at any time. Now when the program runs it can erase itself from memory or it can change just one or two bytes of information. Remember that ROM based programs can not be modified while they run, whereas RAM based programs can be modified at any time. If the cartridge program contains any type of protection it will only affect the program if it resides in RAM.

In order to find the protection placed on a cartridge it will be necessary to load and execute the HIMON.

1). Load the cartridge program from the disk.
2). Change the CBM80 to CBM00. This will prevent the cartridge program from taking over control of your computer if you use the reset button. REMEMBER to change this back to CBM80 prior to saving the program back to disk.
3). Transfer the memory block $8000 - $9FFF to $4000.

```
T 8000 9FFF 4000               'RETURN'
```

This will allow you to have an area in memory to compare the original program with.

4). Find the cold and warm start vectors for the cartridge. Use the 'G' command to execute the cartridge. Try the cold start vectors first. It may be necessary try the warm start vectors after you reload the program.

G 8009 'RETURN' in the preceding example

5). After the program crashes, reset the computer with your reset button. Use the SYS 49152 to re-enter the ML monitor.
6). Use the compare command to find any memory locations which were changed when the program was executed. You previously transferred the program to $4000, use this for the comparison. Remember the ROM based program can not be modified or changed while it is running, only RAM based programs can. Most cartridge programs which contain any protection will usually modify only a few bytes of memory or they will wipe out an entire block of memory.

C 8000 9FFF 4000 'RETURN'

7). Write down the memory locations if only a few byte have been modified. If a block or two of memory has been modified write down the general area of memory that has been modified. This will give you an area to start looking at.

If only one or two bytes of memory have been changed the program will probably do something similar to the following example. For instance, if memory location that was changed was $8400 and its value was changed to 00 then you might find the code:

```
., 8040 A9 00    LDA #$00
., 8042 8D 00 84 STA $8400
```

ROM based programs would not allow this to affect its operation. Storing any value at a location which contains a ROM cartridge will not change the value of the cartridge. If we store a value to an area of memory where a RAM based program resides, the value contained their will be changed. In order to find where the code resides you could either use the 'D' command and search the program line by line or you could use the 'H' to hunt for the affected memory location. Sometime the only way to spot the protection schemes is to look at the code with the 'D' command, line by line. Anytime you see the program storing anything at locations $8000 to $9FFF you can be sure it is part of a protection scheme. The way to prevent the code from modifying the RAM based program is to change the code at $8042 thru $8045 to NOP ($EA). This is the No OPeration command, the do nothing command for ML. The original program would STA (Store The Accumulator) at location $8400. The fixed program will do nothing. The code that resides at $8400 will not be changed. Hence, the program will operate normally. Some programs will use 2 or 3 different locations where they will try to change some of the code. Try to find them one at a time. It's not hard, it just takes a little time.

```
.8000 09 80    ORA #$80
.8002 D4       ???
.8003 80       ???
.8004 C3       ???
.8005 C2       ???
.8006 CD 38 30 CMP $3038
.8009 A9 00    LDA #$00
.800B 8D 00 80 STA $8000
.800E A9 20    LDA #$20
.8010 8D 00 84 STA $8400
.8013 A9 60    LDA #$60
.8015 8D 00 90 STA $9000
.8018 A9 00    LDA #$00
.801A 8D 00 94 STA $9400
.801D 8D 00 95 STA $9500
.8020 8D 20 95 STA $9520
.8023 8D FF 80 STA $80FF
.8026 A9 F7    LDA #$F7
.8028 A2 1F    LDX #$1F
.802A A0 00    LDY #$00
.802C 8C 0E DC STY $DC0E
.802F 8C A9 00 STY $00A9
P
```

COLD START VECTOR

WARM START VECTOR

SHIFTED CBM

ASCII 80

CARTRIDGE PROTECTION

START OF PROGRAM

The preceding example program changed the memory of the RAM based program by directly storing a value in memory. As you would expect this is referred to as direct addressing (sometimes called absolute addressing). Values in memory can also be changed by something called INDEXED addressing, INDIRECT INDEXED addressing and INDEXED INDIRECT addressing. This can be a little hard to understand at first. I will try to make it a clear as possible. Look at the following example:

```
.,8070 A9 00    LDA #$00
.,8072 A0 10    LDY #$10
.,8074 99 40 81 STA $8140,Y   STA $8140 + Y  ($8150)
.,8077 C8       INY           Y=Y+1          ($11)
.,8078 99 40 81 STA $8140,Y   STA $8140 + Y  ($8151)
.,808B A9 42    LDA #$42      START OF MAIN PROGRAM
```

This is an example of INDEXED addressing. This is very easy to understand, just add the value of 'Y' to the value of the memory location to arrive at the actual address. The value of 'Y' can any value from 00 to FF, so the actual memory location must be within 255 bytes of $8140 in the preceding example. Indexed addressing is just as easy to 'fix' as direct addressing. Change the bytes at $8074 thru $8076 to NOP and also change the bytes at $8078 thru $807A to NOP. The address may be indexed with either the 'X' or the 'Y'.

Now lets take a look at INDIRECT INDEXED addressing. This type of indexing uses only the 'Y' register and is fairly common. INDIRECT INDEXED addressing may be used to change a few bytes or a block of memory.

```
.,8080 A9 00    LDA #$00      THE STARING ADDRESS IS $8200
.,8082 85 B8    STA $B8       STA LOW BYTE (00) AT $00B8
.,8084 A9 82    LDA #$82
.,8086 85 B9    STA $B9       STA HIGH BYTE (82) AT $00B9
.,8088 A0 00    LDY #$00
.,808A A9 FF    LDA #$FF
.,808C 91 B8    STA ($B8),Y   STA AT $8200 + Y
.,808E C8       INY           Y=Y+1
.,808F D0 FB    BNE $808C     IF Y  0 THEN BRANCH TO $808C
.,8091 A9 00    LDA #$00      START OF MAIN PROGRAM
```

INDIRECT INDEXED addressing uses the zero page (see your book on ML) to store the actual address as a vector. The address must be stored in the usual low byte, high byte format needed for the 6510 microprocessor. When the computer performs STA ($B8),Y it will Store The Accumulator at the location contained in $B8 and $B9 plus 'Y'. The program has stored the address $8200 at $B9 and $B8, the 'Y' is initially set to 00. So the computer will first STA at $8200. Then 'Y' is incremented, a conditional branch (BNE) is made to see if 'Y' equals 00. If 'Y' does not equal 00, then the program will branch back to address $808C and perform the operation again. This time 'Y' equals 1, so the new address to STA will be $8201. This will continue to happen until 'Y' equals $FF. When we increment $FF it will become 00. The BNE will find that 'Y' now equals 00 and the program will 'fall through' to the next instruction. This routine will fill the memory from $8200 to

$82FF with 00. If you find that your program has a protection routine similar to the one described, there are several options to prevent the program from erasing memory.

First is to change the high byte stored at $B9 from $82 to $20 (or some other value). This will cause the program to fill the memory from $2000 to $20FF with 00. This area of memory in not used by the cartridge, cartridges will only reside at the address of $8000 and above.

Second would be to change the first three bytes of the routine to jump by the program protection routine:

```
.,8080 4C 91 80 JMP $8091     JUMP BY THE PROTECTION
```

The third method would be to change the code at $808C and $808D to NOPs. When the program came to this location it would do nothing. Example:

```
.,808A A9 FF    LDA #$FF
.,808C EA       NOP
.,808D EA       NOP
.,808E C8       INY
```

Either one of the three methods will perform the same task equally well. Choose one and try it if you find similiar code in a program that you are looking at. Just as there are many ways to skin a cat, there are many ways to remove program protection. If you are not sure on how to 'fix' the program, try something that seems to make sense. If what you changed works, fine. If it doesn't work, that's OK. You should restore the code to original or reload the program and then try something else. Remember that you cannot damage you computer by experimenting with the code. Many times I will try two or three things before one works.

The third method of addressing used in cartridge program protection is the INDEXED INDIRECT. This type of indexing uses only the 'X' register. Fortunately, most programmers do not know how to use this type of indexing. This indexed indirect addressing can get very complicated. I would suggest that you get out your book on 6510 or 6502 ML programming. Read and re-read the section on INDEXED INDIRECT addressing. INDEXED INDIRECT addressing may be used to modify a few bytes or to wipe out a block of memory

```
.,8040 A9 00    LDA #$00        LOW BYTE MINUS $10
.,8042 85 FB    STA $FB         STA AT $00FB
.,8044 A9 71    LDA #$71        HIGH BYTE MINUS $10
.,8046 85 FC    STA $FC         STA AT $00FC
.,8048 A2 10    LDX #$10        X ADDED TO LOW & HIGH BYTE
.,804A A9 00    LDA #$00
.,808C 81 FB    STA ($FB,X)     STA AT $8110
.,8050 A9 AF    LDA #$AF        START OF MAIN PROGRAM
```

The desired address is $8110, low byte $10, high byte $81. The 'X' register value is subtracted from both the low byte and the high byte. These new values are then stored on the zero page. When the program is run, the value of the 'X' will be added to both the low and high byte of the address.

Some cartridge programs will use the above methods of addressing for reasons other than program protection. Many cartridge programs will store values at memory locations other than $8000 and above. Game programs will need to store the high scores and other associated information some where in memory. Word processors will need to store text at some area of memory not used by the cartridge. Keep in mind that program protection is only a small part of the actual program. If the code that you are looking at is trying to change the program at $8000 or above, then it is for program protection.

CARTRIDGES 16K or MORE

Some of the newer cartridges contain 16K or more of memory. Most cartridges contain only 8K of memory ($8000 - $9FFF). Some contain 16K of memory ($8000 - $BFFF). Others contain multiple ROM chips and the program will switch between the different banks of memory contained in the cartridge, depending upon the function desired. These are cartridges that contain a word processor, a data base and a spread sheet, all in the same cartridge. I would not recommend that you seriously consider trying to down load a cartridge that uses bank switching techniques. Extensive modification of the ML code is necessary for the proper operation of the program.

16K cartridges are similiar to the 8K cartridges. The main difference is that the memory normally occupied by the BASIC interpreter is also used by the cartridge. When the cartridge is first plugged in and the power turned on, the computer will check the cartridge port for the CBM80. If the computer finds the CBM80, it will check the GAME line (#8) for the presence of the cartridge. If line #8 is present the computer will allocate the memory from $8000 to $9FFF (8K) for the cartridge. If the cartridge uses 16K, the entire memory ($8000 to $BFFF) is made available by modifying memory location $0001. By storing the hex value of $36 at $0001 the BASIC interpreter will be flipped out and the cartridge memory will reside in this memory area. If you suspect a 16K cartridges try storing $36 at location $0001. Examine the code at $A000 and above, if any code resides here the cartridge is 16K. 16K cartridges can be saved to disk in the same manner as the 8K cartridges, just use the following save command.

```
S "NAME",08,8000,C000                    'RETURN'
```

This will save the entire 16K of cartridge memory to disk. Program protection is similiar to the 8K, only there is now more room in which to put program protection. Usually you will find the same protection schemes, only there will be more room for them. In an 8K cartridge there might be one, may be two schemes to defeat, in the 16K expect to find at least one, may be three or four different areas of protection. Use the same techniques as before to locate and fix the program protection schemes. One additional item you will need to add to the 16K programs. It will be necessary to turn off BASIC, the KERNAL or both prior to executing the program. This will prevent any chance of interference between the program and the operating system of your computer. The first program is an example of how to turn off BASIC from ML. The second program will allow you to turn off the KERNAL from ML.

```
.,0A00 A9 36    LDA #$36        REM: TURN OFF BASIC
.,0A02 85 01    STA $01         $01 WILL CONTROL MEMORY
.,0A04 4C 09 80 JMP $8009       START OF PROGRAM

.,0A00 A9 35    LDA #$35        REM: TURN OFF KERNAL
.,0A00 85 01    STA $01
.,0A04 4C 09 80 JMP $8009
```

These programs are completely relocatable and may be moved any where in memory. The jump command can be determined from your particular program.

ADVANCED ML PROTECTION

This chapter on advanced program protection will try to give you clues on where to start looking when you run into some one who has taken additional time with the program protection scheme. Most programmers will use only the easiest methods of program protection. The more sophisticated programmers will use a little more complex method to protect their programs. A few programmers will use some really radical methods to protect their software. The same rules apply no matter how complex the programmers try to make things.

When you are looking at a protected program for the first time it will be necessary to follow a logical procedure in order to find the program protection in the least amount of time. Look at program protection as if it is a big crossword puzzle. The first thing to do is to take care of the obvious and easy to correct protection schemes. Then move on to the more difficult schemes, finally to the hardest. I will try to give you a technique to use when breaking programs.

1). LOAD and RUN the original program. Keep track of how long the program loads until the bad block is encountered, if one is used. This will also give you an idea of just how the program is supposed to load until the protection scheme is encountered.
2). Copy the original disk with BACKUP 228. Many programs only need to be backed up properly to work. BACKUP 228 will make a good copy, without putting any errors on the destination disk.
3). LOAD and RUN the copy. If it works, you're done. If it doesn't work, pay attention to how long the program loads before it crashes. It is important to know how far the program gets before checking for the error.
4). Examine the BAM and directory. Correct the directory and the BAM so that a complete file listing may be obtained.
5). Look for any bogus files on the disk. These are files that do not contain any useful information, they just take up space.
6). Load and execute a ML monitor. Load in each and every program. Hunt for KERNAL calls, 'U1' or 'B-R' in an attempt to find where the program protection is located. Use the 'I' command to see if the program is written in BASIC.
7). Once you find the area of the program that contains the program protection it will be necessary to disassemble the ML code in that area.
8). Find the code that reads the error channel and makes the comparison.
9). Find all the KERNAL calls and write down what each does. Use a separate sheet of paper, if necessary, to keep track of the KERNAL calls. Comments are very important when you are looking at the code. They can be the most important help that there is when you are examining ML.
10). Once you have commented the code, try to decide on a course of action to remove the need for the error to be on the disk.

11). Modify the code as necessary.
12). Save the code back to the disk and try it. If you don't succeed the first time, don't give up. Try another approach to the problem.

This all sounds so simple. Doesn't it?

Most of the time it is. The average program takes me less than an hour to break. This is only an average. Some programs have taken less than 15 minutes, others more than a day. It all depends how much protection is contained on the disk, and how sneaky the programmer is. When you are just starting out it is important to look at as many different programs as possible. Some programs will do some really sneaky tricks, most will not. Set a time limit for yourself, if you can not find the program protection schemes within a hour or so go on to the next program. Do not start out on programs that you know have the best protection schemes. I would recommend that you start out by looking at programs that other people have broken. Once you are able find the program protection, look to see what was done to break the program. The best way to learn any thing is to look at what others have done and then to do it yourself.

All programs loaded from disk will be stored in RAM. Any program stored in RAM can be changed or modified as the program runs. Some programs will load a file from the disk and then modify the file before it runs. Other programs will load a file from the disk and move the code to another location before executing it. A few programs will both modify the code and move it in memory. I have included some commented examples of these types of programs. Take some time to examine the programs before going further.

SUBROUTINE # 4 ORIGINAL CODE

```
,2100 A9 60    LDA #$60      THIS ROUTINE WILL CHECK FOR THE PRESENCE OF A
,2102 8D FF 80 STA $80FF     CARTRIDGE (ML MONITOR). IF PRESENT THE PROGRAM
,2105 20 FF 80 JSR $80FF     WILL CRASH.
,2108 A9 21    LDA #$21      THIS STARTS THE INDIRECT INDEXED ADDRESSING
,210A 85 FC    STA $FC       THAT WILL CHANGE SOME OF THE FOLLOWING CODE.
,210C A2 00    LDX #$00
,210E 86 FB    STX $FB       MANY OF THE FOLLOWING SUBROUTINES WILL BE CHANGED
,2110 A9 FF    LDA #$FF      FROM JSR $20XX TO JSR $FFXX.
,2112 A0 32    LDY #$32
,2114 91 FB    STA ($FB),Y   THIS IS DONE TO HIDE THE CODE FROM THE USER.
,2116 A0 35    LDY #$35
,2118 91 FB    STA ($FB),Y   THIS WILL MAKE THE PROTECTION SCHEME A LITTLE
,211A A0 3E    LDY #$3E      TOUGHER TO SPOT
,211C 91 FB    STA ($FB),Y
,211E A0 4F    LDY #$4F
,2120 91 FB    STA ($FB),Y
,2122 A0 57    LDY #$57
,2124 91 FB    STA ($FB),Y
,2126 A0 70    LDY #$70
,2128 91 FB    STA ($FB),Y   THIS IS THE END OF THE INDIRECT INDEXED ADDR.
,212A A9 0F    LDA #$0F
,212C A2 08    LDX #$08
,212E A0 0F    LDY #$0F
,2130 20 BA FF JSR $FFBA     THIS WILL BE CHANGED TO $FFBA
,2133 20 C0 FF JSR $FFC0     THIS WILL BE CHANGED TO $FFC0
,2136 A9 01    LDA #$01
,2138 A2 F3    LDX #$F3
,213A A0 21    LDY #$21
,213C 20 BD FF JSR $FFBD     THIS WILL BE CHANGED TO $FFBD
,213F A9 05    LDA #$05
,2141 A2 08    LDX #$08
,2143 A0 05    LDY #$05
,2145 20 BA FF JSR $FFBA     SET LOGICAL, 1ST & 2ND ADDR (5,8,5)
,2148 20 C0 FF JSR $FFC0     OPEN FILE
,214B A2 0F    LDX #$0F
,214D 20 C9 FF JSR $FFC9     THIS WILL BE CHANGED TO $FFC9
,2150 A2 00    LDX #$00
,2152 BD F4 21 LDA $21F4,X
,2155 20 D2 FF JSR $FFD2     THIS WILL BE CHANGED TO $FFD2
,2158 E8       INX
,2159 E0 0C    CPX #$0C
,215B D0 F5    BNE $2152
,215D 20 CC FF JSR $FFCC     CLOSE I/O CHANNELS
,2160 A2 0F    LDX #$0F
,2162 20 C6 FF JSR $FFC6     OPEN CHANNEL FOR INPUT
,2165 A2 00    LDX #$00
,2167 20 CF FF JSR $FFCF     INPUT CHARACTER
,216A C9 32    CMP #$32      CMP TO $32 (ASCII 2)
,216C D0 07    BNE $2175
,216E 20 CF FF JSR $FFCF     THIS WILL BE CHANGED TO $FFCF
,2171 C9 37    CMP #$37      CMP TO $37 (ASCII 7)
,2173 F0 04    BEQ $2179
,2175 78       SEI
,2176 4C 76 21 JMP $2176     ENDLESS LOOP
,2179 4C 24 09 JMP $0924     START OF MAIN PROGRAM
```

THE FOLLOWING CODE IS AN EXAMPLE OF THE ROUTINE THAT MAY BE USED TO MOVED ML CODE IN MEMORY. THIS ROUTINE WILL ALSO MODIFY EVERY BYTE THAT IS MOVED. FORTUNATELY MOST PROGRAMS DO NOT USE THIS TYPE OF PROTECTION SCHEME TO HIDE THE CODE. IF YOU DO NOT UNDERSTAND HOW IT WORKS YOU SHOULD TAKE TIME TO EXAMINE THE CODE. THIS MODIFICATION DOES NOT OCCUR IN MANY PROGRAMS, BUT YOU SHOULD BE AWARE THAT IT DOES EXIST.

```
,5FFB A9 00    LDA #$00       THIS IS AN EXAMPLE OF INDIRECT INDEXED ADDRESSING
,5FFD 85 FD    STA $FD        USED TO MOVE ML CODE FROM $2000 - $3000  TO
,5FFF A9 30    LDA #$30       $3000 - 3FFF.
,6001 85 FE    STA $FE
,6003 A9 00    LDA #$00
,6005 85 FB    STA $FB
,6007 A9 20    LDA #$20
,6009 85 FC    STA $FC
,600B A0 00    LDY #$00
,600D B1 FB    LDA ($FB),Y    IT WILL ALSO MODIFY THE CODE BY THE USE OF THE
,600F 49 A0    EOR #$A0       EOR #$A0 COMMAND. THIS IS THE EXCLUSIVE OR
,6011 91 FD    STA ($FD),Y    COMMAND. SEE YOUR BOOK ON ML FOR FURTHER INFO.
,6013 C8       INY
,6014 D0 F7    BNE $600D
,6016 E6 FC    INC $FC
,6018 A5 FC    LDA $FC
,601A C9 30    CMP #$30
,601C D0 ED    BNE $600B
,601E 60       RTS
```

TO SEE THE EFFECTS OF THE CODE WILL REQUIRE YOU TO CHANGE THE RTS ($60) TO BRK ($00). THEN USE THE GO COMMAND TO EXECUTE THE SUBROUTINE: G 5FFB. THE ML CODE FROM $2000 TO $2FFF WILL BE MODIFIED (EOR #$A0) AND MOVED (STA ($FD),Y) IN MEMORY. BY CHANGING THE RTS ($60) TO BRK ($00) YOU CAN EXAMINE THE CODE FROM $3000 TO $3FFF. THE COMPUTER WILL TURN OVER CONTROL OF THE COMPUTER TO THE ML MONITOR WHEN THE BRK COMMAND IS ENCOUNTERED.

SUBROUTINE # 4 MODIFIED CODE

```
,2100 A9 60    LDA #$60
,2102 8D FF 80 STA $80FF
,2105 20 FF 80 JSR $80FF
,2108 A9 21    LDA #$21
,210A 85 FC    STA $FC
,210C A2 00    LDX #$00
,210E 86 FB    STX $FB
,2110 A9 FF    LDA #$FF
,2112 A0 32    LDY #$32
,2114 91 FB    STA ($FB),Y
,2116 A0 35    LDY #$35
,2118 91 FB    STA ($FB),Y
,211A A0 3E    LDY #$3E
,211C 91 FB    STA ($FB),Y
,211E A0 4F    LDY #$4F
,2120 91 FB    STA ($FB),Y
,2122 A0 57    LDY #$57
,2124 91 FB    STA ($FB),Y
,2126 A0 70    LDY #$70
,2128 91 FB    STA ($FB),Y
,212A A9 0F    LDA #$0F      THE FOLLOWING CODE WILL BE APPARENT AFTER THE
,212C A2 08    LDX #$08      INDIRECT INDEXED ADDRESSING HAS RUN.
,212E A0 0F    LDY #$0F
,2130 20 BA FF JSR $FFBA     SET LOGICAL, 1ST & 2ND
,2133 20 C0 FF JSR $FFC0     OPEN FILE (15,8,15)
,2136 A9 01    LDA #$01
,2138 A2 F3    LDX #$F3
,213A A0 21    LDY #$21
,213C 20 BD FF JSR $FFBD     SET FILE NAME (#)
,213F A9 05    LDA #$05
,2141 A2 08    LDX #$08
,2143 A0 05    LDY #$05
,2145 20 BA FF JSR $FFBA     SET LOGICAL, 1ST & 2ND ADDR (5,8,5)
,2148 20 C0 FF JSR $FFC0     OPEN FILE
,214B A2 0F    LDX #$0F
,214D 20 C9 FF JSR $FFC9     OPEN CHANNEL FOR INPUT
,2150 A2 00    LDX #$00
,2152 BD F4 21 LDA $21F4,X
,2155 20 D2 FF JSR $FFD2     PRINT CHARACTERS TO DISK DRIVE (U1: 5 0 01 00)
,2158 E8       INX
,2159 E0 0C    CPX #$0C
,215B D0 F5    BNE $2152
,215D 20 CC FF JSR $FFCC     CLOSE I/O CHANNELS
,2160 A2 0F    LDX #$0F
,2162 20 C6 FF JSR $FFC6     OPEN CHANNEL FOR INPUT
,2165 A2 00    LDX #$00
,2167 20 CF FF JSR $FFCF     INPUT CHARACTER
,216A C9 32    CMP #$32      CMP TO $32 (ASCII 2) - CHANGE TO CMP $30
,216C D0 07    BNE $2175
,216E 20 CF FF JSR $FFCF     INPUT CHARACTER
,2171 C9 37    CMP #$37      CMP TO $37 (ASCII 7) - CHANGE TO CMP $30
,2173 F0 04    BEQ $2179
,2175 78       SEI
,2176 4C 76 21 JMP $2176     ENDLESS LOOP
,2179 4C 24 09 JMP $0924     START OF MAIN PROGRAM
```

There are a few programmers out there that have developed such a sophisticated protection scheme that it is not possible to directly load the program files in to memory and find their protection scheme. Some of these programmers will make the code so confusing that it is not worth the time to decipher and modify their protection scheme. Other programmers will save their programs on a special disk drive that will not store the information in the standard 1541 format. These disks will be read compatible with the 1541. This means that your drive will be able to read the information from the disk drive, but you will not be able to exactly duplicate the original disk when you copy it. Others will save their program on a disk, then put bad blocks on the original disk at the same blocks where the information is stored. When you load the program it will modify the DOS of your drive. They will actually bypass the error checking routine on your disk drive and read information from these bad blocks, without making the head beat up against the end stop. A few programmers will put multiple errors on a single block. When this bad block is read in the conventional manner one error will be returned, when the DOS is modified another error will be returned. Some disks will not contain all the tracks. When the disk was formatted it was done in a special disk drive. This drive may be programmed to skip tracks or to format the disk in any manner.

Any programmer who has the knowledge to modify the DOS is sure to have a good grasp of program protection. At least enough knowledge to keep the average person guessing for many hours (days) as to what is happening and why. The most sophisticated methods of program protection will involve hundreds of hours to write. Fortunately most programs are not protected in this manner. I don't even try to find out how or what the programmer is doing when the protection scheme is this complicated. What I do, is to let the program load into memory. Reset the computer, then dump the ML code to the disk. This is a much easier method of getting the code. Then all that is needed is to find the proper entry point to execute the code. Experience is a great help in finding this entry point, the more programs you have examined the easier it is to find. These types of protection are better left as they are. It is not that they can't be broken, it's that the time required to break them, in the conventional manner, is excessive.

When looking at these few selected programs you will find it easier to let the program load into memory and check for its own particular type of error. Then, just before the program is going to execute, reset the computer. Usually the computer will perform a normal reset. Now the whole program is in memory. Most of these programs will use a separate loader program. This loader program will load in the main program and check for errors. The main program usually does not contain any error checking routine. It just contains the actual program. After the main program has been loaded into memory you can now examine the program with a ML monitor. The program may now be SAVEd with the use of the ML monitor to another disk.

If you use the above method to bypass the error checking routine, you will find it much easier than trying to figure out what some of the more sophisticated protection schemes do. Occasionally, after you load the program into memory and reset the computer you may find that the computer does not want to reset normally. You push the reset button, the screen shrinks and the computer either locks up or the program begins executing. This is a result of the program storing the CBM80 at location $8000+. When ever the computer is reset it will check to see if the CBM80 is present, if it is the computer will turn over control to wherever the vectors point to. This can be particularly hard to get around. If you have a ML monitor that is cartridge based you can turn off your cartridge switch, insert your cartridge, turn the switch back on and then reset the computer. One problem with this is that code located from $8000 to $9FFF will be erased when the ML monitor is activated. Another problem is that when the computer is reset the code from $0000 to $0800 is reinitialized and any subroutines or special values stored there by the program will be erased. Another alternative is to look at each block of information on the disk until you find the CBM80, then change one of the bytes (i.e. change the '8' to '0'). When the reset button is pressed the computer will reset normally. All five bytes of the CBM80 must be present in order for control to be turned over to the vectors. Changing any one of the bytes will prevent this from happening. Don't forget to change the CBM80 back to normal before saving the program to disk.

Following is a step by step procedure to follow if you feel that the program protection scheme is to hard to break. This will allow you to get the program after the protection scheme has checked for its special type of protection.

1). LOAD and RUN the original program. Keep track of how long the program loads until the program executes. This will tell you when the program has passed its protection scheme.
2). LOAD the program a second time. Reset the computer just prior to it executing. It is necessary to reset the computer as close to its execution time as possible, within one second or less.
3). If the computer does not reset it will be necessary to search the disk for the CBM80. Change the '8' to '0' on the disk. This will prevent the auto start feature from taking over control when the computer is reset.
4). Load and execute the HIMON. Examine memory to find out where the program starts and ends. Remember to flip out BASIC and check to see if any code is stored in the RAM that underlies BASIC ROM. BASIC may be flipped out by storing $36 at location $0001.
5). Save the program to disk. Then try to find an entry point that will cause the code to execute (use the 'G' command).
6). Some programs will store valid ML code at the same location that the HIMON uses. To get the program that resides from $C000 to $CFFF it will be necessary to use one of the other ML monitors. Repeat steps 1 thru 4, only this time use one of the other monitors to down load the code from $C000 to $CFFF.

7). It may also be necessary to examine the ML code from $0000 to $0400 to see if any values have been stored at these locations. If they have, it will be necessary to save the area of code to the disk. It may be easier to write a little boot program that will store the proper values in affected locations.
8). Once the code has been saved to disk it will be necessary to properly load all the code back into memory and jump to the proper entry point.

This type of program breaking is the most involved and time consuming. It is not a job for the beginner. Do not start using the techniques presented in this chapter until you have mastered all the other chapters.

One question that every one asks is: How do you find the proper entry point to execute the program? Experience is the only answer, once you have examined enough programs it will become apparent where the program starts. Look for the area of memory that will change the color of the screen (STA $D021). Until you gain the necessary experience this is a good place to start looking.

Another good trick that some of the advanced programmers use is to store ML code on the screen. First they will set the character color to match the screen color or the 'RASTER' may be disabled. Either method will not allow any character to be seen on the screen. Then the program will load a file to the screen memory. The program will then execute the code contained at screen memory. If you were to reset the computer the screen would be initialized and the code would be erased. The best way to examine the code that is supposed to reside at the screen location is to use the program called 'U1 AND U2'. First it will be necessary to find where the information resides on the disk. If the program that loads data to the screen performs a block read, start by looking at the block that is read. Most programs that use this form of protection will do a valid block read. Use 'U1 AND U2' to load the block into memory. Load and execute the HIMON. The block will now be stored at $1000 to $1100. With the ML monitor it will be possible to examine and modify the routine as necessary. Once you have modified the code use the 'X' command to exit to BASIC. RUN the 'U1 AND U2' and use the same program to save the modified code back to the disk.

Another very good method of program protection is thru the use of a 'DONGLE'. A dongle is a plug in device that will act as a security key. This key is necessary for the program to run. The dongle may be inserted in the any port of the computer. Some may be very simple (a resistor across two terminals) or very sophisticated (4 I.C. chips including a shift register). Each dongle will perform in a different manner. Some may be bypassed very easily, others may not be bypassed with complete success.

Start by looking for the memory location that may be affected by the dongle. If the dongle plugs into the joystick port #1 look for the memory location $DC01 (or $D419). When you find the program doing something with this memory location you will probably be in the right area. Try to document what the program

is doing, comment the code fully. Sometimes it will be necessary to modify a few comparisions (CMP) or change the values that will be stored in memory. Many times the function of the dongle will become apparent when the code is fully understood.

Another alternative to the dongle is to make a circuit to duplicate the actions of the original dongle. This may be easier for some people who are familiar with TTL circuitry. By all means use the method that will provide the easiest method to produce a working copy of the original program.

As you can see there is not one sure method of breaking every program. Each program is unique and special in its own way. Programmers are getting more sophisticated every day. Each time someone finds a new method of program protection someone else will find a method of breaking it. I hope that you will be able to use some of the techniques presented in this book. Remember, start out by looking at programs that someone else has broken. If you can not break it yourself, then look to see how the other person did it. It is necessary to progress from the easy programs to the difficult. Don't start out by looking at programs that other people have not been able to break. Start out with the small fish, then go after the bigger ones. Pretty soon you will be able to break most anything that comes along.

GOOD LUCK!

PROTECTING YOUR OWN SOFTWARE

In this chapter I will recommend some methods of program protection that you may want to use on your own software. All the methods contained in this chapter will make use of the principles that you have learned throughout this book. First offer a duplicate disk to the legitimate purchaser at a reasonable price. This way it will not be necessary to pirate your software in order to obtain an archival copy.

The most important lesson that you have learned is that bad blocks are harmful to disk drive. If you are going to use bad blocks on your disk be sure that they are there only to foil pirates, not the legitimate user. By this I mean don't make the program read a bad block in order for the program to work. Place the bad blocks on the disk only to foil the back up programs. Store your files on the disk in such a manner that the blocks of the file are spread throughout the disk. Then place the bad blocks on the disk wherever the program does not reside. This way when the user loads the program they will not encounter the bad blocks. If someone tries to back up the disk up it will play havoc with their disk drive. Remember, the legitimate purchaser can buy the duplicate disk at a nominal fee.

Modify the disk directory so that it is no longer listable. Don't just change one byte of code, change fifteen or twenty bytes of code. Make the pirates earn their treasure. Too many programmers give their programs away by not adequately protecting their disks. Modify the disk directory so that it will become an endless loop. Change the name of the disk, the I.D. and the 2A to un-printable characters. Add a few bogus programs to the disk in order to trip up the file copy programs. These can be programs that will put the drive in an endless loop or send the drive to bad blocks. Add the special characters after the first $A0 in the program name. Use program names which are as unique as possible ($0D, 03).

Use a number of programs that load prior to the main program. Have your BOOT program load another program, this program will store a few values in memory and load another program. The third program will store a few more values in memory and load another program. Then have the fourth program store a few more values in memory and load the main program. The main program will contain a number of places that will check to see if the proper values have been stored in memory by the prior programs. Have the main program check the disk to see if any bytes have been change in the directory. If they have, cause the program to crash. It is possible to put so many small protection schemes on the disk that most any pirate will go crazy trying to find them all. Buy a good BASIC compiler, have some of your code in compiled BASIC. It is not usually worth the time to decipher the compiled code.

I hope that you get the drift of what I am trying to say. Make the disk so hard to crack that the pirates will waste a whole lot of time trying to break your program. If you set the program up

properly it will not be worth the time required to break it.

The biggest problem that software authors have today is the MODEM. Once a program is broken and placed on a bulletin board the program can travel across the country in a few days. Any one can down load your program in a matter minutes. If the program is set up so that it needs many smaller programs to load it into memory and it needs to see special information on the disk, most of the time if it does get broken it will not go very far. If you can keep the pirates from placing your software on the bulletin boards it will be worth the extra time taken to protect it.

Try to find someone who has the special programmable disk drive (good luck). Save the information to the disk in such a manner that it cannot be copied by the 1541 disk drive. Use some of the tips contained in this book to protect your software. Use some of your own. If you have to borrow the techniques used by other programmers, that's OK. Remember it is not the idea that is copyrighted, it is how the programmer expressed the idea.

Finally change the 'A' to an 'E' in the BAM (track 18, sector 0). This can be the most difficult trick for the software pirate to get around.

Any program can be broken. Any protection scheme can be gotten around. Sometimes the protection scheme will provide a challenge to the pirate. Sometimes the pirate will place a greater value on his time than he does on your program. At any rate, don't give your software away. Make the pirate work for the treasure.

PROGRAM DISK

The program disk contains many useful utility routines. These are the programs that I use when looking at program protection schemes. There are two other programs that you will need to purchase in order to have a complete tool kit. These are a track and block editor and a cartridge based ML monitor.

CLONE MACHINE (R) by MICRO-WARE DISTRIBUTING INC. This is an excellent program for editing track/blocks. This not the only good program for this purpose, it just happens to be the program that I use. CLONE MACHINE will allow you to view and edit individual blocks from the disk. This is similiar to DISK DR. DISK DR will work in decimal and will display the characters in screen codes. Whereas CLONE MACHINE will work in hex and ASCII. It may be necessary to use both programs when examining disks. The only function of CLONE MACHINE that I use is the edit track/block.

HESMON (R) by Human Engineered Software is a cartridge based ML monitor. This is an excellent program. It has many more functions than the ML monitors that are supplied on the program disk. The most important feature is that HESMON is contained on a cartridge. The ML monitor does not need to be loaded into memory, just plug in the cartridge. It will also dump the ML code to your printer so that a hard copy of program listing may be obtained.

The following programs are contained on the program disk supplied with this book. These programs are all public domain and may be duplicated by any one. This first group of programs are to be used with the book. These are tutorial programs and are not part of the tool kit.

TEST #1 is a simple BASIC program used to demonstrate the effects of simple pokes on the program.

KEYBRD BUFFER is used to demonstrate one of the uses that the keyboard buffer may be used for.

ERROR CHECK is used to check a specific block for errors. You may input the track and block to be examined. The disk drive will try to read the block. Then the error channel of the drive will be read and the error (if any) will be displayed on the screen. This is a simple and convenient way to check a block for errors.

RESTORE is a ML program that can be used to restore the pointers of a BASIC program after the computer has been reset. When the computer is reset the BASIC program becomes unlistable, the program still resides in memory, you just can't see it or run it. RESTORE will reset all the necessary pointers to make the program listable.

SUPER LINES is a simple BASIC program that has been modified to prevent tampering with the program. Many of the line number are over 64000. BASIC will not allow line number greater than 63999 to be entered. With the use of the ML monitor, the line numbers can be modified to any number up to 65535. Many of the line number have the same number. Yet when the program does a GOSUB, the program will return to the proper place.

MOVE BASIC is a simple program that will allow you to reset the pointers so that BASIC will reside higher in memory.

MOD POINTERS is a BASIC program that has been modified to hide some of the line numbers. The lines do exist, they just will not list when the pointers have been modified.

ML & BASIC is a simple BASIC program that has been relocated higher in memory and a short ML routine placed before the program. The ML routine is entered by the SYS command and the ML will turn over control to the BASIC program that resides higher in memory.

ML & BASIC #2 is a simple BASIC program that has been relocated higher in memory and a short ML routine placed before the program. This program will give some help to those people who can not find where the ML & BASIC program resides in memory.

TEST ML is a short ML routine. This is not a functioning program. It only purpose is to allow you to find where ML routine resides in memory. Look for the KERNAL subroutines and try to decipher the logic of the subroutine from $2100 to $2200.

USER FILES is a BASIC program that will allow a block of data to be loaded into memory. Additional blocks of data may also be loaded into memory and stored at successive memory locations. The data from the disk may be examined with the use of the ML monitors. The program will normally reside from 2048 to 2688 (decimal) in memory. Do not store any data lower than 2690 when using this program or the data will over-ride the program.

USER.WOW is the same program as USER FILES, only this is the compiled version. The compiled program is 32 blocks longer than the original version of the program. It will execute slightly faster than the BASIC version. Use this program for a comparison of a compiled program versus program written in BASIC. The compiled program will occupy memory from 2048 to 10768 (decimal). Do not store any data lower than 10770 when using this program or the data will over-ride the program.

The following programs are the actual utilities that you may find useful when examining program protection schemes. These programs are part of my tool kit.

DISK DR is a track and block editor. Use this program to examine and modify blocks on the disk. DISK DR will not work on disks that have their directories placed in an endless loop. It will show all the code in decimal and will display the screen codes on the screen. Sometimes it will be necessary to use both CLONE MACHINE and DISK DR when examining programs. The program is menu driven and easy to use. List the DISK DR for a pleasant surprise.

LLMON SYS8192, LOMON SYS32768 and HIMON SYS49152 are ML monitors. They are the same program, just relocated in memory. If you load a ML monitor into memory it will over-ride any code contained in that area. Some programs will store values at many different memory locations. It may be necessary to use the LLMON to examine one area of memory, then use the HIMON to examine the area of memory that was occupied by the LLMON. These monitors do not contain all the function of HESMON (R), but they are very good programs.

BLOCK AL & FREE is a short BASIC program that will allow you to ALLOCATE or FREE a block on the disk. This program will only modify the BAM, no data on the disk will be changed. When the directory has been extensively modified it may be necessary to use this program to ALLOCATE or FREE blocks on the disk.

APPEND is a very simple program that will allow you to append (join) two or more BASIC programs. First renumber the program lines of the programs to be appended, SAVE these programs to disk. LOAD and RUN the append program. LOAD the first program, then POKE 44,8:POKE43,1 'RETURN'. Your first program will be appended to the APPEND routine. RUN line 1 again, then load your second program. POKE 44,8:POKE 43,1 'RETURN'. The second program will be appended to the first. You can now delete the APPEND program.

I.D. CHECKER is a BASIC program that will allow you to read the actual I.D. from the disk. A few programs will need a special I.D. in order to RUN properly. Other programs will use an I.D. mismatch in order to generate an error. This program will display the I.D. of track 18 sector 0, then it will display I.D. of the track and sector to be examined, finally the actual track and sector number of the block. All the numbers are displayed in decimal, it will be necessary to convert these numbers into ASCII (use the chart provided in the memory map section). Pay attention to the track and sector numbers that are displayed when the disk contains certain errors. The track and sector numbers do not always contain the proper numbers.

DISK ADDR CHANGE is a program from your disk that was supplied with your disk drive. Use this to change the device number of your disk drive.

BACKUP 228 is a BASIC program that uses a ML routine to speed up

the program. This is a program that I use to make the first copy of a disk. BACKUP 228 will not put any errors on the destination (copy) disk. It will only transfer the data from the original disk. BACKUP 228 will read the actual I.D. from the original disk and will format the destination disk with this I.D. It will also check the original disk for the 'A' on track 18, sector 0. If the 'A' is not present the character contained there will be displayed. If this character is an 'E' special copy methods will have to be used to backup the disk.

1). Copy the disk from track 19, block 0 to track 35. Then copy the disk from track 1, block 0 to track 18.

If you have CLONE MACHINE (R) you can eliminate the 'E' from the destination disk.

1). Copy the disk from track 18, block 1 to track 35. Then copy the disk from track 1, block 0 to track 17. Do not copy track 18, block 0.
2). Load and execute the edit track/block feature of CLONE MACHINE (R). Insert the source disk and examine track 18, block 0. Change the 'E' to an 'A', do not press 'RETURN'. Remove the source disk, insert the destination disk and press 'RETURN'. You're done.

A few disks do not contain all 35 tracks on the disk. The disk will be prepared on a special programmable disk drive that will not format all of the tracks. Use BACKUP 228 to skip over these tracks (i.e. if track #2 is not present, copy the disk from track 1, block 0 to track 1. Then copy track 3, block 0 to track 35.).

DISK CHECKER is a simple BASIC program that will check 3 blocks on each track. Some disks will contain many tracks full of errors (#21 or #27 errors). Usually error #21 or error #27 will be full track errors and these tracks do not contain any information, just errors. If you check the disk with DISK CHECKER before making a copy, you may find these bad tracks. Use BACKUP 228 to make the copy and skip over these bad tracks. This will keep your disk drive from being beat to death.

U1 & U2 is a BASIC program that will allow you to load user files into memory, edit them and save them back to disk. The user files will be stored at memory location: hex $1000, decimal 4096. The ML monitor may be used to examine and modify these files. Use the 'load' function to store a user file in memory. Then load and execute a ML monitor to examine the files (at $1000 to $10FF). Exit ('X') to exit to BASIC. Run the U1 & U2 to save the file back to the disk.

COPYRIGHT NOTICE

WARRANTY AND LIABILITY

Neither C S M SOFTWARE, T. N. Simstad, nor any dealer distributing the product, makes any warranty, express or implied, with respect to this manual, the disks or any related item, their quality, performance, merchantability, or fitness for any purpose. It is the responsibility solely of the purchaser to determine the suitability of these products for any purpose.

In no case neither C S M SOFTWARE nor T. N. Simstad will be held liable for direct, indirect or incidential damages resulting from any defect or omission in the manual, the disk or other related items and processes, including, but not limited to, any interruption of service, loss of business, anticipated profit, or other consequential damages.

THIS STATEMENT OF LIMITED LIABILITY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Neither C S M SOFTWARE nor T. N. Simstad assumes any other warranty or liability. Nor do they authorize any other person to assume any other warranty or liability for it, in connection with the sale of their products.

UPDATES AND REVISIONS

T. N. Simstad and C S M SOFTWARE reserves the right to correct and/or improve this manual and the related disK at any time without notice and with out responsibility to provide these changes to prior purchasers of the program.

Although every attempt to verify the accuray of this document has been made, we cannot assume any liability for errors or omissions. No warranty or other guarantee can be given as to the accuracy or suitability of this book or the software for a particular purpose, nor can we be liable for any loss or damage arising from the use of the same.